EDÍCIA SKRÍPT

ZÁKLADY PROGRAMOVANIA V JAZYKU C

Zuzana Krivá

STU SvF
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
STAVEBNÁ FAKULTA

SPEKTRUM STU

ZÁKLADY PROGRAMOVANIA V JAZYKU C

Zuzana Krivá

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
2020

Recenzenti: Ing. Róbert Čunderlík, PhD.
Ing. Mariana Remešíková, PhD.

ISBN 978-80-227-4982-4

doc. RNDr. Zuzana Krivá, PhD.

ZÁKLADY PROGRAMOVANIA V JAZYKU C

Vydala Slovenská technická univerzita v Bratislave vo Vydavateľstve SPEKTRUM STU, Bratislava, Vazovova 5, v roku 2020.

Edícia skrípt

Rozsah 210 strán, 44 obrázkov, 16 tabuliek, 10,881 AH, 11,187 VH, 1. vydanie,
edičné číslo 6039, vydané v elektronickej forme;
umiestnenie na http://www.svf.stuba.sk

Schválila Edičná rada Stavebnej fakulty STU v Bratislave.

85 – 201 – 2020

ISBN 978-80-227-4982-4
ISBN 978-80-227-2486-6 (r. 2006 – 1. vyd.)

Úvod

Skriptá Základy programovania v jazyku C sú určené predovšetkým študentom ročníka študijného programu Matematicko-počítačové modelovanie ako študijný materiál pre predmet Programovací jazyk C, môžu ich však využiť aj doktorandi, ktorí potrebujú programovať v jazyku C. Sú zamýšľané predovšetkým pre študentov, ktorí nikdy neprogramovali, preto obsahujú veľa upozornení a podrobných vysvetlení. Predpokladá sa, že čitatelia raz budú zameraní na numerické výpočty technickej praxe a zodpovedal tomu aj výber príkladov pre prácu s dátami a jednoduchými dátovými štruktúrami. Odzrkadľujú osobnú skúsenosť autorky a snahu poskytnúť čitateľom vedomosti, ktoré v tejto oblasti budú potrebovať. Okrem toho bola veľká pozornosť venovaná práci so smerníkmi a tiež aj práci s textom. Záverečné kapitoly už skôr predstavujú doplnkový materiál, ktorý môže byť využitý pri práci s dlhšími programami.

Skriptá sú rozčlenené do dvanástich kapitol. Jedenásť kapitol predkladá novú látku a je doplnených o testy, kontrolné otázky, cvičenia a zadania, dvanásta kapitola sa snaží byť súhrnom predchádzajúcich jedenástich. Kapitoly sú pomerne rozsiahle a nie vždy sa ich podarí prebrať v rámci vyučovacieho procesu, preto sú napísané tak, aby sa dali zvládnuť formou samoštúdia. K tomu by mali napomôcť kontrolné testy na konci každej kapitoly (okrem poslednej) ktoré by mali overiť pochopenie pojmov a základných princípov opisovaných v danej kapitole. K testu sú uvedené aj výsledky v závere skrípt. Úlohou kontrolných otázok je overiť schopnosť manipulácie s pojmami a cvičenia už predpokladajú ich dobré porozumenie a využitie v programovaní. Autor považuje schopnosť čítať programy druhých programátorov za rovnako dôležitú ako písať vlastné, a preto zadania obsahujú vždy aj jeden príklad, kde má čitateľ zistiť, čo robí zadaný program.

Veľa príkladov je zameraných na prácu s obrázkami, pretože tie predstavujú spôsob ako ľahko získať „rozumné“ dáta pomerne veľkého rozsahu, na ktoré možno aplikovať rôzne algoritmy, často obsahujúce operácie bežné pri numerických výpočtoch praxi, a tým, že výsledok je grafická informácia, je výsledný efekt ľahko pozorovateľný a pomerne ľahko kontrolovateľný.

Za najpodstatnejšiu časť skrípt považujeme kapitolu 1až 9 . Kapitoly 10 a 11 predstavujú pokročilejšie techniky a ich zaradenie spolu so súhrnnou 12. kapitolou je vedené snahou poskytnúť študentom informácie o týchto technikách, aj keď sa predpokladá, že ich budú používať až po získaní určitých skúseností.

Poďakovanie. Autor ďakuje obom recenzentom Mgr. Mariane Remešíkovej, PhD. a Ing. Robertovi Čunderlíkovi, PhD. za starostlivé prezretie skrípt a za mnoho cenných praktických rád. Za veľa užitočných pripomienok autor ďakuje aj Mgr. Oľge Stašovej, Mgr. Zuzane Minarechovej a Doc. RNDr. Petrovi Volaufovi, CSc.

Autorka

1 Základné zložky počítača a algoritmizácia

Hlavné podtémy kapitoly:

- Pozičné číselné sústavy, dvojková číselná sústava.
- Zložky typického počítača, výpočet.
- Algoritmizácia. Vývojový diagram, pseudokód.
- Príklady algoritmov popísaných slovne: princíp EAN kódov, násobenie dvoch celých čísel, numerické hľadanie koreňa metódou delenia intervalu, Pearsonov korelačný koeficient, filtrácia zašumeného obrázku pomocou mediánu.

1.1 Dvojková (binárna) číselná sústava

Dvojková sústava, podobne ako desiatková sústava, je *pozičná* číselná sústava. Pozičnú číselnú sústavu vytvorili indickí vedci a urobili tak jeden z najdôležitejších matematických objavov. Pri zápise čísla v pozičnej číselnej sústave je hodnota (význam) jeho číslic daná ich pozíciou v zápise čísla. V desiatkovej sústave má číslo 2562 nasledovnú interpretáciu:

$$\begin{aligned} 2 \times 1000\ (10^3) &= 2000 \\ +\ 5 \times 100\ (10^2) &= 500 \\ +\ 6 \times 10\ (10^1) &= 60 \\ +\ 2 \times 1\ (10^0) &= 2 \\ &= 2562 \end{aligned}$$

Všimnite si, že dvojka má na rôznych pozíciách rôznu interpretáciu. Číslicu zapísanú v dvojkovom číselnom systéme interpretujeme podobne ako číslicu zapísanú v desiatkovej sústave, ale základom, ktorý sa umocňuje podľa pozície, je 2 a používajú sa číslice 0 a 1.

Prevod z dvojkovej do desiatkovej sústavy:
Číslo 1001001 predstavuje
$1 \times 2^6+0 \times 2^5+0 \times 2^4+1 \times 2^3+0 \times 2^2+0 \times 2^1+1 \times 2^0=64 + 0 + 0 + 8 + 0 + 0 + 1 = 73.$
Na číslo 73 sa môžeme pozrieť aj inak:
$73 = 2.36 + 1 = 2.(2.18) + 1 = 2^2.(2.9) + 1 = 2^3.(2.4 + 1) + 1 = 2^4.4 + 2^3 + 1 =$
$2^4.(2.2) + 2^3 + 1 = 2^5. 2 + 2^3 + 1 = 2^5.(2.1) + 2^3 + 1 = 2^6.(1) + 2^3 + 1 = 2^6 + 2^3+ 2^0.$
Z tohto zápisu vyplýva návod, ako možno desiatkové číslo previesť do dvojkovej sústavy.

Prevod z desiatkovej do dvojkovej sústavy: Preveďme číslo 75.

75:2=37:2=18:2=9:2=4:2=2:2=1:2=0

Zvyšky: 1 1 0 1 0 0 1

Reprezentácia v dvojkovej sústave: 1001011 (sú to zvyšky zapísané sprava). Podobne, číslo 73 má zápis 1001001.

Sčítavanie čísel v dvojkovej sústave:

```
  1 1 1
  1 0 1
1 1 0 0
```

Bitový posun doľava predstavuje násobenie dvoma.
Bitový posun doprava predstavuje celočíselné delenie dvoma.

Napríklad posuňme číslo 1001001(73) doľava tak, že nová pozícia je doplnená nulou:
1001001(73) ←10010010(146).
Posuňme číslo 1001001(73) doprava: 1001001(73) →100100(36).

Podobne v osmičkovej sústave je základ 8 a používajú sa číslice 0 až 7. V šestnástkovej sústave je základ 16 a používajú sa číslice 0 až 9 a písmená A(10), B(11),..., F(15).

1.2 Reprezentácia čísel, znakov, reálnych čísel

Počítačová pamäť je zariadenie schopné reprezentovať binárnu číslicu, t.j. musí mať dva identifikovateľné stavy, ktoré definujú „bit“ (skratka pre **b**inary dig**it**). Slová sú jednotky rovnakej veľkosti, ktoré majú jednoznačnú adresu. Veľkosť slov sa počíta v bitoch. Najmenšia pamäťová jednotka väčšia ako 1bit je *byte(bajt)* a je to zoskupenie 8 bitov.

Príklad 1
Aké najväčšie číslo sa dá zapísať do 8-bitového, 16-bitového slova?

Riešenie:
$11111111=2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0=2^8-1=255$ (súčet geometrickej postupnosti s kvocientom 2)
$1111111111111111= 2^{15}+2^{14}+...+2^0=2^{16}-1=65535$

Záporné čísla
Obyčajne bit úplne vľavo sa rezervuje na znamienko (znamienkový bit). 0 v znamienkovom bite znamená, že číslo je kladné a 1 že číslo je záporné.

Reálne čísla
Ich reprezentácia závisí od počítača. Číslo reprezentujeme pomocou pohyblivej desatinnej čiarky. Napríklad číslo 3 863 213 632 môžeme napísať ako 0.3863214×10^{10}. Zlomková časť je +.3863214 a exponent: +10. Číslo -0.00000002857 zapíšeme ako -0.286×10^{-7}. Zlomková časť je -.286 a exponent -7. Pri reprezentácii reálneho čísla v počítači vyčleníme priestor pre zlomkovú časť a exponent.

Znaky
Znak je základná textová informácia. Jeho reprezentácia sa rieši pomocou kódovacích schém. Dve najpopulárnejšie kódovacie schémy sú 8-bitová EBCDIC a 7-bitová ASCII. V jazyku C znaku zodpovedá **číselný kód** zodpovedajúci umiestneniu v ASCII tabuľke (pozri dodatok **ASCII tabuľka**).

Rozdiel medzi znakom a číslom
Pri programovaní treba rozlišovať medzi číslom a číslicou (číslica je znak). Klávesnica je znakové zariadenie, načítavame z nej znaky. Aj znak však v počítači musí byť reprezentovaný binárnou číslicou. Napríklad, keď zadáme číslo 255, postupne získame kód znaku 2 (ASCII kód je 50), kód znaku 5 (ASCII kód je 53) a kód znaku 5 (ASCII kód je 53). Ak chceme z týchto číslic získať číslo, musíme číslice na číslo konvertovať, prípadne môžeme použiť konverznú funkciu, ktorá nám tieto tri znaky (zaberajúce tri byty) prevedie na číslo 255, v dvojkovej sústave 11111111 (ktoré sa zmestí do jedného bytu). Okrem znakov tlačiteľných, ktoré majú kód v rozsahu 32 až 127, existujú znaky *riadiace,* ako napr. backspace, tabulátor, odstránkovanie, začiatok nového riadku apod. Tieto sú umiestnené na

začiatku tabuľky. Národné znaky (v prípade slovenčiny ide o znaky s diakritikou) sa kódujú na pozície 128 až 255.

1.3 Zložky typického počítača, výpočet

CPU (Central Processing Unit)
CPU obsahuje 2 zložky:

- *aritmeticko-logickú jednotku* – vykonávajú sa v nej všetky výpočty ako sčítanie, odčítanie, delenie alebo logické operácie ako porovnanie dvoch hodnôt,
- *riadiacu jednotku* – riadi napr. načítavanie dát zo vstupno/výstupných zariadení, posiela údaje z pamäte do aritmeticko-logickej jednotky na výpočet.

Inštrukcie pre riadiacu a aritmeticko-logickú jednotku musia byť vyjadrené v strojovom jazyku príslušného počítača. Je to postupnosť núl a jednotiek. Inštrukcia strojového jazyka obsahuje **operáciu,** ktorá sa má vykonať (napr. sčítanie, odčítanie, uloženie hodnoty) a jej **operandy** (t. j. s čím sa pri vykonávaní danej operácie manipuluje). Hardwarové obvody riadiacej jednotky zabezpečujú čítanie inštrukcií jednu po druhej z operačnej pamäte a indikujú operácie, ktoré sa majú vykonať na špecifikovaných operandoch.
Programy v strojovom jazyku sú vyjadrené tak, aby im počítač rozumel - t.j. jednoduchými operáciami vyjadrenými binárnymi číslami. Napríklad nasledujúca tabuľka uvádza množinu inštrukcií jednoduchého hypotetického počítača. Operácie sú reprezentované jednoduchými trojbitovými kódmi. Tento počítač je „jednoadresový“, čo znamená, že každá inštrukcia má jeden operand, špecifikovaný prostredníctvom svojej adresy (tiež vyjadrenej binárne). Zatiaľ si pod adresou môžeme predstaviť poradové číslo bytu, na ktorom je uložená inštrukcia alebo údaj, pričom číslujeme od nuly.
Predpokladá sa, že *akumulátor* je druhý operand tých inštrukcií, ktoré vyžadujú dva operandy. Pod akumulátorom si môžeme predstaviť špeciálne pamäťové miesto. Pri operácii vloženia hodnoty alebo kopírovania je pôvodná hodnota v pamäťovom mieste prepísaná.

Tabuľka 1 – príklad množiny inštrukcií

Kód operácie	Operácia	Význam
001	Load	Skopíruje hodnotu adresovaného slova do akumulátora
010	Store	Skopíruje hodnotu z akumulátora do adresovaného slova
011	Add	Nahradí momentálnu hodnotu akumulátora súčtom jeho momentálnej hodnoty a adresovaného slova
100	Subtract	Nahradí momentálnu hodnotu akumulátora rozdielom jeho momentálnej hodnoty a adresovaného slova
101	Branch	Skok na inštrukciu s uvedenou adresou
110	Branch if not zero	Skok na inštrukciu s uvedenou adresou, iba ak je obsah akumulátora rôzny od nuly
111	Halt	Zastavenie vykonávania programu

Príklad programu napísaného v takomto jazyku sú uvedené v tabuľke 2.

Tabuľka 2 predstavuje vzorový program. Program má dve časti. Prvá časť, pozostávajúca z desiatich osembitových inštrukcií, končí inštrukciou *Halt* a je uložená na adrese 0 až 9. Druhá časť obsahuje spracovávané údaje uložené na adresách 10 až 14. Kvôli prehľadnosti sú inštrukcie rozdelené do dvoch stĺpcov. Jeden stĺpec, vyznačený tučným písmom, obsahuje iba operačný kód a druhý iba adresu operandu. Ostatné informácie sú iba pomocné.

Tabuľka 2- vzorový program

	Inštrukcia			
Adresa inštrukcie	**Operačný kód**		**Adresa operandu**	
0	**001**	Load	**01010**	10
1	**010**	Store	**01100**	12
2	**001**	Load	**01110**	14
3	**011**	Add	**01011**	11
4	**010**	Store	**01110**	14
5	**001**	Load	**01100**	12
6	**100**	Subtract	**01101**	13
7	**010**	Store	**01100**	12
8	**110**	Bran.if not zero	**00010**	2
9	**111**	Halt	**00000**	

Aby sme porozumeli, ako spracováva riadiaca jednotka program, musíme si objasniť pojem **riadenia toku programu**. Vykonávanie postupuje postupne pamäťou, od jednej inštrukcie k druhej, uloženej fyzicky za ňou, okrem prípadu, keď narazíme na inštrukciu skoku, podmienenú alebo nepodmienenú. Vtedy urobíme odskok na uvedenú adresu. Vykonávanie programu sa zastaví na inštrukcii *Halt.*

Cvičenie 1

Vykonajte program, znázornený Tabuľkou 2. Pripomíname, že operand vždy predstavuje adresu, nie hodnotu. Nech je stav operačnej pamäte na adresách 10-14 takýto:

Adresa:	10	11	12	13	14
Hodnota:	00000011	00000100	00000000	00000001	00000000

Riešenie

Adresa:	10	11	12	13	14
Hodnota:	00000011	00000100	00000000	00000001	00001100

Akumulátor: 00000000

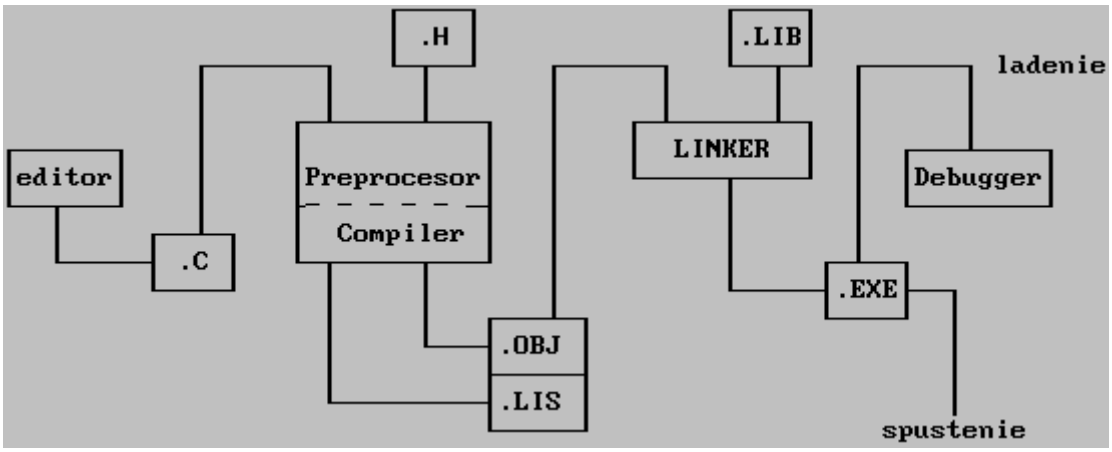
Program vynásobí čísla 3 a 4. Výsledok je uložený na adrese 14.

Kedysi naozaj programátori písali programy podobným spôsobom. Aj keď dnes sú množiny strojových inštrukcií oveľa dokonalejšie a programovacím jazykom vyššej úrovne sa programy píšu oveľa pohodlnejšie, princíp ostáva rovnaký. Výsledný program musí byť zrozumiteľný počítaču, a teda to musí byť postupnosť núl a jednotiek. Prevod do tejto výslednej formy za nás však robia špeciálne programy v spolupráci s operačným systémom.

Použitie počítača pri riešení úlohy programovaním

Dnešné programovacie jazyky umožňujú programátorovi písať programy vo forme viac vyhovujúcej ľudskému vnímaniu. Programy, ktoré sa nazývajú kompilátory, potom transformujú (prekladajú) program napísaný v programovacom jazyku do jazyka strojového kódu príslušného počítača.

Program píšeme obyčajne v nejakom textovom editore vhodnom na dané účely, alebo priamo v nejakom „vývojovom prostredí“. Výsledkom je tzv. *zdrojový súbor* (angl. *source*). Potom musíme dať program preložiť, zlinkovať a nakoniec ho môžeme spustiť, t. j. dať príkaz na jeho vykonanie. Jednotlivé fázy tvorby programu sú znázornené na nasledujúcom obrázku a popísané v ďalšom texte.



Editor: s jeho pomocou sa vytvárajú a opravujú *zdrojové súbory* s príponou .c.

Preprocesor: je to súčasť prekladača, ktorá upravuje zdrojový súbor tak, aby mal prekladač ľahšiu prácu, napr. vynechá komentáre, zaisťuje správne vloženie hlavičkových (**.h**) súborov, rozvoj makier, atď. Výsledkom jeho práce je znova textový súbor, ten si však môžete pozrieť iba vtedy, keď viete spustiť preprocesor samostatne bez kompilátora, inak preprocesor odovzdá výsledky svojej práce priamo svojemu nadriadenemu - kompilátoru (podrobnejšie pozri kapitolu 11).

Compiler: v slovenčine nazývaný tiež prekladač alebo kompilátor, vykonáva preklad zdrojového textu spracovaného už preprocesorom. Počas prekladu kompilátor kontroluje, či boli dodržané pravidlá programovacieho jazyka, robí takzvanú lexikálnu analýzu (kontrolu syntaxe). V prípade chýb, vypisuje chybové hlásenia. Vtedy programátor musí chyby opraviť a znovu dať program preložiť. Program – ešte textový súbor - sa preloží do relatívneho kódu počítača - vzniká **.OBJ** súbor. Relatívny kód je takmer hotový program. Slovo relatívny znamená, že adresy premenných alebo funkcií nie sú ešte celkom známe. Vedľajší produkt prekladača je takzvaný protokol o preklade (.lis), v ktorom sú uložené informácie o chybách nájdených prekladačom.

Linker: alebo zostavovací program prepočíta relatívne adresy na absolútne a prevedie všetky odkazy na dosiaľ neznáme identifikátory na volané knižnicové (už napísané a teraz iba používané) funkcie uložené v súboroch **.LIB.** Výsledkom práce linkera je priamo spustiteľný program **.EXE.**

Debugger: jeho slovenský preklad je "odvšivovač", ale viac sa používa pojem "ladiaci" program. Slúži na ladenie, alebo hľadanie chýb, ktoré nastávajú pri behu programu. Po najdení chyby sa celý cyklus (editor, compiler, linker, debugger) opakuje tak dlho, až si myslíme, že náš program žiadnu chybu neobsahuje.

1.4 Algoritmizácia

Algoritmus

Pri písaní programu musíme pamätať na tieto zásady:

Počítač robí to, čo mu povieme, a nie to, čo chceme, aby urobil.
V programe nesmie byť žiadna dvojznačnosť ani možnosť alternatívnej interpretácie.

Vytváranie programu na počítači môžeme rozdeliť do dvoch fáz:

1. fáza riešenia problému
2. implementačná fáza.

Vo *fáze riešenia problému* sa sústredíme na návrh zoradenej postupnosti krokov, ktorá popisuje riešenie daného problému. Táto postupnosť krokov sa nazýva algoritmus.
Príkladom klasickej formy algoritmu je recept. Algoritmus musí spĺňať dve dôležité vlastnosti: 1. jednotlivé kroky algoritmu musia byť jednoduché a jednoznačné, 2. algoritmus musí byť efektívny, t.j. musí vyriešiť problém v konečnom počte krokov.
Vo fáze riešenia problému najprv sformulujeme algoritmus s minimálnym počtom detailov tak, aby bol vyjadrený všeobecný spôsob, ako riešiť problém. Potom upresňujeme algoritmus dovtedy, pokiaľ sa nepresvedčíme, že je v schopnostiach počítača algoritmus vykonať.

Neexistuje jednoznačný návod ako vyriešiť daný problém. Treba si vybudovať zručnosti získané skúsenosťami a zdokonaľovať schopnosti nachádzať riešenia.

Konečné spresnenie algoritmu by malo vyústiť v počítačový program. Zapísanie algoritmu v konkrétnom programovacom jazyku sa nazýva *implementácia*.

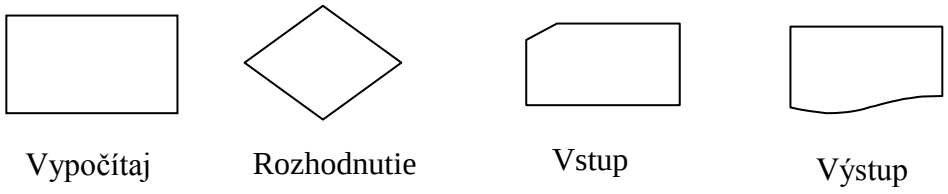
Metódy vyjadrenia algoritmu

1. Verbálne vyjadrenie algoritmu. Typickým príkladom je kuchynský recept alebo návod k spoločenskej hre. Prirodzený jazyk je však často ako nástroj popísania algoritmu nevhodný a nepresný, s rizikom dezinterpretácie alebo straty informácie. Iste ste sa už stretli s prípadom, že pravidlá nepopisovali hru jednoznačne, t.j. nepopisovali každú situáciu, ktorá sa môže vyskytnúť, a keď nastala, nebolo jasné ako ďalej postupovať. Také pravidlá alebo návod sú príkladom nedokonalého algoritmu. Iným známym príkladom nejednoznačného pokynu je veta „Meľte tri dni staré rožky!“

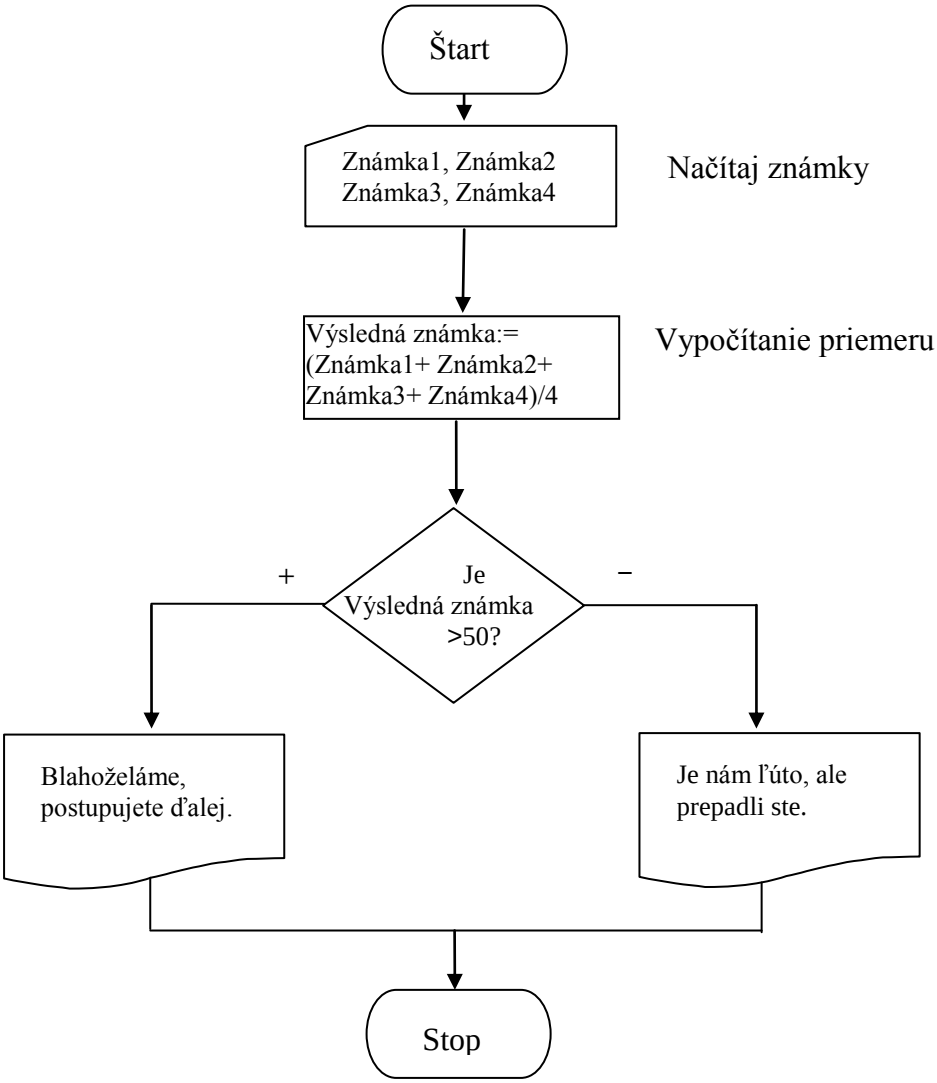
Príklad 2

Majme takýto problém: konečná známka študenta v kurze sa vypočíta ako priemer štyroch známok za štyri testy počas trvania kurzu. Algoritmus má vypočítať konečnú známku a stanoviť, či študent prechádza (známka je väčšia alebo rovná ako 50) alebo neprechádza (známka je menšia ako 50).
Verbálne vyjadrenie algoritmu:
Načítaj množinu štyroch známok. Vypočítaj priemer ich sčítaním a vydelením súčtu štyrmi. Ak je výsledná známka menšia ako 50, označ ju ako „neprechádza“, ak je väčšia alebo rovná 50, označ ju ako „prechádza.“

2. Vyjadrenie pomocou vývojového diagramu. Je to grafické vyjadrenie postupu. Vývojové diagramy ukazujú logiku algoritmu, zdôrazňujú jednotlivé kroky a ich prepojenie, t.j. spôsob, ako riadenie prechádza z jednej činnosti do druhej. Postupne sa pre vývojové diagramy vyvinula pomerne štandardná symbolika.



Svojho času boli vývojové diagramy veľmi populárne. Ich popularita postupne upadala a názory na ne sa rôznia: určité vlastnosti programov sa pomocou vývojových diagramov ťažko vyjadrujú.
Vyjadrenie predchádzajúceho programu pomocou vývojového diagramu:



3. *Algoritmická notácia (pseudokód).* Ide o zmes expresivity a intuície prirodzeného jazyka s logickou presnosťou schémy ako je vývojový diagram. Používame prvky prirodzeného jazyka a programovacieho jazyka. Pseudokód môže byť formalizovaný a nemusí závisieť od konkrétneho programovacieho jazyka.

Teraz si ukážeme návrh programu zhora – nadol, t. j. **top - down**. V prvej fáze popíšeme algoritmus slovne.

1. Načítaj množinu štyroch známok.
2. Vypočítaj ich priemer ich sčítaním a vydelením súčtu štyrmi.

3. Ak je výsledná známka menšia ako 50, označ ju ako „neprechádza“, inak ju označ ako „prechádza.“

V ďalšej fáze vyjadríme algoritmus pomocou pseudokódu. V pseudokóde obyčajne používame pojem premennej. **Premenná** je pomenovaná pozícia v pamäti, prístupná pomocou identifikátora (mena). Hodnota uložená v premennej sa môže v priebehu výpočtu meniť.

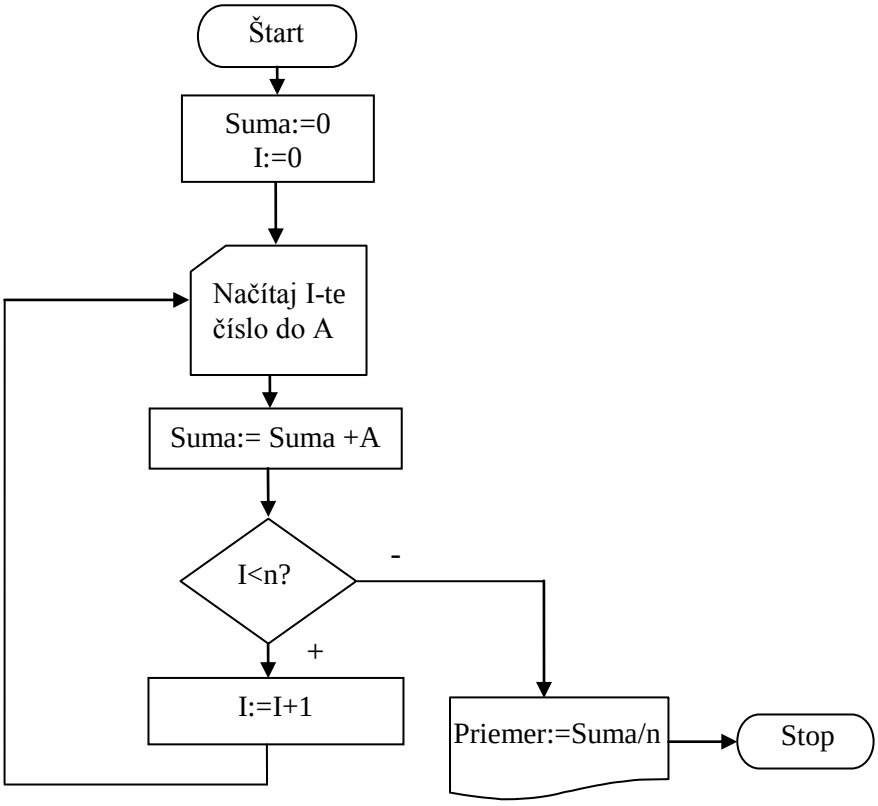
1. Získaj štyri známky zo vstupného zariadenia a ulož ich do premenných Známka1, Známka2, Známka3, Známka4.
2. Výsledná_známka = (Známka1 + Známka2 + Známka3 + Známka4)/4
3. **if** Výsledná_známka<50 (ak)
   then zobraz „Je nám ľúto, ale prepadli ste.“ (tak)
   else zobraz „Blahoželáme, postupujete ďalej.“ (inak)
4. Stop.

Posledná fáza je fáza implementácie algoritmu.

1.5 Príklady algoritmov popísaných slovne, vývojovým diagramom alebo pseudokódom

V ďalšom texte uvedieme niekoľko príkladov algoritmov. Posledné tri z nich sú príkladom numerického a štatistického algoritmu a algoritmu zo spracovania obrazu.

Algoritmus 1 (Výpočet priemeru *n* čísel)
Načítajme *n* čísel zo vstupu, kde *n* je ľubovoľné prirodzené číslo. Vypočítajte priemer z načítaných čísel.
Riešenie
Tento problém je algoritmicky jednoduchý, musíme ho iba vyriešiť „technicky.“



Algoritmus 2

EAN13 kód slúži k jednoznačnej identifikácii výrobkov, objektov, miest, služieb... S týmto kódom sa stretávame denne v obchodoch pri platení. Identifikačné čísla je možné zobrazovať pomocou symbolov čiarového kódu, aby mohli byť čítané čítačkou. Logika celého systému zaručuje, že dáta zosnímané z čiarových kódov produkujú jednoznačné elektronické správy a ich spracovanie je možné v plnom rozsahu vopred naprogramovať.

Čiarový kód EAN-13 sa skladá z 13 číslic, ktoré sú rozdelené do 4 skupín: 1) číselný systém (krajina) 2) kód výrobcu v danej krajine 3) kód výrobku u daného výrobcu a 4) kontrolná číslica. Kód je vymyslený tak, aby pri nesprávnom načítaní jednej číslice kontrolný mechanizmus chybu odhalil.

Kód znázorníme ako $a_1a_2a_3..... a_{12}p$. Kontrolná číslica p je nastavená tak, aby platilo $(a_1 + 3a_2 + a_3 + 3a_4 + ... + a_{11} + 3a_{12} + p) \ mod\ 10 = 0$. ($a\ mod\ b$ je zvyšok po delení celého čísla a celým číslom b). Teda kontrolná číslica je nastavená tak, aby vyznačený súčet bol deliteľný desiatimi.

Ako vyzerá algoritmus rozhodujúci, či je kód načítaný správne alebo nesprávne?

Riešenie:

1. zosnímaj kód
2. vypočítaj hodnotu súčtu $s_t = a_1 + 3a_2 + a_3 + 3a_4 + ... + a_{11} + 3a_{12}$
3. vypočítaj hodnotu p_t kontrolnej číslice ako $(10 - s_t\ mod\ 10)\ mod\ 10$.
4. **if** $p_t \neq p$ **then** výhlas chybu **else** kód považuj za OK.

Porozmýšľajte, prečo je v bode 3 dvakrát použitá operácia *mod* 10.

Algoritmus 3

Násobenie dvoch prirodzených čísel „sedliackou metódou“.

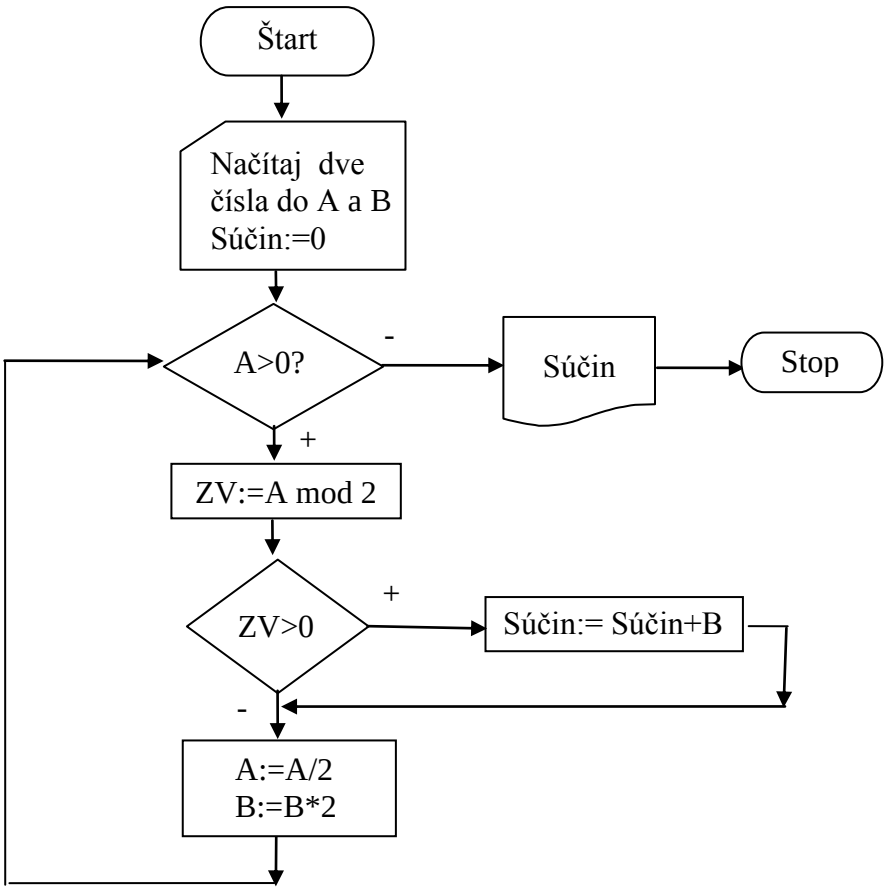
Podľa „sedliackej metódy“ by sme dve celé čísla mohli vynásobiť takto:
Prvé číslo budeme stále celočíselne deliť dvoma a druhé budeme násobiť dvoma.
Vytvoríme nasledovné dva stĺpce:

21	17
~~10~~	~~34~~
5	68
~~2~~	~~136~~
1	272
	357

V ľavom stĺpci párne čísla vyškrtáme a súčasne v pravom stĺpci vyškrtáme čísla v tých istých riadkoch ako boli párne čísla vľavo. V pravom stĺpci sčítame nevyškrtané čísla.

Na akom princípe je tento algoritmus založený?
Číslo 21 môžeme napísať ako:
$21=1*2^0+0*2^1+1*2^2+0*2^3+1*2^4$. V ľavom stĺpci vlastne vytvárame zápis čísla v dvojkovej sústave. Párnym číslam v ľavom stĺpci zodpovedajú koeficienty nula v dvojkovom zápise čísla.
Celý súčin môžeme zapísať nasledovne:
$21*17 = (1*2^0 + 0*2^1 + 1*2^2 + 0*2^3 + 1*2^4)*17 = 1*2^0*17 + 0*2^1*17 + 1*2^2*17 + ...$
Jednotlivé ščítance posledného zápisu vytvárame v pravom stĺpci.

Pomocou vývojového diagramu zapíšeme algoritmus takto:



Iste by ste si vedeli predstaviť aj iný algoritmus na vynásobenie dvoch celých čísel. Napríklad by sme mohli 17x pričítať k nule číslo 21. Mnoho problémov môže byť riešených viacerými algoritmami. Zrejme jedným dobrým kritériom, ktorý z nich je lepší, môže byť rýchlosť výpočtu. Ako už bolo povedané, zápis daného algoritmu v konkrétnom programovacom jazyku sa nazýva implementácia. Aj pre jeden algoritmus môže byť implementácia rôzna. Napríklad násobenie čísla dvomi môžeme urobiť „klasickým“ spôsobom, teda použijeme operáciu *2, môžeme dvakrát sčítať to isté číslo, alebo môžeme urobiť rýchlu operáciu bitového posunu doľava. Veľakrát nám záleží na tom, aby implementácia bola čo najrýchlejšia. Môžu byť však aj iné kritéria ako rýchlosť: napríklad dobrá čitateľnosť alebo modifikovateľnosť.

Algoritmus 4
Numerické hľadanie koreňa metódou postupného delenia intervalu.
Koreňom funkcie jednej premennej $f(x)$ nazývame takú hodnotu x, že $f(x)=0$. Pre lineárne alebo kvadratické funkcie vieme koreň nájsť priamym výpočtom alebo podľa vzorca. Pomerne komplikované vzorce existujú aj pre polynómy stupňa 3 a 4. Pre polynómy stupňa vyššieho ako 4 však už vzorce neexistujú. Pre takéto funkcie (ale aj pre funkcie, ktorých korene vieme vypočítať pomocou vzorcov) existujú pomerne všeobecné numerické metódy, ktoré nájdu koreň s vopred zadanou presnosťou. Známe sú napr. metódy postupného delenia intervalu alebo Newtonova metóda. Ukážeme si prvú.

Majme **spojitú** funkciu. Ak nájdeme dva body x_1 a x_2, v ktorých ma funkcia rôzne znamienka, tak koreň bude určite ležať medzi nimi. Navyše platí $f(x_1) * f(x_2) < 0$. Rozdelíme

interval na dve polovice. Pre jednu z nich musí platiť, že $f(x_1) * f(x_2) < 0$, kde x_1 a x_2 sú krajné body rozdeleného intervalu. V tej musí ležať koreň. Tú ďalej delíme, až kým sa s predpísanou presnosťou nepriblížime ku koreňu rovnice. Procedúry založené na tejto metóde obyčajne skončia po nájdení prvého koreňa, aj keď v zadanom intervale môže byť koreňov viac.

Slovný zápis algoritmu je nasledovný:

1. Zadajte dva body x_1 (ľavá hranica intervalu) a x_2 (pravá hranica intervalu), v ktorých má funkcia rôzne znamienka. (Napríklad ich zistite z grafu funkcie).
2. Nájdite r, t.j. stred (x_1 , x_2).
3. Ak f(r)= 0, tak r je koreňom. Tiež ak $|f(r)|$ je v rámci nejakej predpísanej tolerancie, tak za koreň budeme považovať r.
4. Ak f(r) má to isté znamienko ako $f(x_1)$, tak opakujme proces s $x_1=r$, inak opakuj proces s $x_2= r$, t.j. choď na bod 2..

Algoritmus 5
Spearmanov korelačný koeficient.

Dvojrozmerný štatistický súbor je súbor, ktorý obsahuje vždy dvojice hodnôt (meraní) týkajúcich sa sledovanej udalosti (napr. výška a váha človeka, prah počuteľnosti pravého a ľavého ucha, výška vodného toku na dvoch rôznych miestach a pod). Kvantitatívnu mieru závislosti medzi dvomi zložkami určuje *korelačný koeficient.* Spearmanov korelačný koeficient je číslo z intervalu < -1, 1 >. Ak je rovné nule, zložky sú navzájom nezávislé, ak je Spearmanov korelačný koeficient v absolútnej hodnote blízky jednej, náhodne veličiny sa navzájom ovplyvňujú. Postup pri výpočte tohto koeficientu je takýto: usporiadame vzostupne namerané hodnoty x_i a hodnoty y_i, nech R_i je poradie hodnoty x_i v usporiadaní a Q_i je poradie hodnoty y_i v usporiadaní. Pre rovnaké namerané hodnoty určíme poradie ako aritmetický priemer *i*-poradí rovnakých hodnôt. Nech n je veľkosť štatistického súboru (t.j. počet dvojíc hodnôt), Spear-manov korelačný koeficient vypočítame podľa vzorca

$$r_s = 1 - \frac{6}{n(n^2-1)} \sum_{i=1}^{n} (R_i - Q_i)^2 .$$

Skúste si nakresliť vývojový diagram pre daný postup. V tomto prípade sa stretneme s častou úlohou v programovaní, a to usporiadaním čísel vzostupne alebo zostupne. Skúste si napísať jednoduchý algoritmus pre usporiadanie čísel. Tomuto problému sa budeme ďalej venovať podrobnejšie. V jazyku C existuje aj efektívna funkcia pre usporiadanie, ktorú sa naučíme vyvolať. S usporiadaním prvkov sa stretneme aj v ďalšom príklade.

Algoritmus 6
Filtrácia obrázku pomocou mediánu.

Predstavme si obrázok, ako obdĺžnikovú tabuľku celých čísel v rozmedzí 0 až 255, kde 0 predstavuje čiernu farbu, 255 bielu farbu a ostatné hodnoty predstavujú úrovne šedej. Predstavme si, že máme obrázok, ktorý sa nám „pokazil“ tak, že napr. na 10% jeho pozícii sa skutočné hodnoty nahradili náhodnými číslami (celé čísla v rozmedzí 0 až 255). Na „opravenie“ obrázku použijeme algoritmus, založený na vypočítaní mediánu, t.j. prostrednej hodnoty v usporiadanom súbore.

Medián M_e , t.j. stredná alebo prostredná hodnota v poradí (v zmysle usporiadania) je počítaná podľa vzorca

$$\mathbf{M_e} = \begin{cases} x_{k+1} & , \quad ak \quad n = 2k+1 \\ \dfrac{x_k + x_{k+1}}{2} & , \quad ak \quad n = 2k \end{cases}$$

kde *n* je počet prvkov súboru.

Napr. vezmime si súbor čísel 3 5 4 2 1 7 3 . Čísla usporiadajme, dostaneme: 1 2 3 3 4 5 7 . Prostredné číslo v tomto usporiadaní je číslo 3, teda mediánom je 3. Vezmime si súbor čísel 3 5 4 2 1 7 3 8 . Po usporiadaní dostaneme postupnosť 1 2 3 3 4 5 7 8. Keďže počet prvkov tejto postupnosti je párny, musíme zobrať 4. a 5. číslo a vypočítať ich priemer. Mediánom druhej postupnosti je číslo 3,5.

Pre výpočet mediánu existuje mnoho algoritmov, my použijeme jednoduchý algoritmus založený na usporiadaní vstupnej postupnosti čísel (čo je v súlade s definíciou a uvedeným príkladom) . V C jazyku medián nájdeme ľahko pomocou zabudovanej procedúry pre usporiadanie. Výpočet spočíva v tom, že pre každý bod obrázku – pixel – nájdeme jeho priamych susedov, t.j. pixle, ktoré ho obklopujú, je ich 8. Z týchto deviatich pixlov, t.j. spracovávaného pixlu a jeho ôsmich susedov, nájdeme medián. Pôvodnú hodnotu obrázku nahradíme mediánom. Ak je v „hladkom“ obrázku, t.j. obrázku so spojitými prechodmi nejaká hodnota „uletená“, t.j. ide o šum, dostane sa na začiatok alebo koniec usporiadania. Táto hodnota určite nebude mediánom, a preto bude nahradená hodnotou bližšou okolitým pixlom.

Skúste si zostaviť algoritmus a zamyslieť sa nad tým, aké problémy budete musieť riešiť.

Nasledujúci obrázok ukazuje príklad obrázku zašumeného opísaným typom šumu a výsledok práce mediánového filtra.

☑ Kontrolný test

1. Binárne číslo 1011011 v desiatkovej sústave predstavuje číslo:
   a) 92
   b) 91
   c) 95
   d) 93
2. Desiatkové číslo 137 v dvojkovej sústave predstavuje číslo:
   a) 10001001
   b) 1001001
   c) 10001101
   d) 10001000
3. Osmičkové číslo 765 v desiatkovej sústave predstavuje číslo:
   a) 766
   b) 603
   c) 501

d) 496

4. Desiatkové číslo 5149 v osmičkovej sústave predstavuje číslo:
   a) 5100
   b) 5101
   c) 5102
   d) 12035
5. Šestnástkové číslo 193 v desiatkovej sústave predstavuje číslo:
   a) 402
   b) 403
   c) 411
   d) 405
6. Desiatkové číslo 17 v šestnástkovej sústave predstavuje číslo:
   a) 13
   b) 11
   c) 15
   d) 18
7. Maximálne číslo, ktoré sa zmestí do 32-bitového slova, ak nepoužívame znamienkový bit je:
   a) 4294967295
   b) 4294967966
   c) 2147483647
   d) 4294967296
8. Kompilátor je:
   a) prostredie, v ktorom sa píše program
   b) strojový ekvivalent zdrojového programu
   c) program transformujúci program napísaný v programovacom jazyku do jazyka strojového kódu príslušného počítača
   d) program vyjadrený pomocou vývojového diagramu
9. Implementácia je:
   a) vytvorenie algoritmu pre daný problém
   b) zoradená postupnosť krokov popisujúca riešenie daného problému
   c) zapísanie algoritmu v konkrétnom programovacom jazyku
   d) syntaktická kontrola algoritmu
10. Vyberte nesprávne tvrdenie. EAN kód je kód, ktorý:
    a) Slúži k jednoznačnej identifikácii výrobkov, objektov, miest, služieb...
    b) Identifikačné čísla je možné zobrazovať pomocou symbolov čiarového kódu, aby mohli byť čítané čítačkou.
    c) sa skladá z 13 číslic a jedného kontrolného bitu
    d) kontrolná číslica p je nastavená tak, aby platilo
    $(a_1 + 3a_2 + a_3 + 3\,a_4 + \ldots\ldots + a_{11} + 3\,a_{12} + p)$ *mod* $10 = 0$ (*a mod b* je zvyšok po delení celého čísla *a* celým číslom *b*).
11. Máme dvojrozmerný štatistický súbor. Spearmanov korelačný koeficient je:
    a) kladné číslo vyjadrujúce závislosť medzi dvoma veličinami
    b) číslo z inetrvalu <-1,1> vyjadrujúce závislosť medzi dvoma veličinami
    c) číslo, ktoré ak rovné nule, tak sú veličiny silne závislé
    d) číslo, ktoré ak je záporné, tak sa veličiny silne ovplyvňujú.
12. Medián je číslo, ktoré:
    a) sa vždy rovná aritmetickému priemeru daných dát

b) počíta sa len pre nepárny počet dát
c) vždy sa rovná jednému z dát, pre ktoré sa počíta
d) pre nepárny počet dát sa rovná prostrednej hodnote v zmysle usporiadania.

Kontrolné otázky

1. Uveďte názvy dvoch schém, pomocou ktorých sa v počítači kódujú znaky.
2. Vymenujte základné zložky CPU a popíšte ich význam.
3. Čo je to kompilátor a čo je to linkovací program?
4. Čo je to implementácia?
5. Aké môžu byť kritériá pre kvalitu implementácie?
6. Aké vlastnosti musí mať algoritmus?
7. Čo je to pseudokód?
8. Vynásobte čísla 8 a 16 „sedliackou metódou“.
9. Čo sa stane ak číslo zapísané v dvojkovej sústave posunieme o bit doľava alebo doprava? Vyskúšajte.
10. Popíšte postup, ako hľadáme koreň funkcie metódou postupného delenia intervalu. Akú vlastnosť musí mať daná funkcia? Vymyslite funkciu a interval, pre ktoré by daný algoritmus nefungoval.
11. Aký je význam korelačného koeficientu?
12. Charakterizujte medián.
13. Popíšte princíp filtrovania mediánom. Na aké obrázky je vhodný a na akom pricípe je založený?
14. Čo je to premenná?

Cvičenia

1. Uveďte príklad číselnej sústavy, ktorá nie je pozičná.
2. Pomocou inštrukcií hypotetického počítača napíšte program, ktorý vydelí dve prirodzené čísla. Prvé je väčšie, druhé je menšie a prvé je deliteľné druhým.
3. Ako by ste čo najjednoduchším spôsobom sčítali 10 čísel uložených v pamäti za sebou?
4. Zostrojte vývojový diagram pre výpočet faktoriálu.
5. Zostrojte algoritmus pre vyhľadanie minimálneho a maximálneho čísla v číselnej postupnosti.
6. Slovne popíšte jednoduchý algoritmus pre usporiadanie čísel vzostupne.
7. Ukážte, že EAN kód naozaj nájde jednu chybu, zatiaľ čo chybu v nesprávnom načítaní dvoch susediacich čísel nemusí odhaliť.
8. Pre Algoritmus 6 vypíšte niekoľko príkladov činností, ktoré budete musieťzvládnuť.
9. V tabuľke je daný nasledovný dvojrozmerný súbor:

x_i	23,1	12,8	17,8	21,3	18,5	93,5
y_i	25,9	15,1	20,4	23,5	21	105,9

Vypočítajte Spearmanov korelačný koeficient [výsledok 1].

10. V tabuľke je daný nasledovný dvojrozmerný súbor:

X	5	6	11	13	20	21	22	22	25	26	27	31	34	37	37	41	42	44	49	50
Y	6	7	6	8	13	20	21	23	15	20	22	17	30	31	33	14	39	31	39	41

Vypočítajte Spearmanov korelačný koeficient (dáta sa opakujú)[0,852256].

11. Vypočítajte medián pre dáta: 3, 1, 6, 9, 11, 15, 15, 15, 6, 6, 9 [9].
12. Vypočítajte medián pre dáta: 3, 1, 6, 9, 11, 15, 15, 15, 6, 6 [7,5].

13. V poslednej dobe je veľmi populárna hra, ktorá sa nazýva Sudoku. Majme nasledovnú mriežku.

		8			5	1	4	
2					4			3
3			9					8
	4			9				
		3	4		6	7		
				7			8	
7					9			2
1			2					6
	6	2	1			5		

Mriežku treba vyplniť tak, aby každý stĺpec, riadok i štvorec s rozmermi 3x3 políčka obsahoval čísla od 1 do 9. V štvorci, riadku a stĺpci sa čísla nesmú opakovať. Hra sa lúšti iba logickou úvahou. Obrázok vpravo predstavuje riešenie pre zadanie vľavo.

6	9	8	3	2	5	1	4	7
2	5	1	7	8	4	9	6	3
3	7	4	9	6	1	2	5	8
5	4	7	8	9	3	6	2	1
8	2	3	4	1	6	7	9	5
9	1	6	5	7	2	3	8	4
7	3	5	6	4	9	8	1	2
1	8	9	2	5	7	4	3	6
4	6	2	1	3	8	5	7	9

Skúste vymyslieť algoritmus, ktorý by vyriešil ľubovoľné Sudoku. Popíšte ho verbálne. Zamyslite sa nad obtiažnosťou problému, čo sa týka nájdenia postupu. Myslíte, že existuje iba jedno riešenie? Aký máte názor na obtiažnosť programovania takéhoto algoritmu? V čom je zložitejší trebárs proti algoritmu filtrovania mediánom?

Na precvičenie uvádzame nasledovné príklady. Prvé dva sú ľahké, tretí stredne obtiažny a posledný je ťažký.

4		3				1		8
	8	2				3	7	
			3		9			
		9	1		6	4		
	2						9	
		7	4		8	2		
			7		3			
	3	1				7	2	
7		8				6		3

4		3				1		8
	8	2				3	7	
			3		9			
		9	1		6	4		
	2						9	
		7	4		8	2		
			7		3			
	3	1				7	2	
7		8				6		3

		8				9		
	7			9			3	
3			8	5	2			6
2			5		8			9
		6				4		
1			6		3			2
9			1	2	7			4
	6			8			9	
		4				1		

3			7				6	
4					1			
	8				6		1	3
2							8	
		4	9		3	5		
	7							9
9	5		1				7	
			2					6
	1				5			8

Zadania

1. Analyzujte nasledovný program a určte obsah akumulátora po jeho dokončení. Prepíšte program pomocou binárnych čísel.

1	LOAD	9
2	SUB	11
3	STORE	9
4	LOAD	10
5	SUB	12
6	STORE	10
7	BRANCH IF NOT ZERO	1
8	HALT	
9	100	
10	5	
11	10	
12	1	

2. Napíšte program, v ktorom sa vypočíta priemer štyroch čísel deliteľných štyrmi v strojových inštrukciách hypotetického počítača z kapitoly 1.

2 Základné pojmy v jazyku C

Hlavné podtémy kapitoly:

- Príklady iných programovacích jazykov: Pascal, Ada, Lisp...
- Vznik a vývoj C. Základné pojmy v C: premenná, dátový typ...
- Základná štruktúra programu. Spôsob spracovania programu.
- Premenná a typ premennej. Priraďovací príkaz. Aritmetické, relačné a logické operátory.
- Ladenie programov. Formátovaný výstup a vstup a funkcie pre ne: `printf` a `scanf`.
- Riadiace štruktúry: blok, podmienený a rozhodovací príkaz. Práca s logickými výrazmi.

2.1 Príklady programovacích jazykov

V kapitole 1 sme uviedli príklad strojového jazyka hypotetického počítača. Strojový kód sa používal na riešenie algoritmov pre počítače prvej generácie (50. roky 20. storočia). Každý počítač mal vlastný strojový kód, v ktorom boli číselne zakódované príkazy na vykonávanie elementárnych operácií. Programovanie v strojovom kóde bolo ťažkopádne a zdĺhavé. Ďalším vývojovým stupňom boli strojovo - orientované jazyky symbolických inštrukcií. Namiesto číselného používali symbolický zápis príkazov a adries v pamäti. Program zapísaný v tomto jazyku bolo treba preložiť a zostaviť v strojovom kóde, aby s ním počítač mohol pracovať. Takto vynikol nový druh programového vybavenia – prekladacie a zostavovacie (linkovacie) programy. Podľa anglického označenia zostavovacieho programu (assembly program) dostala neskôr táto skupina programovacích jazykov názov *assemblery.*
Assemblerový jazyk mal oproti strojovému jazyku štyri hlavné výhody:

- je mnemonický, napr. namiesto bitovej konfigurácie pre inštrukciu môžeme napísať text
- adresovanie je symbolické, nie absolútne
- čítanie je ľahšie
- zápis dát do programu je ľahší.

Vyšším vývojovým stupňom programovania sú algoritmicky orientované programovacie jazyky. Oslobodzujú programátora od sledovania hardwaru a umožňujú úplne sa sústrediť na riešený problém.

Postupná kategorizácia používateľských problémov do takých oblastí ako je vedecký výskum, ekonomika, štatistika... viedla k vytvoreniu špecializovaných jazykov (jazyky vyššej úrovne – strojovo nezávislé jazyky), ktoré umožňujú používateľovi vyjadriť problémy stručne a ľahko.

Príklady iných programovacích jazykov (okrem C-jazyka):

- **Programovací jazyk Pascal** bol navrhnutý profesorom Niklausom Wirthom ako jazyk pre vyučovanie programovania ako systematickej disciplíny.
- **Programovací jazyk ADA** vyvinulo americké ministerstvo obrany. V Ade je napríklad napísaný riadiaci software k raketoplánu.
- **Programovací jazyk Lisp** bol pôvodne určený pre výskum v oblasti umelej inteligencie, postupne sa vyvinul na pomerne všeobecný jazyk.

- **Programovací jazyk Fortran** (Formula Translation) patrí medzi skutočne prvé programovacie jazyky. Bol vyvinutý v roku 1957. Používa sa prevažne na numerické vedecké výpočty.
- **Programovací jazyk Cobol** (Common Businnes Oriented Language) bol vytvorený v roku 1960 pre aplikácie v oblasti administratívy, kde sa stále používa.
- **Programovací jazyk C++** je v súčasnosti pravdepodobne najrozšírenejší programovací jazyk zvlášť profesionálov a ďalších odborníkov. Jazyk C++ je jazyk vychádzajúci z C, ale je doplnený o mnohé vylepšenia. Predovšetkým je to plná podpora objektovo orientovaného programovania. Jedným z dôvodov veľkej obľúbenosti jazykov C/C++ je to, že sa v nich dajú používať podobné operácie a tvary ako pri programovaní v jazyku *assembler* (jazyk najnižšej úrovne), predovšetkým veľké možnosti pri práci s adresami dát a funkcií. Pri vhodnej voľbe algoritmov sa potom programy zapísané v C/C++ môžu svojou rýchlosťou takmer rovnať rýchlosti rovnakého programu zapísaného v assembleri, ale s omnoho menším úsilím. Existuje pre ne množstvo vývojových prostredí, medzi najznámejšie patria: Visual C++, C++ Builder, Borland C++.
- **Programovací jazyk JavaScript** je kompaktný objektovo orientovaný skriptový jazyk, ktorý slúži k začleneniu malých programov do siete WWW. JavaScript čiastočne odstraňuje nedostatky HTML-dokumentov tým, že im dodáva interaktívnosť. Umožňuje vytvárať takzvané dynamické HTML-dokumenty. Hlavným dôvodom jeho vzniku bola potreba testovať správnosť vstupov zadávaných vo formulároch skôr, než cieľový WWW-server obdrží od nich vyplnené dáta. V prípade chybných vstupov sa tým šetrí čas a znižuje zaťaženie siete.
- **Programovací jazyk Java** je objektovo orientovaný programovací jazyk, ktorý vyvinula firma Sun a predstavila ho 23. mája 1995. Java je jedným z najpoužívanejších programovacích jazykov na svete. Vďaka svojej prenositeľností je používaný pre programy, ktoré majú pracovať na rôznych systémoch od čipových kariet, po mobilné telefóny a rôzne zabudované zariadenia. Je to objektovo orientovaný jazyk vychádzajúci z C++, ku ktorému má aj syntakticky najbližšie. Vďaka obrovskej popularite internetovej služby WWW bola podpora Javy integrovaná do prehliadačov Netscape Navigátor (NN) 2.0+ a MS Internet Explorer 3.0+ (MSIE). Práve na WWW stránkach je možné na programy v Jave naraziť vo forme tzv. appletov. Existuje tiež veľa samostatných aplikácií.
- **Programovací jazyk C#** vyvinula firma Microsoft. Bol predstavený spolu s celým vývojovým prostredím .NET. Ako názov napovedá, vychádza z programovacího jazyka C/C++, ale v mnohom je ďaleko bližší programovaciemu jazyku Java. Bol vytváraný tak, aby bol jednoduchým, moderným, objektovo orientovaným jazykom pre všeobecné použitie. Napríklad často sa využíva pri programovaní hier.
- **Programovací jazyk Python** patrí medzi jeden momentálne z najpoužívanejších jazykov vôbec. Vznikol v roku **1991** a vytvoril ho *Guido van Rossum*. Ak sa mu budete venovať, budete môcť programovať rôzne aplikácie nie len na PC s Windows, či Linuxom, ale aj napríklad na minipočítač Raspberry Pi a vďaka nemu si budete nôcť naprogramovať rôzne veci - od semafora, až po detektor pohybu. Jazyk Python je veľmi jednoduchý na pochopenie, odpustí vám mnoho chýb a naviac je na internete mnoho tutoriálov, kde sa ho môžete zadarmo naučiť.
- **SQL,** alebo inak *Structured Query Language* je tzv. dopytovací jazyk pre manipuláciu s údajmi v databázach. Ak sa chcete venovať databázam, nie je nič jednoduchšie, naučiť sa aspoň základy tohto jazyka. SQL nie je príliš zložitý a za jeho vedomosti vás odmenia zlatom.

2.2 Jazyk C, vznik, vývoj a charakteristika

Jazyk C je univerzálny programovací jazyk, ktorý nie je špecializovaný na určitú oblasť používania. Vyvinuli ho Ken Thompson a Dennis Ritchie, pre potreby operačného systému Unix. V súčasnej dobe je prakticky na každom operačnom systéme. Je to najpopulárnejší jazyk pre písanie systémového softwaru a tiež veľmi rozšírená voľba pre písanie aplikácií. Používa sa tiež vo výuke počítačových vied. Mnoho ďalších moderných programovacích jazykov preberá spôsob zápisu (alebo syntax) z jazyka C. Patria medzi ne napríklad Java či Perl a PHP. Vývoj jazyka C začal v Bellových laboratóriách AT&T v rokoch 1969 až 1973. Ritchie tvrdí, že najprínosnejším bolo obdobie v roku 1972. Pomenovanie "C" zvolili, pretože mnoho vlastností preberali zo staršieho jazyka nazývaného "B". V roku 1973 sa stal jazyk C dostatočne stabilným. Väčšina zdrojového kódu jadra operačného systému Unixu, pôvodne napísaného v assembleri PDP-11, bola prepísaná do C. Unix teda patrí medzi prvé operačné systémy, ktoré boli napísané v inom než strojovom jazyku či assembleri. V roku 1978, Dennis Ritchie a Brian Kernighan vydali prvé vydanie knihy The C Programming Language. Táto kniha, medzi programátormi C známa ako "K&R", slúžila po mnoho rokov ako neformálna špecifikácia jazyka. Verzia C, ktorú takto popísali, býva označovaná ako "**K&R** C". Druhé vydanie knihy popisovalo novší štandard **ANSI C**, ktorý je popísaný aj v tejto učebnici. Neoceniteľnou výhodou ANSI C je, že program napísaný podľa tohto štandardu a iba s využitím štandardných knižníc je takmer 100%-ne prenositeľný na ľubovoľný počítač pod ľubovoľný operačný systém. Pokiaľ je nejaká zmena nutná, tak potom je to zmena naozaj minimálna.

2.3 Základné pojmy v C

Programy v jazyku C obsahujú niekoľko druhov symbolov, medzi ktoré patria predovšetkým

- **špeciálne symboly**, napr. symboly pre aritmetické a relačné operátory (+,-,<,>...), zátvorky a rezervované (kľúčové) slová jazyka, napr. if, else, for, while.....
- **identifikátory,** napr. mená premenných, konštánt, procedúr....

Programovací jazyk C je „case-sensitive“, t.j. záleží v ňom na veľkosti písmen!!!

Príklad 1
V nasledujúcom príklade programu sú hrubým písmom vyznačené príklady špeciálnych symbolov a šikmým písmom príklady identifikátorov.

```
#include <stdio.h>
main(){
int i,j,n,fakt;
for (i=0;i<10;i++)
 { scanf("%d",&n);
  if (n==0) printf("faktorial z 0 = 1\n");
  else
   {fakt=1;
    for(j=2;j<=n;j++)
      fakt= fakt*j;
    printf("faktorial z %d = %d\n",n,fakt);
   }}}
```

V moderných editoroch sa rezervované slová zvýrazňujú farebne.

Pri voľbe mien funkcií máme úplnú voľnosť (musia však byť rôzne od rezervovaného slova). Identifikátor **`main`** je však špeciálne meno, program sa vždy začne vykonávať na začiatku tejto funkcie. To znamená, že v každom C-programe musí byť niekde funkcia `main`, ktorá zvyčajne na vykonanie určitej činnosti volá ďalšie funkcie. Niektoré z týchto funkcií sa nachádzajú v samotnom programe, iné zasa v knižniciach: ak máme súbor podprogramov, ktoré často používame, môžeme ich zoskupiť dohromady do niekoľkých *zdrojových súborov*, každý zdrojový súbor preložiť do relatívneho kódu a vytvoriť z nich knižnicu. Knižnice sú teda súbory dávnejšie vytvorených funkcií, či už nami, alebo sú súčasťou samotného jazyka C.
Každý program musí obsahovať

```
main()
{
  ...
}
```

Ako už bolo povedané, súbor, ktorý obsahuje text programu, sa nazýva *zdrojový súbor*. Má príponu `.c`. Zdrojový súbor, v ktorom je náš program, je pre jeho správne fungovanie väčšinou nutné doplniť vložením jedného alebo viac tzv. hlavičkových súborov, ktoré majú príponu `.h`. Používajú sa preto, lebo program väčšinou volá (zabudované) *knižničné* programy (napr. pre tlač, matematické funkcie, prácu s časom...), ktorých správne využitie umožní práve vloženie príslušného h-súboru pomocou príkazu `#include`. Napr. v príklade 1 to bolo

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

Názov `stdio` je skratka pre standard input/output (vstup/výstup). Po vložení hlavičkového súboru môžeme teda použiť funkcie `scanf` a `printf`, pomocou ktorých sa dá načítať číslo z klávesnice `(scanf)` a vypísať výsledok na obrazovku `(printf)`. (V mnohých implementáciách kompilátora C pre `printf` a `scanf` tento súbor vkladať netreba.)

V príklade 1 boli použité štyri premenné typu `int`. Išlo o premenné `i,j,n` a `fakt`, ktoré boli celočíselného typu `(int` je skratka od `integer)`. Jednotlivé príkazy sú od seba oddelené bodkočiarkou. Býva zvykom písať jeden príkaz do jedného riadku, aj keď to nie je podmienka

2.3.1 Jednoduché číselné dátové typy

V predchádzajúcom texte sme sa stretli s pojmom premenná typu `int`. Zopakujme si, že premenná je pamäťové miesto, prístupné pomocou identifikátora, t.j. mena. V predchádzajúcej kapitole sme si ukázali, že do jedného bytu sa zmestí, v prípade, že nepoužívame znamienko, maximálne číslo 255. Je nám jasné, že aj pre úplne bežné výpočty, je takáto maximálna hodnota čísla nepostačujúca a že od pozície, na ktorú identifikátor premennej ukazuje, musíme brať viacero bytov, ak chceme dostať vyššiu hodnotu, prípadne hodnotu reálneho čísla. To, koľko bytov máme zobrať pre premennú a akým spôsobom sa číselná hodnota ukladá, určuje typ premennej.

Premenná je vždy združená s nejakým typom!!!

Hlavným celočíselným typom v C je typ **`int`**. Je dobré preferovať práve tento dátový typ, pretože je prispôsobený k tomu, aby operácie s ním prebiehali čo najrýchlejšie. V rôznych implementáciách prekladačov C môže premenná typu `int` obsadiť 2 alebo 4 byty. Rozsah dátového typu v bytoch môžeme zistiť nasledovným krátkym programom pomocou funkcie `sizeof`:

Príklad 2

```
main()
{
 printf("typ int ma rozsah %d\n", sizeof(int));
 printf("typ double ma rozsah %d\n", sizeof(double));
}
```

O tom, ako môžeme tento program spustiť, si viac povieme v sekcii 2.4.

V C existujú 3 typy reprezentujúce reálne čísla. Najúspornejší z nich, čo sa týka pamäte, je typ `float`, ktorý ale väčšinou neposkytuje dostatočnú presnosť. Tú by mal vo väčšine prípadov zaistiť typ `double`. Rovnako, ako by sa mal medzi celými číslami preferovať typ `int`, medzi reálnymi by to mal byť typ `double`.

Dátový typ	**Rozsah**		
Celé čísla:			
char	1 byte		
int	závisí od implementácie (2 alebo 4 byty)		
long int(long)	4 byty		
short int(short)	2 byty		
Reálne čísla:		Maximum:	Presnosť:
float	4 byty	$3{,}40282*10^{38}$	6 des. miest
double	8 bytov	$1{,}79769...*10^{308}$	15 des.miest

V príkladoch je „*“ operátor pre násobenie. (Tento operátor sa pre násobenie používa aj v C jazyku). Použitím slov `signed` a `unsigned` v špecifikácii typu premennej môžeme rozhodnúť o znamienkovosti typu. Asi tušíte, že použitím `signed` si vynútime, aby bol typ znamienkový a `unsigned` neznamienkový. Pokiaľ tieto slová nepoužijeme, môžeme sa spoľahnúť, že typy `int` `short` a `long` budú považované za `signed`, u `char` to však záleží na konkrétnej implementácii prekladača. Premenné typu `unsigned` majú rozsah od 0 do 2^n-1, kde *n* je počet bitov premennej. Rozsah `signed` premenných je od -2^{n-1} do $+2^{n-1}$-1.

2.3.2 Definícia a deklarácia premenných

Po tom, ako sme sa oboznámili s typom premenných, môžeme si povedať niečo o ich definícii.

Pod **definíciou** sa myslí príkaz, ktorý zavedie premennú daného typu: priradí jej dané meno a pamäť podľa typu. Pri definícii najprv píšeme typ premennej, potom identifikátor. Definícia je ukončená bodkočiarkou.

```
int i;
```

Súčasne môžeme definovať aj viac premenných toho istého typu.

```
int i,j,n,fakt;
```

Definíciu môžeme spojiť *s inicializáciou*, t.j. premennej hneď pri definícii priradíme nejakú hodnotu.

```
int i = 5;
int j = 2*10 + 4;
```

Deklarácia je príkaz, ktorý iba udáva typ premennej a jej meno. Deklarácia nepridelíuje žiadnu pamäť. Vlastne iba oznamuje, že daná premenná existuje. Deklarácia sa využíva pri dlhých programoch, napr. keď je program rozdelený do viacerých súborov.

Upozornenie: v niektorej literatúre sú významy slov definícia a deklarácia práve opačné. Pri nejasnostiach je treba zistiť na príkladoch, čo tým pojmom autor označuje.

2.3.3 Aritmetické a logické operátory

Operátory "+", "-" , "*" netreba komentovať, treba dávať pozor na prioritu (t.j. poradie vykonávania operácií) a v prípade potreby použiť zátvorky. Priorita operátorov je uvedená v prílohe v tabuľke Priorita operátorov.

Operátor "/" predstavuje celočíselné alebo reálne delenie. To, či sa vykoná celočíselné alebo reálne delenie, závisí na type operandov, ako ukazuje jeden z príkladov v nasledujúcom odseku. Ak je aspoň jeden z operandov reálneho typu, vykoná sa reálne delenie. V prípade, že chceme vykonať reálne delenie na operandoch celočíselného typu, musíme aspoň jeden z operandov pretypovať pomocou operátora pretypovania (pozri 2.3.5).

Operátor "%" nazývaný „modulo" predstavuje zvyšok po celočíselnom delení, napr. 7%5=2, 6%3=0. Pomocou neho môžeme napr. otestovať deliteľnosť čísla iným číslom.

Operátor "++" je operátor inkrementácie, t.j. hodnota operandu zvýši o 1.

Operátor "--" je operátor dekrementácie, t.j. hodnota operandu zníži o 1.

Napr. príkaz `i++` zvýši hodnotu premennej `i` o 1.

Obidva posledné operátory sa dajú použiť aj pred operandom, aj za ním. Rozdiel je v tom, či sa hodnota premennej najprv použije a potom zvýši o 1, alebo najprv zvýši a potom použije. My však tieto príkazy najčastejšie budeme používať spôsobom, ako je uvedené v príklade, t. j. za operandom.

2.3.4 Priraďovanie premenným

Premenným priraďujeme pomocou znaku „=".

Napríklad:

```
X =  100;
Z =  (100*5)-10;
Y =  5/2;
Y =  5/2.0;
```

Pozor, v prípade predposledného príkazu, sa do premennej Y uloží výsledok je `2`, aj v prípade, ak bola definovaná ako reálna. V prípade posledného príkazu, sa príkaz na pravej strane vyhodnotí ako `2.5`.

V programovaní sa často vyskytuje nasledujúci zápis: `J = J + 1.`
Z matematického hľadiska sa nám javí ako veľmi pochybný, ale tu znak "=" má funkciu priradenia a nie rovnosti. Príkaz pracuje tak, že sa najprv vypočíta hodnota výrazu `J + 1` a potom sa hodnota tohto výrazu sa uloží do `J`. Takýto zápis sa dá nahradiť zápisom `J += 1`. To isté platí aj pre ďalšie matematické operátory `-=, *=, /=, %=.`

Príklady:

`J += 2`	má význam	`J = J + 2`
`J -= 10`		`J = J - 10`
`J /= 5`		`J = J / 5`
`J *= 4`		`J = J * 4`

2.3.5 Operátor pretypovania

Aj keď sa pri nezhodách typu pri priraďovaní vykonáva automatické pretypovanie (t.j. zmena typu premennej), sú situácie, keď potrebujeme explicitne (t.j. my dávame podnet, nie je to vnútorná záležitosť kompilátora) pretypovať jeden typ na iný. To vykonáme pomocou operátora pretypovania. Zápis potom vyzerá tak, že pred samotný výraz, ktorý chceme pretypovať, uvedieme v okrúhlych zátvorkách meno nového typu.

```
int  a=10, b=3;
double c;
c= a /(double)b;
```

Výsledok je 3,3333..., bez pretypovania by výsledkom bolo číslo 3.

2.4 Spôsob spracovania programu

V tomto momente by sme vedeli napísať krátky program a zaujíma nás, ako ho vykonáme v počítači. Ukážme si to na príklade programu, ktorý sa píše ako prvý vo všetkých jazykoch, ktorého úlohou je vytlačiť slová

```
Hello, world!
```

V prvom rade musíme vedieť napísať text programu a napísať ho do počítača. Ďalej ho musíme preložiť, zaviesť, spustiť a zistiť, kde sú jeho výsledky (v prípade, že sa nevypisujú na obrazovku).

Zodpovedajúci program v C-jazyku je takýto:

```
#include <stdio.h>
main()
{
printf("Hello, world!\n");
}
```

Pokiaľ pracujeme vo vývojovom prostredí, prekladanie zvyčajne vykonáme klikaním na zodpovedajúce ikony. IDE je skratka pre interaktívné vývojové prostredie. Medzi najjednoduchšie vývojové prostredie v OS Windows patria voľne dostupné DevCpp a náročnejšie Code:Blocks, najčastejšie používané profesionálne prostredie je Visual Studio.

2.4.1 Spôsob preloženia z príkazového riadku

Spôsob spustenia tohto programu závisí od systému a prostredia, ktoré používame. Napr. pod operačným systémom Unix alebo Linux musíme vytvoriť zdrojový program v súbore, ktorého meno končí príponou .c, napr. *hello.c* a potom program preložiť povelom

```
gcc -o hello hello.c.
```

V tomto príkaze `gcc` je názov kompilátora `-o` je prepínač udávajúci ako sa bude volať výstupný (output) spustiteľný súbor, teda namiesto `hello` by sme mohli napísať ľubovoľné meno. Ďalej nasleduje meno zdrojového (existujúceho) súboru.

Prekladač `gcc` (GNU compiler collection) je súčasťou väčšiny unixovských a linuxovských distribúcií. Je to typický zástupca „osamoteného“ prekladača, ktorý nemá žiadny editor ani integrované prostredie. Tento prekladač sa ovláda z príkazového riadku.
Program spustíme pomocou povelu

```
hello  (prípadne ./hello).
```

Výsledok sa objaví na obrazovke, vypíše sa text `hello, world.`

Môže byť užitočné vedieť, že pokiaľ neuvediem pomocou prepínača `-o` meno výstupného súboru, súboru bude priradené automatické meno *a.out* Program potom spustíme napísaním tohto mena.

Uveďme niektoré ďalšie možnosti pri preklade a linkovaní súboru a zhrňme ich do tabuľky.

Tabuľka 2.1

1.	`gcc -o hello hello.c`	vytvorenie spustiteľného programu s menom programu `hello`
2.	`gcc hello.c`	vytvorenie spustiteľného programu s menom programu `a.out`
3.	`gcc -c hello.c`	iba kompilácia, výstup je objektový súbor `hello.o`
4.	`gcc hello.o`	linkovanie, výstupom je program `a.out`
5.	`gcc -o hello hello.o`	linkovanie, výstupom je program `hello`
6.	`gcc - g -o hello hello.c`	umožnenie ladenia spustiteľného programu `hello` (pozri 2.6)

V operačnom systéme Windows môžeme použiť napríklad vývojové prostredie DevCpp, ktoré je pre svoju jednoduchosť vhodné pre začiatočníkov. Informácie a inštalačný balík sú dostupné na internetovej adrese http://www.bloodshed.net/devcpp.html.

Cvičenie 1
Napíšte a spustite príklad 2.

2.5 Matematické funkcie

V jazyku C je už vytvorených veľa prostriedkov potrebných pri matematických výpočtoch. Sú to predovšetkým funkcie uložené v knižnici matematických funkcií. Ak chceme použiť matematickú funkciu, musíme na začiatok programu vložiť hlavičkový súbor pre matematické funkcie príkazom `#include <math.h>` a pri preklade musíme ešte pripojiť samotnú knižnicu. Príkaz prekladu bude

```
gcc -o hello hello.c -lm
```

Funkcie, ktoré často využívame, sú:

Tabuľka 2.2

`sqrt`	druhá odmocnina
`sin`	sínus
`cos`	kosínus
`fabs`	absolútna hodnota
`pow(x,y)`	x^y
`tan`	tangens
`asin`	arkussínus
`acos`	arkuskosínus
`atan`	arkustangens
`exp(x)`	e^x
`log(x)`	prirodzený logaritmus
`log10(x)`	dekadický logaritmus
`fmod(x,y)`	reál. zvyšok po del. x a y

Všetky tieto funkcie majú jeden alebo dva vstupy (argumenty), ktorý píšeme do zátvorky za meno funkcie a výsledky týchto funkcií priraďujeme do premennej typu `double`.

Príklad 3
V premenných `x` a `y` majme uložené dve čísla, ktoré predstavujú odvesny pravouhlého trojuholníka. Vypočítajme dĺžku prepony z.

```
#include <math.h>
main()
{
double x=4,y=3,z,pomoc;
pomoc=x*x+y*y;
z= sqrt(pomoc);
}
```

Ak všetko prebehlo v poriadku, program sa vykonal, my však nič nevidíme. Výsledok je uložený v premennej `z` niekde v operačnej pamäti. Ak sa tam chceme pozrieť na výsledok, môžeme použiť program nazývaný *debugger*, ktorý sa používa pre hľadanie logických chýb v programe a obsahuje prostriedky pre zastavenie programu a prezeranie obsahu premenných. Okrem týchto základných funkcií obsahuje mnohé ďalšie užitočné pri hľadaní chýb v programoch a určite sa ho oplatí naučiť. Druhý, oveľa bežnejší postup, je dať vypísať obsah premennej `z` na obrazovku. Tejto možnosti sa budeme venovať v nasledujúcom odstavci. Zatiaľ sa chvíľu zastavme pri chybách a ladiacom programe - debuggeri.

2.6 Ladenie programov (Debugging)

Ladenie je proces, pomocou ktorého overujeme správnosť programu. Spočiatku sa stretávame s tzv. *syntaktickými* chybami: sú to chyby, ktoré nám obyčajne hlási kompilátor výpisom na obrazovku (závisí to však od prostredia, v ktorom pracujeme), ide o nesprávne zadaný príkaz (preklep, nesprávny počet zátvoriek, neexistujúci objekt, nedodržané pravidlá pre písanie príkazu a pod.). V C - jazyku sa program môže vykonať, až keď sú chyby tohto druhu odstránené. (Ešte existujú tzv. interpretre, kde sa po preložení každého príkazu tento aj vykoná).
Logické chyby programu. Po odstránení syntaktických chýb môžeme program spustiť, ale pred jeho používaním v praxi by sme mali overiť, či funguje správne, t.j. či robí to, na čo bol určený. Niekedy je správnosť/nesprávnosť programu očividná, avšak korektným overením programu je až získanie formálnych dôkazov o jeho správnosti. Pre zložité programy je však takýto spôsob overenia ich správnosti obťažný a niekedy aj nedosiahnuteľný. V praxi sa preto program overuje na vhodne zvolenej množine skúšobných vstupných údajov, pre ktoré sú k dispozícii skutočné správne výsledky. Na základe analýzy programu a vypočítaných výsledkov sa táto čiastková správnosť zovšeobecní.

Ak program nepracuje správne, musíme hľadať chybu, t.j. program musíme ladiť. Robíme to väčšinou **zastavovaním programu** na "podozrivých miestach" a **výpismi obsahu sledovaných premenných**. K tomu často používame ladiaci program (debugger), ktorý môže byť alebo samostatný, alebo môže byť súčasťou vývojového prostredia. Napríklad v linuxe môžeme použiť program *gdb,* v OS Xwindows jeho grafickú nadstavbu *ddd*. V Dev-C++ je debugger zabudovaný. Ak používame ladiaci program samostatne, musíme ho preložiť s prepínačom *–g*. (Pozri tabuľku v odseku 2.4.1.)

Program určený na ladenie obsahuje nasledujúce funkcie:

Zastavovanie programu:

- Pred spustením programu si nastavíme tzv. *breakpoint – bod prerušenia.* Program sa vykoná iba po *breakpoint.*
- V prípade, že to ladiaci program umožňuje, vykonávame programu sa zastaví pred kurzorom, ktorý nastavíme na podozrivý príkaz.

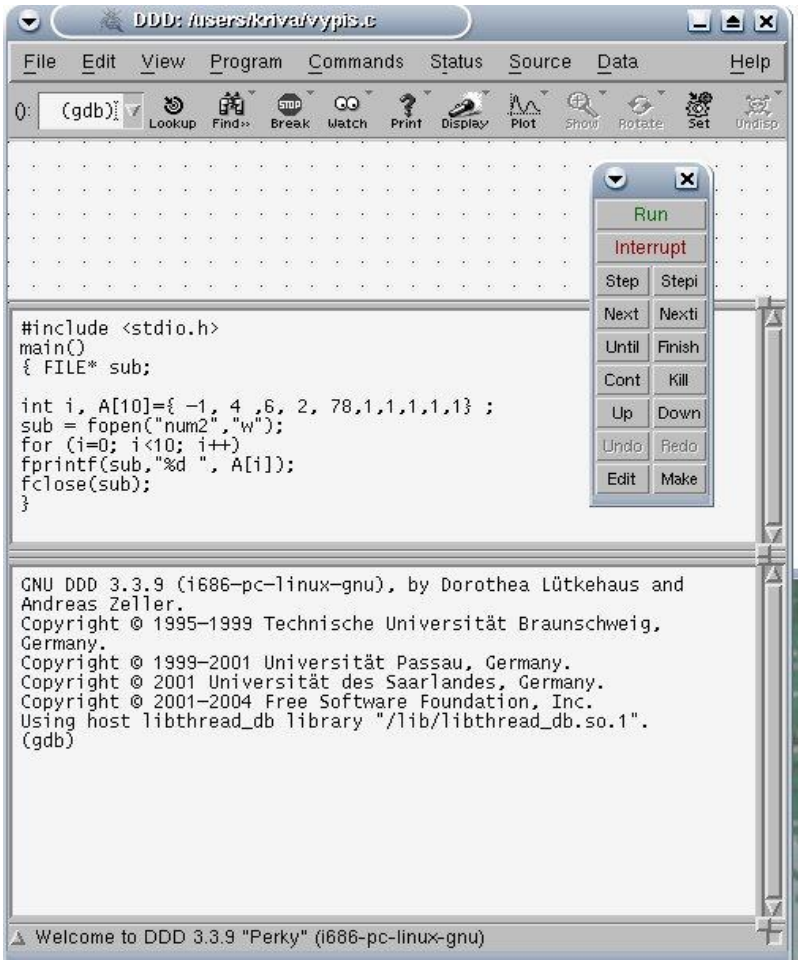
Pre pokračovanie programu mávame k dispozícii:

- príkaz **Continue** - program sa vykoná do konca, alebo po ďalší bod prerušenia (*continue* – pokračovať).
- príkaz na vykonanie jednej ďalšej inštrukcie programu (*Step*).
- príkaz na vykonanie jedného príkazu programu, pričom za jeden príkaz sa môže považovať celá procedúra, (t.j. pomenovaný blok kódu) (*Next*).

Po zastavení programu si môžeme pozrieť obsah premennej .

Sledovanie obsahu premennej:

- obyčajne sa po nastavení kurzora myši na názov premennej po chvíľke objaví jej hodnota
- príkazom pre výpis hodnoty premennej alebo výrazu jednorazovo
- príkazom pre výpis hodnoty premennej alebo výrazu do sledovacieho okienka. V špeciálnom okienku sa zobrazuje obsah premennej v každom kroku programu.



Obrázok vľavo znázorňuje vývojové prostredie programu *ddd* v OS Linux.. Tento program sa dá pomerne ľahko používať intuitívne. Pracovná plocha je rozdelená na tri časti. Do hornej časti sa vypisujú okienka so sledovanými premennými, v strednej časti je zdrojový kód a do spodnej môžeme zadávať príkazy a objavujú sa v ňom niektoré výpisy, vyvolané aj z ponuky, v nej objavujú. Vypisujú sa v nej aj hlásenia, ktoré vzniknú v dôsledku ladenia, napr. že program bol zastavený, zhavaroval apod. Príkazy sa vyvolávajú z hlavnej ponuky alebo z horného alebo plávajúceho panelu s nástrojmi. Bod zastavenia nastavíme tak, že sa s kurzorom nastavíme na riadok, kde chceme program nastaviť a klikneme na tlačidlo s názvom *Break.* Po zastavení obsah premennej vypíšeme tak, že ju vyberieme pomocou myši a klikneme na tlačidlo *Print* pre jednorazový výpis a na tlačidlo

Display v prípade, že chceme obsah premennej sledovať v sledovacom okienku. Ako napovedá záhlavie, program, ktorý ladíme sa nazýva *vypis.c*. Museli sme ho najprv preložiť napr. príkazom `gcc -g –o vypis vypis.c`. Potom do príkazového riadku napíšeme `ddd vypis` a objaví sa okno prgramu spolu so zdrojovým textom. V programe Dev-C++ je debugger zabudovaný priamo. Jeho príkazy sú v položke Debug hlavného menu a niektorým z nich sú priradené tlačidlá. Jeho použitie je principiálne rovnaké ako v *ddd*, intuitívne jednoduchšie. V mnohých debuggeroch sa obsah premennej dá pozrieť aj tak, že nastavíme kurzor myši na danú premennú a nehýbeme s ním. Obsah premennej sa vypíše v malom okienku pri myši.

Cvičenie 2

Napíšte a vykonajte príklad 3. V prostredí, ktoré máte k dispozícii ho „laďte“ tak, že nastavte bod zastavenia na prvý príkaz a program vykonávajte po jednej inštrukcii. V každom kroku si pozrite obsah použitých premenných.

2.7 Základné použitie formátovaného výstupu a vstupu

Súčasťou každého programu býva výpis výsledkov. Spočiatku nám úplne postačí výpis výsledkov na obrazovku, neskôr sa naučíme zapísať výsledky do súboru. Podobne nám spočiatku pre vstup údajov do programu postačí klávesnica, neskôr sa naučíme načítať dáta zo súboru.

2.7.1 Formátovaný výstup – funkcia printf

Formátovaný výstup má svoju štandardnú funkciu `printf`. Táto funkcia umožňuje výstup údajov na obrazovku v predpísanom tvare. Počet výstupných údajov (t.j. argumentov funkcie `printf`) je voliteľný, je možný aj počet `n = 0`. Základné použitie tejto funkcie je napr. takéto:

```
printf (“%d“,i);
```

- “`%d`“ určuje formát výpisu, tu dekadický celočíselný
- `i` je vypisovaná premenná.

Keby sme napísali:
`printf (“%d`**`\n`**`“,i);`
tak ďalší výpis sa vypíše na nový riadok, t.j. tento výpis sa „odriadkuje.“

Viacero premenných vypíšeme jedným príkazom nasledovným spôsobom:
`printf (“%d %d %d“,i,j,n);`

Keď máme v programe viac výpisov, je rozumné vypísať k premennej aj textový reťazec oznamujúci, akú premennú vypisujeme.
`printf (“ Obsah premennej i je:%d`**`\n`**`“,i);`

Pri výpise často potrebujeme skombinovať text a číslo, napríklad:
`printf (“Dĺžka prepony pre odvesny %lf a %lf je %lf`**`\n`**`“,x,y,z);`
`%lf` je konverzný reťazec pre typ `double` a napíšeme ho na to miesto do textu, kde chceme zaradiť výpis premennej. Za čiarku napíšeme mená vypisovaných premenných v príslušnom poradí.

Zložky popisu formátu pre ďalšie typy sú takéto:

celé čísla:

%d celé dekadické číslo typu `signed int`

%i celé dekadické číslo typu `signed int`

%u celé dekadické číslo typu `unsigned int`

čísla v pohyblivej rádovej čiarke:

%f číslo v desatinnom tvare typu `float`

%lf číslo v desatinnom tvare typu `double` (niekedy sa nedá použiť pre `printf`)

%e číslo v semilogaritmickom tvare, t.j. základ, mantisa

%g použije sa jedna z konverzií `f`, `e`. Výstupný tvar sa volí podľa hodnoty exponentu (ak je menší ako -4 alebo väčší ako presnosť, použije sa `e`, inak `f`)

znaky a reťazce:

%c jeden znak typu `char`

%s reťazec ukončený znakom '\0' (netlačí sa)

%% tlač znaku %

Všeobecnejší popis formátu je:

```
printf ("%10d %5d %3d",i,j,n);
```

V tomto prípade udávame, na koľko pozícii sa má číslo napísať.

```
printf ("%6.2f",x);
```

V tomto prípade udávame, že reálne číslo bude vytlačené na 6 znakov, z nich budú dva za desatinnou bodkou a jeden bude desatinná bodka.

Skúšobný program z príkladu 3 si môžeme doplniť takto:

```
main()
{
int i=5;
    j=-1;
    j=j+2*3;
prinft("Obsah premennej j: %d",j) ;
}
```

2.7.2 Formátovaný vstup – funkcia scanf

Keby sme si chceli krátke programy vytvorené v predchádzajúcich príkladoch odskúšavať, t.j. spúšťať ich s rôznymi vstupnými hodnotami, mohli by sme to robiť v programe tak, že by sme príslušné premenné inicializovali rôznymi hodnotami a zakaždým preložili a zlinkovali. Je to však nešikovné, a preto si ukážeme, zatiaľ len veľmi jednoducho, ako môžeme zadať vstup z klávesnice.

Napríklad načítanie dvoch celých čísel môžeme urobiť takto:

```
scanf("%d%d", &i,&j)
```

Znak & pred názvom premennej je veľmi dôležitý. Jeho presný význam bude vysvetlený neskôr.

2.8 Riadiace štruktúry

S doteraz získanými vedomosťami by sme vedeli napísať program, v ktorom zadefinujeme premenné, priradíme im hodnoty, vykonáme s nimi jednoduché aritmetické operácie, prípadne použijeme zabudované matematické funkcie. Nakoniec výsledok vypíšeme na obrazovku. Počas výpočtu sa však často stretneme s tým, že niektoré operácie budeme chcieť vykonať len za určitých podmienok, napríklad deliť číslom budeme len vtedy, keď je rôzne od nuly, alebo sčítame len tie čísla, ktoré sú väčšie ako nejaká hodnota. V príklade 2 v kapitole 1 sme vypísali reťazec „Blahoželáme, postupujete ďalej“, ak výsledná známka bola väčšia ako 50, a reťazec „Je nám ľúto, ale ste prepadli“ inak. Inokedy budeme opakovane vykonávať nejakú činnosť, pokiaľ bude splnená určitá podmienka.

V tejto časti opíšeme podstatnú časť jazyka C – riadenie behu programu. V prvej kapitole sme spomínali, že hardwarové obvody riadiacej jednotky zabezpečujú čítanie inštrukcií jednu po druhej z operačnej pamäte a indikuje operácie, ktoré sa majú vykonať na špecifikovaných operandoch. Niektoré inštrukcie sú jednoduché, napr. sčítanie, priradenie, niektoré však majú v sebe zabudovaný odskok do inej časti programu. **Riadiacimi štruktúrami,** ktorými sa vyjadruje postupné, podmienené, či opakované vykonávanie príkazov, budeme rozumieť :

- **zložený príkaz - blok,**
- **podmienený príkaz,**
- **príkazy cyklu.**

Je pre ne charakteristické, že obsahujú výrazy, udávajúce podmienky pre vetvenie a opakovanie.

2.8.1 Blok

Syntax bloku je veľmi jednoduchá. Začína znakom **{** a končí znakom **}**. Medzi týmito zátvorkami môžu byť ľubovoľné iné príkazy včítane ďalších blokov. Okrem toho môžu byť ešte na začiatku bloku, teda pred prvým príkazom, lokálne **deklarácie** a definície. Premenné takto definované sú potom viditeľné iba vnútri bloku a v ďalších vnorených blokoch. Blok je v C chápaný ako jediný príkaz, a preto sa tiež dá použiť všade tam, kde je možné použiť príkaz.

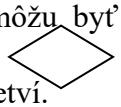
2.8.2 Podmienený príkaz

Tento príkaz má v sebe zabudovanú podmienku, od ktorej splnenia závisí, či sa nejaký príkaz vykoná alebo nie, prípadne, ktorý z dvoch príkazov sa vykoná.

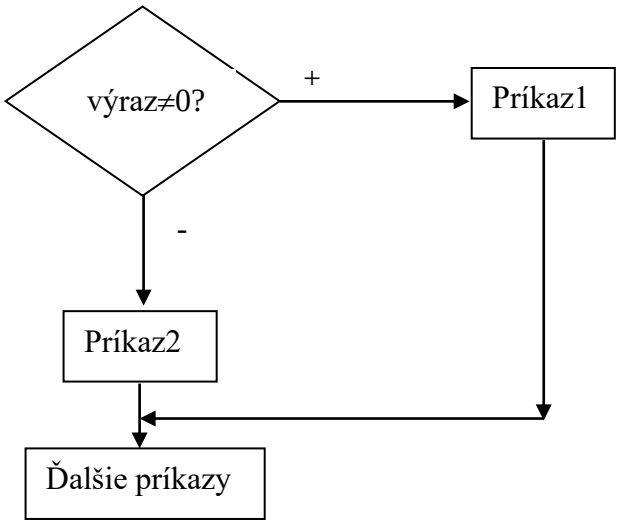
```
if (výraz) príkaz;  /* skrátený */
if (výraz) príkaz1;
else príkaz2;       /* úplný */
```

`príkaz`, `príkaz1` a `príkaz2` môžu byť aj bloky.

Nezabúdajme za príkaz časti `if` (pred `else`) písať bodkočiarku a testovaný výraz dať do zátvorky, aj keď pozostáva len z jednej premennej, napr. `if(z)...` !!!

Ak je v príklade hodnota výrazu `výraz` nenulová, vykoná sa `príkaz`, resp. v prípade úplného tvaru `príkaz1`. Ak je hodnota výrazu nulová, vykoná sa iba v druhom prípade `príkaz2`. Po vykonaní `príkaz1` alebo `príkaz2` riadenie pokračuje na nasledujúcom príkaze. `príkaz1` a `príkaz2` môžu byť aj bloky. Vo vývojových diagramoch zodpovedalo tomuto príkazu rozhodnutie  . Vychádzajú z neho vždy dve vetvy + a -, a preto sa často hovorí, že program sa vetví.

Vývojovým diagramom by sme príkaz `if (výraz) príkaz1; else príkaz2;` mohli vyjadriť takto:

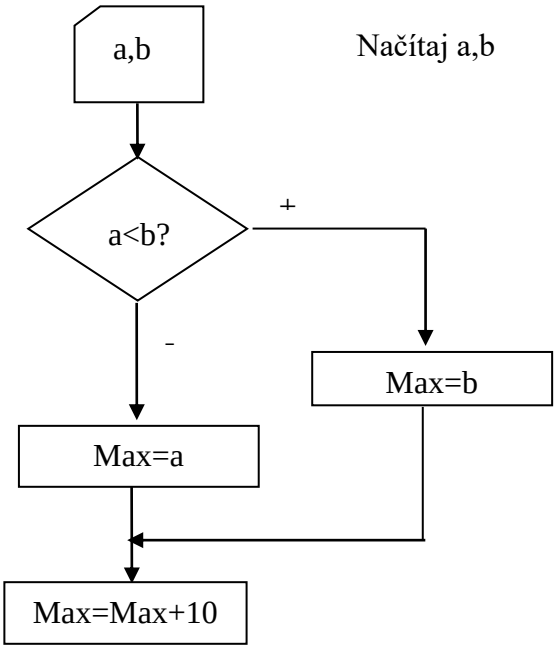


`Príkaz`, `Príkaz1` a `Príkaz2` môžu byť aj bloky príkazov. V tom prípade musia byť ohraničené zátvorkami {}. Podmienené príkazy možno do seba vnárať.

Vetvenie sa vykoná na základe vyhodnotenia podmienky, ktorou je *výraz* **v zátvorkách**. Pod výrazom si môžeme predstaviť aj výraz logický. Logický výraz je taký výraz, ktorého pravdivosť sa dá jednoznačne vyhodnotiť jedným z dvoch možných výsledkov: **pravda (true)** alebo **nepravda (false)**. V ANSI C neexistuje nejaký špeciálny logický typ, ako napr. `Boolean` v jazyku Pascal. "Pomáhame si" číselným typom, pričom platí, že nula znamená nepravdu a nenulová hodnota (obyčajne sa používa 1) znamená pravdu. Operácia negácie ! pracuje tak, že ak `D` bola pravda, `!D` bude nepravda a naopak. Napríklad testujeme, či sa rovnajú dva číselné výrazy. Ak je logický výraz splnený (pravdivý), zodpovedá mu nenulová číselná hodnota, ak nie je splnený, zodpovedá mu nulová číselná hodnota.

Príklad 4

Načítajme zo vstupu dve čísla. Zistime, ktoré z nich je väčšie, pripočítajme k nemu 10 a dajme ho vypísať na obrazovku.



Vývojovému diagramu zodpovedá nasledujúci kód:

```
if (a<b)
   max=b;
else max=a;
   max+=10;
printf(„max=  %d",max) ;
...
```

Aby sme mohli pohodlne používať podmienený príkaz, je potrebné si doplniť znalosti o logické a relačné operátory.

Relačné operátory

<,<=,>,>=	menší, menší alebo rovný, väčší, väčší alebo rovný
!=	nerovnosť
==	rovnosť

Častou chybou býva použitie `if(a=b)` namiesto `if(a==b)`. Použitie `a=b` je legálne, a preto kompilátor nevyhlási chybu a program zdanlivo funguje! ! !

Uvedomme si rozdiel medzi `if (a=b) .... a if (a==b) ....`
V C platí, že priradenie, ktoré nie je ukončené bodkočiarkou, napr. `j=5,` nie je priraďovacím príkazom, ale priraďovacím výrazom, ktoré má svoju hodnotu, v tomto prípade hodnotu 5, avšak v jeho rámci dôjde aj k priradeniu čísla 5 do premennej `j`. Čo sa teda stane, ak namiesto `if (a==b)` použijeme `if (a=b)`? Ak `b` je rôzne od nuly, výraz bude vyhodnotený ako pravdivý, ak sa `b` rovná nule, vyraz bude vyhodnotený ako nepravdivý.

Príklad 5

Vyskúšajte si tento príklad:

```
main(){
double i=1,j=0.3;
if (i=j)
      printf("pravda,%lf\n",i);
else
      printf("nepravda,%lf\n",i);
 }
```

Logické operátory.

&&	logický súčin (AND)
||	logický súčet (OR)
!	negácia

Takisto nezabúdajme, že logické operátory treba zdvojovať!!!

Pre úplnosť uvádzame tabuľky vyhodnotenia logických operátorov. && a || sú binárne operátory a negácia ! je unárny operátor.

		&&	||		!
PRAVDA	PRAVDA	PRAVDA	PRAVDA	PRAVDA	NEPRAVDA
PRAVDA	NEPRAVDA	NEPRAVDA	PRAVDA	NEPRAVDA	PRAVDA
NEPRAVDA	PRAVDA	NEPRAVDA	PRAVDA		
NEPRAVDA	NEPRAVDA	NEPRAVDA	NEPRAVDA		

2.8.3 Rozhodovací príkaz switch

Príkaz **`switch`** je podmieňovací príkaz a je to príkaz pre mnohonásobné vetvenie. Pokiaľ potrebujeme tok programu vetviť do viac ako dvoch smerov, môžeme namiesto niekoľko príkazov `if-else` využiť možnosti, ktoré nám v C poskytuje príkaz `switch`.

```
switch (výraz1) {
                case konštantný_výraz1:
                                        príkaz1;
                case konštantný_výraz2:
                                        príkaz2;
                    ⋮                       ⋮
                case konštantný_výrazn:
                                        príkazn;
                              default:
                                        príkazm;
                             }
```

Pri vykonávaní príkazu `switch` je najskôr vyhodnotený *konštantný_výraz1* a potom sa postupne vyhodnocujú konštantné výrazy v návestiach `case`. V prípade, že je nájdené návestie, kde je *konštantný_výrazi* rovný hodnote výrazu *výraz1,* začnú sa vykonávať všetky príkazy až do konca príkazu `switch`. To ale znamená, že sa vykonajú aj všetky príkazy v nasledujúcich návestiach. Možnosť `default` znamená niečo ako "ostatné prípady". Ak chceme, aby sa vykonala vždy len jedna vetva, dá sa vykonávanie príkazu `switch` okamžite zastaviť použitím príkazu `break`.

```
switch (výraz1) {
                case konštantný_výraz1:
                                        príkaz1;
                                        break;
                case konštantný_výraz2:
                                        príkaz2;
                                        break;
                                        ⋮
⋮               case konštantný_výrazn:
                                        príkazn;
                                        break;
                              default:
                                        príkazm;
                             }
```

Ako napovedá samotný názov, každý *konštantný_výrazn* musí byť konštanta, premenné nie sú povolené.

Príklad 6
Načítajte zo vstupu číslo medzi 1 a 7. Vypíšte príslušný deň v týždni.

```
int c;

...
scanf("%d", &c);

  switch (c)
```

```
{case 1: printf("Pondelok\n"); break;
 case 2: printf("Utorok\n");   break;
 case 3: printf("Streda\n");   break;
 case 4: printf("Štvrtok\n");  break;
 case 5: printf("Piatok\n"); break;
 case 6: printf("Sobota\n"); break;
 case 7: printf("Nedeľa\n"); break;
 default: printf("Nesprávne číslo\n");
}
```

Príklad 7

```
int c;
...
switch (c)
  {
    case 1:
    case 2:
    case 3: printf("Číslo 1,2 alebo 3") ; break;
    case 4: printf("Číslo 1,2 alebo 4") ; break;
    default: printf("Iné číslo  ako 1,2 alebo 3") ;
}
```

V tomto príklade vezmeme celé číslo `c`, a podľa toho, akú má hodnotu, vypíšeme výpis na obrazovku. Aký by malo byť zrejmé zo samotného algoritmu.

☑ Kontrolný test

1. Vyberte správne tvrdenie:
   a. jazyk C vyvinuli Dennis Ritchie a Brian Kernighan
   b. bol vyvinutý pre aplikácie v oblasti administratívy
   c. jeho názov súvisí s jazykom B, z ktorého prebral veľa vlastností
   d. bol vyvinutý pre potreby operačného jazyka Windows

2. Každý program musí obsahovať funkciu
   a. `include`
   b. `printf`
   c. `scanf`
   d. `main`.

3. Dátový typ `int` zaberá:
   a. 2 byty
   b. 4 byty
   c. 8 bytov
   d. v závislosti od implementácie 2 alebo 4 byty

4. Vyberte nesprávne tvrdenie:
   a. reálny typ `float` zaberá vždy 4 byty
   b. reálny typ `double` poskytuje väčšiu presnosť ako dátový typ `float`

c. *n*-bitové premenné typu `unsigned` majú rozsah od 0 do 2^n
d. použitím `signed` stanovíme, že typ bude znamienkový.

5. Nech celočíselná premenná `J` má hodnotu 5. Výsledkom operácie `J--` bude:
a. 5
b. 6
c. 4
d. 0.

6. Nech celočíselná premenná `J` má hodnotu 5. Výsledkom operácie `J*=2` bude:
a. 2
b. 10
c. 7
d. 3.

7. Majme program:

```
…
int i=6,j=4;
double x;
x=i/j;
…
```

Výsledkom operácie je:
a. 2,5
b. 1,5
c. 1
d. 2

8. Vyberte nesprávne tvrdenia:
a. syntaktická chyba je zapríčinená tým, že príkaz je zapísaný nesprávne z hľadiska pravidiel výstavby (zápisu) programovacieho jazyka
b. na príkaze `j=(a+b*c;` vyhlási kompilátor syntaktickú chybu
c. príkaz `j=(a+b*c;` obsahuje logickú chybu
d. logická chyba je chyba, ktorá sa nemusí prejaviť tým, že program „padne“. Program sa môže vykonať zdanlivo bez chyby, ale nevykoná to, čo bolo jeho úlohou.

9. Vyberte príkazy, ktorými dáme vypísať do jedného riadku dve celé čísla uložené v premenných `i` a `j` a jedno reálne číslo typu float, uložené v premennej `x`:
a. `printf( "%d %d %f\n", i,j,x);`
b. `printf( "%d ",i); printf( "%d ",j); printf( "%f \n",x);`
c. `printf( "%f ",i); printf( "%f ",j); printf( "%d \n",x);`
d. `printf( "%d\n",i); printf( "%d\n",j); printf( "%f\n",x);`

10. Nech `i=4` a `j=3`. Po vykonaní príkazov

```
if (i>j) i++;
i=+1;
```

bude sa premenná `i` rovnať:
a. 4
b. 5
c. 6
d. 8

11. Nech `test=0` a `j=1`. Po vykonaní príkazov

```
if (!test) j+=2;
else j*=2;
j-=2;
```

sa bude premenná `j` rovnať:

a. 0
b. 4
c. 1
d. 2

12. Nech `a=2, b=2` a `c=3`. Označte odpovede, ktorých výsledkom je nenulová hodnota, (t. j. logické výrazy sú pravdivé):

a. `(a==b) || (b==c)`
b. `(a!=c) && (b!=c)`
c. `(a==b) && (a==c)`
d. `(!(a==b)) && (b==c)`

Kontrolné otázky

1. Za akým účelom sa postupne vyvíjali vyššie programovacie jazyky? Vymenujte niektoré a charakterizujte ich.
2. Charakterizujte jazyk C. V čom spočíva jeho prínos?
3. Charakterizujte celočíselné typy v jazyku C: aké sú, koľko zaberajú miesta, aký je rozsah premenných, ktoré sú daného typu, ako sa ich obsah vypisuje na obrazovku?
4. Charakterizujte reálne typy v jazyku C: aké sú, koľko zaberajú miesta, aká je presnosť a rozsah premenných, ktoré sú daného typu, ako sa ich obsah vypisuje na obrazovku?
5. Aký je rozdiel medzi definíciou a deklaráciou premenných?
6. Charakterizujte proces ladenia. Predstavte si, že máte program na výpočet faktoriálu. Aké chyby sa v ňom môžu vyskytnúť a ako by ste ich odstraňovali?
7. Čo chápeme pri ladení programov pod pojmom *breakpoint*?
8. Čo rozumieme pod riadiacimi štruktúrami?
9. Charakterizujte podmienený príkaz. Čo je jeho účelom a kedy sa používa?
10. Popíšte postup tvorby programu od jeho napísania po jeho spustenie.
11. Akým príkazom dáme zdrojový program preložiť a ako ho spustíme?

Cvičenia

1. Načítajte číslo zo vstupu a zistite, či je párne alebo nepárne. Výsledok vypíšte. (použite operáciu modulo). Načítajte postupnosť piatich čísel, vypíšte, koľko z nich je párnych.
2. Načítajte z klávesnice postupnosť piatich čísel 0 alebo 1. Vypíšte počet zadaných núl a jednotiek. Načítajte postupnosť desiatich čísel 10, 20 alebo 30 a vypíšte, koľko bolo ktorých.
3. Načítajte zo vstupu postupnosť piatich reálnych čísel a vypíšte ich:
    a. súčet
    b. súčet druhých mocnín
    c. súčet druhých odmocnín
    d. aritmetický priemer
    e. minimum
    f. maximum.

4. Ako zistíte, aká je v danom čísle číslica na pozícii jednotiek, desiatok, stoviek?
5. Ako zistíte, že načítané celé číslo je trojmiestne?
6. Sformulujte v C test, ktorým zistíte, či číslo patrí do intervalu <0,20 > príp. (0,20)?
7. Predstavte si, že v x a y sú uložené výsledky vášho výpočtu. Vypíšte na obrazovku reťazec: Hodnoty výpočtu sú: x = ? , y = ? , kde za ? dosadíte skutočné hodnoty x a y.
8. Načítajte zo vstupu (klávesnice) trojmiestne číslo. Zistite, či:
    a) je naozaj trojmiestne
    b) obsahuje dve sedmičky
    c) číslo obsahuje 7 alebo 1 alebo 5?
9. Načítajte zo vstupu čísla A,B,C ako koeficienty kvadratickej funkcie. Riešte kvadratickú funkciu, vypíšte riešenia. Nájdite bod, v ktorom má parabola vrchol.
10. V rovine je zadaný trojuholník súradnicami jeho bodov. Vypočítajte plochu. Použite analytickú geometriu a vektorový súčin. Nezabudnite vnoriť súradnice do priestoru.
11. Sú dané koeficienty dvoch priamok zapísaných v tvare ax + by +c = 0. Zistite,či sú priamky rovnobežné

Zadania

1. Zistite, čo robí tento program:

```
main()
{
double a,b,c,d,x;
printf("Zadaj a,b,c:");
scanf("%lf%lf%lf",&a,&b,&c);
d=b*b-4*a*c;
if (d<0) printf("smola\n");
else if (!d) printf("vysledok je %lf\n",(-b+sqrt(d))/(2*a));
     else printf("vysledok je %lf %lf\n",(-b+sqrt(d))/(2*a),(-b-
sqrt(d))/(2*a));
system("pause");
}
```

Na záver upresnite výpisy.

2.
Sú dané koeficienty dvoch priamok zapísaných v tvare ax+by+c=0. Zistite ich priesečník.

3 Cykly

Hlavné podtémy kapitoly:

- Príkaz skoku `goto`, príkazy `break` a `continue`, komentáre.
- Príkazy na opakované vykonávanie príkazov – iteračné cykly.
- Rôzne typy cyklov: cykly `while` a `for`, cyklus `do while`.
- Vzorové príklady na typické použitie jednotlivých typov cyklov.
- Násobenie dvoch celých čísel `(while)`, numerické hľadanie koreňa metódou postupného delenia intervalu `(do while)`, numerická integrácia lichobežníkovou metódou `(for)`.

3.1 Príkaz skoku `goto`

Tento príkaz nepatrí medzi veľmi používané príkazy a vždy sa mu dá vyhnúť. Používa sa iba v ojedinelých prípadoch, keď by sa program bez jeho použitia veľmi zneprehľadnil. Teraz však pomôže urobiť si predstavu, aká programátorská konštrukcia sa za cyklom skrýva.

Príkaz `goto` je vždy spojený s návestím. *Návestie* sa píše pred príkaz a je to identifikátor nasledovaný dvojbodkou. Návestia nemusia byť vopred deklarované a ich identifikátory môžu byť zhodné s identifikátormi použitých premenných bez toho, aby vznikol zmätok. Skočiť sa dá takmer ľubovoľne, čo robí tento príkaz obzvlášť nebezpečný.

```
goto Nav;
...
Nav: príkaz;
```

3.2 Cykly

Veľakrát potrebujeme vykonať ten istý príkaz opakovane. Napríklad chceme zo vstupu načítať 20-krát jeden riadok, alebo chceme sčítať desať po sebe idúcich čísel....V princípe chceme veľa krát vykonať skupinu tých istých operácií, ktoré spracovávajú údaje. Koľkokrát závisí od splnenia nejakej podmienky. Napríklad môžeme stanoviť, že skupina operácií sa má vykonať *n* - krát, napr. v algoritme 1 sme *n* – krát pričítali k premennej `Suma` načítané číslo uložené v `A`. V príklade s hľadaním numerického koreňa sme skupinu operácií opakovali, pokiaľ rozdiel dvoch hodnôt nebol dostatočne malý. Táto podmienka sa môže testovať pred vykonaním skupiny príkazov alebo po ich vykonaní.
V programovacích jazykoch existujú pre takéto výpočty konštrukcie, ktoré sa nazývajú *cykly*. Jednotlivé cykly sa líšia podľa toho, ako sa zapisujú a kedy sa testuje podmienka cyklu.

V C-jazyku máme nasledovné cykly:

- **`while`**
- **`for`**

} cykly s podmienkou na začiatku

- **`do ... while`** cyklus s podmienkou na konci

3.2.1 Cyklus `while`

Príkaz **`while`** realizuje v C cykly s podmienkou na začiatku.

`while (výraz1) príkaz1`

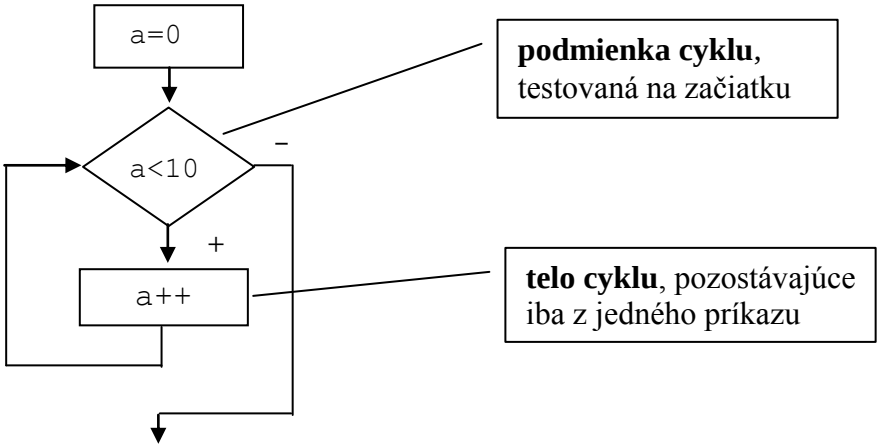
Príkaz **`while`** pracuje tak, že sa najprv vyhodnotí **`výraz1`** a potom, ak je vyhodnotený ako nenulový, sa vykoná **`príkaz1`**. Po jeho vykonaní sa znovu vyhodnotí **`výraz1`** a prípadne sa znovu vykoná **`príkaz1`**. Cyklus končí v okamžiku, keď je podmienka vyhodnotená ako 0. **`Príkaz1`** sa už nevykoná a program pokračuje ďalším príkazom po **`while, príkaz1`** môže byť aj blok. Testovaný výraz musí byť vždy v zátvorkách.

Príklad 1

```
int a=0;
while (a<10) a++;
```

V tomto príklade sa desaťkrát vykoná *inkrementácia* `a,` t.j. hodnota `a` sa 10-krát zvýši o 1.

Tento príklad by sme mohli znázorniť nasledovným vývojovým diagramom:



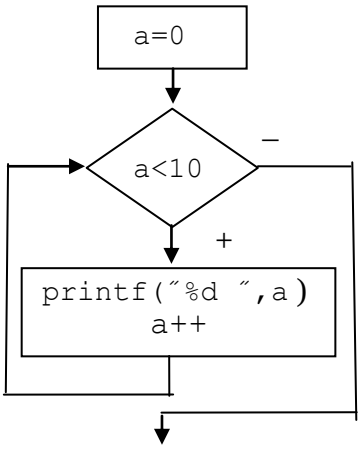
Tento istý počtový úkon by sme vedeli zapísať aj prostriedkami nám už doteraz známymi:

```
     a=0;
Nav:
     if (a<10)
        {a++;
         goto Nav;}
     ...
```

Príkaz cyklu vlastne nahrádza skupinu takýchto príkazov.

Príklad 2

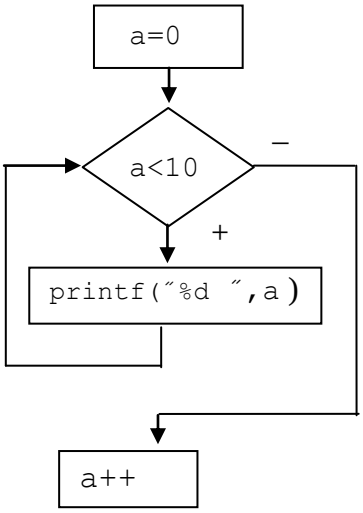
Do predchádzajúceho príkladu dodajme ešte výpis hodnoty premennej `a` pred každým jej zvýšením. Nech je algoritmus vyjadrený nasledujúcim vývojovým diagramom:



Otázka.
Zodpovedá mu nasledujúci kód?

```
int a=0;
while (a<10)
printf(˝%d ˝,a);
a++;
```

Riešenie:
Nie, uvedenému kódu zodpovedá takýto diagram:



Tento program sa „zacyklí“: bude sa neustále vypisovať hodnota 0. Podmienka cyklu je na začiatku splnená, jeho telo tvorí iba jeden príkaz, v ktorom sa hodnota `a` nemení. Podmienka cyklu je stále splnená a vznikne nekonečný cyklus.

Správny zápis kódu je:

```
int a=0;
while (a<10)
   {printf(˝%d ˝,a);
    a++;}
```

Keďže za `while` nasleduje iba jeden príkaz, v prípade viacerých inštrukcií ich musíme uzavrieť do bloku (ktorý je považovaný za jeden príkaz). Pri tomto zápise sme už použili odsadenie. Je dobrým zvykom takto zapisovať príkazy tela cyklu kvôli lepšej čitateľnosti.

Príklad 3
Aká hodnota `a` bude vypísaná po vykonaní nasledujúcej skupiny príkazov:

```
int a=0;
while (a<10);
     {printf(˝%d ˝,a);
      a++;}
```

Riešenie:
Ako sme povedali, za `while` sa vždy vykoná iba jeden príkaz. V tomto prípade sme tam nesprávne napísali bodkočiarku, ktorá predstavuje prázdny príkaz. V tomto prípade, pokiaľ je splnená podmienka cyklu, bude sa vykonávať prázdny príkaz (teda nič), a teda hodnota `a` sa meniť nebude. Podmienka cyklu bude stále splnená, cyklus nikdy neskončí.

Za podmienku cyklu `while` sa nepíše bodkočiarka. Podmienka musí byť v zátvorke, akýmkoľvek jednoduchým výrazom je tvorená!!!!

3.2.2 Cyklus `for`

Cyklus **`for`** je len trochu rozšírený príkaz `while`, pomocou ktorého môžeme prehľadnejšie zapisovať iteračné cykly. Kto má skúsenosti s inými programovacími jazykmi vidí, že príkaz **`for`** sa od podobných príkazov v iných jazykoch líši už na prvý pohľad. Avšak jeho hlavné uplatnenie bude podobné ako v iných jazykoch - v iteračných cykloch. Často ho používame **pri práci s poliami**, ktorá bude vysvetlená v závere tejto kapitoly.
Ide o typ cyklu, kde sa podmienka testuje na začiatku.

```
for (inicializačný výraz; koncový výraz; iteračný výraz) príkaz1;
```

Jednotlivé časti príkazu `for` majú širší význam ako ďalej popíšeme, my však opis tohto príkazu zjednodušíme, pretože tieto skriptá sú určené predovšetkým začiatočníkom.

Príkaz `for` sa vykonáva v týchto krokoch:

1. *Inicializačný výraz* je určený v prvom rade pre nastavenie *riadiacej premennej cyklu (iteračnej premennej).* Táto premenná sa bude v priebehu cyklu meniť a jej hodnota je rozhodujúca pre ukončenie cyklu. Tento krok sa vykoná iba raz na začiatku a ďalej sa vykonávajú už len kroky 2, 3 a 4.
2. Je vyhodnotený *koncový výraz*. Ak je vyhodnotený ako 0, je vykonávanie cyklu `for` ukončené a pokračuje sa prvým príkazom uvedeným po cykle. Ak je ale vyhodnotený ako nenulový, pokračuje sa krokom 3.
3. Vykoná sa *príkaz1* (ktorý môže byť aj blok).
4. Vyhodnotí sa *iteračný výraz*. Opäť je už z názvu zrejmé, že slúži predovšetkým k iterácii (pozmenení) riadiacej premennej cyklu. Nasleduje skok na krok 2.

Príklad 4
Použitie cyklu `for` pre príklad 2 a súčasne jeho typické použitie:

```
int a;
for (a=0; a<10; a++)
   printf(˝%d ˝,a);
```

Ako inicializačné a iteračné výrazy môžu byť použité výrazy akéhokoľvek typu. Je tiež možné vynechať ľubovoľný z týchto výrazov. Napríklad vynechaním inicializačného a iteračného výrazu vlastne získame cyklus `while`.

Príklad 5

```
int a=0;
for (;a<10;)
    {printf(˝%d ˝,a);
     a++};
```

Tu vidíme, že zápis z príkladu 4 je elegantnejší a hlavne prehľadnejší.
Vynechaním koncového výrazu dosiahneme nekonečný cyklus, pretože chýbajúci terminálny výraz bude nahradený nejakou nenulovou konštantou, a tá bude vždy vyhodnotená ako rôzna od nuly.

Inicializačný, koncový a iteračný príkaz sú oddelené bodkočiarkou, a nie čiarkou. Za `for(...)` sa nepíše bodkočiarka, tá sa chápe ako prázdny príkaz!!!

Príklad 6

Načítajme z klávesnice 10 celých čísel a spočítajme, koľko z nich je

a) kladných,
b) párnych,
c) nie je deliteľných tromi.

Riešenie:

a)

```
main(){
int i,cislo,pocitadlo=0;
for (i=0;i<10;i++)
     {scanf("%d", &cislo);
     if (cislo>0) pocitadlo++;
     }
printf("pocet hladanych cisel je: %d\n", pocitadlo);
}
```

b) Zvýraznený riadok nahradíme príkazom

```
if (!(cislo%2)) pocitadlo++;
```

...

c) Zvýraznený riadok nahradíme príkazom

```
if (cislo%3) pocitadlo++;
```

...

Príklad 7

Vypočítajte súčet druhých odmocnín čísel 5 až 10.

Riešenie:

```
#include <stdio.h>
#include <math.h>
main(){
double x,suma=0;
for (x=5;x<=10;x++)
     suma+=sqrt(x);
printf("hladany sucet je: %lf\n",suma); }
```

Príklad 8
Naprogramujme algoritmus 1 z kapitoly 1 pre výpočet priemeru *n* čísel načítaných zo vstupu. Premennú *n* nastavme na 10.

Riešenie:

```
#include <stdio.h>
main(){
int I,n=10,A;
float Suma,Priemer;
Suma=0;
for (I=0; I<n; I++)
   {scanf(˝%d˝,&A);
    Suma+=A; }
Priemer=Suma/n;
printf("priemer cisel je: %f\n",Priemer);
}
```

Ďalšie príklady použitia

Nezamýšľajte sa nad praktickým použitím, všímajte si iba spôsob zápisu.

```
for (x=100;x>50; x--)
for (x=0;((x>3) && (x<9)); x++)
for (x=0,y=4;((x>3) && (y<9)); x++,y+=2)
for (x=0,y=4,z=4000;z; z/=10)
```

V treťom prípade si všimnite, že môžete inicializovať a modifikovať aj viacero výrazov, ktoré sú oddelené čiarkami. V štvrtom prípade sa cyklus bude vykonávať, pokiaľ `z` bude rôzne od nuly.

Niekedy, väčšinou omylom, použijeme v cykle premennú s rovnaký menom ako riadiaca premenná. Nie je to samozrejme chyba syntaktická, ale pre cyklus to môže mať nevyspytateľné dôsledky. Zistite, čo urobí nasledujúci program, ktorého pôvodnou úlohou bolo vypísať obsah riadiacej premennej pri každom vykonaní tela cyklu. Cyklus sa mal vykonať 10 krát. Omylom sa nám „priplietol“ riadok `i++`.

Príklad 9

```
#include <stdio.h>
main()
{
int i;

for (i=0;i<10;i++)
  {i++;
   printf("%d \n",i);
  }
}
```

Použitie príkazu `while` alebo `for` je rovnocenné, pričom výber závisí od toho, ktorý príkaz je v danej situácii jasnejší.

3.2.3 Cyklus `do - while`

Podobný cyklu `while` je príkaz `do – while`. Tento cyklus ale **vyhodnocuje podmienku pokračovania (`výraz1`) až po vykonaní tela cyklu**. To znamená, že aspoň raz sa telo cyklu vždy vykoná.

```
do príkaz1 while (výraz1);
```

Najprv sa vykoná `príkaz1` a až po jeho vykonaní je vyhodnotený výraz podmienky. Pokiaľ je nenulový, znovu sa vykoná `príkaz1`. To pokračuje a cyklus skončí, keď je podmienka vyhodnotená ako nulová.

Príklad 10
V nasledujúcom príklade 10 krát vypíšeme riadok s textom `ahoj`.

```
a=0;
do
  {printf(„ahoj\n“) ;
   a++;}
while (a!=10);
```

Na tomto príklade je vidieť, že sa dá, hlavne u zložitejších programov, ľahko popliesť so zápisom obyčajného cyklu `while`. Z toho dôvodu sa ukončovacia zátvorka bloku často píše pred slovo `while`. Tak programátor naznačí sám sebe, ale aj druhým ľuďom, ktorí budú s programom pracovať, že nejde o cyklus `while` ale `do - while`. Odporúčaný zápis je:

```
a=0;
do
  {printf(„ahoj\n“) ;
   a++;
}while (a!=10);
```

Cykly sa dajú do seba vnárať.

```
for (i=0;i<5; i++)
   for (j=0;j<5; j++)
     printf(“i=%d, j=%d \n“,i,j);
```

Napíšte program a spustite ho. Všimnite si, ako sa menia hodnoty `i` a `j`.

3.2.4 Príkazy `break` a `continue`

`break;`

Príkaz `break` môže byť použitý v tele cyklov a v tele príkazu `switch`, pričom jeho použitie spôsobí okamžité opustenie cyklu (alebo príkazu `switch`). Ide teda o skok na prvý príkaz za cyklom (alebo za príkazom `switch`). V prípade vnorených cyklov pomocou neho opustíme vždy najvnútornernejšiu neuzavretú slučku.

`continue;`
Príkaz `continue` skočí na koniec najvnútornejšej neuzavretej slučky a tým si vynúti ďalšiu iteráciu slučky. Tento príkaz cyklus neopúšťa.

3.2.5 Znovu príkaz `goto`

Jedným z rozumných použití tohto príkazu je vyskočenie z vnorených cyklov.

```
double x,y;
int i,j,k;
.....
for (i=0;i<10; i++)
   for (j=0;j<10; j++)
     for (k=0;k<10; k++)
          {
```

```
            scanf("%lf",&y);
            if (y==0)
               goto Error;
            else {x=(i*j*k)/y;
                   printf("x= %lf\n",x);
                  }
             }

...

Error:
   printf("nulový deliteľ\n");

.......
```

Ak chceme vyskočiť iba z jednej slučky, je namiesto príkazu `goto` vhodnejšie použiť príkaz `break`.

3.3 Príklady typického použitia typov cyklov

V prípade cyklov `while` a `do while` nevieme vopred, koľkokrát sa má cyklus vykonať. Dobrým príkladom môže byť násobenie dvoch čísel „sedliackou metódou“ alebo hľadanie koreňov funkcií algoritmom 3. Vezmime metódu polenia intervalu: dovtedy zmenšujeme nejaký interval, až kým sa funkčná hodnota v jeho strede nelíši od nuly o nejakú malú vopred stanovenú hodnotu. Cyklus `for` väčšinou používame vtedy, keď vieme, koľkokrát sa má cyklus vykonať (buď vopred alebo ako dôsledok nejakého výpočtu) a jeho riadiacu premennú aj reálne používame.

- Cyklus `while`

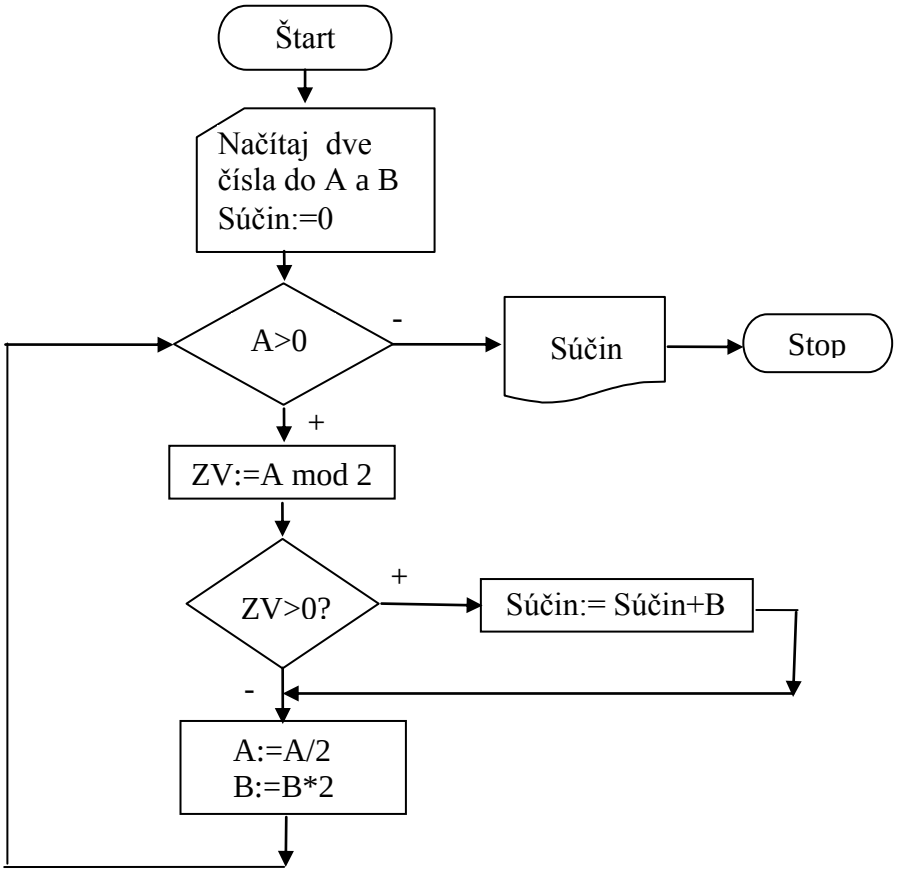
Príklad 11 (Algoritmus 3)
Násobenie čísel „sedliackou" metódou

Príklad postupu na papieri z prvej kapitoly:

Vynásobme čísla 21 a 17.

21	17
~~10~~	~~34~~
5	68
~~2~~	~~136~~
1	272
	357

Celý postup môže byť vyjadrený nasledujúcim vývojovým diagramom z kapitoly 1.



V tomto algoritme nevieme vopred, koľkokrát sa cyklus vykoná. Pohodlne preň môžeme využiť cyklus `while`. Algoritmus môže vyzerať takto:

```
#include <stdio.h>
main()
{
int a, b, zv, súčin;
súčin = 0;
scanf("%d%d",&a,&b);
while(a)
  {zv=a % 2;

  if (zv) súčin = súčin + b;
  a=a/2;
  b=b*2;
  }
 printf("%d \n", súčin);
}
```

Všimnite si použitie `while(a)` a `if(zv)` namiesto `while(a>0)` a `if(zv>0)`.

- Cyklus `do while`

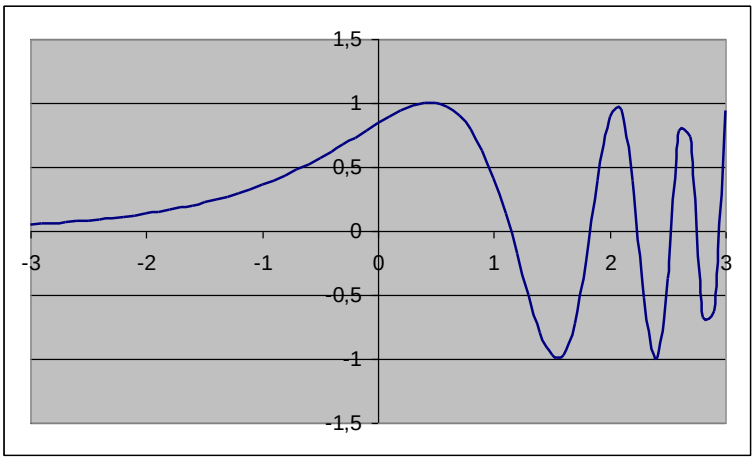
Príklad 12 (Algoritmus 4)
Numerické hľadanie koreňa metódou postupného delenia intervalu

V kapitole 1 si prečítajte podmienky, ktoré musia byť splnené, aby tento algoritmus robil to, čo má.

Slovný zápis algoritmu je nasledovný:

1. Zadajte dva body *DH* (ľavá, dolná hranica intervalu) a *HH* (pravá, horná hranica intervalu), v ktorých má funkcia f rôzne znamienka. (Napríklad ich zistite z grafu funkcie).
2. Nájdite *S*, stred (*DH* , *HH*).
3. Ak $|f(S)|$ je v rámci nejakej predpísanej tolerancie, t.j. menšia ako vopred dané malé číslo, tak za koreň budeme považovať *S*.
4. Ak f(*S*) má to isté znamienko ako *f(DH)*, tak opakujme proces s *DH*=*S*, inak opakuj proces s *HH*= *S*, t.j. choď na bod 2.

Nájdime nejaký koreň funkcie $sin(e^x)$. Vyberme si interval `<1;1.5>`.



Príklad programu pre riešenie tohoto problému:

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

main()
{
double DH,HH,S,hodnota_v_s;
double tol=0.0001;

printf("zadaj dolnu a hornu hranicu:\n");
scanf("%lf %lf",&DH,&HH);

/* test spravnosti zadania hranic*/
d=sin(exp(DH));
h=sin(exp(HH));
if (fabs(d)<tol){printf("DH je korenom\n");return;}
if (fabs(h)<tol){printf("HH je korenom\n");return 1;}

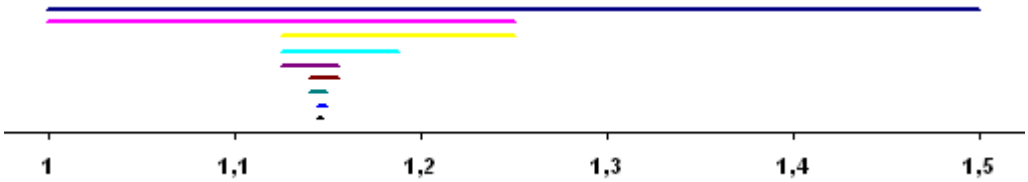
if (d*h>0) {printf("zle zadane hranice\n");return 1;}

do
  {S=(DH+HH)/2.;
   if (sin(exp(DH))*sin(exp(S))<0) HH=S;
   else DH=S;
   hodnota_v_s=fabs(sin(exp(S)));
   fprintf(f,"%lf %lf\n", DH,HH);
  }while( hodnota_v_s>=tol);
```

```
printf("korenom je %lf\n",S);
}
```

Tento krát sme hodnotu, ktorú bolo treba testovať, t.j. `hodnota_v_s`, dostali až po vykonaní tela cyklu, a preto je testovanie podmienky zaradené až na konci a je použitý cyklus s testovaním podmienky na konci. Príkaz `return`, ktorý v tomto prípade ukončí vykonávanie hlavného programu, sme použili aj s návratovou hodnotou aj bez nej. Niektoré prekladače návratovú hodnotu vyžadujú. V našom príklade na nej nezáleží.

Postupnosť vypísaných intervalov znázormíme graficky:

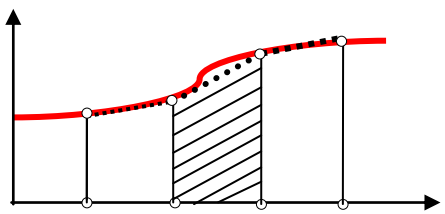


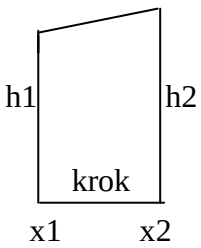
Koreňom je hodnota 1,144775.

Text uzavretý medzi dvojicu zátvoriek /* a */ je *komentár*. Je to text, ktorý kompilátor ignoruje a slúži k sprehľadneniu, niekedy na prvý pohľad dosť nepochopiteľného programu. Komentár slúži k tomu, aby sa v našich programoch vyznal aj niekto cudzí, prípadne my sami, keď pristúpime k programu po dlhšej dobe (hlavne, ak sú v ňom použité nejaké „fígle“.) Nezabúdajme komentár ukončiť. Ak zabudneme, aj časť kódu je považovaná za komentár a prekladač obyčajne býva zmätený. ANSI C nedovoľuje vnorené komentáre.

- Cyklus `for`

Príklad 13
Numerická integrácia lichobežníkovou metódou





Vypočítať určitý integrál na intervale <a,b> znamená vypočítať na danom intervale plochu, ohraničenú funkciou a osou *x*, pričom plocha pod osou *x* má záporné znamienko. Numerická integrácia spočíva v rozdelení intervalu <a,b> bodmi, v ktorých sú známe funkčné hodnoty, na podintervaly a na tých pôvodnú funkciu nahradíme funkciami jednoduchšími, v prípade lichobežníkovej metódy nahradíme funkciu po čiastkach lineárnymi funkciami (na obrázku znázornenými bodkovanou čiarou). Určitý integrál na danom podintervale nahradíme plochou príslušného lichobežníka. Tá sa vypočíta ako `(a+b)*v/2`, kde `a` a `b` sú základne a `v` je výška lichobežníka. Čím jemnejšie bude delenie (t.j. čím viac bude deliacich bodov), tým presnejší bude vypočítaný integrál. Na obrázku vpravo je nakreslený lichobežník, ktorého plochu vlastne počítame. `h1=f(x1)=a` a `h2=f(x2)=b` sú základne a dĺžka intervalu, t.j. krok, je výška lichobežníka.

Vypočítajme určitý integrál funkcie *sin(x)* na intervale <0,3.14>. Vstupom do programu bude dolná a horná hranica intervalu a počet bodov delenia.

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

main()
{
double  a,b,pocet_krokov, krok,P;
double x1,x2,h1,h2;
int i;

printf("Zadaj a,b a pocet bodov delenia:");
scanf("%lf %lf %lf",&a,&b,&pocet_krokov);

krok=(b-a)/pocet_krokov;

P=0;
for (i=0;i<pocet_krokov;i++)
      {
      x1=i*krok;
      x2=x1+krok;
      h1=sin(x1);
      h2=sin(x2);
      P=P+(h1+h2)*krok/2;
      }
printf("Urcity  integral je: %lf\n",P);
}
```

Pre počet bodov delenia 100 je výsledok `1.999834`.
Pre počet bodov delenia 1000 je výsledok `1.999997`.
Pre počet bodov delenia 10000 je výsledok `1.999999`.

☑ Kontrolný test

1. Vyberte správne tvrdenia:
   a. návestie pre príkaz `goto` nesmie mať meno zhodné so žiadnou premennou
   b. návestie musí byť vopred deklarované
   c. rozumné použitie príkazu `goto` je vyskočenie z do seba vnorených cyklov
   d. v prípaden dvoch a viac do seba vnorených cyklov má príkaz `break` rovnaký význam ako `goto`

2. Vyberte správne tvrdenia. Cyklus `while`
   a. sa musí vykonať vopred daný počet krát
   b. podmienka cyklu `while` sa testuje vždy na začiatku cyklu
   c. podmienka cyklu `while` sa testuje vždy po vykonaní tela cyklu
   d. cyklus sa vykonáva, pokiaľ podmienka cyklu predstavuje výraz rôzny od nuly.

3. Príkaz `while ((i>1)&&(i<1));` predstavuje
   a. cyklus, ktorý sa ani raz nevykoná
   b. nekonečný cyklus

c. jeho správanie závisí od nastavenia premennej `i`
d. je to chybný príkaz, na ktorom prekladač vyhlási chybu.

4. Vyberte nesprávne tvrdenia:
   a. cyklus `for` je rozšírením cyklu `while`
   b. cyklus `while` je rozšírením cyklu `for`
   c. z jednoduchého cyklu for môžeme kedykoľvek vyskočiť pomocou `break`
   d. z jednoduchého cyklu for môžeme kedykoľvek vyskočiť pomocou `goto`.

5. Vyberte nesprávne tvrdenia:
   a. cyklus `do while` testuje podmienku vždy po vykonaní tela cyklu
   b. použitie príkazu `break` zapríčiní, že sa riadenie nastaví na posledný príkaz tela cyklu
   c. použitie príkazu `break` zapríčiní, že sa riadenie nastaví na prvý príkaz za cyklus
   d. podmienka cyklu `do while` môže byť ľubovoľný logický výraz.

6. Majme príkazy:

```
j=0; k=0;
for(i = 0; i< 5; i++)
    j+=2;
    k++;
```

Po vykonaní príkazov budú hodnoty premenných `j` a `k` nasledovné:
   a. `j=10; k=5;`
   b. `j=10; k=0;`
   c. `j=10; k=1;`
   d. `j=2; k=1;`

7. Majme príkazy:

```
j=0; k=0;
for(i = 0; i< 5; i++);
          j+=2;
          k++;
```

Po vykonaní príkazov budú hodnoty premenných `j` a `k` nasledovné:
   a. `j=10; k=5;`
   b. `j=10; k=0;`
   c. `j=10; k=1;`
   d. `j=2; k=1;`

8. Majme program:

```
int  A=10, B=5;
for  (;A;A/=2)
      B=B*2;
```

Výsledkom jeho vykonania je :

   a. `B=10`
   b. `B=20`
   c. `B=40`
   d. `B=80.`

9. Majme príkazy :

```
int i,n=100;
double DI,S,K,x;
DI=1; S=0;K=DI/(n-1);
for(i = 0; i<(n-1); i++)
   {x=i*K;
    S=S+x*x*K;
    }
```

Vyberte správne tvrdenia:

a. program numericky vypočíta určitý integrál funkcie $y=x$
b. program numericky vypočíta určitý integrál funkcie $y=x^2$
c. ak n zvýšime, výsledok bude presnejší
d. ak n znížime, výsledok bude presnejší .

10. Majme príkazy:

```
l=0;
for(i = 0; i<= 2; i++)
   {for(j = 0; j< 5; j++)
      { l++;
        if (j>2) break;
        l++;}
    l++;
    }
```

Poskončení cyklus bude hodnota l rovná:

a. 7
b. 8
c. 24
d. 10.

11. Majme príkazy:

```
l=0;
for(i = 0; i<= 2; i++)
   {for(j = 0; j< 5; j++)
      { l++;
        if (j>2) continue;
        l++;}
    l++;
    }
```

Poskončení cyklu bude hodnota l rovná:

a. 7
b. 27
c. 9
d. 10.

12. Majme príkazy:

```
z=12;x=0;
do
   {z--;
    x++;
    } while (!z)
```

Po skončení cyklus sa bude hodnota `x` rovnať:

a. 12
b. 1
c. 13
d. 10.

Kontrolné otázky

1. Charakterizujte rozdiely medzi príkazmi `goto`, `break` a `continue`.
2. Nájdite ďalšie typické príklady na použitie všetkých troch typov cyklov.
3. Prejdite všetky vzorové algoritmy z prvej kapitoly. Ak je príslušný algoritmus vhodný na použitie cyklu, premyslite si, aký typ cyklu by ste použili.
4. Skúste vymenovať najčastejšie chyby, ktoré sa vyskytujú pri práci s cyklami.
5. Vysvetlite rozdiel v použití `break` a `goto NAV` v tele dvoch do seba vnorených cyklov.
6. Ako je chápaná bodkočiarka za záhlavím cyklu `for`, prípadne `while`?
7. Čo je to nekonečný cyklus? Vymyslite príklad nekonečného cyklu pre cyklus `for`.
8. Napíšte, čo bude výsledkom cyklu:

```
s=0;
for (i=0;i<10;i+1)
  s++;
```

9. Napíšte, čo bude výsledkom cyklu:

```
s=0;
for (i=0;i<10;i+=2)
  s++;
```

Cvičenia

1. Vypočítajte všetky korene uvedených funkcií (zvoľte vhodné intervaly) s presnosťou na 5 desatinných miest:
   a. $x^2 - 2$
   b. $\ln(3x - 4)$
2. Lichobežníkovou metódou s počtom bodov delenia 1000 vypočítajte určitý integrál zadaných funkcií na zadaných intervaloch:
   a. x^3 na <0,1>,
   b. e^{x*x} na <0,1>,
   c. $\sqrt{(1 - x^2)}$ na <0,1>.
3. Vypočítajte určitý integrál obdĺžnikovou metódou, t.j. funkciu nahraďte na danom intervale konštantnou funkciou danou hodnotou funkcie v ľavom bode intervalu.
4. Napíšte program pre výpočet faktoriálu. Použite cyklus `for`.
5. Napíšte program pre výpočet faktoriálu. Použite cyklus `while`.
6. Vypíšte tabuľku funkčných hodnôt funkcie e^x pre $1 \le x \le 10$ s krokom 0,25.
7. Do tabuľky 10 x 10 vypíšte 100 členov postupnosti

$$\frac{\sqrt[3]{n^2 + n}}{n + 1}.$$

8. Vypočítajte súčet:

$$\sum_{n=1}^{100}\left(\sqrt{n}-\sqrt{n-1}\right)$$

Ako sa zmení súčet, keď sčítame 200 členov?

9. Vypíšte hodnoty funkcie $\frac{\sin(x)}{x}$ pre *x* idúce od 0.1 do 0.01 s veľkosťou kroku -0.005.

10. Vymyslite spôsob, ako by ste odhadli

$\lim_{x\to\infty}\left(1+\frac{1}{n}\right)^n$ s presnosťou na 4 desatinné miesta.

Zadania

1. Zistite, čo robí nasledujúci program:

```
#include <stdio.h>
main()
{
int i,j;
char znak;

for (i=0;i<16;i++)
{for (j=0;j<16;j++)
 printf("%d %c\n",i*16+j,i*16+j);
 scanf("%c",&znak);}
}
```

2. Napíšte program pre výpočet čiastkových súčtov radu:

$\sum_{n=1}^{k}\frac{\ln n}{\sqrt[4]{n^5}}$,

pre k= 100, 200, 300, 400, 500.
Výsledky vypíšte vo forme tabuľky:

k	súčet(k)
100	...
200	...
300	...
400	...
500	...

Výsledky testu:1C, 2BD, 3A, 4B, 5B, 6C, 7D, 8D, 9BC, 10C, 11B, 12B.

4 Statické polia. Funkcie a procedúry

Hlavné podtémy kapitoly:

- Jednorozmerné statické polia, ich definícia. Prístup k prvkom poľa.
- Najčastejšie chyby pri práci s poliami, minimálny a maximálny index.
- Vzťah medzi jednorozmerným a viacrozmerným statickým poľom.
- Funkcie, ich význam a definícia. Rozdiel medzi funkciou a procedúrou.
- Rekurzívne funkcie. Príklad výpočtu faktoriálu pomocou cyklu a rekurzívnou funkciou.

4.1 Polia

4.1.1 Jednorozmerné pole

Vezmime si príklad 1 z kapitoly 1 (sčítanie 10 celých čísel a vypočítanie ich priemeru), kde 10 čísel postupne načítame z klávesnice, sčítame ich a vypočítame ich priemer. Teraz si predstavme, že chceme uložiť 10 čísel do pamäte. V priebehu výpočtu niektoré z nich pozmeníme výpočtom, znovu ich chceme sčítať a vypočítať priemer. Zásadný rozdiel však teraz spočíva v tom, že ich máme uložené v pamäti. Jedna z možností je, že ich uložíme do 10 premenných, z ktorých každá má svoje meno, napr. X1, X2 ...., X10 a vytvoríme príslušný aritmetický výraz

```
Suma= X1+X2+...+X10.
```

Ale predstavme si, že máme sčítať 100, 1000 alebo milión čísel (čo je pri numerických výpočtoch celkom bežné). Veríme, že okrem mimoriadne usilovných jedincov s perspektívou dlhého života by sa to nikomu nechcelo a že priemerne lenivý tvor sa okamžite začne obzerať po menej namáhavom riešení. Využime algoritmus 1 z kapitoly 1. Najprv musíme pouvažovať ako sa odvoláme na premenné inak ako ich (samostatným) menom. Urobíme to tak, že premenné uložíme do pamäte postupne za sebou do postupnosti, ktorej priradíme meno. Na jednotlivé prvky sa odvoláme pomocou mena a indexu, t.j. poradového čísla v postupnosti. Ak pomenujeme postupnosť A, tak na prvky sa budeme odvolávať ako na A[0], A[1], A[2]... Takúto postupnosť musíme vopred zadefinovať, podobne ako premenné, aby kompilátor vedel, koľko treba vyhradiť miesta. K tomu potrebuje vedieť, koľko prvkov bude mať dané pole a akého typu tie prvky budú. Pre definíciu platí šablóna

```
typ_prvkov_poľa  názov_poľa[počet_prvkov_poľa];
```

Príklad definície

```
int a[10];
double x[2*10+3];
```

Pole by sme teda mohli charakterizovať ako dátovú štruktúru zloženú z rovnakých prvkov. V ANSI C musí byť počet prvkov poľa (akýkoľvek) konštantný výraz, ktorý sa dá vyhodnotiť už počas prekladu[*)]. V C indexy polí začínajú vždy od nuly, a teda na prvý prvok sa dá odvolať ako na `a[0], x[0]`. Najvyšší index je o 1 menší ako počet prvkov poľa.

Takto definované pole potom môžeme výhodne spracovávať pomocou cyklu `for`.

[*)] Pozri dodatok.

V C indexy polí začínajú vždy od nuly. Najvyšší index je o 1 menší ako počet prvkov poľa !!!

Príklad 1
Dajme vypísať na obrazovku prvky poľa `a`.

```
...
int a[10];
...
for (i=0;i<10; i++)
    printf(˝%d ˝,a[i]);
```

Ak je pole definované takýmto spôsobom, teda počet prvkov poľa je vopred známy, označujeme ho ako **statické**. Niekedy nevieme vopred, aké veľké budeme pole potrebovať: jeho veľkosť bude určená nejakým výpočtom. V C je možnosť priradiť (alokovať) poľu miesto aj až počas výpočtu. Také pole sa nazýva **dynamické**. Pri práci s ním sa využíva technika smerníkov, ktorou sa budeme zaoberať neskôr.

Časté chyby pri práci s poľami

1. Väčšinou si každý uvedomí, že dolný index poľa je 0, ale často zabúda, že horný je o 1 menší ako veľkosť poľa. Majme pole definované príkazom `int x[10];` Čo urobí príkaz `x[10]=1?` V prvom rade si treba uvedomiť, že **C nekontroluje hranice polí,** teda prekladač nevyhlási žiadnu chybu. Po spustení programu tento príkaz priradí hodnotu 1 do pamäti bezprostredne za poľom `x`. Môže teda prepísať hodnotu nejakej premennej a zapríčiniť tak ťažko odhadnuteľnú chybu. Kontroly hraníc sú časovo veľmi náročné a C ako jazyk nižšej úrovne ich z dôvodu efektivity nevykonáva. Ak teda predpokladáme, že by v programe mohla nastať situácia, že by hranice mohli „pretiecť“ (nedodrží sa horná hranica) alebo „podtiecť“ (nedodrží sa dolná hranica), potom to musíme sami ošetriť.
2. Pretečenie alebo podtečenie hraníc je väčšinou zdrojom zle odhaliteľných chýb. Najčastejšie sú tzv. „chyby+1“. Sú to chyby, kde index o jednotku presahuje rozsah poľa. Z toho plynie poučenie, že keď program pracuje nesprávne a je podozrenie na zlú prácu s poľami, treba sa najskôr zamerať na „chybu+1“.

Príklad 2
Naprogramujme algoritmus 1 z kapitoly 1 pre výpočet priemeru *n* čísel. Čísla sú načítané zo vstupu do poľa. Priemer vypočítajme z prvkov poľa. Počet čísel *n* nastavme na 10.

Riešenie:

```
...
int a[10],i;
float Suma,Priemer;

/* načítanie čísel zo vstupu */
for (i=0; i<10; i++)
     scanf(˝%d˝,&a[i]);

/* vypočítanie súčtu*/
Suma=0;
for (i=0; i<10; i++)
     Suma+= a[i];
```

```
/* vypočítanie priemeru*/
Priemer=Suma/10;
...
```

V tomto prípade je dobré si uvedomiť, že záleží na tom, akého typu je premenná `Suma`. Keby bola typu `int`, delenie pri výpočte priemeru by sa vykonalo ako celočíselné a bez pretypovania by sme dostali nesprávny výsledok. Všimnite si, že kompilátoru nevadí, že sme do reálnej premennej priradili celé číslo. Niektoré typy pretypovaní sa dejú automaticky.

Príklad 3

Zmeňte v predchádzajúcom príklade počet prvkov poľa 10 na 100. Vidíme, že program musíme zmeniť na 4 miestach (a to máme len krátky program). Elegantnejšie by bolo namiesto 10 použiť nejaký symbol, ktorému na začiatku priradíme nejakú hodnotu. Z predchádzajúceho textu však vieme, že v ANSI C to nemôže byť premenná, pretože pri definícii statického poľa jeho veľkosť nemôže byť daná premennou, ale musí byť známa ešte pri preklade. Môžeme použiť symbolickú **konštantu**, ktorá sa definuje pomocou príkazu `#define`.

```
...
#define n 100
int a[n],i;
float Suma,Priemer;

/* načítanie čísel zo vstupu */
for (i=0; i<n; i++)
     scanf(˝%d ˝,&a[i]);
/* vypočítanie súčtu*/
Suma=0;
for (i=0; i<n; i++)
     Suma+= a[i]};
/* vypočítanie priemeru*/
Priemer=Suma/n;
...
```

Pri zmene počtu čísel tak program stačí modifikovať na jednom mieste - na začiatku.

Príklad 4

Vypočítajme vektorový súčin dvoch vektorov. Vektory uložme do polí `a` a `b` a inicializujme ich priamo v programe. Príklady budeme inicializovať takto:

```
double a[3]={3.0,2.0,1.0};
```

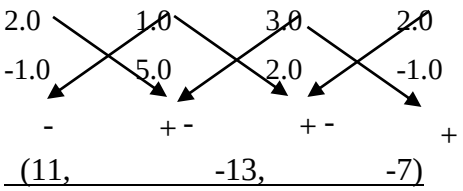
Tento príkaz nahradí príkazy `a[0]=3.0; a[1]= 2.0; a[2]= 1.0;`

Rozmer poľa môžeme vynechať, kompilátor ho doplní sám podľa počtu inicializovaných prvkov: `double b[]={2.0,-1.0,5.0};`

Na výpočet vektorového súčinu použijeme takýto postup: do dvoch riadkov napíšeme dve štvorčíslia tak, že v každom riadku bude 2., 3., 1. a 2. súradnica vektora. V našom prípade dostaneme:

2.0	1.0	3.0	2.0
-1.0	5.0	2.0	-1.0

Postupne zľava vypočítame 3 determinanty 2x2 krížovým pravidlom, a tak dostaneme postupne tri súradnice vektorového súčinu.



V programe urobíme ešte skúšku správnosti: vektorový súčin je taký vektor, ktorý je kolmý na obidva zadané vektory. Kolmosť vektorov overíme pomocou skalárneho súčinu, ktorý musí byť rovný nule.

Riešenie:

```
#include <stdio.h>
main()
{
double a[] = {3.0,2.0,1.0};
double b[] = {2.0,-1.0,5.0};
double c[3],ss1,ss2;
int i,k;
printf("Vektorovy sucin je:");
for (i=0; i<3; i++)
      {
      k = (i+1)%3;
      c[i] = a[k]*b[(k+1)%3] - a[(k+1)%3]*b[k];
      printf("%lf,",c[i]);
      }
printf("\n");
printf("Skuska spravnosti:\n");
ss1 = ss2 = 0; /*tieto vyrazy sa vyhodnocuju sprava*/
for (i=0; i<3; i++)
      {ss1 += a[i]*c[i];
       ss2 += b[i]*c[i];}
printf("a.c,b.c: %lf %lf\n",ss1,ss2);
}
```

Príklad výpisu:

```
Vektorovy sucin je:11.000000,-13.000000,-7.000000,
Skuska spravnosti:
a.c,b.c: 0.000000 0.000000
```

Pristavme sa pri príkaze `ss1=ss2=0`. Takýto príkaz, v C často používaný, sa vyhodnotí sprava, t.j. najprv sa do `ss2` priradí 0 a potom do `ss1`obsah `ss2` (teda 0).

4.1.2 Viacrozmerné polia

Dvojrozmerné (a podobne aj viacrozmerné) pole definujeme takto:

```
int a[2][5];
```

Toto pole si môžeme predstaviť ako tabuľku s 2 riadkami a 5 stĺpcami, **kde prvý index je číslo riadku a druhý index je číslo stĺpca**. Podobne ako jednorozmerné pole aj dvoj-

rozmerné pole sa dá inicializovať priamo v programe. Napríklad pole z nasledujúceho príkladu 3 môžeme inicializovať príkazom

```
int tab[2][2] = {{11,200},{75,1}};
```

Pri inicializácii vlastne vymenujeme riadky tabuľky. Každá zložená zátvorka obsahuje toľko čísel, koľko je stĺpcov a predstavuje jeden riadok tabuľky. Výrazov v zložených zátvorkách je toľko, koľko je riadkov v tabuľke. Ďalším príkladom inicializácie by mohlo byť:

```
int a[2][5] = {{1,1,1,1,1},{5,5,5,5,5}};
```

Príklad 5

Máme údaje zadané frekvenčnou tabuľkou (tabuľkou početností), t.j. pre každý údaj máme zadanú jeho hodnotu a koľkokrát sa v súbore nachádzal. Vypočítajme priemernú hodnotu dát. Takúto funkciu budeme nazývať *vážený priemer.* Napr. nech nasledujúce hypotetické údaje predstavujú počet a vek účastníkov, ktorý sa zúčastnili nejakej akcie (samé jedenásťročné deti a jeden penzista) .

vek	**počet**
11	200
75	1

Riešenie:

Pokiaľ sú údaje zadané frekvenčnou tabuľkou, nemôžeme použiť algoritmus 1. Je jasné, že priemer nie je ani 43 rokov (priemer z 11 a 75), ani 71,75 rokov (priemer z 200, 1, 11 a 75). Najprv musíme vypočítať súčet rokov všetkých účastníkov (200*11+1*75), ktorý potom vydelíme počtom všetkých účastníkov (200+1). Priemerný vek je 11,32.

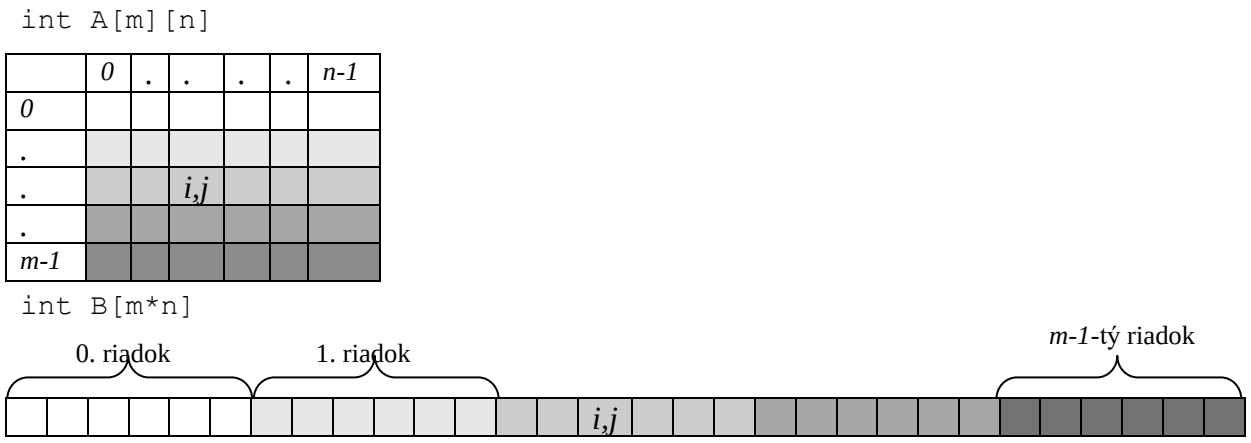
Čo sa týka organizácie údajov, mohli by sme použiť dve jednorozmerné polia `vek` a `počet`, aby sme si však precvičili dvojrozmerné polia, zavedieme pole `tab`, v ktorom prvý stĺpec bude určený pre vek a druhý pre frekvencie, t.j. počet účastníkov. V tejto tabuľke budú vždy len 2 stĺpce, ale počet riadkov má zmysel meniť, preto ho zavedieme ako konštantu `n`. Dáta do tabuľky budeme načítavať v dvoch do seba vnorených cykloch. Všimnime si, že pre pevný riadok meníme najprv index stĺpca (vo vnútornom cykle), a preto údaje budeme zadávať v poradí 11 200 75 1.

```
#define n 2
int tab[n][2];
int i,j,súčet,počet;
double priemer;
/*priradenie údajov zo vstupu do poľa*/
for(i = 0; i< n; i++)
  for(j = 0; j< 2; j++)
       scanf(˝%d˝,&tab[i][j]);
/*vypočítanie celkového súčtu*/
súčet = 0;
počet = 0;
for(i = 0; i< n; i++)
    {počet  += tab[i][1];
     súčet  += tab[i][0] * tab[i][1]; }
/*vypočítanie váženého priemeru*/
priemer = (double)súčet / počet;
```

4.1.3 Vzťah jednorozmerného a dvojrozmerného poľa

Aj viacrozmerné pole je často v pamäti uložené ako jednorozmerné, t.j. postupnosť za sebou idúcich prvkov. V prípade dvojrozmerného poľa sa za sebou ukladajú riadky tabuľky. Povedzme si, aký je vzťah medzi indexmi jednorozmerného a viacrozmerného poľa.

Majme dvojrozmerné pole o *m* riadkoch a *n* stĺpcoch. Predstavujeme si ho ako tabuľku (maticu), kde polohu každého prvku označujeme najprv riadkovým a potom stĺpcovým indexom. Dvojrozmernému poľu A bude zodpovedať jednorozmerné pole B.



Prvku `A[i][j]` zodpovedá prvok `B[i*n+j]`.
Prvku `B[k]` zodpovedá prvok `A[k/n][k%n]`.

4.2 Funkcie

Funkcie predstavujú základnú programovú jednotku, ktorá rieši nejaký problém. Poskytujú možnosť ako uzavrieť časť výpočtu do „čiernej skrinky“, ktorú je možné neskôr použiť bez ohľadu na to, čo je vo vnútri. Je to cesta, ako zvládnuť potenciálnu zložitosť veľkých programov. Správne navrhnuté funkcie umožňujú nevšímať si, *ako* sa určitá činnosť vykonáva, stačí len vedieť, *čo* sa vykonáva. Je možné ukryť podrobnosti jednotlivých operácií pred časťami programu, ktoré o nich nemusia vedieť, čím sa celok stáva jasnejším a vykonávanie zmien menej namáhavým. Okrem toho, že funkcie umožňujú rozčleniť veľké výpočtové úlohy na menšie celky, umožňujú aj stavať na tom, čo už urobili iní, a nie začínať z ničoho. Funkcia môže aj skracovať dĺžku programu, ak viackrát opakujeme tú istú činnosť.
Doteraz sme sa stretli s funkciou `main`, funkciami `sizeof`, `printf` a `scanf` a matematickými funkciami uloženými v knižnici, ale neskúšali sme si vytvoriť vlastnú funkciu.
Zatiaľ predpokladajme, že program píšeme do jedného súboru. **V ANSI C funkcie nemôžu byť do seba vnárané, t.j. jedna funkcia nemôže obsahovať definíciu inej funkcie[*)].** Nami vytvorené funkcie budeme písať pred funkciu `main`. Tieto funkcie potom budú vo funkcii `main` volané.

4.2.1 Definícia funkcie

Každá funkcia má meno, zvyčajne jeden alebo niekoľko vstupných parametrov (ale nemusí byť žiadny) a jej výsledkom je nejaká hodnota. Pri definícii funkcie definujeme hlavičku funkcie (záhlavie) a jej telo. Hlavička funkcie určuje jej meno, typ návratovej hodnoty a počet a typ jej parametrov. Funkcie v C sú skutočné funkcie, teda vracajú hodnotu (aj keď je

[*)] Pozri dodatok

možnosť použiť ich ako procedúry, t.j. žiadnu hodnotu nevrátia, „iba“ vykonajú nejakú činnosť).

Príklad 6

```
int Abs(int a)
{
  if (a<0)
     a=-a;
  return a; }
```

Táto funkcia vráti absolútnu hodnotu celého čísla. Jej vstupom je 1 celočíselný parameter a jej výstup – výsledok je typu `int`. Telo funkcie tvorí jeden podmienený príkaz. Príkaz `return` predstavuje mechanizmus, pomocou ktorého volaná funkcia vracia hodnotu volajúcej funkcii.

Pri definícii sa za záhlavím funkcie nepíše bodkočiarka!!!

Všeobecne by sme mohli funkciu s dvoma parametrami charakterizovať:

```
návratový_typ meno_funkcie (typ1 meno_parametra1, typ2 meno_parametra2)
   {
    ...
   return návratová_hodnota;
   }
```

Ak neuvedieme návratový typ, návratová hodnota je implicitne (automaticky) typu `int`.

Príklad 7

Napíšme funkciu, ktorá umocní celé číslo `x` na *n*-tú. Všimnite si, že funkcia nemá uvedený návratový typ - ten je automaticky celočíselný.

```
power(int x,int n)
{
  int p;
  for (p=1;n>0;n--)
       p=p*x;
  return p;
}
```

Pri funkciách s iným návratovým typom ako `int` nie je možné označenie typu vynechať, lebo inak by bola návratová hodnota na typ `int` konvertovaná!!!

Funkcia, ktorá nemá žiadne parametre, musí byť definovaná aj volaná včítane oboch guľatých zátvoriek, napr. `funkcia1()`, a nie `funkcia1`!!!

Parametre funkcie, ale aj akékoľvek premenné definované nám doteraz známym spôsobom vo funkcii sú známe iba vo vnútri funkcie. Po vykonaní funkcie sa takto zadefinované hodnoty neuchovávajú, a teda sa nemôžeme spoliehať na to, že po opätovnom volaní budú hodnoty premenných také, aké boli po predchádzajúcom volaní funkcie. Na takéto premenné sa nemôžeme odvolávať z inej funkcie alebo hlavného programu. Parametre funkcie v jej definícii voláme **formálne** parametre.

4.2.2 Volanie funkcie

Funkciu voláme tak, že uvedieme jej meno a namiesto formálnych parametrov uvedieme skutočné (teda konkrétne hodnoty, priamo čísla alebo hodnoty uložené v premenných. Poradie formálnych a skutočných parametrov si musí zodpovedať. Návratovú hodnotu priradíme do premennej, prípadne použijeme tam, kde je očakávaná hodnota. Príklad volania funkcie z príkladu 7.

```
...
printf("štvrtá mocnina z troch je %d \n",power(3,4));
```

Príklad 8 – pole ako argument funkcie

Napíšme program, ktorý nájde minimum desať prvkového poľa.
Všimnite si, že v nasledujúcom príkaze je pole inicializované vymenovaním jeho prvkov.

```
int minimum(int a[10])
{int i,index_min=0;
for (i=1;i<10;i++)
if(a[i]< a[index_min]) index_min=i;
return a[index_min]; //pripadne index_min
}

main()
{
int i,minimum,pole[10]={3,2, 5, 7, -1,45,23,4,12,0};
printf("minimum:%d\n",minimum(pole));
}
```

Funkciu najprv zadefinujeme a potom ju voláme v hlavnom programe. Volanie funkcie je zvýraznené farebne. Pri volaní funkcie sa vykoná táto postupnosť krokov:

- prenesú sa parametre: vo funkcii minimum kompilátor vie (podľa hranatých zátvoriek), že na vstup príde adresa v počítači, kde pole začína. Adresa poľa `pole` sa skopíruje do `a`. pomocou adresy v pamäti, kde pole začína a pomocou indexov sa robia posuny od začiatku poľa. Ak `pole` a `a` predstavujú tú istú adresu, tak pracujeme s tým istým poľom, teda v tomto prípade s poľom definovaným v hlavnom programe.
- uchová sa návratová adresa, t.j. adresa inštrukcie za volaním funkcie,
- vykoná sa skok do funkcie `minimum`, jej kód sa vykoná,
- vykonávanie programu sa vráti z funkcie do miesta volania (na návratovú adresu) a zrušia sa použité parametre, pričom sa návratová hodnota priradí do premennej alebo sa použije na mieste, kde sa očakáva hodnota.

V jazyku C **sa všetky argumenty funkcie odovzdávajú hodnotou**. To znamená, že volanej funkcii sa neodovzdávajú adresy premenných použitých ako argumenty, ale (iba) ich hodnoty sa skopírujú do dočasných premenných toho istého typu (uložených v zásobníku). V jazyku C volaná funkcia nemôže zmeniť premennú vo volajúcej funkcii; zmeniť môže iba svoju súkromnú dočasnú premennú. (Aj to sa však dá dosiahnuť, ak použijeme techniku smerníkov vysvetlenú neskôr.) V prípade poľa sa však používa práve táto technika, tým, že sa prenáša adresa , a teda na rozdiel od obyčajnej premennej pole pomocou procedúry môžeme zmeniť - nerobí sa s jeho kópiou (iba s kópiou jeho počiatočnej adresy).

Častou chybou, ktorú neodhalí kompilátor (niektoré však vypíšu varovanie) je volanie funkcie s argumentom poľa ako `minimum(pole[10])`!!! Siahne sa mimo poľa, lebo najvyšší index v poli je o jedna menší ako počet jeho prvkov, a toto sa bude chápať ako adresa, od ktorej sa bude indexovať. Program alebo padne, ak adresa spadá

do oblastí zakázaných adries (to je ten lepší prípad), alebo sa poprepisuje pamäť a program sa správa veľmi záhadne. Stále treba mať na pamäti, že v prípade poľa prenášame adresu.

4.2.3 Procedúry

Procedúra je nejaká činnosť, ktorej výsledkom nie je číselná hodnota. Formálne v C procedúry neexistujú, ale dá sa to obísť. Funkcia sa zadefinuje ako funkcia vracajúca dátový typ `void („prázdny“)`. Príkaz `return` potom nie je nutný, snáď len v prípade núteného ukončenia funkcie pred dosiahnutím jej konca, napr. po nejakej podmienke.

Príklad 9

Máme zadefinované a inicializované pole desiatich prvkov. Najprv ho dáme vypísať pomocou procedúry. Potom pomocou funckie dáme vypočítať jeho priemer a vypíšeme ho.

```
#include <stdio.h>

double priemer(int x[]){
double pr;
int i;
pr=0;
for (i=0;i<10;i++)
     pr+=x[i];
printf("suma:%lf\n",pr);
return pr/10;
}

void vypis_pola(int pole[]){
int i;
for (i=0;i<10;i++)
     printf("%4d",pole[i]);
printf("\n");
}
/*-------------------------------------------------*/
main(){
double x;
int i,j,n,pole[10]={3,2, 5, 7, -1,45,23,4,12,0};
printf("vypis pola:\n");
vypis_pola(pole);
x=priemer(pole);
printf("priemer:%lf\n",x);
}
```

V tomto príklade si všimnite niekoľko vecí. V prvom rade ako sa líši definícia a volanie funkcie a procedúry. Ďalej ako sa v definícii funkcie definuje parameter, ktorý predstavuje pole. Zadáva sa typ poľa, jeho názov a prázdne hranaté zátvorky. Rozmer jednorozmerného poľa nie je potrebné uvádzať. Na mene parametra nezáleží. Môže a nemusí sa volať ako skutočné pole (to platí pre všetky parametre, nielen typu pole). **V prípade, že sa ako argument použije meno poľa,** pri volaní hodnotou odovzdávanou funkcii je umiestnenie, čiže adresa začiatku poľa. (Prvky poľa sa nekopírujú). Indexovaním tejto hodnoty má funkcia prístupný ľubovoľný prvok poľa a môže ho teda zmeniť. Porovnajte s prípadom, keď sa odovzdá jeden prvok poľa.

Cvičenie 1
Pozmeňte program tak, aby sa prvky poľa nezískavali inicializáciou, ale aby sa načítavanie prvkov poľa vykonávalo pomocou klávesnice a aby ho zabezpečila procedúra.

4.2.4 Rekurzívne funkcie

Rekurzia je metóda ako rozdeliť problém na podproblémy rovnakého typu. Takmer všetky programovacie jazyky dovoľujú použitie rekurzívnych funkcií, t.j. funkcií, ktoré môžu volať sami seba. Funkcie v C vyzerajú rovnako ako všetky ostatné funkcie.

Typickým príkladom rekurzívne definovanej funkcie je nasledujúca definícia faktoriálovej funkcie *f*(*n*):

f(0) = 1
f(n) = n · f(n-1) pre nejaké prirodzené číslo *n > 0*.

Napr. vypočítajme *f*(*3*) takto:
f(3) = 3 · *f*(3-1) = 3 · *f*(2) = 3 · 2 · *f*(2-1) = 3 · 2 · *f*(1) = 3 · 2 · 1 · *f*(1-1) = 3 · 2 · 1 · *f*(0) = 3 · 2 · 1 · 1 = 6.

Príklad 10
S použitím rekurzie napíšte program pre výpočet faktoriálu.

```
int fakt(int n) {
if (n==0)
      return 1;
else
return n*fakt(n-1); }
main(){
printf("5!=%d\n",fakt(5));
}
```

Výsledok je 120.

Cvičenie 2
Zistite, pre aké maximálne číslo viete vypočítať hodnotu faktoriálu, ak použijete pre premennú `n` typ `int` a `long double`.

Príklad 11
Napíšte program na výpočet faktoriálu bez použitia rekurzie, pomocou cyklu `for`.

```
main(){
int i,f=1,n=5;
for (i=2;i<=n;i++)
       f=f*i;
printf("5! =%d\n",f); }
```

Funkcie si zvyknite starostlivo dokumentovať a archivovať. Medzi dokončením funkcie a jej opätovným použitím môže uplynúť veľa času.!!!

☑ **Kontrolný test**

1. Pole definované príkazom `double x[n],` kde `n` je konštanta, sa nazýva:
   a. dynamické
   b. konštantné

c. statické
d. stále.

2. Majme pole `double x[100]`. Vyberte nesprávne tvrdenia.
   a. prvky poľa sú reálne premenné typu `double`
   b. pole má 100 prvkov
   c. minimálny index poľa je 1
   d. maximálny index poľa je 100.

3. Majme pole `int a[k]`. Vyberte nesprávne tvrdenia. V ANSI C
   a. takéto pole sa nazýva statické
   b. takéto pole sa nazýva dynamické
   c. `k` musí byť konštanta
   d. `k` môže byť premenná.

4. Majme príkazy:

```
int a[10],i;
for(i = 0; i<= 10; i++);
    a[i]=i;
```

Vyberte nesprávne tvrdenia:
   a. tento kód zapríčiní práve to, že priradíme prvkom poľa čísla od 0 do 10
   b. tento kód môže zapríčiniť nevyspytateľnú chybu
   c. prekladač vyhlási počas prekladu chybu, lebo chceme priradiť do `a[10]` a maximálny index je 9
   d. prekladač vyhlási počas prekladu chybu, lebo chceme priradiť do `a[0]` a minimálny index je 1.

5. Majme 5-prvkové pole `a` definované príkazom `int a[5] = {1,2,3,4,5}`. Výpis 5, 4, 3, 2, 1 sa nedosiahne príkazom/príkazmi
   a. `for(i = 5; i>0; i--) printf("%d,",a[i]);`
   b. `for(i = 4; i>-1; i--) printf("%d,",a[i]);`
   c. `for(i = 4; i>=0; i--) printf("%d,",a[i]);`
   d. `for(i = 0; i<=4; i++) printf("%d,",a[4-i]);`

6. Majme pole definované príkazom `int a[2][3]={{1,2,3}{10,11,12}};` Vykonajme príkazy:

```
suma=0;
for(i = 0; i<= 3; i++)
   for(j = 0; j< 2; j++)
         suma+=a[i][j];
```

Vyberte správne tvrdenia:
   a. výsledkom bude `suma = 39`
   b. nevieme povedať, čo sa stane
   c. výsledkom bude `suma = 24`
   d. kompilátor vyhlási chybu.

7. Majme pole definované príkazom `int a[3][3];` Jeho hodnoty sa budú načítavať z klávesnice v cykle

```
for(j = 0; j< 3; j++)
   for(i = 0; i< 3; i++)
      scanf(("%d",&a[i][j]);
```

a z klávesnice budeme zadávať hodnoty v poradí: 1 2 3 14 5 6 7 8 9. Výsledkom príkazov

```
suma=0;
for(i = 0; i< 3; i++)
     suma+=a[1][i];
```

bude

a. `suma = 25`
b. `suma = 15`
c. `suma = 18`
d. `suma = 6.`

8. Vyberte správne tvrdenia. V ANSI C
   a. funkcia vždy musí mať aspoň jeden parameter
   b. vždy treba uviesť návratový typ funkcie, inak kompilátor vyhlási chybu
   c. funkcia môže byť definovaná v inej funkcii
   d. hodnotu funkcie vrátime pomocou príkazu `return`.

9. Funkcia, ktorá môže volať sama seba sa nazýva:
   a. rekurentná funkcia
   b. rekurzívna funkcia
   c. dynamická funkcia
   d. procedúra.

10. Vyberte správne tvrdenia:
   a. procedúra vždy vráti hodnotu
   b. návratový typ procedúry je `void`
   c. procedúra musí mať vždy aspoň jeden parameter
   d. záhlavie `void vypis(int a[])` je syntakticky nesprávne.

11. Vyberte správne tvrdenia: príkaz `#define n 10`
   a. zadefinuje celočíselnú premennú `n` a jej hodnotu nastaví na 10
   b. zadefinuje celočíselnú konštantu `n` a jej hodnotu nastaví na 10
   c. `n` môžeme použiť pri definícii statického poľa
   d. hodnota `n` sa nastaví počas behu programu.

Kontrolné otázky

1. Prečo môže byť výhodnejšie definovať rozsah statického poľa symbolickou konštantou a nie číslom?
2. Majme pole `int a[2][4].` Ak si toto pole predstavíme ako jednorozmerné, koľký v poradí od počiatku bude prvok s indexom `a[2][2]` ?
3. Majme pole `int a[5][4].` Aký index bude mať prvok, ktorý je uložený ako 10. v poradí od začiatku poľa?
4. Čo sú to “chyby+1“ ?
5. Vysvetlite, prečo sa používajú funkcie.
6. Vysvetlite, aký je rozdiel medzi funkciou a procedúrou.
7. Akým príkazom vrátime riadenie a vypočítanú hodnotu na miesto volania funkcie?
8. Môžu byť funkcie do seba vnárané? Čo to znamená?
9. Aký je návratový typ funkcie, pokiaľ ho neuvedieme pri definícii?

10. Aká je syntax, keď ako argument funkcie používame meno poľa? Charakterizujte rozdiel v organizácii prenášania parametrov v prípade, že je ním celočíselná premenná a v prípade, že je ním celočíselné pole.
11. Charakterizujte princíp rekurzie. Aká je to rekurzívna funkcia?

Cvičenia

1. Načítajte do poľa desať celých čísel a nájdite ich minimum a maximum.
2. Načítajte do poľa desať celých čísel a usporiadajte ich.
3. Načítajte do poľa *n* celých čísel, kde *n* môže byť párne aj nepárne a nájdite medián.
4. Načítajte dva *n*-rozmerné vektory reálnych čísel a vypočítajte ich skalárny súčin.
5. Pomocou vektorové súčinu vypočítajte plochu trojuholníka zadaného súradnicami jeho vrcholov.
6. Načítajte do poľa *n* reálnych čísel a vypočítajte harmonický priemer. Vypočítajte harmonický priemer, ak sú dáta dané tabuľkou početností.
7. Načítajte maticu 3x3 celých čísel a Sarusovým pravidlom spočítajte determinant matice.
8. Načítajte do vhodnej matice (vhodných matíc) sústavu troch rovníc o troch neznámych. Zistite, či sa dá riešiť Cramerovým pravidlom a ak áno, vyriešte ju.
9. Načítajte 10 čísel zo vstupu. Vypíšte ich v opačnom poradí, ako ste ich načítali.
10. Napíšte funkciu, ktorej argumentom bude pole celých čísel a funkcia z nich vráti maximum (minimum, súčet, priemer, usporiada ho).
11. Napíšte funkciu, ktorej argumentom bude trojrozmerný vektor a funkcia vráti jeho dĺžku.
12. Napíšte funkciu, ktorej argumentom budú dva trojrozmerné vektory reálnych čísel a funkcia z nich vráti dĺžku vektorového súčinu.
13. Napíšte funkciu, ktorej argumentom budú dva *n*-rozmerné vektory reálnych čísel a funkcia z nich vráti skalárny súčin.
14. Napíšte funkciu, ktorej argumentom bude matica 3x3 celých čísel a ktorá vráti hodnotu jej determinantu.
15. Napíšte program, ktorý načíta maticu, v pamäti vytvorí k nej transponovanú a vypíše ju.
16. `N`ech n je definované ako konštanta. Majme `n` - rozmerný vektor `x` s `i` -tou zložkou $\left(1+\frac{1}{i}\right)^i$. Pri jeho vytváraní použite matematickú funkciu `pow`. Vytvorte vektor `y` dĺžky `n`, toho istého typu, pre ktorý platí:

    a) $y(1)=x(n), y(2)=x(n-1). \ldots . y(n)=x(1)$
    b) $y(1)=x(1), y(n)=x(n), y(i)=\frac{x(i-1)+x(i+1)}{2}$
    c) $y(1)=0, y(n)=0, y(i)=\frac{1}{4}x(i-1)+\frac{1}{2}x(i)+\frac{1}{4}x(i+1)$
    d) vypočítajte súčet zložiek vektora
    e) vypočítajte súčin zložiek vektora
    f) vypočítajte priemer prvkov poľa
    g) vypíšte indexy tých prvkov poľa, ktoré sú väčšie ako priemer.
17. Zadefinujte 10-zložkový vektor s položkami 3, 9, 5, 4, 7, 7, 3, 8, 9, 6.
    a) nájdite minimum a vypíšte index jeho "prvého výskytu"
    b) nájdite maximum a vypíšte index jeho "posledného výskytu"
    c) nájdite prvé párne číslo a vypíšte jeho index
    d) nájdite posledné párne číslo a vypíšte jeho index

e) vypíšte všetky prvky medzi prvým a posledným párnym číslom
f) spočítajte, koľko prvkov je nepárnych
g) vypíšte indexy tých prvkov polí, ktoré sú nepárne
h) vypíšte, koľko prvkov sa rovná číslu 7
i) vypíšte index prvého prvku z takej dvojice susedov, ktorí obsahujú tie isté hodnoty
j) zistite, či existuje taká dvojica, že jeden prvok je násobkom druhého, vypíšte index
k) zadajte zo vstupu číslo a zistite, či vektor dané číslo obsahuje.

18. Dané sú 3 body v priestore. Vypočítajte koeficienty *a ,b ,c ,d* roviny $ax+ by + cz + d = 0$, ktorá prechádza danými tromi bodmi. V rovine zvoľte 2 vektory, pomocou vektorového súčinu vytvorte normálový a dopočítajte *d*.

19. Zadefinujte 5-prvkové pole, do ktorého zapíšeme polynóm štvrtého stupňa. Vytvorte program, ktorý do poľa rovnakej dĺžky vytvorí deriváciu tohto polynómu. Napíšte program, ktorý vypočíta koeficienty *k* a *q* dotyčnice k danému polynómu v bode, pre ktorý zadáte *x*-ovú súradnicu.

20. Nech *n* je definované ako konštanta. Z klávesnice načítajte reálne číslo blízke nule a priraďte ho do premennej *x*. Vytvorte *n* - rozmerný vektor *u* s i -tou zložkou rovnou $(-1)^i \frac{x^{2i+1}}{(2i+1)!}$. Napíšte algoritmus, ktorý zostrojí vektor *v* s $v(i)=\sum_{j=1}^{i} u(j)$ a vektor *w* s $w(i)=abs(v(i)-w(i))$.

Zadanie

1. Fibonacciho postupnosť je nasledovná rekurzívne definovaná postupnosť:
   $F(0) = 0$
   $F(1) = 1$
   $F(n + 2) = F(n) + F(n + 1)$ *pre* $n > 1$.

Napíšte rekurzívny program pre výpočet *n*-tého Fibonacciho čísla. V postupnosti Fibonacciho čísel F(10) až F(20) dajte vypísať podiely dvoch po sebe idúcich členov.

Podobne ako v príklade 11 skúšajte, aký veľké číslo Vám dovolí použiť celočíselný typ. Pri práci s veľkými číslami buďte opatrní a radšej použite typ `float` alebo `double`.

5 Zložený dátový typ - štruktúra

Hlavné podtémy kapitoly:

- Zložený dátový typ - štruktúra. Charakteristika a význam.
- Spôsoby definovania štruktúry a jej prislúchajúcemu typu.
- Prístup k jednotlivým prvkom štruktúry. Polia štruktúr.
- Vzorový príklad na pole štruktúr – triangulácia danej oblasti.
- Generovanie pseudonáhodných čísel v určitom intervale. Meranie dĺžky trvania programu.

5.1 Zložený dátový typ

5.1.1 Štruktúra a jej charakteristika

Pole je postupnosť za sebou idúcich premenných **rovnakého typu. Zložený dátový typ** (v K&R nazývaný štruktúra, v Pascale záznam) je súhrn jednej alebo viacerých premenných, i **rôzneho typu**, ktoré sú kvôli výhodnej manipulácii združené pod jediným menom.

Tradičným príkladom štruktúry je *mzdový záznam*. Zamestnanec je popísaný množinou atribútov ako je meno, adresa, číslo poistky, mzda, pohlavie, stav atď. Niektoré z atribútov môžu byť opäť štruktúry: meno má viacero zložiek, takisto adresa aj mzda.

V matematike by ako príklad mohla poslúžiť štruktúra *komplexné číslo*, ktorá sa skladá z dvoch zložiek, reálnej a imaginárnej. Z troch zložiek: deň, mesiac a rok by sa mohla skladať štruktúra *dátum*.

Štruktúry pomáhajú organizovať zložité dáta, najmä v rozsiahlych programoch, pretože v mnohých prípadoch umožňujú zaobchádzať so skupinou súvisiacich premenných ako s celkom a nie ako so samostatnými objektmi.

5.1.2 Spôsoby definovania štruktúry

V ďalšom texte opíšeme ako môžeme definovať štruktúru. Uvedieme niekoľko spôsobov ako sa dá zadefinovať štruktúra a jej prislúchajúci typ a ako sa dajú definovať premenné daného typu. Čitateľ si samozrejme môže zapamätať a používať jeden spôsob: všetky uvádzame preto, kedy súčasne so štruktúrou definujeme aj nový dátový typ a aby sme boli schopní čítať programy po druhých, ktorý môžu použiť iný spôsob. Kvôli prehľadnosti ich všetky uvádzame na jedno miesto.

Predstavme si štruktúru, ktorá bude predstavovať kružnicu a bude mať 3 zložky: *x*-ovú a *y*-ovú súradnicu stredu a veľkosť polomeru. Chceme mať 3 premenné typu štruktúra: `kružnica1, kružnica2` a `kružnica3.`

1. spôsob: vytvorená štruktúra nie je pomenovaná a nedá sa v programe nikde ďalej využiť. Dajú sa využívať definované premenné `kružnica1, kružnica2` a `kružnica3.`

```
struct {
  double x;
  double y;
  float r;
 } kružnica1,kružnica2,kružnica3;
```

2. spôsob: štruktúru pomenujeme `kružnica` a dá sa teda využiť ďalej v programe.

```
struct kružnica{
      double x;
      double y;
      float r;
     } kružnica1, kružnica2, kružnica3;
```

3. spôsob: je to vlastne predchádzajúci spôsob, ale definícia štruktúry je oddelená od definície premenných. Definície premenných sa môžu robiť aj viackrát, čo nebolo možné v prvom prípade.

```
struct kružnica{
      double x;
      double y;
      float r;
     };
struct kružnica kružnica1, kružnica2;
struct kružnica kružnica3;
```

Na `struct` nemožno zabudnúť, je to častá chyba.

4. spôsob: definícia nového typu – štruktúry pomocou príkazu **`typedef`**. Pomocou operátora **`typedef`** sa dá vytvoriť nový dátový typ. Nie je to definícia premennej, ktorá prideľuje pamäť, ale definícia nového typu, ktorá iba určuje vzorec (šablónu) pre ďalšie akcie. Pri tomto spôsobe definície štruktúry táto nie je pomenovaná, ale je pomenovaný nový typ a dá sa teda ľubovoľne používať, napr. pre definície premenných, pretypovanie apod.

```
typedef struct{
      double x;
      double y;
      float r;
     }KRUŽNICA;
KRUŽNICA kružnica1, kružnica2;
KRUŽNICA kružnica3;
```

Pri definícii premenných už nie je použité slovo `struct`.

5. spôsob: je to modifikácia predchádzajúceho spôsobu, kde pomenujeme aj štruktúru aj nový typ. Pre tento príklad nemá tento spôsob opodstatnenie, ale sú prípady, keď opodstatnenie má.

```
typedef struct kružnica{
      double x;
      double y;
      float r;
     }KRUŽNICA;
KRUŽNICA kružnica1, kružnica2;
KRUŽNICA kružnica3;
```

Odporúča sa pomenovať nový typ aj štruktúru rovnako, rozlíšiť ich iba veľkosťou písmen.

Prax ukazuje, že najvýhodnejšie sú posledné dva spôsoby.

5.1.3 Prístup k prvkom štruktúry

K prvkom štruktúry sa pristupuje pomocou bodky.

Napr. `kružnica1.x, kružnica1.y, kružnica3.r.`

Príklad 1

Zadefinujme štruktúru pre komplexné číslo, tri premenné `k1`, `k2` a `k3` danej štruktúry a ukážme, ako by sme `k1` a `k2` sčítali do `k3`. Pre definovanie štruktúry si vyberme spôsob číslo 4.

Riešenie:

```
…
typedef struct{
      double re;
      double im;
      }KOMPLEX;
KOMPLEX k1, k2, k3;
…
k3.re = k1.re + k2.re;
k3.im = k1.im + k2.im;
…
```

Pomocou jedného priraďovacieho príkazu je možné priradiť obsah jednej štruktúry do druhej, napr. `kružnica1 = kružnica2.`

Príklad 2

Presvedčte sa, že priraďovací príkaz `a = b` v nasledujúcom príklade zapríčiní výpis chyby „nekompatibilné priradenie“.

```
main()
{
int a[10],b[10];
a = b;
}
```

Ak však použijeme „trik“ z nasledujúceho príkladu, môžeme s poľom pracovať ako s celkom.

```
main()
{
typedef struct {
        int pole[10];
        }POLE;

POLE a,b;
a = b;
}
```

Príklad 3

Vychádzajme z predchádzajúceho príkladu. Vynulujme prvky poľa `a` a priraďme hodnoty poľa `a` poľu `b`. Obsah poľa `b` dajme vypísať.

```
main()
{int i;
typedef struct {
        int p[10];
        }POLE;
POLE a,b;

for (i=0; i<10; i++)
     a.p[i]=0;
b = a;
for (i=0; i<10; i++)
     printf("%d\n",b.p[i]);
}
```

5.2 Pole štruktúr

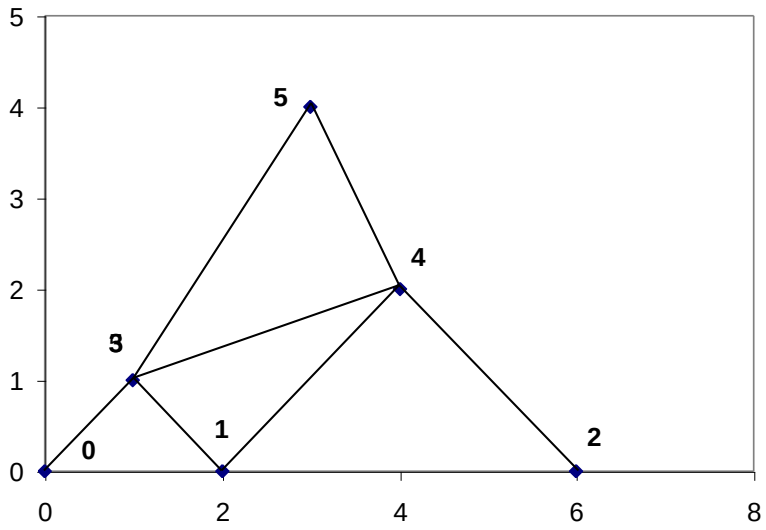
Jedna štruktúra môže byť podčasťou inej štruktúry. Prvky typu štruktúra môžu tvoriť pole.

5.2.1 Vzorový príklad – triangulácia oblasti

Pri mnohých výpočtových problémoch sa stretávame s triangulániami oblastí. Budeme mať danú oblasť, ktorú budeme musieť „poskladať“ z trojuholníkov: trojuholníky musia oblasť pokryť, ale nemôžu sa pretínať. Nasledujúci príklad bude príkladom jednoduchej triangulácie v rovine. Podobne by však mohla byť zorganizovaná aj triangulácia v priestore.

Príklad 4

Triangluláciu v rovine môžeme mať zorganizovanú napríklad tak, že množinu bodov, ktoré budú tvoriť vrcholy trojuholníkov, si uložíme do poľa. Jeden bod bude tvoriť štruktúru BOD.



Pre jeden trojuholník zostrojíme štruktúru TROJ, ktorá bude pozostávať z trojice vrcholov. Tie však nebudú predstavovať skutočné súradnice bodov, ale príslušné indexy do poľa bodov. Navyše štruktúra TROJ bude obsahovať položku pre plochu trojuholníka, ktorú bude treba vypočítať. Keby sme mali trianguláciu v priestore, ďalšou rozumnou položkou by mohol byť napr. vektor normály na trojuholník. Obrázok zobrazuje množinu šiestich vrcholov a štyroch trojuholníkov, ktoré sú nimi preložené.

Uvedomme si, že z hľadiska množstva použitej pamäte je tento spôsob výhodnejší ako keby sme vrcholy trojuholníka reprezentovali priamo súradnicami bodov (ešte väčšia úspora by vznikla v trojrozmernom priestore). Navyše, ak trochu posunieme súradnicu niektorého bodu, okrem samotných súradníc nič viac meniť nemusíme. V prípade, že by sme používali priamo súradnice vrcholov, museli by sme meniť viacero vrcholov, v závislosti od triangulácie.

Plocha trojuholníka sa počíta pomocou vektorového súčinu. Vzorec si najprv odvoďte. Trojicu vrcholov vnorte do priestoru, a to do roviny XY tak, že bodom priradíte z-ovú súradnicu nastavenú na nulu. Bodmi preložíme dva rôznobežné vektory a vypočítame vektorový súčin. V prípade nulových z-súradníc je výsledný vzorec veľmi jednoduchý.

Riešenie:

```
/*uloha - pre kazdy z trojuholnikov vypocitajte plochu*/
#include <math.h>
#define m 6
#define n 4

/*definicia typov a premennych*/
typedef struct {
 double  x;
 double  y;
} BOD;

typedef struct {
 int V1;
 int V2;
 int V3;
 double plocha;
} TROJ;

BOD vrcholy[m];
TROJ triangulacia[n];

/*definicia funkcii*/
double plocha(TROJ T)
{double ax,ay,bx,by;
 ax=vrcholy[T.V1].x-vrcholy[T.V2].x;
 ay=vrcholy[T.V1].y-vrcholy[T.V2].y;
 bx=vrcholy[T.V2].x-vrcholy[T.V3].x;
 by=vrcholy[T.V2].y-vrcholy[T.V3].y;
 return fabs(ax*by-ay*bx)/2.;
}

/*_____________________hlavny program_____________________________*/
main(){
int i;
/*nastavenie bodov */
vrcholy[0].x=0;vrcholy[0].y=0;
```

```
vrcholy[1].x=2;vrcholy[1].y=0;
vrcholy[2].x=6;vrcholy[2].y=0;
vrcholy[3].x=1;vrcholy[3].y=1;
vrcholy[4].x=4;vrcholy[4].y=2;
vrcholy[5].x=3;vrcholy[5].y=4;

/*zadefinovanie trojuholnikov*/
triangulacia[0].V1= 0; triangulacia[0].V2= 1; triangulacia[0].V3= 3;
triangulacia[1].V1= 3; triangulacia[1].V2= 1; triangulacia[1].V3= 4;
triangulacia[2].V1= 1; triangulacia[2].V2= 2; triangulacia[2].V3= 4;
triangulacia[3].V1= 3; triangulacia[3].V2= 4; triangulacia[3].V3= 5;

for (i=0; i<n; i++)
   {
    triangulacia[i].plocha = plocha(triangulacia[i]);
    printf("plocha %d -iteho trojuholnika je:%lf \n", i,
           triangulacia[i].plocha);
   }
}
```

5.3 Generátor pseudonáhodných čísel

Budeme používať dve funkcie: `rand` a `srand`. Funkcia `srand` inicializuje generátor náhodných čísel `rand`. Aby sme ich mohli použiť, potrebujeme vložiť hlavičkový súbor `stdlib.h`. Navyše si ešte pomôžeme funkciou `time`, pre ktorú treba vložiť hlavičkový súbor `time.h`.

1. `int rand(void)` je funkčný prototyp prvej funkcie. Volá sa bez parametrov a vracia celé číslo v rozsahu `0` až `RAND_MAX`, kde `RAND_MAX` je konštanta definovaná v hlavičkovom súbore `stdlib.h`, v nami používanej verzii C má hodnotu `2147483647`. Ak potrebujeme náhodné čísla v určitom rozsahu, použijeme operáciu *modulo.*
   Napr. `rand() % 20` vracia čísla od 0 do 19.

   Ak potrebujeme náhodné čísla v rozsahu <0,1), môžeme s výhodou použiť konštantu `RAND_MAX`, napríklad
   `x=rand()/(double) (RAND_MAX+1)`

2. `void srand(unsigned int start)` je funkčný prototyp druhej funkcie. Táto funkcia inicializuje generátor `rand()` počiatočnou hodnotou `start`. Ak je `start` konštanta, generátor generuje pri každom spustení rovnakú postupnosť. Ak potrebujeme naozaj náhodné čísla, musí byť aj hodnota `start` náhodná. Obyčajne pre ňu používame nejaký časový údaj. Budeme používať funkciu `time`.

3. volanie `time(NULL)` vráti počet sekúnd od 1. januára 1970.

Príklad 5

Do celočíselného poľa veľkosti 10 vygenerujte náhodné čísla v rozmedzí:

a) 1 až 100,
b) 0 až 100,
c) 20 až 30,
d) -10 až 10.

Riešenie:

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

main()
{
int i,start, nah[10];
srand ((unsigned int ) time(NULL));

for (i=0;i<10;i++)
  {
  nah[i]=rand()%100 + 1; /* pripad a)*/
  printf(„%d \n",nah[i]);
  }
}
```

V ostatných prípadoch nahradíme vyznačený príkaz takto:

b) `nah[i] = rand()%101;`
c) `nah[i] = 20 + rand()%11;`
d) `nah[i] = -10 + rand()%21;`

5.4 Meranie dĺžky trvania programu

V hlavičkovom súbore `<time.h>` sú popísané dátové typy a funkcie slúžiace pre prácu s časom a dátumom.

`clock_t`	- je dátový typ (4-bytové celé znamienkové číslo) slúžiace pre funkciu `clock()`.
`clock()`	- je počet tikov procesora od spustenia programu. Jej návratový typ je `clock_t`.
`CLOCKS_PER_SEC`	-je symbolická konštanta, vyjadrujúca počet tikov procesora za sekundu. Používame ju k prevodu tikov na sekundy.

Príklad 6
Porovnajte časovú náročnosť výpočtu *n*-tého Fibonacciho čísla rekurzívne a iteračne.

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

unsigned int  fib_rek(int x)
{
  if (x<2)
    return 1;
  else
    return fib_rek(x-1) + fib_rek(x-2);
}
```

```
unsigned int fib_iter(int x)
{
  int i;
  unsigned int f1,f2,f3;
   f1 = 1;
  f2 = 1;
  for(i=2; i<=x; i++)
    {f3 = f2 + f1;
     f1 = f2;
     f2 = f3;
    }
return f3;
}

int main()
{
  unsigned int f;
  clock_t zac, koniec;
zac=clock();
f=fib_rek(45);
koniec=clock();
printf("Cislo:%u trvanie:%lf [sec]\n",
       f,(koniec-zac)/(double)CLOCKS_PER_SEC);

zac = clock();
f = fib_iter(45);
koniec = clock();
printf("Cislo:%u trvanie:%lf [sec]\n",
       f,(koniec-zac)/(double)CLOCKS_PER_SEC);
}
```

☑ Kontrolný test

1. Vyberte správne tvrdenia:
   a. v štruktúre môžu byť len prvky rovnakého typu
   b. v štruktúre musia byť prvky rôzneho typu
   c. v štruktúre môžu byť prvky rovnakého i rôzneho typu
   d. prvky štruktúry sú kvôli výhodnej manipulácii združené pod rovnakým menom.

2. Chceme vytvoriť štruktúru obsahujúcu prvky *výška* a *hmotnosť*. Príkaz

```
struct osoba{
     float hmotnosť;
     float výška; }
```

   a. zadefinuje premennú osoba obsahujúcu položky hmotnosť a výška
   b. zadefinuje nový dátový typ nazvaný osoba

c. zadefinuje štruktúru nazvanú osoba. Premennú tohto typu zadefinujeme príkazom `osoba Jano, Peter.`
d. zadefinuje štruktúru nazvanú osoba. Premennú tohoto typu zadefinujeme príkazom `struct osoba Jano, Peter.`

3. Chceme vytvoriť štruktúru obsahujúcu prvky *výška* a *hmotnosť*. Príkaz

```
typedef struct osoba{
     float hmotnosť;
     float výška } OSOBA;
```

a. zadefinuje premennú osoba obsahujúcu položky hmotnosť a výška
b. zadefinuje nový dátový typ nazvaný osoba
c. zadefinuje štruktúru nazvanú `osoba` a typ `OSOBA`. Premennú tohto typu zadefinujeme príkazom `osoba Jano, Peter`
d. zadefinuje štruktúru nazvanú `osoba` a typ `OSOBA`. Premennú tohto typu zadefinujeme príkazom `OSOBA Jano, Peter.`

4. Majme premenné definované príkazom

```
struct{
     float hmotnosť;
     float výška;} Jano,Peter,Kata;
```

Vyberte syntakticky nesprávne príkazy (t.j. kde kompilátor hlási chybu):

a. `Jano = Peter;`
b. `Jano.hmotnosť = Peter.hmotnosť;`
c. `Kata.hmotnosť = Jano.výška;`
d. `Jano=Peter.výška;`

5. Majme tieto definície:

```
typedef struct {
int p[10];
}POLE;
POLE a,b;
```

Vyberte nesprávne tvrdenia. Príkaz `a = b`

a. vyhlási chybu nekompatibilné priradenie
b. priradí í-temu prvku poľa `a` í-ty prvok poľa `b` pre `i` = 0...9
c. adresa poľa `a` sa prepíše adresou poľa `b`, t.j. pole `a` prestane existovať
d. prvky poľa `b` môžeme priradiť prvkom poľa `a` jedine v cykle.

6. Majme tieto definície:

```
typedef struct {
int p[10];
}POLE;
POLE a,b;
```

Piaty prvok poľa `b` priradíme piatemu prvku poľa `a` príkazom:

a. `a[5] = b[5];`
b. `a[4] = b[4];`
c. `a.p[5] = b.p[5];`
d. `a.p[4] = b.p[4];`

7. Majme definíciu:

```
typedef struct osoba{
     float hmotnosť;
     float výška; } OSOBA;
```

Počet bytov, ktoré štruktúra zaberá, zistíme príkazom:

a. `sizeof(osoba)`
b. `sizeof(struct osoba)`
c. `sizeof(OSOBA)`
d. `sizeof(struct OSOBA)`.

8. Celé čísla v rozsahu 0 až 8 vygenerujeme príkazom/príkazmi:

a. `c = rand() % 8`
b. `c = rand() % 9`
c. `c = rand(8)`
d. `c = rand(9)`.

9. Celé čísla v rozsahu 10 až 20 vygenerujeme príkazom/príkazmi:

a. `c = rand() % 20`
b. `c = 10 + rand() % 10`
c. `c = 10 + rand() % 11`
d. `c = 10 + rand() % 20`.

10. Reálne čísla v rozsahu <0,1> vygenerujeme príkazom/príkazmi:

a. `c = rand()/(double)RAND_MAX;`
b. `c = (double) rand()/RAND_MAX;`
c. `c = rand()/(double)RAND_MAX + 1;`
d. `c = rand()/(double)(RAND_MAX + 1);`

11. Reálne čísla v rozsahu <0,10> vygenerujeme príkazom/príkazmi:

a. `c = 10 *rand()/(double)RAND_MAX;`
b. `c = 10 * ((double) rand()/RAND_MAX);`
c. `c = 11 *rand()/(double)RAND_MAX + 1;`
d. `c = 11 *rand()/(double)(RAND_MAX + 1);`

12. Reálne čísla v rozsahu <10,20> vygenerujeme príkazom/príkazmi:

a. `c = 10 + 10 *rand()/(double)RAND_MAX;`
b. `c = 20 - 10 * ((double) rand()/RAND_MAX);`
c. `c = 20 *rand()/(double)RAND_MAX +1;`
d. `c = 20 *rand()/(double)(RAND_MAX + 1);`

Kontrolné otázky

1. Charakterizujte štruktúru.
2. Aký je význam štruktúry? V čom je nápomocná?
3. Ako pristupujeme k jednotlivým prvkom štruktúry?
4. Ako zadefinujeme nový typ, ktorý predstavuje štruktúru?
5. V čom nám môže pomôcť štruktúra pri práci s poľom?

6. Vytvorte pole 10-tich kružníc. Kružnicu definujte ako štruktúru.
7. Ako vypočítate pomocou vektorového súčinu plochu trojuholníka?
8. Opíšte prácu s generátorom náhodných čísel. Aké čísla generátor generuje, celé alebo reálne?
9. Ako dáte vygenerovať reálne čísla pochádzajúce z určitého intervalu?
10. Opíšte, ako odmeriate dĺžku vykonávania programu, alebo nejakej jeho časti?

Cvičenia

1. Napíšte program, ktorý pomocou funkcie vypočíta absolútnu hodnotu komplexného čísla definovaného ako v príklade 1.
2. Dodefinujte do štruktúry pre trojuholník z príkladu 4 vektor normály a napíšte funkciu, ktorá ho pre každý trojuholník vypočíta. Čo treba pozmeniť?
3. Množinou bodov z príkladu 4 sa dajú preložiť ďalšie dva trojuholníky. Doplňte o ne program. Vypočítajte v cykle plochu všetkých trojuholníkov v triangulácii. Ako by ste skontrolovali, či je výsledný súčet správny?
4. Zadefinujte pole 10 kružníc definovaných ako štruktúry. Súradnice štruktúr nech sú celé náhodné čísla v rozsahu 0-200 a polomery reálne čísla v rozsahu 1 až 5. Vypočítajte, koľko kružníc má polomer väčší ako 3.
5. Zadefinujte pole, ktorého prvkami budú rodné čísla, napr. študentov v krúžku (desaťmiestne čísla). Vytvorte pole rovnakej dĺžky, ale obsahujúce štruktúry `student`, ktoré budú obsahovať tieto položky: vnorenú štruktúru `datum` , predstavujúcu deň, mesiac a rok narodenia , položku `vek` a položku pohlavie (0 v prípade muža a 1 v prípade ženy). Na základe rodných čísel novo zadefinované položky naplňte. Pri prevode rodného čísla na rok, deň a dátum môžte použiť cvičenia z kapitoly 2. Na záver pre daný krúžok vypočítajte, aký v ňom bol priemerný vek dievčat a aký bol priemerný vek chlapcov.

Zadania

1. Zistite, čo robí nasledujúci program:

```
main()
{
 int i,j,suma;
 typedef struct{
         int r[3];
               }RI;
 RI M[10];

 for (i=0; i<3;i++)
   for (j=0;j<3; j++)
        scanf("%d",  &M[i].r[j]);

 suma=0;
 for (i=0; i<3;i++)
      suma+=M[0].r[i]*M[1].r[(i+1)%3]*M[2].r[(i+2)%3];

 for (i=0; i<3;i++)
```

```
        suma-=M[2].r[i]*M[1].r[(i+1)%3]*M[0].r[(i+2)%3];

printf("vysledok je %d\n",suma);
}
```

2. Budeme simulovať aditívny šum. Predstavme si, že máme čiernobiely obrázok, ktorý je reprezentovaný celočíselnými hodnotami od 0 do 255. Ku každej hodnote pričítajme alebo odčítajme nejaké číslo v rozmedzí napríklad 0 až 30. Vizuálny efekt je znázornený na obrázku. S obrázkami budeme pracovať neskôr. Zatiaľ napíšte procedúru, ktorá k číslam od 0 do 255 pričíta alebo odčíta náhodné číslo v rozmedzí 0 až 50. Musíte zabezpečiť, aby číslo ostalo v rozmedzí 0 až 255. Čísla väčšie ako 255 nech ostanú číslom 255 a zo záporného čísla urobte nulu.

6 Smerníky a príklady ich použitia

Hlavné podtémy kapitoly:

- Charakteristika smerníka a jeho definícia, operátory `&` a `*`.
- Smerníková aritmetika. Vzťah medzi smerníkmi a poliami.
- Smerník na štruktúru. Význam typu pri definícii smerníka.
- Základy práce so súbormi. Otvorenie súboru na čítanie a na zápis a jeho uzavretie. Príklady základných funkcií pre čítanie a zápis do súboru. Kontrola správnosti otvorenia súboru.
- Funkcia C-jazyka na triedenie poľa prvkov `qsort` a funkcia na vyhľadávanie `bsearch`.

6.1 Smerníky

Smerníky sú fundamentálnou časťou C. Ak nevieme poriadne používať smerníky, potom strácame flexibilitu a možnosti, ktoré C poskytuje.
C často používa smerníky. Prečo?

- Je to jediný spôsob, ako vykonať niektoré operácie.
- Dôsledkom ich použitia býva kompaktný a efektívny kód
- Predstavujú veľmi silný (ale aj nebezpečný) programátorsky nástroj, napr. na šetrenie pamäti.

C využíva vnútorne (explicitne) smerníky pri:

- poliach,
- štruktúrach,
- funkciách.

6.1.1 Čo je to smerník

Smerník (ukazovateľ, pointer) je premenná na uchovanie adresy inej premennej. Pri práci s touto premennou sa často stretneme s dvoma operátormi: **&** na získanie adresy a ***** na získanie obsahu z danej adresy.
Nech smerník obsahuje adresu objektu premennej `x`. Hovoríme, že smerník ukazuje na x. Ak pristupujeme k premennej pomocou inej premennej, hovoríme o *nepriamom* prístupe. Výhody takéhoto prístupu sa nám objasnia neskôr. Predpokladajme, že `x` je premenná typu `int` a že `px` je smerníková premenná, vytvorený nejakým doteraz nešpecifikovaným spôsobom. Unárny operátor & poskytuje adresu objektu, a teda príkaz

```
px=&x;
```

priradí adresu premennej `x` do premennej `px`. Operátor & môžeme aplikovať len na premenné a na prvky polí.
Unárny operátor * zaobchádza so svojim operandom ako s adresou a jeho výsledkom je daný obsah. Teda ak `y` je tiež typu `int`, tak

```
y = *px;
```

priradí premennej `y` obsah toho, na čo ukazuje `px`, v tomto prípade celé číslo. Teda postupnosť

```
px = &x;
y = *px ;
```

priradí y rovnakú hodnotu ako

```
y = x;
```

Operátory * a & sú silnejšie (majú vyššiu prioritu) ako aritmetické. Operátor & sa nazýva **referenčný** a operátor * sa nazýva **dereferenčný** operátor.
Premenné, ktoré sa na uvedených operáciách zúčastňujú, treba tiež definovať. Pri priradení `y=*px` vás možno napadlo, že ak `px` ukazuje na nejakú adresu, na výpočet hodnoty je možné od tejto adresy zobrať 1 byte, 2 byty, 4 byty atď. a že napr. na 4 bytoch môže byť uložené celé aj reálne číslo, ktoré sú v pamäti inak reprezentované.

6.1.2 Definovanie smerníka

Pri definícii smerníka vlastne stanovujeme, akého typu je obsah adresy, na ktorú ukazuje smerník. Napr.

```
int *smerník;
```

Teraz už vieme, že pri čítaní hodnoty z adresy, na ktorú ukazuje smerník, máme zobrať 4 (prípadne 2, podľa systému) byty a hodnotu na nich máme chápať ako celé číslo. Príklad, ako sa to urobí, si popíšeme neskôr.

Smerník je vždy spojený s určitým typom!!!

Príklad 1

Majme nasledujúci hypotetický kód a predpokladajme, že `x` je v pamäti uložená na pozícii `100`, `y` na `200` a `ip` na `1000`.
Majme takéto definície:

```
int x = 1, y = 2;
int *ip;
```

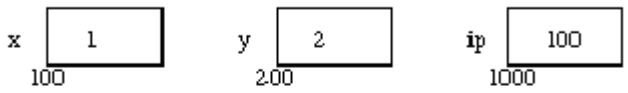
a skupinu (nezávislých) inštrukcií:

a. `ip = &x;`
b. `y = *ip;`
c. `*ip = 3;`

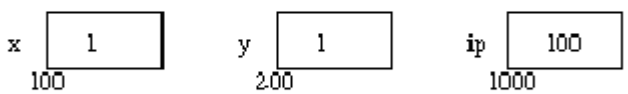
Celú situáciu si znázornime graficky. Údaje v rohu obdĺžnikových buniek predstavujú adresy premenných.

```
int x = 1, y =2;
int *ip;
```

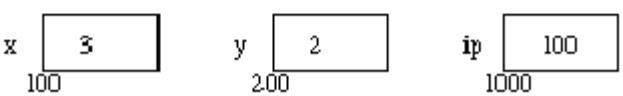
```
ip = &x;
```




```
y = *ip;
```




```
*ip = 3
```



Ešte raz pripomíname, že operátor * je závislý od toho, akého typu je smerník. Podľa toho vie, po koľkých bytoch treba „siahnuť“, aby získal hodnotu a akým spôsobom má hodnotám v bytoch rozumieť (napr. či je to celé alebo reálne číslo).

Cvičenie 1

Vyskúšajte si nasledujúci príklad:

```
main()
{
char *p1;
short int *p2;
int  *p3;
char a[4]={1,1,1,1};
printf("short %d \n",sizeof(short int));
printf("int %d \n",sizeof( int));  /*výpis pomocnej informácie*/
p1=&a[0];
p2=&a[0]; /* kompilátor hlási varovanie*/
p3=&a[0]; /* kompilátor hlási varovanie*/
printf("char *p1 %d \n", *p1);
printf("short *p2 %d \n", *p2);
printf("int *p3 %d \n", *p3);
}
```

Výsledok závisí od toho, koľko miesta zaberá celočíselný typ `int`. Všetky smerníky ukazovali na tú istú adresu, ale čítal a interpretoval sa rôzny počet bytov. Výsledok môže byť napríklad:

```
short 2
int 4
char *p1 1
short *p2 257
int *p3 16843009
```

Pri označených príkazoch kompilátor hlásil varovanie. Program sa však vykonal správne. Ako sa varovaniam tohto typu vyhnúť si ukážeme v závere kapitoly. Niektoré kompilátory namiesto varovania môžu hlásiť chybu.

Pri definovaní sa smerník nenastaví, a teda „neukazuje nikde“. Treba ho nastaviť skôr ako ho použijeme, inak môže zapríčiniť padnutie programu!!!

Napr.

```
    int *ip;
    *ip = 100;
```

zapríčiní padnutie programu. Správne použitie je:

```
   int *ip;
   int x;
   ip = &x;
  *ip = 100;
```

V ďalšom texte si objasníme význam smerníka na niekoľkých príkladoch. Ukážeme si, ako sa smerník využíva pri práci s funkciami a pri práci so štruktúrami

6.1.3 Smerník a funkcia

V časti 4.2.2 sme hovorili, že v jazyku C sa všetky argumenty funkcie odovzdávajú hodnotou. To znamená, že volanej funkcii sa odovzdávajú kópie hodnôt pomocou dočasných premenných (uložených v zásobníku). V jazyku C volaná funkcia nemôže zmeniť premennú vo volajúcej funkcii; zmeniť môže iba svoju súkromnú dočasnú premennú. Zmeniť premennú mimo volanej funkcie však možno a vlastne sme to už aj robili, keď sme používali funkciu scanf. Ako jej parameter sme zadali ***adresu*** premennej, do ktorej sme chceli priradiť hodnotu načítanú zo vstupu. Uveďme si ďalší podobný príklad.

Príklad 2

V hlavnom programe máme celočíselnú premennú x, ktorú budeme chcieť meniť v závislosti od reálnej premennej a, ktorá bude predstavovať náhodné číslo od -10 do 10. Funkcia nastaví premennú x na 0, ak je a záporné a na 1, ak je a kladné alebo rovné nule.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
void nastav(int a, int *b){
if (a>=0) *b=1;
else *b=0;}
main(){
int x,a;
srand((unsigned)time(NULL));  /*inicilizácia generátora náh. čísel*/
a=rand()%21 -10;              /*vygenerovanie  náh. čísel*/
nastav(a,&x);
printf("a=%d, x=%d\n",a,x);
}
```

Príklady výpisov:

```
a=-7, x=0
a=15, x=1
```

6.1.4 Smerník a štruktúra

Smerník môže ukazovať aj na premennú predstavujúcu štruktúru. Majme štruktúru z predchádzajúcej hodiny a definujme smerník ukazujúci na danú štruktúru.

```
typedef struct kružnica{
     double x;
     double y;
     float  r;
     }KRUŽNICA;
 KRUŽNICA *p_kr;
```

Priraďme smerníku p_kr nejakú existujúcu adresu. Z toho, čo bolo doteraz povedané, by sme sa k jednotlivým položkám štruktúry vedeli dostať takto:

```
(*p_kr).x, (*p_kr).y, (*p_kr).r.
```

Zátvorku musíme použiť, pretože operátor . má vyššiu prioritu ako operátor *. V C-jazyku však používame skrátený zápis

```
p_kr->x, p_kr->y, p_kr->r.
```

(*p_kr).r= p_kr->r

V prípade, že sme definovali premennú typu `KRUŽNICA`, v pamäti sa skutočne vyhradilo miesto pre uloženie `x`, `y` a polomeru a premenná predstavovala tento pamäťový blok. V prípade, že definujeme smerník na štruktúru `kružnica`, tak si môžeme predstaviť, že adresa uložená v smerníku bude predstavovať miesto, od ktorého priložíme „šablónu" štruktúry pre kružnicu. Treba si uvedomiť, že ak pošleme do funkcie štruktúru ako parameter, vždy sa vytvára jej kópia. Ak chceme štruktúru modifikovať pomocou funkcie, musíme ju vrátiť, inak zmodifikujeme kópiu, ktorá po skončení vykonávania funkcie zanikne. Ak je štruktúra veľká, je tento spôsob nevýhodný a používa sa smerník, kde sa prenáša vždy len adresa, 4 alebo 8 bytov, v závislosti od operačného systému.

6.2 Základy práce so súbormi

Vo väčšine prípadov vyskytujúcich sa v praxi, máme dáta, ktoré spracovávame, uložené v súboroch. Napríklad, keď chceme spracovať nejaký obrázok (napríklad zlepšiť jeho kvalitu), musíme ho počas behu programu načítať z disku do vyhradeného miesta v pamäti - do nejakého poľa.
Keď pracujeme so súborom, musíme mať vložený hlavičkový súbor `<stdio.h>`. Pre prácu so súborom je v C vyhradená štruktúra `FILE`. Môžeme si ju predstaviť ako tabuľku, do ktorej sa zapisujú údaje potrebné na prácu operačného systému so súborom. Pre konkrétny súbor definujeme smerník na túto štruktúru. Súbor musíme otvoriť. Pri operácii otvorenia sa vytvorí a naplní konkrétna tabuľka a v prípade úspešného vytvorenia sa súborový smerník nastaví na začiatok tabuľky. Pomocou tohto smerníka sa potom budeme na súbor odvolávať, meno súboru už viackrát v programe nebude vystupovať. Zo súboru budeme čítať alebo doň môžeme zapisovať. Po skončení práce súbor zatvoríme.

Základný dátový typ pre prácu so súborom v jazyku C je `FILE`:
`FILE*` - je to smerník na objekt typu `FILE`.
Definícia premennej `súbor` pre prácu so súborom potom bude:

```
FILE *súbor;
```

Identifikátor `FILE` musí byť napísaný veľkými písmenami. Ak definujeme viac smerníkov na `FILE`, musíme napísať `FILE *f1,*f2`, a nie `FILE* f1,f2`!

6.2.1 Otvorenie súboru na čítanie

```
súbor =  fopen("pokus", "r");
súbor =  fopen("D:/data/blany.tif","r");
```

Prvým príkazom otvoríme súbor s menom „pokus" v aktuálnom adresári, pri druhom príkaze uvádzame celú cestu k súboru. Ako si otvoríme súbor tak, že jeho meno zadáme po spustení programu alebo pri spustení programu, si povieme neskôr.
Príkaz pre formátované čítanie zo súboru je:

```
fscanf(súbor, "formát", argumenty);
```

6.2.2 Ukončenie práce so súborom

```
fclose(súbor);
```

Príklad 3

Načítajme zo súboru `num` v aktuálnom adresári 10 celých čísel a priraďme ich do poľa `A`.

Riešenie:

```
#include  <stdio.h>
main()
{
FILE *sub;
int i, A[10];
sub =  fopen("num", "r");
for (i=0; i<10; i++)
   fscanf(sub, "%d", &A[i]);
fclose(sub);
}
```

Príkaz `fscanf` pracuje podobne ako príkaz `scanf`, ale údaje nenačítava z klávesnice, ale zo súboru. V tomto prípade, po otvorení súboru prečíta prvý reťazec zo súboru až po biely znak (medzeru, oddeľovač...), na základe špecifikácie "`%d`" skonvertuje tento reťazec na celé číslo a uloží ho do premennej `A[0]`. Znova načíta ďalší reťazec po biely znak, skonvertuje ho, uloží do `A[1]`, a tak pokračuje, spolu 10 krát.

Ak súbor uzatvoríme a znovu ho otvoríme na čítanie, údaje sa budú čítať od začiatku. Ak pri čítaní potrebujeme načítavať znova od začiatku súboru, netreba súbor zatvárať a otvárať. Riešenie si ukážeme v jednej z ďalších kapitol. Podobne si, ukážeme, ako zistíme, že sme už na konci súboru.

6.2.3 Otvorenie súboru na zápis

```
súbor =  fopen("pokus", "w");
```

Príkaz pre formátovaný zápis do súboru je:

```
fprintf(súbor,"formát",argumenty);
```

Príklad 4

Zapíšme do aktuálneho adresára do súboru `num` 10 celých čísel uložených v poli `A`.

Riešenie:

```
#include  <stdio.h>
main()
{FILE* sub;
  int i, A[10];
sub =  fopen("num", "w");
for (i=0; i<10; i++)
   fprintf(sub, "%d",A[i]);
fclose(sub);
```

```
}
```

Ak súbor uzavrieme a znovu otvoríme a začneme doň zapisovať, obsah súboru sa prepíše – načítava sa vždy od začiatku súboru. Ak chceme po otvorení súboru zapisovať na jeho koniec, musíme ho otvoriť v režime *append*, t. j. použiť `"a"` namiesto `"w"`.

6.2.4 Testovanie správnosti otvorenia a uzavrenia súboru

Ak by sme v príklade1 a príklade2 zadali meno neexistujúceho súboru, program by zhavaroval. V prípade, že sa súbor nepodarí otvoriť (najčastejšia príčina býva, že si popletieme meno), vráti `fopen` hodnotu `NULL`, ktorá býva definovaná v `stdio.h` a má väčšinou hodnotu `0`. Preto je dobré zaradiť za príkaz `sub = fopen(„num","w");` test `if(sub == NULL) {printf("súbor sa neotvoril\n"); return;}`.
Pri neúspešnom uzatvorení súboru (napr., je otvorený v inom programe) vracia `fclose()` hodnotu `EOF`.

```
if (fclose(sub) ==  EOF)
          printf("súbor sa neuzatvoril\n"); ...
```

6.3 Smerníková aritmetika

So smerníkmi môžeme vykonávať aritmetické operácie podobne ako s celými číslami, ale s výraznými obmedzeniami. Smerníková aritmetika disponuje 4 aritmetickými operátormi +, `-`, `++`, `--`, kde v prípade oprátorov + a – je druhý operand celé číslo. **Dva smerníky môžeme odčítať, nemôžeme ich však sčítať.**

Dôvodom, prečo spájame smerník pri definícii s dátovým typom je, že treba vedieť, v koľkých bytoch sú uložené dáta. Keď inkrementujeme smerník, posunieme sa o jeden pamäťový blok, zodpovedajúci premennej.

Pre smerník ukazujúci na **znak**, `ch_ptr++` pridáva k adrese 1 byte.
Pre smerník ukazujúci na **float**, `fl_ptr++` pridáva k adrese 4 byty.

Všeobecne, ak `p` je smerník na daný typ, potom výraz `(p+a)` ukazuje na adresu vzdialenú práve o `(a*sizeof(typ p))` bytov. Pri aritmetických operáciách so smerníkmi sa teda zohľadňuje dĺžka typu, na ktorý smerník ukazuje. Obsah premennej typu smerník môžeme vytlačiť s použitím riadiaceho reťazca `%p`.

Príklady operácii so smerníkmi:

```
...
short int *p1, *p2, x;

p2 = &x;
p1 = p2;      printf(„%p \n",p1);
p1++;         printf(„%p \n",p1);
p2--;         printf(„%p \n",p2);
p1 = p2 + 1; printf(„%p \n",p1);
p1 = p2 +10; printf(„%p \n",p1);
p1 = p2 - 5; printf(„%p \n",p1);
...
```

Rozdiel dvoch smerníkov si môžeme predstaviť takto: majme `int *p1,*p2;` Nech `p1=p2+5,` teda `p1-p2=5.` **Je to počet pamäťových blokov zodpovedajúcich premennej typu smerníka, a nie vzdialenosť počítaná v bytoch.**

Príklad 5

```
...
short int *p1, *p2, a[5]={1,2,3,4,5};
int i;
p1  = & a[1]; printf(„p1: %p \n”,p1);
p2  = & a[4]; printf(„p2: %p \n”,p2);
i = p2 – p1;  printf(„ i: %d \n”,i);
...
```

Príklad 6 (Vzťah poľa a smerníkov)

Majme desaťprvkové pole definované príkazom `double a[10].` Chceme definovať smerník ukazujúci na začiatok poľa. Použime príkazy

```
double *p;
p=& a[0];
```

V C jazyku je identifikátor poľa vlastne smerník na jeho počiatok, t. j adresa jeho začiatku. Teda nastavenie smerníka `p` na začiatok poľa dosiahneme aj jednoduchším príkazom

```
p= a;
```

p=&a[0]; je to isté ako p=a;

V súlade s predchádzajúcimi poučkami, príkaz `p+1` ukazuje o 8 bytov ďalej ako smerník `p,` teda na druhý prvok poľa, `p+2` na tretí atď. `*p,*(p+1)` a `*(p+2)` predstavujú reálne hodnoty uložené na príslušných adresách, teda prvky `a[0],a[1]` a `a[2].` Keby sme pre každú hodnotu `i` dali vypísať `*(p+i),` dostali by sme postupne hodnoty všetkých prvkov poľa. Také isté pravidlá platia aj pre `a+i` a `*(a+i).` Vo všeobecnosti pre ľubovoľné pole platí:

a[i]= *(a+i)
&a[i]= a+i

Jediný rozdiel medzi prácou so smerníkom `p` a názvom poľa `a` je, že hodnotu `p` môžeme meniť, ale hodnotu `a` nie. Keby sme sa o to pokúsili, kompilátor vyhlási chybu. Skúste si napísať nasledujúci program:

```
main() {
double a[10],*p;
p=a;
p=a+1;
a=p+1;
}
```

Pozrite sa, akú chybu vám vyhlási kompilátor na vyznačenom príkaze.

6.4 Pretypovávanie smerníkov

Majme súvislý pamäťový priestor definovaný príkazom

```
char a[100];
```

Predstavme si, že toto pole využívame v programe striedavo. V niektorých momentoch ho nepotrebujeme a môžeme ho využívať na pomocné výpočty. V nasledujúcom príklade voľný priestor využijeme raz ako pole reálnych čísel a druhýkrát ako pole premenných typu KRUZNICA, pričom najprv si vždy vypočítame, koľko prvkov môže dané pole obsahovať.

a) reálne pole typu `double`

```
char a[100];
double *p;
int m,i;
…
m = 100 / sizeof(double);
p = (double *)a;
for (i = 0; i < m; i++)
    p[i] = …
```

b) pole premenných typu `KRUZNICA`

```
char a[100];
typedef struct {
  float x;
  float y;
  float r;
     }KRUZNICA;
KRUZNICA *p;
int m,i;
m = 100 / sizeof(KRUZNICA);
p = (KRUZNICA *)a;
for (i = 0; i < m; i++)
 {    p[i].x = …
      p[i].y = …
      p[i].r = …
```

V nasledujúcom príklade vychádzajme z cvičenia 1, kde sme si ukázali, aký je rozdiel v tom, keď smerník síce ukazuje na to isté miesto, ale je iného typu. Smerníky môžeme pretypovávať ako „obyčajné“ premenné.

Príklad 7

```
main()
{
char *p1; int *p2;
char i; int j; short int k;
char a[4]={1,1,1,1};

p1 = &a[0];
i = *p1;
p2 = (int *) p1;
j = *p2;
k = *((short int *)p1);     /*dva kroky v jednom*/
printf("p1,*p1: %p %d \n", p1,i);
printf("p2 *p2 %p %d \n",  p2,j);
printf("k %d \n", k);
```

```
}
```

Všimnite si, že tu (ale aj v predchádzajúcom príklade) sme pretypovanie použili pri priraďovaní smerníkov rôzneho typu, pri priradení do `p2`. Priradenie by sa vykonalo správne, ale kompilátor by vypísal varovanie. V prípade priradenia do `k` by sme však bez pretypovania dostali iný výsledok.

6.4.1 Smerník na dátový typ `void`

Definuje sa príkazom

```
void *p;
```

Takýto smerník neukazuje na žiadny dátový typ. Voľne by sme ho mohli charakterizovať ako smerník, pri ktorom je dôležitá iba jeho adresa. Smerník typu `void` sa dá využiť na ukazovanie na ľubovoľný typ, môžeme ho priradiť do smerníka iného typu bez toho, že by kompilátor hlásil varovanie, takisto bez varovania môžeme do smerníka typu `void` priradiť hodnotu smerníka iného typu.

Na smerník typu `void` nefunguje smerníková aritmetika. Nemôžeme napríklad použiť dereferenčný operátor *, ani nevieme, o koľko sa máme posunúť v pamäti, keď k nemu pričítame jednotku. Smerník typu `void` obyčajne používame v spojení s dôkladným pretypovávaním.

Pri preklade cvičenia 1 kompilátor vygeneroval varovanie, že v priraďovacom príkaze neboli smerníky rovnakého typu. Program sa síce vykonal správne, ale typ `void` predstavuje možnosť, ako sa generovaniu takýchto hlásení vyhnúť.

Príklad 8

```
main()
{
char *p1; short int *p2;
int  *p3; void *p;
char a[4]={1,1,1,1};

p=&a[0];
p1=p;
p2=p;
p3=p;

printf("char *p1 %d \n", *p1);
printf("short *p2 %d \n", *p2);
printf("int *p3 %d \n", *p3);
}
```

V C existuje dátová konštanta `NULL` pre označenie, že smerník „neukazuje na nič“. Je možné ho priradiť bez pretypovania všetkým typom smerníkov. Je to symbolická konštanta definovaná v `stdio.h` ako napríklad:

`#define NULL 0` alebo `#define NULL ((void *)0)`

6.5 Funkcia na triedenie `qsort`

Typické použitie smerníkov v jazyku C si môžeme ukázať na funkcii `qsort`. Úlohou tejto funkcie je usporiadať pole prvkov (jednoduchého typu alebo štruktúr) vzostupne alebo zostupne. Pri triedení poľa (bez použitia iného poľa) vždy vymieňame dvojice prvkov. To sa dá robiť rôznym spôsobom. Zvolená stratégia ovplyvňuje rýchlosť utriedenia.

Algoritmus použitý v procedúre `qsort` je veľmi efektívny a nazýva sa *quick sort*.
Táto metóda vychádza z princípov triedenia výmenou a jej autor C.A.Hoare ju nazval rýchle triedenie – *quick sort*. Princíp algoritmu je založený na skutočnosti, že najefektívnejšie sú tie výmeny, ktoré sa vykonávajú na veľké vzdialenosti. Stratégia výmen (t.j. postup, ktorý prvok s ktorým sa má vymeniť v prípade, keď ich porovnanie stanoví, že ich vymeniť treba) je naprogramovaná v procedúre `qsort`. Čo musíme urobiť my, je naprogramovať práve procedúru pre porovnanie dvoch prvkov. Procedúra nevie, čo triedi, to vieme my. My jej len musíme dodať informáciu, kedy máme prvky vymeniť.
Triediť môžeme znakové, celočíselné, reálne polia a aj polia štruktúr. Preto sa funkcia porovnania dvoch prvkov nedá naprogramovať všeobecne, ale musíme ju naprogramovať my. Pritom bude zabezpečené, že do našej porovnávacej funkcie prídu dve adresy prvkov, ktoré treba porovnať. Adresy prídu ako dva smerníky na typ `void`. Tieto smerníky treba pretypovať podľa typu prvkov, ktoré porovnávame. Pomocou dereferenčného operátora * môžeme potom získať hodnoty prvkov a môžeme ich porovnať. Podľa výsledkov porovnania vrátime hodnoty 1, 0 alebo -1.
Aby sme mohli funkciu `qsort` použiť, musíme vložiť hlavičkový súbor `<stdlib.h>`.
Majme pole s názvom `t_pole`, ktoré má `n` prvkov dĺžky `dlzka_prvku` (meranej v bytoch). Aby funkcia `qsort` mohla usporiadať toto pole, musíme doprogramovať funkciu `porov` (meno je voliteľné), ktorá rozhodne o porovnávacom kritériu a o tom, či sa pole usporiada vzostupne alebo zostupne.

Funkčný prototyp funkcie je nasledujúci:

```
void qsort(void *t_pole, int n, int dlzka_prvku,
                   int(*porov)(const void *, const void *));
```

Funkcia `porov` má tieto vlastnosti:

- má dva parametre typu smerník na `const void`
- funkcia vracia:
  - záporné číslo, ak je prvý parameter menší než druhý,
  - nulu, ak sú parametre zhodné,
  - kladné číslo, ak je prvý parameter väčší ako druhý.

Takýmto spôsobom dostaneme usporiadanie vzostupne. Ak prehodíme význam kladného a záporného čísla, dostaneme usporiadanie zostupne.
Pri tejto funkcii sa stretneme s typovým modifikátorom `const`. Tento špecifikuje, že definovanému objektu nesmie byť po jeho inicializácii už menená hodnota. Tento typ umožňuje kompilátoru robiť určité kontroly.

Príklad 9

Usporiadajte celé čísla uložené v poli dĺžky `n` vzostupne.

Riešenie:

```
#include <stdlib.h>
#define n 6

int porov(const void* prve,const void* druhe)
{
int cislo1,cislo2;
cislo1 = *((int *)prve);
```

```
/*najprv pretypovat na konkretny typ a potom zobrat hodnotu*/
cislo2 = *((int *)druhe);

if (cislo1 < cislo2) return (-1);
if (cislo1 > cislo2) return (1);
return 0;
}
main()
{
int i, X[n]={3,-1,5,3,8,7};

/*vypisanie X  a jeho usporiadanie */
for (i = 0; i < n; i++)
   printf("%d ",X[i]);
printf("\n");
qsort((void*)X,n,sizeof(int),porov);
for (i = 0; i < n; i++)
   printf("%d ",X[i]);
printf("\n");
}
```

6.6 Funkcia na vyhľadávanie v utriedenom poli `bsearch`

Veľakrát potrebujeme vyriešiť takúto úlohu: máme pole a chceme zistiť, či sa v ňom nachádza nejaká konkrétna hodnota. Vyhľadávanie sa dá urýchliť, ak prvky poľa sú už usporiadané. V takom prípade je najzaužívanejšou technikou opakované delenie intervalu na podintervaly, v ktorých sa zadaný prvok hľadá. Tento spôsob vyhľadávania sa nazýva *bisekcia* alebo binárne vyhľadávanie. Takýto algoritmus predstavuje aj funkcia C jazyka `bsearch`.
Okrem toho, že chceme zistiť, či sa zadaný prvok v poli nachádza, chceme často zistiť aj jeho pozíciu v danom utriedenom poli. Funkcia `bsearch` nám nevráti priamo index hľadaného prvku v poli, ale smerník na prvok ukazujúci, pomocou ktorého môžeme potom index vypočítať.

Majme usporiadané pole `u_pole`, ktoré má `n` prvkov dĺžky `dlzka_prvku` meranej v bytoch.

Funkčný prototyp funkcie `bsearch` je nasledujúci:

```
void *bsearch(const void *kľúč, const void *u_pole, n,
    velkost_prvku, int(*porov)(const void *, const void *));
```

Funkcia vráti smerník na typ `void`, ktorý si podľa potreby môžeme pretypovať. Parameter `kľúč` je hodnota, ktorú vyhľadávame. Zadávame však smerník na túto hodnotu (pretypovaný na `void`, prečo??), a nie samotnú hodnotu. Funkcia `porov` má ten istý význam ako vo funkcii `qsort`. V prípade, že vyhľadávaný prvok sa v usporiadanom poli opakuje, hodnota vráteného smerníka je nešpecifikovaná v tom zmysle, že napríklad sa nemôžeme spoľahnúť, že je to smerník na prvý zo skupiny rovnakých prvkov. Program potom treba príslušne upraviť.

Príklad 10
Vychádzajme z predchádzajúceho príkladu. Usporiadajme celé čísla uložené v poli dĺžky `n` vzostupne a zistime, koľký je v usporiadanom poradí prvý prvok pôvodného poľa (t.j. určime

index prvku s hodnotou 3 v usporiadanom poli). Použijeme rozdiel dvoch smerníkov. Uvedomme si, že sa pole utriedi do toho istého pamäťového priestoru (ak neurobíme kópiu), a teda pôvodné usporiadanie sa stratí.

Riešenie:

```
#include <stdlib.h>
#define n 6


int porov(const void* prve,const void* druhe)
{
int cislo1,cislo2;
cislo1=*((int *)prve);
/*najprv pretypovat na konkretny typ a potom zobrat hodnotu*/
cislo2=*((int *)druhe);

if (cislo1<cislo2) return (-1);
if (cislo1>cislo2) return (1);
return 0;
}

main()
{
int i, a, X[n]={3,-1,5,3,8,7};
a = X[0];

/*vypisanie x  a jeho usporiadanie */
for (i=0;i<n;i++)
   printf("%d ",X[i]);
printf("\n");
qsort((void*)X,n,sizeof(int),porov);
for (i=0;i<n;i++)
   printf("%lf ",X[i]);
printf("\n");
s=(int *)bsearch((const void *)&a,
    (const void *)X,n,sizeof(int),porov);
printf("%d ",s-&X[0]);
}
```

Program po spustení vypísal nasledovné riadky:

3 -1 5 2 8 7

-1 2 3 5 7 8

2

☑ Kontrolný test

1. Vyberte správne tvrdenia:
   a. smerník je premenná, ktorá obsahuje adresu inej premennej,
   b. smerník musí byť vždy spojený s nejakým typom,
   c. nech `p=&x`, kde `p` je smerník na `int`, operácia `*p` vráti obsah bytu s adresou uloženou v `p`,

d. nech `p=&x`, kde `p` je smerník na `int`, operácia `*p` vráti obsah `sizeof(int)` bytov od adresy uloženej v `p`.

2. Smerník `p` ukazujúci na `int` definujeme príkazom:
   a. `*int p;`
   b. `int *p;`
   c. `int p;`
   d. `int p*;`

3. Majme príkazy:

```
int i=1, j=2, *p1, *p2;
p1 = &j;
p2 = &i;
i = *p1;
j = *p2;
```

Výsledkom týchto operácií je:
   a. `i = 2; j = 2;`
   b. `i = 1; j = 1;`
   c. `i = 1; j = 2;`
   d. `i = 2; j = 1;`

4. Majme príkazy:

```
char a[4]={0,1,0,1};
char *p1; short int *p2;
p1 = &a[0];
p2 = &a[2];
```

Platí:
   a. `(*p1) = 1, (*p2) = 1`
   b. `(*p1) = 0, (*p2) = 1`
   c. `(*p1) = 1, (*p2) = 0`
   d. `(*p1) = 0, (*p2) = 256.`

5. Majme definíciu `char *p, x;`
   Príkaz `x=*p;`
   a. môže zapríčiniť zhavarovanie systému
   b. nikdy nezapríčiní zhavarovanie systému. Vždy vráti obsah adresy p, nech tá je akákoľvek.
   c. či systém zhavaruje závisí od obsahu p, ale ten je náhodný
   d. aby sme mohli túto operáciu vykonať, p treba najprv korektne nastaviť.

6. Majme definíciu `int *p;`
   a príkaz `p = NULL;` Vyberte správne tvrdenie.
   a. kompilátor vyhlási chybu nekompatibilné priradenie
   b. `NULL` treba pretypovať na smerník na int
   c. priradenie je korektné
   d. priradenie naznačuje, že `p` neukazuje nikam.

7. Majme definície:

```
typedef struct {
     char  deň;
      char mesiac;
     short int rok; } DATUM;
DATUM *d;
char a[4]={25,10,170,7};
```

Vyberte správne tvrdenie o získaní položky `rok`:

a. `(*d).rok  = 1707;`
b. `d->rok =  1962;`
c. `*d.rok  = 5;`
d. `*(d.rok)  = 1707;`

8. Nech `f` je smerník na objekt typu `FILE` úspešne nastavený príkazom `f = =fopen("data","w");` ďalej majme premennú `int x=10;`

Vyberte správne tvrdenia:

a. hodnotu `x` zapíšeme do súboru príkazom `printf("%d",x);`
b. hodnotu `x` zapíšeme do súboru príkazom `printf(f,"%d",x);`
c. hodnotu `x` zapíšeme do súboru príkazom `fprintf(f,"%d",x);`
d. `"data"` musí byť súbor v aktuálnom adresári.

9. Nech `f` je smerník na štruktúru `FILE` úspešne nastavený príkazom `f = =fopen("data","r");` ďalej majme premennú `int x;`

Vyberte príkazy, ktoré môžu predstavovať korektné načítavanie zo súboru `"data"` do `x`:

a. `scanf("%d", x);`
b. `fscanf("%d",x);`
c. `fscanf(f,"%d",x);`
d. `fscanf(f,"%d",&x);`

10. Majme definície a príkazy

```
int  *p1, *p2,i ,a[6];
p1 = &a[2];
p2 = &a[4];
```

Výsledkom operácie `i = p2 - p1` bude:

a. `i = 2;`
b. `i = 8;`
c. `i = 3;`
d. `i = 1;`

11. Majme definície

```
int  i ;
char a[6] = {1,0,1,1,0,0};
```

Výsledkom operácie `i = *((int *)( &a[2]));` bude:

a. `i = 11;`
b. `i = 1;`
c. `i = 256;`
d. `i = 257;`

12. Vyberte nesprávne tvrdenia:
    a. `qsort` slúži na usporiadanie poľa vzostupne alebo zostupne;
    b. funkcia, ktorá rozhoduje o porovnávacom kritériu sa musí volať `porov;`
    c. funkcia `bsearch` vyhľadáva prvok s danou hodnotou v poli a v prípade úspechu vráti smerník typu `void`, ktorý naň ukazuje;
    d. funkcia `bsearch` vyhľadáva prvok s danou hodnotou v usporiadanom poli a v prípade úspechu vráti smerník typu `void`, ktorý naň ukazuje;

Kontrolné otázky

1. Charakterizujte smerník – čo predstavuje, ako sa definuje, aké operácie preň môžeme použiť.
2. Prečo treba smerník nastaviť skôr ako ho použijeme?
3. Vysvetlite, ako funguje dereferenčný operátor *.
4. Akým príkazom vypisujeme obsah smerníka?
5. Aký význam má pretypovanie smerníka? Uveďte príklady použitia.
6. Popíšte prácu so súborom: definíciu, otvorenie, funkcie pre načítanie a zápis a jeho uzatvorenie.
7. Ako sa robí kontrola správneho otvorenia a uzatvorenia súboru?
8. Aký význam má smerník na dátový typ `void?`
9. Kedy sa priraďuje smerníku hodnota `NULL?`
10. Opíšte význam a použitie funkcie `qsort.`
11. Čo predstavuje rozdiel dvoch smerníkov?
12. Opíšte, ako zistíte, či sa v usporiadanom poli nachádza nejaká hodnota. Použite funkciu `bsearch.`
13. Ako zistíme index hľadaného prvku, ak predpokladáme, že sa v (usporiadanom) poli nachádza iba raz?
14. Ako zistíme, či sa v usporiadanom poli nachádza nejaká hodnota práve jeden krát?

Cvičenia

1. Načítajte 10 celočíselných hodnôt zo súboru s menom *vstup* a zapíšte ich do súboru s menom *vystup* v opačnom poradí. Súbor *vstup* vytvorte, súbor *vystup* vznikne.
2. Napíšte program, ktorý spojí obsah dvoch súborov, obsahujúcich 20 celých čísel, do jedného. Mená súborov si zvoľte ľubovoľne.
3. Pomocou funkcie `qsort` usporiadajte 20-prvkové pole typu `unsigned char.` Predpokladajte, že prvky sú uložené v súbore „`data`" v aktuálnom adresári.
4. Pomocou funkcie `qsort` usporiadajte 20-prvkové pole typu `double.` Prvky poľa zadajte pri inicializácii.
5. Zadefinujte 10 prvkové pole typu `int.` Jeho hodnoty zadajte z klávesnice. Pri načítavaní nepoužite indexáciu pomocou hranatých zátvoriek, ale prístup popísaný v smerníkovej aritmetike. Takisto aj pri výpise použite dereferenčný operátor namiesto hranatých zátvoriek.
6. Vytvorte si 100000 prvkové pole čísel od 1 do 100000 (prvku v poli priraďte hodnotu jeho indexu). V tomto poli budeme vyhľadávať zadané číslo dvoma metódami. Najprv pole budeme prehľadávať postupne a potom pomocou binárného vyhľadávania pomocou funkcie `bsearch.` Pre oba postupy naprogramujte výpisy časov a porovnajte ich. Voľte rôzne čísla, napr. zo začiatku, prostriedku a konca poľa.

7. Majme usporiadané pole, v ktorom sa prvky opakujú. napríklad sú v ňom tri štvorky. Takéto pole si utvorte jednoduchým spôsobom – inicializujte ho pri jeho definícii. Ako sme už uviedli, nevieme povedať, na ktorú štvorku bude ukazovať smerník, ktorý nám vráti funkcia `bsearch`. Upravte program tak, aby ukazoval na prvú. (Vo všeobecnosti, aby pri viacnásobných hodnotách ukazoval na prvý prvok.)
8. Zadefinujte si pole rovnakým spôsobom, ako v predchádzajúcom príklade. Napíšte procedúru, ktorá zistí, koľkokrát sa zadaný prvok nachádzal v usporiadanom poli.
9. Napíšte funkciu, ktorá vráti hodnotu prvku, ktorý sa v zadanom poli nachádza najviac krát. Takáto hodnota sa nazýva MODUS. Pole je zadané ako neusporiadané.
10. Majme neusporiadané pole. Napíšte funkciu, ktorá vráti jeho medián. Medián bol definovaný v prvej kapitole.
11. Napíšte procedúru, pomocou ktorej vymeníte hodnoty dvoch premenných.

Zadania

1. Zistite, čo robí nasledujúci program.

```
#include <stdlib.h>
typedef struct{
        float x;
        float y;
        } BOD;

int porov(const void* prve,const void* druhe)
{
BOD cislo1,cislo2;
cislo1 = *((BOD *)prve);
/*najprv pretypovat na konkretny typ a potom zobrat hodnotu*/
cislo2 = *((BOD *)druhe);

if (cislo1.x < cislo2.x) return (-1);
if (cislo1.x > cislo2.x) return (1);
return 0;
}

main() {
       int i;

BOD a[20];
for (i=0;i<20;i++)
     {a[i].x=10*rand()/(double)RAND_MAX;
      a[i].y=10*rand()/(double)RAND_MAX;
      printf("%f %f \n", a[i].x,a[i].y);
      }
qsort((void*)a,20,sizeof(BOD),porov);
printf("\n");
for (i=0;i<20;i++)
      printf("%f %f \n", a[i].x,a[i].y);
}
```

2. Zadefinujte 32-prvkové pole typu `unsigned char`. Vygenerujte doň náhodné čísla od 0 do 2. Na pole sa pozrite jednoducho ako na úsek pamäte. Dajte ho cyklom vypísať tak, aby bolo chápané ako:
   a. pole typu `unsigned char`,
   b. pole typu `short int`,
   c. pole typu `int`,
   d. pole typu `double`.

   Dĺžky polí jednotlivých typov si vypočítajte v programe. Výsledky porovnajte.

7 Dynamické polia. Smerníky a polia

Hlavné podtémy kapitoly:

- Spearmanovho korelačný koeficient a spôsob jeho výpočtu.
- Smerníková aritmetika. Vzťah medzi smerníkmi a poliami.
- Dynamické polia. Funkcie pre prideľovanie a správu pamäte.
- Výpočet Spearmanovho korelačného koeficentu s použitím statických polí.
- Výpočet Spearmanovho korelačného koeficentu s použitím dynamických polí.

7.1 Výpočet Spearmanovho korelačného koeficientu s použitím statického poľa

V tejto kapitole sa budeme venovať implementácii algoritmu 5 z kapitoly 1. Potrebné prostriedky pre vyhľadanie a utriedenie boli popísané v kapitole predchádzajúcej. Uvedený algoritmus je vhodný pre prípady, keď sa dáta v jednotlivých vstupných súboroch neopakujú. Pre druhý prípad existujú presnejšie algoritmy, počítajúce buď z iného vzorca, alebo modifikujúce poradia, pre väčšie dátové súbory sa však ich výsledky vo väčšine prípadov líšia od výsledkov algoritmu 5 iba veľmi málo.

7.1.1. Spearmanov korelačný koeficient

Spearmanov korelačný koeficient sme už spomínali v prvej kapitole. Ešte raz zopakujme jeho význam a spôsob jeho výpočtu: dvojrozmerný štatistický súbor je súbor, ktorý obsahuje vždy dvojice hodnôt (meraní) týkajúcich sa sledovanej udalosti (napr. výška a váha človeka, prah počuteľnosti pravého a ľavého ucha, výška vodného toku na dvoch rôznych miestach, množstvo konzervačnej látky a trvanlivosť výrobku a pod). *Spearmanov korelačný koeficient* je charakteristika, ktorá určuje kvantitatívnu mieru závislosti medzi dvomi zložkami. (viď kapitola 1, algoritmus 5).

Spearmanov korelačný koeficient je založený na výpočte poradí a má tieto vlastnosti:
- nadobúda hodnoty z intervalu < -1, 1 >,
- pri úplnej zhode poradí sa rovná 1 a ide o priamu funkčnú závislosť,
- ak sa rovná -1, ide o nepriamu funkčnú závislosť,
- pri korelačnej nezávislosti sa rovná 0.

Postup pri výpočte tohto koeficientu je takýto:
krok1 - usporiadame vzostupne namerané hodnoty x_i a hodnoty y_i ,
krok2 - nech R_i je poradie hodnoty x_i v usporiadaní a Q_i je poradie hodnoty y_i v usporiadaní, určíme tieto poradia pre každú hodnotu x_i a y_i.,
krok3 - nech n je veľkosť štatistického súboru, Spearmanov korelačný koeficient vypočítame podľa vzorca

$$r_s = 1 - \frac{6}{n(n^2-1)} \sum_{i=1}^{n} (R_i - Q_i)^2 .$$

7.1.2 Výpočet Spearmanovho korelačného koeficientu za predpokladu, že ani v jednom súbore sa hodnoty neopakujú

Polia `X` a `Y` skopírujeme do polí `U_X` (usporiadané `X`) a `U_Y` (usporiadané `Y`), ktoré dáme usporiadať. Zavedieme polia `P_X` a `P_Y`, kde na í-tu pozíciu uložíme poradie í-teho prvku z `X` resp. `Y` v usporiadanom poli `U_X` resp. `U_Y`, vypočítanú pomocou funkcie `bsearch`. Funkcia `bsearch` vráti smerník na daný prvok, a teda ak chceme získať jeho index, musíme od neho odčítať adresu začiatku poľa.

Na výpočet kroku 1 použijeme funkciu **`qsort`**.
Na výpočet kroku 2 použijeme funkciu **`bsearch`**.
Predpokladajme, že robíme s dátami typu `double`.

```
#include <stdlib.h>
#define n 6

/* funckia pre qsort a bsearch*/
int sort_cisla(const void* prve,const void* druhe)
{double cislo1,cislo2;
cislo1=*((double *)(prve));
cislo2=*((double *)(druhe));

if (cislo1<cislo2) return (-1);
if (cislo1>cislo2) return (1);
return 0;
}

main()
{
int i,j,suma=0;
double *p;

double X[n]={23.1,12.8,17.8,21.3,18.5,93.5};
double Y[n]={25.9,15.1,20.4,23.5,21,105.9};

double U_X[n],U_Y[n];/*usporiadané polia X a Y*/
int P_X[n],P_Y[n];   /*poradia prvkov X a Y v usporiadaných
                     poliach*/

/*X,Y skopírujeme do  U_X a  U_Y a tie potom utriedime*/
for (i=0;i<n;i++)
{
 U_X[i]=X[i];
 U_Y[i]=Y[i];
}

qsort((void*)U_X,n,sizeof(double),sort_cisla);
qsort((void*)U_Y,n,sizeof(double),sort_cisla);

/*určenie poradí jednotlivých prvkov polí  X ,Y a ich uloženie do
P_X a P_Y*/

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

```
{p=(double *)bsearch((const void *)&X[i],
    (const void *)U_X,n,sizeof(double),sort_cisla);
P_X[i]=(p-U_X);}

for (i=0;i<n;i++)
{p=(double *)bsearch((const void *)&Y[i],
    (const void *)U_Y,n,sizeof(double),sort_cisla);
P_Y[i]=(p-U_Y);}

for (i=0;i<n;i++)
   suma=suma+(P_X[i]-P_Y[i])*(P_X[i]-P_Y[i]);

printf("Spearmanov korelacny koeficient je   %lf:\n",1-
        (double)(6*suma)/(n*n*n-n));
}
```

Zadávanie údajov pri inicializácii je nešikovné, pretože pri každej zmene dátových dvojíc treba program preložiť a zlinkovať. Výhodnejšie je dáta načítavať zo súboru. Keďže dĺžka statického poľa musí byť v ANSI C vopred známa, môžeme ju nastaviť na dostatočne veľké číslo, napr. 1000 a do súboru ako prvý údaj zadať počet dvojíc. Teraz už program po zmene aktualizovať netreba, treba len dbať o to, aby počet dvojíc nepresiahol 1000. Je zrejmé, že v mnohých prípadoch bude pole zbytočne veľké a naopak, môže sa stať, že sa dáta do statického poľa nezmestia.
V novších štandardoch C môže byť dĺžka poľa aj premenná a môžeme ju napríklad načítať z klávesnice. Veľa krát však potrebnú veľkosť poľa zistíme až z nejakého výpočtu. Elegantnejšie tento problém vyriešia *dynamické polia*, vysvetlené v jednom z ďalších odsekov tejto kapitoly. V jednej z nasledujúcich kapitol sa dozvieme aj to, ako dáta v súbore spočítať, t.j. ako sa vyhnúť zadávaniu prvého údaju – počtu dvojíc v súbore.

7.2 Smerníky a polia

Najprv zopakujme, čo sme už povedali v kapitole 6 o smerníkoch.
V C sú smerníky a polia veľmi úzko prepojené. Uvažujme nasledujúcu postupnosť príkazov:

```
...
int a[10]={1,2,3,4,5,6,7,8,9,10}, x;
int *pa;

pa = a;        /* pa ukazuje addresu a[0] */
 x = *pa;      /* x = obsah pa (v tomto prípade a[0]),teda 1 */
pa = a+1;      /* pa ukazuje addresu a[1] */
 x = *pa;      /* x = obsah pa+1 (a[1]),teda 2 */

/* vo vseobecnosti*/
for (i=0;i<10;i++)
   printf("%d %d\n",a[i], *(pa+i));
...
```

Všetky riadky budú obsahovať rovnaké dvojice hodnôt

Platí: `pa + i = &a[i]`

V C sa nekontrolujú hranice polí, a teda ľahko môžeme pri nesprávnom použití poprepisovať hodnoty v pamäti !!!

V C ďalej môžeme písať:

`pa = a` namiesto `pa = &a[0]`

`a + i` namiesto `&a[i]`

`a[i]` je to isté ako `*(a + i)`

`pa[i]` je to isté ako `*(pa + i).`

Medzi smerníkovou premennou a názvom poľa je však rozdiel v tom, že

- **Smerník je premenná. Môžeme vykonať** `pa = a` **a** `pa++.`
- **Názov poľa nie je premenná.** `a = pa` **a** `a++` **nie sú povolené.**

7.3 Dynamické polia

Statické polia majú svoju veľkosť presne určenú v okamihu prekladu zdrojového textu a počas behu programu sa nedá meniť[*)]. Pracuje sa s nimi jednoducho. Či však náš program potrebuje dát menej alebo viac, nedá sa s tým nič robiť. Preto často radšej volíme statické pole naddimenzované, aby náš program zvládol "väčšinu" úloh pre riešený typ problému. Zamýšľali sme sa nad tým v prvej časti pri výpočte Spearmanovho korelačného koeficientu pomocou statického poľa. Uveďme si ďalší príklad. Ak napíšeme program pre prácu s maticami postavený na statických poliach, musíme určiť ich dimenziu (rozmer). Dimenzia 3 x 3 asi nepostačí, tak radšej siahneme po dimenzii 10 x 10, alebo 20 x 20. Ak však budeme potrebovať maticu 21 x 21, program si s ňou už neporadí. Navyše kladieme na operačný systém vždy rovnaké, a to väčšinou premrštené požiadavky na operačnú pamäť. V jednoúlohovom systéme to obyčajne nevadí. Vo viacúlohovom, a ešte k tomu viacužívateľskom prostredí však takéto plytvanie nie je žiadúce.

Dynamické polia nemajú svoju veľkosť pri preklade určenú. Pri preklade sú vytvorené iba premenné vhodného typu *smerník na*, ktoré nám počas vykonávania programu slúžia ako pevné body pre prácu s dynamickými poliami. O požadovaný pamäťový priestor však musíme požiadať operačný systém (OS). Môže sa stať aj to, že OS našu žiadosť neuspokojí. V každom prípade však žiadame vždy len toľko pamäti, koľko sa počas behu programu ukázalo potrebné. Pri práci s dynamickými poliami vznikajú mnohé nebezpečenstvá, napríklad, že zabudneme o pamäť požiadať, alebo že chceme zapisovať do nepridelených priestorov alebo opakovane žiadame o jej prideľovanie a zabúdame ju uvoľňovať.

Aby časti programu mohli jednak žiadať o pridelenie dynamické pamäti a jednak už nepotrebnú pamäť mohli vrátiť, musí existovať aspoň základná programová podpora. Tu si však nebudeme vytvárať sami. Je definována ANSI C normou jazyka, a teda ju dostávame spolu s prekladačom. Deklarácie funkcií pre prácu s pamäťou sú umiestnené v `stdlib.h`, prípadne v `alloc.h`.

7.3.1 Funkcie pre prideľovanie pamäti

Štandardnou a najčastejšie používanou funkciou pre pridelenie pamäti je

[*)] Pozri dodatok

void *malloc(unsigned int *počet_bytov*);

a predstavuje požiadavku na pridelenie súvislého bloku pamäti o veľkosti počet_bytov.

V prípade úspešného pridelenia, dostávame smerník na void, ktorý predstavuje adresu prvého prideleného prvku.

Tento smerník je veľmi vhodné pretypovať na smerník na zodpovedajúci typ. V prípade, že pamäť nebolo možné prideliť, vráti funkcia hodnotu NULL. Pri zadávaní počtu bytov obyčajne využívame funkciu sizeof. Po úspešnom pridelení už môžeme s pamäťou pracovať ako s poľom, kde názov smerníka predstavuje názov poľa.

Príklad 1

```
...
main() {
int *p_i, suma, i , n;
scanf(„%d“,&n);
p_i = (int *)malloc(n * sizeof(int));
if (p_i==NULL) {printf(“málo pamäte\n”); return(1);}
for (i=0; i<n; i++)
    p_i[i]=0;
}
```

Spočiatku sa často stáva, že síce zadefinujeme smerník, ale zabudneme požiadať o pamäť. Smerník ostáva nenastavený (t.j. je nastavený náhodným spôsobom) a pri jeho použití chceme obyčajne zapisovať, alebo aspoň pristupovať do nepridelenej pamäti. Predstavuje to jedno z nebezpečenstiev pri práci s dynamickými poľami.

Ďalej si všimnite, že malloc je funkcia, a nie procedúra, ako by sa mohlo zdať na prvý pohľad podľa špecifikácie void v záhlaví. Návratový typ nie je void, ale smerník na void. Tento smerník treba následne pretypovať v závislosti od typu poľa.

void *calloc(unsigned int *n_prvkov*, **sizeof(***prvok***));**

Táto funkcia urobí to isté, ako by urobila funkcia malloc(n_prvkov*sizeof(prvok)), navyše je pridelená pamäť vyplnená nulami.

void *realloc(void *), ***p_block*, unsigned int** *nový_počet_bytov*);

Tento príkaz umožňuje zmeniť veľkosť alokovanej pamäte, na ktorú ukazuje p_block na novú veľkosť určenú hodnotou nový_počet_bytov. V prípade potreby (požiadavka je väčšia než pôvodný blok) je obsah pôvodného bloku prekopírovaný. Vracia smerník na nový blok.

7.3.2 Príkazy pre uvoľňovanie pamäti

Uvoľňovanie pamäti je akcia opačná, ako prideľovanie pamäti. Platí všeobecná zásada, že už nepotrebnú pamäť je dobré okamžite vrátiť a nečakať až na koniec programu.

void free((void *) *p_blok*);

Príkaz uvoľní pamäť, na ktorú ukazuje p_block. Dôležité je si uvedomiť, že procedúra free nemení hodnotu svojho parametra. To znamená, že smerník stále ukazuje na to isté miesto v pamäti. S touto pamäťou sa teda dá ďalej pracovať, ale v skutočnosti už programu nepatrí. Využívanie takejto pamäte môže potom spôsobiť množstvo problémov. Po príkaze

`free(void *p_blok)` je teda vhodné uviesť príkaz `p_blok=NULL`, čím zabránime možnému prístupu do uvoľnenej pamäte.
V niektorých systémoch je nutné pamäť pridelenú pomocou funkcie `calloc` uvoľniť pomocou procedúry `cfree`, a nie `free`.

7.4 Výpočet Spearmanovho korelačného koeficientu pomocou dynamických polí

Tento algoritmus sa od predchádzajúceho líši tým, že dáta sa nachádzajú v súbore a nie sú zapísané priamo v programe. Súbor sa volá `data` a nachádza sa v aktuálnom adresári. Najprv obsahuje údaj o veľkosti dát a potom samotné dáta, v každom riadku sú údaje jednej zložky. Použité sú dynamické polia, používame rôzne spôsoby prístupu k prvkom poľa. Oproti predchádzajúcemu príkladu je zaradených niekoľko kontrolných výpisov.

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

int sort_cisla(const void* prve,const void* druhe)
{double cislo1,cislo2;
cislo1=*((double *)(prve));
cislo2=*((double *)(druhe));
if (cislo1<cislo2) return (-1);
if (cislo1>cislo2) return (1);
return 0;
}

main()
{
FILE *f;
int i, j, n, suma = 0;
double *p;

double  *X, *Y, *U_X, *U_Y;
int *P_X, *P_Y;

f = fopen("data","r");
if (f == NULL){ printf("neotvoril sa subor\n"); return;}

fscanf(f,"%d", &n);  /*načítanie rozsahu súboru, t.j. počtu dvojíc*/
X   = (double*) malloc(n*sizeof(double));
Y   = (double*) malloc(n*sizeof(double));
U_X = (double*) malloc(n*sizeof(double));
U_Y = (double*) malloc(n*sizeof(double));
P_X = (int*) malloc(n*sizeof(int));
P_Y = (int*) malloc(n*sizeof(int));
/* tu by bolo treba skontrolovať smerníky*/

for (i=0; i<n; i++)
     fscanf(f,"%lf", X+i);

for (i=0; i<n; i++)
     fscanf(f,"%lf", Y+i);
```

```
for (i=0; i<n; i++)
      {
       U_X[i] = X[i];
       *(U_Y+i) = *(Y+i);
      }

/*usporiadanie a vypisanie x*/
for (i=0; i<n; i++)
     printf("%lf ",U_X[i]);
printf("\n");
qsort((void*)U_X,n,sizeof(double),sort_cisla);
for (i=0;i<n;i++)
     printf("%lf ",U_X[i]);
printf("\n");


/*usporiadanie a vypisanie y*/
for (i=0;i<n;i++)
     printf("%lf ",U_Y[i]);
printf("\n");

qsort((void*)U_Y,n,sizeof(double),sort_cisla);
for (i=0;i<n;i++)
     printf("%lf ",U_Y[i]);
printf("\n");
for (i=0;i<n;i++)
     {p = (double *)bsearch((const void *)(X+i),
          (const void *)U_X,n,sizeof(double),sort_cisla);
     P_X[i] = (p-U_X);}

for (i=0;i<n;i++)
     printf("%d ",P_X[i]);
printf("\n");

for (i=0;i<n;i++)
     {p = (double *)bsearch((const void *)(Y+i),
          (const void *)U_Y,n,sizeof(double),sort_cisla);
      P_Y[i] = (p-U_Y);}

for (i=0;i<n;i++)
     printf("%d ",P_Y[i]);
printf("\n");

for (i=0;i<n;i++)
   suma=suma+(P_X[i]-P_Y[i])*(P_X[i]-P_Y[i]);
printf("Spearmanov korelacny koeficient je   %lf:\n",1-
(double)(6*suma)/(n*n*n-n));
fclose(f);

free((void*) X);
free((void*) Y);
free((void*) U_X);
free((void*) U_Y);
free((void*) P_X);
free((void*) P_Y);
}
```

☑ Kontrolný test

1. Majme príkazy:
   ```
   int a[5]={11,12,13,14,15};
   ```
   Vyberte správne tvrdenia:

   a. `(a + 2) = 13;`
   b. `*(a + 1) = 12;`
   c. `*(a + 3) = a[3];`
   d. `a + 4 = &a[4];`

2. Majme definície:
   ```
   int a[5],*p_a,*p_b;
   ```
   Vyberte nekorektné priradenia:

   a. `p_a = a;`
   b. `p_b = p_a + 2;`
   c. `p_b = a + 3;`
   d. `a = p_b;`

3. Majme definície:
   ```
   short n = 5, int *u;
   ```
   Chceme, aby `u` bol smerník na `n`-prvkové dynamické pole s prvkami typu `short int`.
   Vyberte príkaz, ktorým sa vyhradí miesto pre `n` prvkové pole typu `short int` tak, že `u` bude ukazovať na začiatok tohto poľa:

   a. `u = (short int*) malloc(n);`
   b. `u = (* short int) malloc(2*n);`
   c. `u = (short int *) malloc(2*n);`
   d. `u = (short int *) malloc(sizeof(short int)*n);`

4. Nech je `p_a` smerník na `int`, ukazujúci na dynamické celočíselné pole dĺžky *n*. Pamäť tohto dynamického poľa bude korektne uvoľnená príkazom
   a. `free(n*p_a);`
   b. `free(&p_a);`
   c. `free(void p_a);`
   d. `free((void *) p_a);`

5. Vyberte správne tvrdenia:
   a. medzi dynamickým a statickým poľom nie ju žiadny rozdiel v prístupe k jednotlivým prvkom
   b. názov statického poľa je vlastne smerník ukazujúci na jeho začiatok
   c. počet prvkov dynamického poľa musí byť známy už v čase kompilácie
   d. veľkosť dynamického poľa sa v priebehu výpočtu nedá meniť.

6. Majme nasledovný dvojrozmerný štatistický súbor:
   X = {4,3,5,1};
   Y = {6,5,7,2};
   Spearmanov korelačný koeficient tohto štatistického súboru bude rovný:

a. 1
b. -1
c. 0
d. 0,5.

7. Vyberte správne tvrdenia:
   a. pre výpočet Spearmanovho korelačného koeficientu (SKK) musíme použiť statické polia takej dĺžky, koľko je dvojíc v štatistickom súbore.
   b. pre výpočet SKK môžeme použiť dynamické polia: program potom bude pracovať pre ľubovoľný počet dvojíc
   c. funkcia `qsort` sa dá použiť len pre statické pole
   d. funkcia `qsort` sa dá použiť len pre dynamické pole

8. Majme pole `X` typu `double`, obsahujúce 10 reálnych čísel typu `double` a porovnávaciu procedúru `porov` pre porovnanie dvoch reálnych čísel. Volanie funkcie `qsort`, ktorá toto pole utriedi, bude:

   a. `qsort(X,10, double,porov)`
   b. `qsort(X,10, sizeof(double),porov)`
   c. `qsort((double *)X,10, double,porov)`
   d. `qsort((void *) X,10, sizeof(double),porov)`

9. Majme pole `X` typu `int`, obsahujúce 20 čísel typu `int` a porovnávaciu procedúru `porov` pre porovnanie dvoch celých čísel. Volanie funkcie `qsort`, ktorá toto pole utriedi, bude:

   a. `qsort((int)X,20, int,porov)`
   b. `qsort(X,20, sizeof(int),porov)`
   c. `qsort((void *)X,20, int,porov)`
   d. `qsort((void *)X,20, sizeof(int),porov)`

10. Vyberte správne tvrdenia: pri výpočte Spearmanovho korelačného koeficientu (SKK) kópie U_X a U_Y pôvodných polí boli:
    a. získané príkazmi U_X=X a U_Y=Y
    b. vytvorené preto, lebo pôvodné polia X a Y by boli prepísané usporiadanými poliami, a pritom X a Y sú ešte potrebné pri utváraní poradí
    c. kópie polí vôbec nie sú potrebné
    d. vytvorené preto, lebo prvky pôvodných polí aj ich kópií sú využívané vo funkcii `bsearch`: funkcia `bsearch` vráti priamo index prvku poľa `X` v usporiadanom poli `U_X`.

11. Nech `p_i` je definované príkazom `int *p_i;` Nech premenná typu `int` zaberá v pamäti 4 byty. Pamäť pre desať prvkové dynamické pole s prvkami typu `int` získame príkazmi:
    a. `p_i=(int *)malloc(10) ;`

b. `p_i=(int *)malloc(40);`
c. `p_i=(void *)malloc(40);`
d. `p_i=(int *)malloc(10*sizeof(int));`

12. Nech `p_i` je smerník na desať prvkové dynamické pole s prvkami typu `int`. Na posledný prvok tohto poľa sa môžeme odvolať príkazmi:
a. `p_i[10];`
b. `*p_i+9;`
c. `*(p_i+10);`
d. `*(p_i+9);`

Kontrolné otázky

1. Charakterizujte význam Spearmanovho korelačného koeficientu.
2. Opíšte postup, akým sa počíta
3. Opíšte súvislosť medzi smerníkom a statickým poľom.
4. Názov poľa je smerník. Aké operácie sú preň povolené?
5. Charakterizujte rozdiel medzi statickým a dynamickým poľom.
6. Aké chyby môžete urobiť pri práci s dynamickým poľom?
7. Líši sa prístup k prvkom statického a dynamického poľa?
8. Presne opíšte postup vytvorenia dynamického poľa.
9. Opíšte funkcie pre prácu s pamäťou.
10. Ako zistíme, či sa podarilo prideliť pamäť?
11. Prečo je dobré po uvoľnení bloku pamäte funkciou `free` nastaviť smerník naň ukazujúci na `NULL`?

Cvičenia

1. Upravte úvodný príklad z tejto kapitoly tak, aby sa údaje načítavali zo súboru, pričom usporiadanie údajov je takéto:

n
x_1 y_1
x_2 y_2
…
x_n y_n

2. Prerobte príklad 3 z kapitoly 4 pomocou dynamického poľa.
3. Príklad 1 prerobte pomocou funkcie `calloc`.
4. Automobil šiel úsek dlhý 10 km rýchlosťou 60 km/hod., úsek dlhý 20 km rýchlosťou 60 km/hod. a úsek dlhý 30 km rýchlosťou 80 km/hod.

Použijeme vzorec: $v_p = \frac{s}{t} = \frac{\sum_{i=1}^{3} s_i}{\sum_{i=1}^{3} \frac{s_i}{v_i}}$

Funkciu napíšte tak, aby sa dala spočítať pre ľubovoľný počet úsekov.

5. Napíšte funkciu, ktorej argumentom bude *n*-rozmerný vektor a funkcia vráti jeho dĺžku. Rozmer *n* načítajte ako úvodnú informáciu a použite dynamické pole.
6. Načítajte dva *n* - rozmerné vektory reálnych čísel a vypočítajte ich skalárny súčin. Rozmer *n* načítajte ako úvodnú informáciu a použite dynamické pole.
7. Napíšte program, ktorý načíta maticu rozmerov *n* x *n*, v pamäti vytvorí transponovanú maticu a vypíše ju. Použite dynamické pole, rozmer *n* načítajte ako úvodnú informáciu. Použite prevod medzi jednorozmerným a dvojrozmerným poľom uvedený v časti 4.1.3.
8. Takzvané *Erastenovo sito* je jedna z najstarších metód získavania prvočísel. Algoritmus spočíva v nasledujúcom filtrovaní čísel:

   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... N

   Odstránime čísla väčšie ako 2, deliteľné dvoma:

   1 2 3 * 5 * 7 * 9 * 11 * 13 * 15 ... N

   Odstránime čísla väčšie ako 3, deliteľné troma:

   1 2 3 * 5 * 7 * * * 11 * 13 * * ... N

   Odstránime čísla väčšie ako 5, deliteľné piatimi,

   atď:

Napíšte program, ktorý vypočíta všetky prvočísla menšie ako N.

Zadania

1. Zistite, čo robí nasledujúci program.

```
...
int main()
{
  int i,j,n1;
  float s1,s2,rr;
  FILE   *vystup;
  char line[]="otazka";
  float *u,dist;

  printf("Zadaj rozmer pola:\n");
  scanf("%d",&n1);
  u=(float *) malloc(n1*n1*sizeof(float));

  s1=s2=n1/2.;
  rr=n1/3.;
  printf("rr=%f\n",rr);

  for (i=0;i<n1;i++)
    for (j=0;j<n1;j++)
      {
       dist=sqrt((i-s1)*(i-s1)+(j-s2)*(j-s2));
       if ((dist<rr-1)||(dist>rr+1))u[i*n1+j]=0;
       else u[i*n1+j]=1;
     }

 vystup=fopen(line,"w");
 for (i=0;i<n1;i++)
       {for (j=0;j<n1;j++)
          if (u[i*n1+j]==1) fprintf(vystup,"*");
          else fprintf(vystup," ");
        fprintf(vystup,"\n");
```

```
          }
fclose(vystup);
}
```

2. Vygenerujte do poľa *n* reálnych náhodných čísel v rozmedzí 0 až 10. Zistite

a) koľko z nich je menších ako 5
b) koľko z nich je v intervale <5,6>.
c)vypočítajte ich aritmetický priemer.

Rozmer *n* načítajte ako úvodnú informáciu a použite dynamické pole.

8 Reťazce. Práca s obrázkom

Hlavné podtémy kapitoly:

- Vytvorenie reťazca a jeho definícia. Statický a dynamický reťazec.
- Funckie pre prácu s reťazcami: `strcpy`, `strlen`, `strcat`, `strcmp`.
- Konverzia číslicového reťazca na číslo. Funkcie `sprintf` a `sscanf`.
- Práca s obrázkom. Načítanie a zápis riadku zo a do súboru. Spracovanie záhlavia obrázku.
- Parametre funkcie `main`: ich zadávanie pri spúšťaní programu z príkazového riadku.

8.1 Reťazce

Pre prácu s **reťazcami** (angl. **strings**) musíme dobre ovládať problematiku polí. Pre prácu s reťazcami sú kľúčové tieto dve skutočnosti:

1. v jazyku C sú reťazce implementovamé ako polia typu `char`
2. reťazce sú ukladané do pamäti tak, že za ich posledný znak sa pridá znak `'\0'` (myslí sa znak s ASCII kódom 0, t.j.znak na prvej pozícii v tabuľke ASCII).

Od samého začiatku musíme pamätať na to, že dĺžka reťazca je o jednotku dlhšia ako počet znakov v ňom obsiahnutých a s ukončovaciou nulou musíme počítať pri alokáciipamäte pre reťazec: ak potrebujeme uložiť päťznakový reťazec, v pamäti musíme vyhradiť šesť bytov, aby na konci reťazca mohla byť uložená ukončovacia nula.

Ak totiž neobsahuje reťazec na svojom konci znak `'\0'`, bude za reťazec považovaný celý nasledujúci úsek pamäte až po ďalší výskyt tohto znaku !!!

8.1.1 Vytvorenie reťazca

Vytvorenie reťazca vyzerá rovnako ako vytvorenie jednorozmerného poľa.

Statická verzia. Vytvorenie reťazca vyzerá rovnako ako vytvorenie jednorozmerného statického poľa:

```
char retazec_s[10];
```

Do takto definovaného reťazca môžeme uložiť maximálne 9 znakov.

Pri definícii môžeme reťazec inicializovať, napr. `char retazec_s[10]="abcdef";`
`char string1[]="ahoj";`
Druhá definícia, spojená s inicializáciou, definuje päťprvkové pole typu `char`, ktoré je súčasne nastavené, t.j. `string1[0]='a'; string1[1]='h'; string1[2]='o'; string1[3]='j'; string1[4]= '\0'.` Priradeniu hodnoty reťazcu v programe, nie pri inicializácii, sa budeme venovať v odseku 8.1.2. Zatiaľ si povedzme, že sa to nedá urobiť príkazom `string1="ahoj";`

Ak chceme získať reťazec z klávesnice, môžeme použiť funkciu `scanf` s formátovou špecifikáciou `"%s"`. Napr.

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

Ako už vieme z predchádzajúcej kapitoly, názov statického poľa predstavuje adresu jeho začiatku, a to je to, čo pri špecifikácii `"%s"` potrebujeme. Preto nepíšeme znak `&`.

Dynamická verzia. Podobne ako pri práci s ľubovoľným poľom, najprv musíme zadefinovať premennú pre smerník a potom alokovať pamäť. Keďže reťazec je smerník na pole typu `char`, zadefinujeme smerník na `char`. Nazveme ho `reťazec_d` a v pamäti vyhradíme 10 bytov, na ktoré bude ukazovať.

```
char *retazec_d;
retazec_d= (char *)malloc(10);
```

Do reťazca opäť načítame príkazom

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

Príklad 1

Doteraz sme pri otváraní súborov zadávali meno súboru v programe priamo v príkaze `fopen`. Teraz už máme prostriedky na to, ako meno súboru zadať počas vykonávania programu z klávesnice.

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

FILE *f;
char *meno_suboru;
meno_suboru = (char *)malloc(10);
printf("Zadaj meno súboru:");
scanf("%s", meno_suboru);
f = fopen(meno_suboru, "r");
...
```

8.1.2 Načítavanie reťazcov zo súboru

Načítavajme slovo za slovom z textového súboru s názvom `"DATA.TXT"`. Každé slovo vypíšme na obrazovku a na záver vypíšme počet slov v celom súbore. Slovo je chápané ako postupnosť znakov medzi dvoma *bielymi znakmi* (angl. *white spaces*): biele znaky sú významné znaky, ktoré nie je vidieť. Sú to tzv. oddeľovacie znaky ako medzera, tabulátor, nový riadok, nová stránka atď. Funkcia `scanf` aj `fscanf` načítava reťazce vždy len po biely znak, potom preskočí všetky biele znaky a hľadá začiatok ďalšieho reťazca. To je dôvodom, prečo nemusíme nijako zvláštne spracovávať oddeľovanie jednotlivých slov.

Príklad 2

```
#include <stdio.h>

#define MAX_SLOVO 80            /* maximálna dĺžka slova */
int main(void)                  /* uplna, neskratená forma */

{
  int spolu = 0;                /* počitadlo slov */
  FILE *fr;                     /* smerník  na súbor */
  char slovo[MAX_SLOVO + 1];    /* reťazec pre načítané slovo*/


  if ((fr = fopen("DATA.TXT", "r")) == NULL)
```

```
      {prinf("chyba otvorenia\n"); return;}
 while (fscanf(fr, "%s", slovo) != EOF)
    {
    printf("%s\n", slovo);
    spolu++;
    }
  printf("\n Súbor obsahuje %d slov.\n", spolu);
  fclose(fr);                    /* uzavretie súboru */
}
```

Dva vyznačené príkazy sú príkladom toho ako sa v C často spája viacero príkazov do jedného. Prvý vyznačený príkaz sme doteraz rozpisovali ako dva príkazy:

```
fr = fopen("DATA.TXT", "r");
if (fr==NULL) {printf....
```

V druhom vyznačenom príkaze sa nám môže zdať nepochopiteľným test `fscanf(fr, "%s", slovo) != EOF`.
Funkcia `fscanf` má návratový typ `int`, to znamená, že jej zavolanie vráti nejakú hodnotu, ktorú môžeme použiť. Táto hodnota predstavuje počet načítaných údajov.

Pri čítaní konca súboru sa automaticky vracia konštanta `EOF`, ktorá je väčšinou definovaná v `stdio.h` a má hodnotu -1.

Rozmyslite si, ako rozumným spôsobom zaradiť do programu výpis návratovej hodnoty funkcie `fscanf`. (Funkcia vždy načítavala jeden reťazec, teda mala by sa vypisovať hodnota 1 v prípade načítania reťazca a -1, ak funkcia prečítala hodnotu `EOF`.)

8.1.3 Priradenie hodnoty reťazcu

V odseku 8.1.1 sme upozornili, že priradenie hodnoty reťazcu sa nedá urobiť príkazom

```
retazec = "abcdef"; resp.
retazec_d = "abcdef";
```

ani v prípade statického ani v prípade dynamického reťazca.
Ak vytvoríme textovú postupnosť *abcdef*, vytvoríme vlastne reťazcovú konštantu. Táto konštanta je uložená niekde v pamäti, a má teda nejakú adresu. A práve túto adresu sa snažíme priradiť v priraďovacom príkaze. V prípade statického reťazca vyhlási chybu kompilátor, lebo vieme, že prepísať meno poľa nie je možné. V prípade dynamického reťazca stratíme 10 alokovaných bytov, lebo adresu ich začiatku prepíšeme adresou reťazca. **Na priradenie hodnoty reťazcu sa používa funkcia `strcpy`.** Aby sme ju mohli použiť, musíme vložiť hlavičkový súbor `<string.h>`.
Funkčný prototyp funkcie je

`char *strcpy(char *`*`s1`*`, char *`*`s2`*`);`
Skopíruje obsah reťazca `s2` do `s1`. Vráti smerník na prvý znak reťazca `s1`.
Obe hore uvedené priradenia budú teda správne fungovať, ak použijeme príkazy

```
strcpy(retazec, "abcdef");
strcpy(retazec_d, "abcdef");
```

8.1.4 Ďalšie štandardné funkcie pre prácu s reťazcami

`int strlen(char *`*`s`*`);`
Vráti dĺžku reťazca bez ukončovacieho znaku. Napríklad
`strlen("abcdef") = 6;`

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

Pripojí reťazec s2 k reťazcu s1. Vráti smerník na prvý znak reťazca s1.

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

Vráti 0, ak sú reťazce s1 a s2 rovnaké. Vráti záporné číslo, ak je s1 lexikograficky menšie ako s2 a kladné číslo v opačnom prípade. Lexikografické usporiadanie je ako usporiadanie v slovníku – podľa abecedy.

Funkcií pre prácu s reťazcami existuje viac. Všetky predpokladajú, že reťazce sú ukončené nulovým znakom. Pokiaľ potrebujeme reťazce spracovať inak, musíme ich spracovať „ručne“.

Príklad 3

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

int main()
{
    char string1[]="ahoj";
    char string2[5]="dovi";
    printf("ret1: %s,dlzka: %d\n",string1, strlen(string1));
    printf("ret2: %s\n ",string2);
    strcat(string1,string2);
    printf("ret1: %s, dlzka: %d\n",string1, strlen(string1));
}
```

Výsledok:

```
ret1: ahoj,dlzka: 4
ret2: dovi
ret1: ahojdovi, dlzka: 8
```

Pri poslednom príkaze si treba uvedomiť, že zreťazenie neprideľuje novú pamäť. Keby za reťazcom s1 v pamäti nasledovali trebárs riadiace premenné cyklu a reťazec by nebol dostatočne dlhý, chyba v programe by mohla byť veľmi závažná. To isté platí aj pre príkaz strcpy.

Príklad 4

Keď riešime úlohy s evolúciou (vývojom), riešenie sledujeme v rôznych momentoch vývoja. Výstupom býva často viacero súborov, ktorých meno získame z mena vstupného súboru pridaním čísla časového kroku. Napríklad majme vstupný súbor data, postupne budú vznikať súbory data_1,data_2, data_3 atď. Nasledujúci kód je príkladom toho, ako možno naprogramovať postupné vytváranie mien súborov.

```
main()
{
FILE   *f;
char   *meno,*koncovka,*zac_mena;
/* vyhradenie pamate pre retazce*/
meno=(char *)malloc(20);
zac_mena=(char *)malloc(15);
koncovka=(char *)malloc(6);
```

```
/*nacitanie mena suboru*/
printf("zadaj meno suboru:");
scanf("%s",zac_mena);
strcpy(meno,zac_mena);

/* tvorba mena v cykle */
for (i=1;i<100;i++)
 {sprintf(koncovka,"_%d",i);/*cislo koncovky zmenene na retazec*/
  strcat(meno,koncovka);
  f=fopen(meno,"w");
       …
  fclose(f);
  strcpy(meno,zac_mena); /* nachystane pre dalsie meno*/
  }
…
}
```

Funckia **sprintf** nebola v predchádzajúcom texte popísaná. Jej význam by mal byť jasný z príkladu: je rozdiel medzi číslom 1 a znakom 1 (zistite ASCII kód číslice 1). Táto funkcia prevedie číslo dané riadiacou premennou cyklu na znak prípadne znaky, ktoré sa spolu s podčiarknikom pridajú k menu súboru. Funkcia pracuje presne tak ako funkcia **printf**, ale výsledok sa nevypíše na obrazovlu, ale zapíše sa do reťazca. Podobne pracuje funkcia **sscanf** v príklade 6. Tá načíta z reťazca (podobne ako z terminálu) dva číslicové reťazce, skonvertuje ich podľa formátových špecifikácií a zapíše do premenných zadaných pomocou ich adries. Pri tvorbe prípony je niekedy výhodné použiť formátovú špecifikáciu **%03d,** ktorá vypíše celé číslo na 3 miesta a ak je kratšie, zvyšok **doplní nulami.**

8.1.5 Prístup k reťazcu znak po znaku

Ďalším spôsobom, ako nastaviť hodnotu reťazca je prístup znak po znaku. Vychádza z toho, že reťazec je pole znakov. Príklad si uvedieme aj na staticky aj na dynamicky definovanom reťazci.

Príklad 5

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

int main(void) {
int i;
char ret_s[20];                  /* statický reťazec */
char *ret_d;                     /* dynamický reťazec */

ret_d = (char *) malloc(20);     /* alokácia pamäte pre  dynamický
                                 reťazec */

for (i = 0; i < 19; i++)        /* priradnie hodnôt do oboch
   {                             reťazcov*/
    ret_s[i] = 'x';
    ret_d[i] = 'y'; }


  ret_s[19] = '\0';              /* uloženie ukončovacieho znaku*/
  ret_d[19] = '\0';              /* uloženie ukončovacieho znaku*/

  printf("%s  -  %s\n", ret_s, ret_d);     /* kontrolný  výpis */
```

```
  return;
}
```

8.2 Práca s obrázkami

Obrázkové dáta môžu byť reprezentované pomocou rôznych formátov. My budeme používať formát, pri ktorom obrázok nie je komprimovaný a je reprezentovaný dvojrozmernou tabuľkou obrázkových bodov - pixelov, ktorej rozmery sú dané šírkou a výškou obrázku. Každému obrázkovému bodu je priradená intenzita, t.j. úroveň šede v rozmedzí 0 až 255. Obmedzíme sa na čiernobiely obrázok. Práca s farebným nekomprimovaným obrázkom je podobná, do hry však vstupuje viacero kanálov – tabuliek s hodnotami úrovne šedej a veľa algoritmov je len jednoduchým zovšeobecnením. Budeme používať formát, ktorému zodpovedá prípona *pgm.* Súbor môže byť ASCII, t.j. textový alebo RAW, t.j.binárny. V tejto kapitole sa obmedzíme na textový formát a čiernobiele obrázky. Binárnemu formátu sa budeme venovať v časti 10.5. **Textový formát** v tomto prípade znamená, že hodnoty intenzít sú v súbore reprezentované znakovými reťazcami, t.j. reťazcami číslic, ktoré sa pri priraďovaní do premenných konvertujú na čísla. Doteraz sme to vlastne tak robili: funkcia `scanf` alebo `fscanf` načítala znakový reťazec, ktorý bol podľa formátovej špecifikácie konvertovaný na číslo a zapísaný do premennej, ktorej adresa bola uvedená.

Konverziu ľubovoľného obrázkového formátu na tento formát môžeme v linuxe získať napríklad programom *xv* a *gimp* a v OS Windows napr. aplikáciou *Photoshop*, príp. *gimp*. Z Internetu možno zadarmo získať aj rozširenú aplikáciu *Irfanview*, ktorej inštalácia je veľmi jednoduchá. Okrem samotných hodnôt intenzít obrázky obsahujú textové záhlavia, ktoré dodržujú určitý formát.

Majme obrázok napr. *kvety.jpg*. Konverzia v programom *gimp* (verzia 2.8 a viac) pozostáva z nasledujúcich krokov:

1. otvorme obrázok *kvety.jpg*
2. vyberme z menu *Súbor* /*Export as*
3. nastavme formát na *pgm,* kliknime na *OK* a z vyskakovacieho menu vyberme *ascii* (nie *binary*)

Ak používame iný program, ktorý vie robiť konverziu na tento formát, pričom sa meno formátu nevyskytuje v ponuke, skúsme rovno napísať príponu *.pgm.* Veľa programov vie vybrať výstupný formát podľa prípony. Čiernobiely *textový (ascii)* formát má prvý reťazec P2. Ak sa namiesto P2 vyskytuje P3, znamená to, že obrázok sa zakódoval ako farebný, a nie ako čiernobiely.

8.2.1 Záhlavie obrázku

Ako príklad uvádzame vzorové záhlavie, ktoré patrí k textovému čiernobielemu *pgm* - obrázku, spracovanom programom *xv.* P2 označuje spôsob kódovania (textový čiernobiely), ďalej nasleduje riadok s komentárom, začínajúci s **#,** môže ich byť aj viac. Dvojica čísel udáva rozmer obrázku a tretie číslo maximálnu intenzitu obrázku. Ďalej už nasledujú údaje o intenzite.

```
P2
# CREATOR: XV version 3.10a-jumboFix+Enh of 20040523
# CREATOR: XV version 3.10a-jumboFix+Enh of 20040523
440 601
255
85  70  56  48  41  37  39  46  35  39  42  42  40  39  43  47
34  35  42  42  37  35  38  40  38  39  42  41  40  43  47  36
 ...
```

Pod spracovaním záhlavia budeme rozumieť jeho načítanie po prvý údaj o intenzite a vyčítanie potrebných údajov, napr. rozmerov obrázka. Užitočnú funkciu, ktorá je nová, predstavuje funkcia pre načítanie celého riadku.

8.2.2 Čítanie riadku zo súboru – funkcia `fgets`

Funkcia pre vstup riadku zo súboru má funkčný prototyp

`char* fgets(char *`*`str`*`, int `*`max`*`, FILE `*`fr`*`),`

kde `str` je smerník na znakové pole, do ktorého bude riadok uložený a `max` je maximálny počet znakov, ktorý bude načítaný zo súboru `fr`. **Načítava sa aj znak konca riadku `'\n'`.** Funkcia vracia alebo smerník na `str`, alebo po dosiahnutí konca súboru vracia `NULL`. Nasledujúci príklad je príkladom načítania záhlavia zo súboru a jeho opätovného zapísania po spracovaní obrázku. Komentár nezapíšeme. Pre jednoduchosť definujme pole pre obrázok staticky a predpokladajme, že vyhradená pamäť je dostatočne veľká. Vstupný obrázok sa bude volať *kvety* a výstupný *kvety_vystup*.

Príklad 6

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define v_max 1000                  /*maximálna výška obrázku*/
#define s_max 1000                  /*maximálna šírka obrázku*/
...
char pom1[5];
char pom2[80]="";
int a,s,v,i,j;
FILE *vstup,*vystup;
unsigned char  pic[v_max][s_max];
...

vstup=fopen("kvety","r");
vystup=fopen("kvety_vystup","w");

fgets(pom1,5,vstup);              /*načítavanie 1.riadku záhlavia*/

do{
  fgets(pom2,80,vstup);
  } while(pom2[0]=='#');

sscanf(pom2,"%d %d",&s,&v);
fscanf(vstup,"%d",&a);

for (i=0;i<v;i++)
    for (j=0;j<s;j++)
      fscanf(vstup,"%d",&pic[i][j]);

...
fprintf(vystup,"%s",pom1);
fprintf(vystup,"%d %d\n%d\n",s,v,a);

for (i=0;i<v;i++)
   for (j=0;j<s;j++)
     fprintf(vystup,"%d ",pic[i][j]);
fclose(vystup);
```

```
}
```

V programe je tučným písmom vyznačená časť, kde sa načítava a zapisuje záhlavie. Prvý riadok sa načíta do znakového poľa `pom1`, potom sa komentár/viacero komentárov postupne načíta do reťazca `pom2`. Cyklus skončí vtedy, keď už nejde o komentár, ale dvojicu čísel udávajúcich rozmer obrázku, ktoré sú po jeho skončení uložené v `pom2`. Tieto čísla sú však ešte stále znakové reťazce a na čísla ich treba skonvertovať. Na to môžeme použiť funkciu `sscanf`.

Funkcia pre zápis riadku do súboru má funkčný prototyp

`char *fputs(char *str, FILE *fr).`

8.3 Parametre funkcie `main`

Doteraz sme používali funkciu **`main`** ako funkciu bez parametrov s implicitnou návratovou hodnotou `int`. Ak má funkcia **`main`** parametre, sú z historických dôvodov pomenované vždy `argc` a `argv`. Ich účelom je predať programu parametre z príkazového riadku. Ide teda o parametre, s ktorými bol spustený program. Napríklad príkazom:

`nasob` `matice vysledok`

môžeme spustiť program `nasob` pre násobenie matíc, pričom `matice` bude súbor, kde sú uložené matice, ktoré treba vynásobiť a `vysledok` je súbor, do ktorého sa uloží matica výsledku. V IDE DevCpp sa parametre (bez mena programu) zadávaju v menu do *Execute/Parameters* (pozri ďalej).

Ak to chceme urobiť takto, funkcia main musí mať záhlavie

`main(int argc, char* argv[]),`

kde

`argc` `bude` udávať počet reťazcov na vstupnom riadku

`argv` je pole smerníkov na reťazce v príkazovom riadku, ktoré je v prípade **`nasob`** `matice vysledok` bude nastavené takto:

- `argv[0]` bude ukazovať na reťazec `nasob`
- `argv[1]` bude ukazovať na reťazec `matice`
- `argv[2]` bude ukazovať na reťazec `vysledok`.

Príklad 7

Pri spustení chceme zadať dva parametre, ako bolo uvedené v predchádzajúcom prípade. Budú predstavovať meno vstupného súboru a meno súboru, do ktorého sa zapíše výsledok. Hodnoty `argc` aj `argv` budú nastavené po spracovaní príkazového riadku.

Riešenie:

```
#include <stdio.h>

int  main(int argc, char* argv[]){

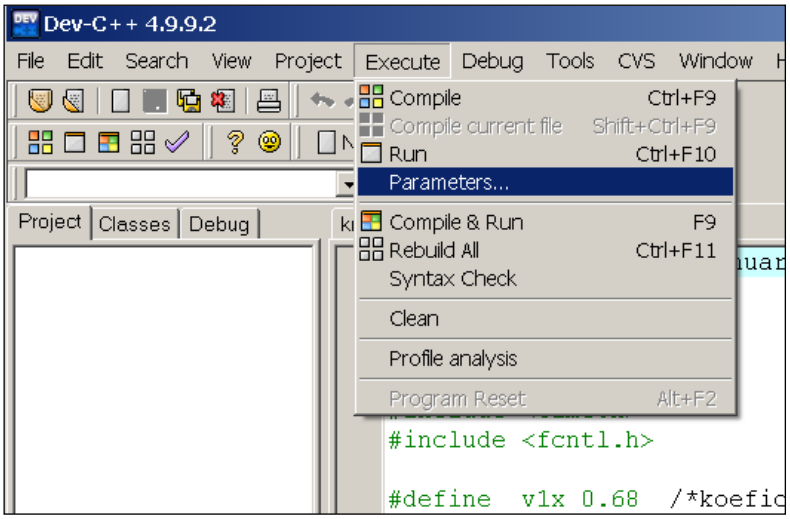
FILE *vstup,*vystup;
if(argc != 3)
    {printf("nesprávny počet argumentov");return;}
```

```
else
    {
     vstup=fopen(argv[1],"r");
     if (vstup==NULL) {printf("neotvoril sa subor1\n");return;}

vystup=fopen(argv[2],"w");
if (vystup==NULL) {printf("neotvoril sa subor12");return;}}

...

 fclose(vstup);
 fclose(vystup);
}
```



Ak spúšťame programy z príkazového riadku, argumenty zadávame za meno programu tak, ako bolo uvedené v príkladoch. Ak robíme v iných programoch alebo vývojových prostrediach, treba pohľadať v ponuke príkaz, ktorý zadávanie parametrov umožňuje. Obrázok uvádza príklad, ako je možné zadať parametre pri spúšťaní programu v prostredí Dev-C++.

☑ Kontrolný test

1. Vyberte nesprávne tvrdenia:
   a. reťazce sa ukladaj[do pamäte tak, že za ich posledný znak sa pridá znak `'\0'`
   b. reťazec `"data.txt"` môžeme uložiť do znakového poľa dĺžky 8
   c. ak neobsahuje reťazec na svojom konci znak '\0', bude za reťazec považovaný celý nasledujúci úsek pamäte až po ďalší výskyt tohto znaku.
   d. reťazec môže byť vytvorený aj ako statické aj ako dynamické pole.

2. Napíšte, ktoré príkazy predstavujú správnu definíciu a inicializáciu reťazca:
   a. `char retazec[]="ahoj";`
   b. `string retazec[]="ahoj";`
   c. `string retazec[4]="ahoj";`
   d. `char retazec[10]= "abcdef";`

3. Majme korektne definovaný reťazec ako pole `retazec` typu `char` dĺžky 10. Vyberte správne tvrdenia:
   a. ak chceme získať reťazec z klávesnice, môžeme použiť funkciu `scanf` s formátovou špecifikáciou `"%c"`, napr. `scanf("%c", retazec);` načítaný reťazec môže mať najviac 10 znakov;

b. ak chceme získať reťazec z klávesnice, môžeme použiť funkciu `scanf` s formátovou špecifikáciou "`%c`", napr. `scanf("%c", retazec);` načítaný reťazec môže mať najviac 11 znakov;
c. ak chceme získať reťazec z klávesnice, môžeme použiť funkciu `scanf` s formátovou špecifikáciou "`%s`", napr. `scanf("%s",&retazec);` načítaný reťazec môže mať najviac 10 znakov;
d. ak chceme získať reťazec z klávesnice, môžeme použiť funkciu `scanf` s formátovou špecifikáciou "`%s`", napr. `scanf("%s",&retazec);` načítaný reťazec môže mať najviac 9 znakov;

4. Majme reťazec `ret` definovaný príkazom `char ret[10].` Do reťazca chceme priradiť text `"vystup"`. Vyberte príkazy, ktorými je to možné urobiť.
   a. `ret = "vystup";`
   b. `&ret = "vystup";`
   c. `*ret = "vystup";`
   d. `strcpy(ret,"vystup");`

5. Majme reťazec `ret` definovaný príkazom `char ret[10].` Vyberte správne tvrdenie. Príkaz `ret = "superstar"`:
   a. vloží 9 znakov do poľa `ret;`
   b. kompilátor vyhlási chybu;
   c. prepíše sa adresa poľa `ret;`
   d. čo sa stane závisí od veľkosti definovaného poľa;

6. Majme nasledujúcu definíciu:
```
char *retazec_d;
retazec_d = (char *)malloc(10);
```
   Napíšte, ktorým z uvedených spôsobov môžeme do pamäti uložiť od adresy `retazec_d` textovú postupnosť "`superstar`":
   a. `retazec_d = "superstar";`
   b. `strcpy(retazec_d, superstar);`
   c. `strcpy(retazec_d, "superstar");`
   d. `strcpy(retazec_d, &superstar);`

7. Majme takúto postupnosť príkazov:
```
char *ret1, *ret2, *ret3;
ret1 = (char *)malloc(30);
ret2 = ret1 + 7;
ret3 = ret2 + 5;
strcpy(ret1, "Martin");
strcpy(ret2, "Pyco");
strcpy(ret3, "Rausch");
strcat(ret1, ret3);
```
   Príkaz `printf("%s",ret2)` vypíše reťazec:
   a. Pyco;
   b. Martin;
   c. Rausch;
   d. ausch;

8. Majme takúto postupnosť príkazov:
```
char *retazec;
int i=100;
retazec = (char *)malloc(20);
sprintf(retazec, "Projekt %d",i);
```
Príkaz
`printf("%s",retazec);` vypíše:

   a. Projekt100;
   b. Projekt 100;
   c. Projekt i;
   d. príkaz môže zapríčiniť zhavarovanie programu;

9. Majme takéto príkazy:
```
char ret1[] = "zeny 100 muzi 200";
char ret2[5];
int i, j;
```
Po vykonaní ríkaz
`sscanf(ret1, "%s%d%s%d",ret2,&i,ret2,&j);` budú pravdivé tieto tvrdenia:
   a. `ret1 = "muzi" , ret2 = "zeny"`
   b. (i + j) / 0.5 = 150
   c. i = "100", j = "200"
   d. `ret2 = "muzizeny".`

10. Majme krátky textový súbor nazvaný `"múdrosti"`, v ktorom je uložený takýto text:
```
Kto chce málo, dostane ešte menej.
Dnes ma konečne navštívilo šťastie. Prišlo sa rozlúčiť.
```
Majme príkazy:
```
char pom1[80], pom2[80]; FILE *f;
f = fopen("múdrosti", "r",);
```
Do reťazca `pom1` chceme zapísať 1.riadok, do `pom2` 2.riadok.
Urobíme to príkazmi:
   a. `fgets(pom1,80,f); fgets(pom2,80,f);`
   b. `fgets(f,80,pom1); fgets(f,80,pom2);`
   c. `fgets(pom1,strlen(pom1),f); fgets(pom2, strlen(pom2),f);`
   d. `fgets(pom1,80, "múdrosti"); fgets(pom2,80, "múdrosti");`

11. Program s menom `oprav` chceme spustiť s dvoma parametrami `obr1` a `obr2`, ktoré budú predstavovať meno vstupného a výstupného súboru. Príkazový riadok pri spustení programu teda bude `oprav obr1 obr2`. Správne spustenie programu čo sa týka počtu parametrov skontrolujeme príkazom:
   a. `if (argc != 2) {printf("nesprávny počet...;`
   b. `if (argc != 3) {printf("nesprávny počet...;`
   c. `if (argv != 2) {printf("nesprávny počet...;`
   d. `if (argv != 3) {printf("nesprávny počet...;`

12. Majme záhlavie hlavnej funkcie `main` definované príkazom
`int main(int argc,char* argv[]).` Vyberte správne tvrdenie:

   a. `argv` je pole smerníkov na reťazce a `argc` je počet zapísaných smerníkov,
   b. `argv` je pole smerníkov na reťazce, `argc` je maximálna dĺžka reťazca,

c. namiesto `argv` a `argc` by sme mohli použiť aj iné názvy, ale nerobí sa to,
d. ak okrem mena programu zadáme ešte meno jedného súboru, `argc=1`.

Kontrolné otázky

1. Opíšte proces vytvorenia statického reťazca. Akým spôsobom mu možno priradiť konkrétnu znakovú postupnosť?
2. Opíšte proces vytvorenia dynamického reťazca. Akým spôsobom mu možno priradiť konkrétnu znakovú postupnosť?
3. Ako zistíme pri načítavaní zo súboru pomocou funkcie `fscanf`, že sme na konci súboru?
4. Prečo sa priradenie hodnoty statickému reťazcu `retazec` nedá urobiť príkazom `retazec="Agent 007"`?
5. Opíšte prácu príkazov `strcpy` a `strcat`. Zamyslite sa ako je to s pridelenou pamäťou. Na čo si treba dávať pozor?
6. Funkcia `strlen` vráti dĺžku reťazca, nech je táto dĺžka rovná `x`. Stačí pre prácu s týmto reťazcom `x` bytov?
7. Opíšte príkazy `sprintf` a `sscanf`. Ktorým príkazom z predchádzajúcej kapitoly sa najviac podobajú?
8. Ako sa pracuje s reťazcami znak po znaku?
9. Povedzte všetko, čo viete o obrázku a o formáte *pgm*. Ako vyzerá záhlavie tohto formátu?
10. Podrobne vysvetlite každý riadok z príkladu 6.
11. Ako môžeme do programu zadať parametre zo vstupného riadku? Opíšte, ako je to možné urobiť v jednotlivých prostrediach.

Cvičenia

1. Premyslite, ako prerobiť príklad 6 tak, aby sa uchovali všetky komentáre.
2. Majme tri krátke reťazce. Napíšte funkciu, ktorá ich ztreťazí do jedného a vráti smerník na výsledný reťazec. Zamyslite sa nad prácou s pamäťou.
3. Spočítajte počet riadkov zadaného textového súboru a vypíšte ich.
4. Načítavajte riadky textového súboru a postupne vypíšte, koľko je v každom riadku núl**.**
5. Načítavajte riadky textového súboru a postupne vypíšte, koľko je v každom riadku číslic**.**
6. Načítavajte riadky textového súboru a postupne vypíšte, koľko je v každom riadku malých písmen**.**
7. Načítavajte riadky textového súboru a postupne vypíšte, koľko je v každom riadku písmen `a` alebo `A`**.**
8. Príklady 1, 5, 6 a 7 z predchádzajúcej kapitoly prerobte tak, aby v dátovom súbore nebol použitý údaj `n` o rozmeroch súboru.

Zadania

1. Zistite, čo robí nasledujúci program:

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

```
#include <stdio.h>

int min(int a, int b)
{if (a<=b)
    return a;
  else
    return b; }

main(){
char *pom1,*pom2;
int  a,i,s,v,j,in,K;
FILE *vstup, *vystup;

vstup=fopen("subor","r");
vystup=fopen("vystup","w");
pom1=(char *)malloc(5);
pom2=(char *)malloc(80);

printf("zadaj hodnotu od 1 do 50:");
scanf("%d",&K);

fgets(pom1,5,vstup);
fputs(pom1,vystup);

do{
  fgets(pom2,80,vstup);
  fputs(pom2,vystup);
  }
  while(pom2[0]=='#');
sscanf(pom2,"%d%d",&s,&v);
fgets(pom2,80,vstup);
fputs(pom2,vystup);
sscanf(pom2,"%d",&a);

for (i=0;i<v;i++)
    for (j=0;j<s;j++)
      {fscanf(vstup,"%d",&in);
        fprintf(vystup,"%d ",min(in+K,255));}

fclose(vstup);
fclose(vystup);
}
```

2. Napíšte program na porovnanie dvoch súborov, ktorý vytlačí prvý riadok a pozíciu znaku, v ktorom sa líšia.

9 Dvojrozmerné dynamické polia

Hlavné podtémy kapitoly:

- Definícia poľa smerníkov. Polostatické a dynamické dvojrozmerné polia.
- Definícia poľa reťazcov. Statické, polostatické a dynamické polia reťazcov.
- Rôzne spôsoby načítavania dvojrozmerného poľa. Načítavanie obrázku.
- Použitie funkcie na výpis systémového chybového hlásenia `perror` pri otvorení súboru.
- Použitie príkazu `fseek` na návrat na začiatok súboru bez jeho uzavretia a znovuotvorenia.
- Dvojrozmerné pole ako parameter funkcie.Vykonanie príkazu OS pomocou príkazu `system`.

9.1 Pole smerníkov

V závere predchádzajúcej kapitoly sme sa stretli s pojmom *pole smerníkov na reťazec* – príkladom takého poľa bolo `argv`. Bolo to pole, ktoré obsahovalo adresy. Uveďme si niekoľko príkladov polí smerníkov rôznych typov.

Príklad 1

```
...
int i = 1, j = 2, k =3;
int *pole1[3];
pole1[0] = &i;
pole1[1] = &j;
pole1[2] = &k;
printf("%p %p %p\n",pole1[0], pole1[1], pole1[2]);
printf("%d %d %d\n", *pole1[0],*pole1[1], *pole1[2]);
...
```

V tomto príklade sme definovali trojprvkové pole smerníkov na celé čísla.

Príklad 2

```
...
char A[3]={1,1,1};
char B[3]={2,2,2};
char C[3]={3,3,3};
char *pole2[3];
pole2[0] = A;
pole2[1] = B;
pole2[2] = C;
```

Takýmto spôsobom vlastne dostávame dvojrozmerné pole. Druhý prvok poľa `A` dostaneme príkazom (`pole2[0])[1] = pole2[0][1];`

```
...
printf("%p %p %p\n",pole2[0], pole2[1], pole2[2]);
printf("%d %d %d\n", pole2[0][1],pole2[1][1], pole2[2][1]);
...
```

Polia smerníkov môžu byť statické aj dynamické, prípadne môžu byť kombináciou statického a dynamického poľa. Poliami smerníkov sa budeme zaoberať v nasledujúcich odsekoch.

9.2 Dvojrozmerné dynamické polia

9.2.1 Polostatické dvojrozmerné pole

Majme dvojrozmernú tabuľku prvkov. Vezmime si situáciu, keď máme vopred známy počet riadkov a až programom určený počet stĺpcov. V tomto prípade jeden rozmer poľa budeme definovať staticky a druhý dynamicky. Nech programom určená šírka riadku bude uchovaná v premennej `n`.

Zadefinujeme si dvojrozmerné celočíselné pole takto:

```
int  *pole[4];
```

Štvorprvkové pole `pole` bude pole smerníkov na `int`. Každý z týchto smerníkov nastavíme na pridelenú pamäť získanú funkciou `malloc`. Riadky nemusia byť uložené v pamäti za sebou.

Príklad 3A

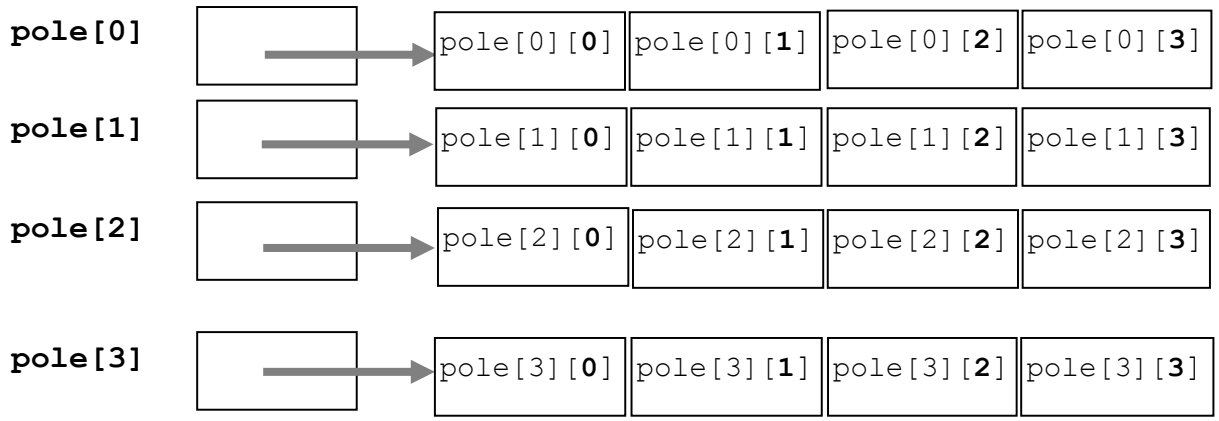
```
/* zadefinovanie polostatického celočíselného poľa*/
/* riadky nemusia nasledovať za sebou*/
int *pole[4];
int i,n;
...
for (i=0; i<4; i++)
  pole[i] = (int *)malloc(n*sizeof(int));
```

V prípade, že by sme chceli, aby celé dvojrozmerné pole bolo v pamäti uložené v súvislej oblasti, mohli by sme postupovať takto:

Príklad 3B

```
/* zadefinovanie polodynamického celočíselného poľa*/
/* riadky budú nasledovať za sebou*/
int *pole[4];
int i;
pole[0] = (int *)malloc(4*n*sizeof(int));
for (i=1; i<4; i++)
  pole[i] = pole[0] + i*n;  /*príp. pole[i] = pole[i-1]+n;*/
```

Z nasledujúceho obrázku vidíme, že k prvkom poľa môžeme pristupovať pomocou dvoch indexov ako pri statickom dvojrozmernom poli.



V prípade príkladu 3B, keď v pamäti nasledujú prvky za sebou a aj v prípade statického dvojrozmerného poľa, môžeme prvky načítať, vypísať, prípadne v niektorých prípadoch aj spracovať pomocou jedného indexu. Napríklad v prípade dvojrozmerného statického poľa sa vnútorne prepočítavajú dva indexy na jeden (pozri kapitolu 2), čo vyžaduje násobenie a sčítanie, a v prípade jedného indexu sa robí iba jedna inkrementácia (pozri príklad v závere kapitoly). Pri definícii z príkladu 3B by sa možno mohlo zdať, že ak `pole` predstavuje adresu začiatku poľa, k jeho prvkom môžeme pristupovať jednoducho takýmto spôsobom:

```
/* výpis  poľa*/
for (i=0; i<4*n; i++)
  printf("%d", pole[i]);
```

Nie je to však tak. Nejde to preto, lebo `pole[i]` je aj v prípade polostatického, aj v prípade statického dvojrozmerného poľa *smerník,* a nie hodnota. V prípade príkladu 3B by sme vypísanie dvojrozmerného poľa pomocou jedného indexu mohli urobiť takto:

```
/* zadefinovanie polodynamického celočíselného poľa*/
/* riadky nasledujú  za sebou*/
int *pole[4];
int i,*p,n;
...
pole[0] =(int *)malloc(4*n*sizeof(int));
for (i=1; i<4; i++)
  pole[i]= pole[0]+i*n;
...
/* výpis  poľa*/
p = (int *)pole;
for (i=0; i<4*n; i++)
  printf("%d ", p[i]);
```

Nie je však celkom jasné, či práca s jedným indexom ušetrí čas, mnohé kompilátory kód optimalizujú. Odporúčame vyskúšať príklad 5, ktorý používa oba spôsoby a súčasne meria čas.

Nevýhoda prístupu uvedeného pri polostatickom poli v porovnaní so statickým poľom je v tom, že okrem pamäte pre samotné pole potrebujeme ešte pamäť pre smerníky ukazujúce na začiatok riadkov.

Dealokácia pamäte. V príklade 3A môžeme dealokovať len pamäť pre jednotlivé riadky príkazmi:

```
for (i=0; i<4; i++)
  free((void*)pole[i]);
```

9.2.2 Dynamické dvojrozmerné pole

V predchádzajúcom prípade sme mali vlastne statické pole smerníkov ukazujúcich na začiatky riadkov. Keď počet riadkov nie je známy, mohli by sme zo smerníkov ukazujúcich na začiatky riadkov vytvoriť dynamické pole. Obsahom tohto poľa sú smerníky na `int`, a teda smerník ukazujúci na začiatok poľa je *smerník na smerník* na `int`,

```
(int *) *pole;
```

Skrátene sa však takáto definícia zapisuje ako

```
int **pole;
```

Pre pole musíme alokovať pamäť, ktorá zodpovedá *m* smerníkom na `int`, kde *m* zisíme až v priebehu výpočtu.

```
/*definicia pola rozmerov m x n*/
 int m, n, i, **pole;
/*popridelovanie pamate a nastavenie smernikov*/
pole=(int**)malloc(m*sizeof(int *));
...
for (i=0;i<m;i++)
  pole[i]=(int *)malloc(n*sizeof(int));
```

Takto alokovanú pamäť uvoľňujeme príkazmi:

```
for (i=0; i<m; i++)
  free((void*)pole[i]);
free((void*)pole);
```

Príklad 4

Majme zadané matice reálnych čísel ľubovoľných rozmerov, ale tak, že sa dajú navzájom násobiť. Naprogramujme násobenie matíc pomocou dynamického dvojrozmerného poľa. Prvok s indexami `i` a `j` výslednej matice dostaneme ako skalárny súčin `i`-teho riadku matice `A` a `j`-teho stĺpca matice `B`.

```
/*majme maticu A = [a_ik]_m x n a maticu B = [b_kj]_n x r,*/
/*potom súčin matice [A]a matice [B] je: A x B = C, kde C = [c_ij]_mxr*/
...
int m,n,r, i,j;
double **sucin;
m=...
n=...
r=...
...
/*vzpocet vyslednej matice C = [c_ij]_m x r*/
sucin = (double **) malloc(m*sizeof(double *));
for (i = 0; i < m; i++)
  sucin[i] = (double *) malloc(r*sizeof(double));

for (i = 0; i < m; i++)
  for (j = 0; j < r; j++)
   {
    sucin[i][j] = 0;
    for (k = 0; k < n; k++)
      sucin[i][j] += a[i][k]*b[k][j];
   }
...
```

Ako sme povedali v závere odseku 4.1 a aj v tejto kapitole, dvojrozmerné pole môžeme reprezentovať pomocou jednorozmerného. Ak by sme mali program s dvojrozmerným statickým poľom a chceli by sme ho prerobiť na program s dynamickými poliami, je jednoduchšie prerobiť ho niektorým zo spôsobov, uvedených v tejto kapitole, pretože stačí prerobiť len definície a prístup k prvkom zostáva rovnaký.

9.3 Polia reťazcov

Pole reťazcov je príklad poľa, ktorého prvky sú síce rovnakého typu - reťazce, ale nemusia mať rovnakú dĺžku. Pole reťazcov môžeme opäť definovať troma spôsobmi: ako statické, „polostatické“ a dynamické.

9.3.1 Statické polia reťazcov

Majme 5 reťazcov, o ktorých vieme, že ich maximálna dĺžka je 10. Utvoriť z nich pole môžeme nasledujúcimi príkazmi:

```
char pole_ret[5][10];
strcpy(pole_ret[0],"Zuza");
strcpy(pole_ret[1],"Marka");
strcpy(pole_ret[2],"Katrena");
strcpy(pole_ret[3],"Ferino");
strcpy(pole_ret[4],"Jan");
...
```

Vypísať adresy, na ktorých začínajú jednotlivé reťazce by sme mohli príkazmi:

```
for (i=0; i<5; i++)
  printf("%p\n ", pole_ret[i]);
```

Vypísať reťazce by sme mohli príkazmi:

```
for (i=0;i<5;i++)
  printf("%s\n", pole_ret[i]);
```

Vypísať prvý znak z každého reťazca by sme mohli príkazmi:

```
for (i=0;i<5;i++)
  printf("%c ", pole_ret[i][0]);
```

Príklad 5A

Pole reťazcov `pole_ret` usporiadajme podľa abecedy.

Riešenie:

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

int sort_meno(const void* prvy,const void* druhy)
{
return(strcmp((char*)prvy, (char*)druhy));
}

main(){
char pole_ret[5][10]; int i;
strcpy(pole_ret[0],"Zuza");
strcpy(pole_ret[1],"Marka");
strcpy(pole_ret[2],"Katrena");
strcpy(pole_ret[3],"Jurino");
strcpy(pole_ret[4],"Jan");

/* vypis povodnych retazcov*/
for (i=0; i<5; i++)
  printf("%s\n", pole_ret[i]);

qsort((void*)pole_ret,5,sizeof(meno),sort_meno);

/* vypis zoradenych  retazcov*/
```

```
printf("\n\n zoradene:\n");
for (i=0; <5; i++)
  printf("%s\n", pole_ret[i]);
}
Vypis z programu:
Zuza
Marka
Katrena
Jurino
Jan

Zoradene:
Jan
Jurino
Katrena
Marka
Zuza
```

9.3.2 Polostatické polia reťazcov

9.3.2.1 Pevná dĺžka reťazcov, neznámy počet riadkov

Najprv si vezmime situáciu, keď máme pevnú dĺžku reťazca, ale neznámy počet riadkov. Oproti predchádzajúcej podsekcii sa zmení málo. Zadefinujeme si nový typ pre riadok s menom: bude to pole desiatich znakov. **Pri uvedenej definícii typu je názov nového typu určený menom poľa.** Ďalej postupujeme ako pri jednorozmernom poli.

Príklad 5B
Takto definované pole reťazcov `pole_ret` usporiadajme podľa abecedy.

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

typedef  char meno[10];  /* tento typ sa bude volať meno*/
int sort_meno(const void* prvy,const void* druhy)
{
return( strcmp((char*)prvy, (char*)druhy));
}

main(){
meno *pole_ret;
int i, n = 5;
pole_ret = (meno*)malloc(n * sizeof(meno));
strcpy(pole_ret[0],"Zuza");
strcpy(pole_ret[1],"Marka");
strcpy(pole_ret[2],"Katrena");
strcpy(pole_ret[3],"Jurino");
strcpy(pole_ret[4],"Jan");

/* vypis povodnych retazcov*/
for (i=0; i<5; i++)
  printf("%s\n", pole_ret[i]);
```

```
qsort((void*)pole_ret,5,10*sizeof(char),sort_meno);

/* vypis zoradenych  retazcov*/
printf("\n\nZoradene:\n");
for (i=0; i<5; i++)
  printf("%s\n", pole_ret[i]);
}
```

Výpis z programu bude rovnaký ako v predchádzajúcom prípade.

9.3.2.2 Premenlivá dĺžka reťazcov, pevný počet riadkov

Ďalej si vezmime situáciu, keď máme pevný počet riadkov a premenlivú dĺžku reťazca. Zadefinujeme si statické pole smerníkov na reťazce. Každému z týchto smerníkov pridelíme pamäť. Ďalej postupujeme ako v predchádzajúcich príkladoch. Tentokrát však reťazce nemusia byť uložené v pamäti za sebou.

Príklad 5C

```
#include <string.h>
main(){
char *pole_ret[5];
int i;
pole_ret[0]=(char *)malloc(5);
pole_ret[1]=(char *)malloc(5);
pole_ret[2]=(char *)malloc(8);
pole_ret[3]=(char *)malloc(6);
pole_ret[4]=(char *)malloc(5);
strcpy(pole_ret[0],"Zuza");
strcpy(pole_ret[1],"Marka");
strcpy(pole_ret[2],"Katrena");
strcpy(pole_ret[3],"Jurino");
strcpy(pole_ret[4],"Jan");
/* vypis  retazcov*/
for (i=0;i<5;i++)
  printf("%s\n", pole_ret[i]);
}
```

9.3.3 Dynamické polia reťazcov

V predchádzajúcom prípade sme mali vlastne statické pole smerníkov dynamických polí. Napokon by sme ešte mohli vytvoriť dynamické pole smerníkov dynamických polí. Opäť sa dostávame ku konštrukcii *smerník na smerník*.

Príklad 5D

```
#include <string.h>
main(){
char **pole_ret;
int i,n=5;
```

```
/* popridelovanie pamate a smernikov*/
pole_ret=(char**)malloc(n*sizeof(char *));
pole_ret[0]=(char *)malloc(5);
pole_ret[1]=(char *)malloc(5);
pole_ret[2]=(char *)malloc(8);
pole_ret[3]=(char *)malloc(6);
pole_ret[4]=(char *)malloc(5);
strcpy(pole_ret[0],"Zuza");
strcpy(pole_ret[1],"Marka");
strcpy(pole_ret[2],"Katrena");
strcpy(pole_ret[3],"Jurino");
strcpy(pole_ret[4],"Jan");
/* vypis povodnych retazcov*/
for (i=0;i<5;i++)
  printf("%s\n", pole_ret[i]);
...
}
```

9.4 Rôzne spôsoby načítania obrázku

Nasledujúci príklad ukazuje tri rôzne spôsoby načítania obrázku a odmeria ich čas. Časové porovnanie môže vyjsť na rôznych kompilátoroch rôzne. Niektoré kompilátory totiž robia optimalizáciu kódu, a tak si možno vysvetliť, prečo niektorý spôsob nie je pomalší ako iný, hoci by sa zdalo, že pomalší byť musí. Na porovnanie treba zobrať čo najväčšie obrázky. Prvé dva prípady chápu obrázok ako dvojrozmerné pole, tretí ako jednorozmerné. Po prvom načítaní sa na začiatok obrázku vrátime, bez toho, že by sme ho uzatvorili a znovu otvorili, príkazom:

```
fseek (f,(long)0,SEEK_SET),
```

kde `f` je smerník na otvorený súbor a `SEEK_SET` je prednastavená konštanta.

Príklad 6

Meno obrázku zadáme do príkazového riadku za meno programu.

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

#define s 1000
#define v 1000

int  main(int argc, char* argv[]){
int  pic[v][s],*pic2, pic3[v*s];
char pom1[5];
char pom2[80]="";
char pom3[80]="";

int a1,a2,a3,i,j,z,k;
FILE *vstup;
clock_t zac,konec;

if(argc != 2)
    {printf("nespravny pocet argumentov");exit(1);}
```

```
else
     {
 vstup=fopen(argv[1],"r");
 if (vstup==NULL) {perror("chyba otvorenia");exit(1);
     }
    }

/*nacitanie prvym sposobom*/
/*nacitanie zahlavia*/
 fgets(pom1,5,vstup);
 do{
  strcpy(pom3,pom2);
  fgets(pom2,80, vstup);}
 while(pom2[0]=='#');
 sscanf(pom2,"%d%d",&a1,&a2);
 fscanf(vstup,"%d",&a3);

zac=clock();
for (i=0;i<a2;i++)
   for (j=0;j<a1;j++)
      fscanf(vstup,"%d",&pic[i][j]);
konec=clock();
printf("nacitavanie1:%f [sec]\n", (konec-
                           zac)/(double)CLOCKS_PER_SEC);

/*nacitanie obrazku do pola pic druhym sposobom*/
fseek(f, (long)0, SEEK_SET);
/*nacitanie zahlavia*/
fgets(pom1,5,vstup);
 do{
  strcpy(pom3,pom2);
  fgets(pom2,80, vstup);}
 while(pom2[0]=='#');
 sscanf(pom2,"%d%d",&a1,&a2);
 fscanf(vstup,"%d",&a3);
 pic2=(int *)pic;
 zac=clock();
 z=a1*a2;
 for (i=0;i<z;i++)
   fscanf(vstup,"%d",pic2+i);
konec=clock();
printf("nacitavanie2:%f [sec]\n", (konec-zac)/(double)
                                          CLOCKS_PER_SEC);
/*nacitanie tretim sposobom, do jednorozmerneho pola pic3*/
fseek(f, (long)0, SEEK_SET);
/*nacitanie zahlavia*/
 fgets(pom1,5,vstup);
 do{
  strcpy(pom3,pom2);
  fgets(pom2,80, vstup);}
 while(pom2[0]=='#');
 sscanf(pom2,"%d%d",&a1,&a2);
 fscanf(vstup,"%d",&a3);
```

```
 zac=clock();
 z=a1*a2;
 for (i=0;i<z;i++)
   fscanf(vstup,"%d",pic3+i);
 konec=clock();
 fclose(vstup);
 printf("nacitavanie2:%f [sec]\n", (konec-
                   zac)/(double)CLOCKS_PER_SEC);

strcpy(pom2,"xv ");
system(strcat(pom2,argv[1]));
}
```

V podprograme boli použité dve doteraz neznáme funkcie, **perror** a **system.** Pre obe treba vložiť <stdio.h>

Prvá funkcia slúži na vypísanie systémového chybového hlásenia. Jej funkčný prototyp je:

void perror(const char *s).

Funkcia najprv vypíše reťazec, na ktorý ukazuje smerník dodaný ako argument a potom vypíše systémové chybové hlásenie, napríklad, že súbor nebol otvorený preto, lebo operačný systém ho nenašiel alebo napríklad neboli správne nastavené prístupové práva.

Druhá funkcia zapríčiní, že operačný systém vykoná príkaz daný reťazcom, ktorý sa dodá ako argument. V našom prípadesa spustí program *xv* a otvorí sa v ňom obrázok, zadaný ako argument. Príkaz *xv meno_obr*, ktorý spustí program *xv* s otvoreným obrázkom so zadaným menom, sa získa zreťazením príslušných reťazcov pomocou funkcie strcat. Funkčný prototyp funkcie system je:

```
int system(const char *string);
```

9.5 Dvojrozmerné pole ako parameter funkcie

9.5.1 Dvojrozmerné **statické** pole ako parameter funkcie

Definujme funkciu, ktorej argumentom bude statické pole. Pri jednorozmernom poli nemusíme, ale môžeme dĺžku poľa udať. Na rozdiel od jednorozmerného poľa, pri dvojrozmernom poli musíme už pri definícii zadať aspoň druhý rozmer, predstavujúci šírku riadku (t.j. počet stĺpcov). Môžeme však zadať rozmery aj obidva.

Príklad 7

Napíšme program pre načítanie a vypísanie matice, kde načítanie aj vypísanie budú robiť procedúry volané v hlavnej funkcii.

```
#include <stdio.h>
#define RIADKY 2
#define STLPCE 4
void citaj(float pole[RIADKY][STLPCE])
/* v hlavicke funkcie musime udat rozmery pola */
{
  int i, j;
  for (i = 0; i < RIADKY; i++)
    for (j = 0; j < STLPCE; j++) {
```

```
            scanf("%f", &pole[i][j]);}
}


void vypis(float pole[RIADKY][STLPCE])
{
  int i, j;
  for (i = 0; i < RIADKY; i++) {
    for (j = 0; j < STLPCE; j++)
      printf("%8.2f ", pole[i][j]);
    printf("\n");
  }
}

main()
{
  float matica[RIADKY][STLPCE];
  citaj(matica);
  vypis(matica);
}
```

Príklad 8

Napíšme program, ktorý z daného čiernobieleho obrázku urobí negatív, t.j. bielej hodnote priradíme čiernu, čiernej bielu a vo všeobecnosti intenzite *i* priradíme intenzitu *255-i*. Načítanie, urobenie negatívu a zápis napíšeme ako procedúry. V tomto prípade šírka a výška súboru je daná pomocou konštánt `s` a `v`, a načítané hodnoty rozmerov sa pri počítaní nevyužívajú. Ak zmeníme hodnoty rozmerov obrázku, musíme hodnoty prestaviť a program znovu skompilovať.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define s 460
#define v 601
/*definicia globalnych premennych*/
char pom1[5];
char pom2[80]="";
char pom3[80]="";
int a1,a2,a3;
FILE *vstup,*vystup;
/*definicia procedur*/
void nacit(int pict[v][s])
{
 int i,j;
 fgets(pom1,5,vstup);
 do{
    strcpy(pom3,pom2);
    fgets(pom2,80, vstup);}
 while(pom2[0]=='#');
 sscanf(pom2,"%d%d",&a1,&a2);
 fscanf(vstup,"%d",&a3);
```

```
 for (i=0;i<v;i++)
    for (j=0;j<s;j++)
fscanf(vstup,"%d",&pict[i][j]);
fclose(vstup);
}
void zapis(int pict[v][s])
{
int i,j;
fprintf(vystup,"%s",pom1);
if (!strlen(pom3)) fprintf(vystup,"%s",pom3);
fprintf(vystup,"%d %d\n%d\n",a1,a2,a3);
for (i=0;i<v;i++)
   { for (j=0;j<s;j++)
        fprintf(vystup,"%d ",pict[i][j]);
     fprintf(vystup,"\n");
   }
fclose(vystup);
}
void negat(int pict[v][s])
{
 int i,j;
 for (i=0;i<v;i++)
    for (j=0;j<s;j++)
       pict[i][j]=255-pict[i][j];
}
int  main(int argc, char* argv[]){
 char  pic[v][s];
 if(argc != 3)
    {printf("wrong number of arguments");exit(1);}
 else
    {
     vstup=fopen(argv[1],"r");
     if (vstup==NULL) {perror("chyba vstupu");exit(1);}
     vystup=fopen(argv[2],"w");
     if (vystup==NULL) {perror("chyba vystupu");exit(1);}
    }
nacit(pic);
negat(pic);
zapis(pic);
strcpy(pom2,"xv ");
system(strcat(pom2,argv[2]));
}
```

9.5.2 Dvojrozmerné **dynamické** pole ako parameter funkcie

Situácia je veľmi podobná ako v prípade statického poľa. Predchádzajúci príklad prerobíme pomocou dynamického poľa.

Príklad 9

Napíšme program pre načítanie a vypísanie matice, kde načítanie aj vypísanie budú robiť procedúry volané v hlavnej funkcii.

```
#include <stdio.h>
#include <stdlib.h>
/* v hlavicke funkcie musime udat rozmery pola */
void citaj(float **pole, int R, int S)
{
  int i, j;
  for (i = 0; i < R; i++)
    for (j = 0; j < S; j++) {
              scanf("%f", &pole[i][j]);}
}

void vypis(float **pole, int R, int S)
{
  int i, j;
  for (i = 0; i < R; i++) {
    for (j = 0; j < S; j++)
      printf("%8.2f ",pole[i][j]);
    printf("\n");
  }
}

int main(void)
{
 int i, RIADKY, STLPCE;

 RIADKY=2;
 STLPCE=4; /* hodnoty su ziskane az pocas behu programu*/
 float **matica;                        /* dynamické pole */
 matica = (float **) malloc(RIADKY * sizeof(float *));
 for (i = 0; i < RIADKY; i++)
     matica[i] = (float *) malloc(STLPCE * sizeof(float));
 citaj(matica, RIADKY, STLPCE);
 vypis(matica),RIADKY, STLPCE;
 for (i = 0; i < RIADKY; i++)
     free((void*)matica[i]);
 free(matica);
}
```

Počet riadkov a stĺpcov by sme mohli zadefinovať aj ako globálne premenné, a v tom prípade by sme ich nemuseli prenášať ako parametre funkcií. Globálnymi premennými sa budeme zaoberať v nasledujúcej kapitole.

☑ Kontrolný test

1. Majme korektne definované pole `char* a[5]`. Ktorý z nasledujúcich príkazov bude predstavovať hodnotu typu `char`.
   a. `*(a[1])`

b. `(a+1)`
c. `a[1]`
d. `*(a+1)`.

2. Majme skupinu nasledujúcich definícií:

```
char AA[4]={0,1,0,1};
char BB[4]={2,3,2,3};
char CC[4]={3,5,3,5};
char *pole[3];
pole[0] = AA;
pole[1] = BB;
pole[2] = CC;
```

Príkazom `*(pole[2] +1)` získame:

a. 0
b. 5
c. kompilátor vyhlási chybu
d. pri vykonávaní takéhoto príkazu by program mohol padnúť.

3. Po spracovaní nasledujúcich definícií prvky `pole[i][i]` budú tvoriť:

```
int i;
float *pole[5];
for (i = 0; i < 5; i++)
   pole[i] = (float *)malloc(28);
```

a. pole 5 x 28 typu `char`
b. pole 5 x 7 typu `float`
c. pole 7 x 5 typu smerník na `float`
d. pole 5 x 7 typu smerník na `float`.

4. Majme dve polia definované príkazmi:

```
int m = 3, n = 3, i, **pole1;
pole1 = (int**)malloc(m*sizeof(int *));
for (i=0; i<m; i++)
  pole1[i] = (int *)malloc(n*sizeof(int));
int  pole2[3][3];
```

Vyberte nesprávne tvrdenie. Polia `pole1` a `pole2` sa líšia tým, že:

a. v poli `pole1` prvky nemusia ísť za sebou, v druhom poli áno
b. zaberajú nerovnaké množstvo pamäte
c. majú prvky rôzneho typu
d. prvé pole je dynamické a druhé statické.

5. Majme dvojrozmerné pole `char b[10][10]`. Ktorými príkazmi ho nemôžeme vynulovať?

a.
```
for (i=0; i<10; i++)
   for (j=0; j<10; j++)
      b[i][j] = 0;
```

b.
```
for (i=0; i<100; i++)
      b[i] = 0;
```

c.
```
char *p;
p=b;
for (i=0; i<100; i++)
```

```
        p[i] = 0;
```

d.
```
char *p;
p=b;
for (i=0; i<10; i++)
   for (j=0; j<10; j++)
     p[i*10+ j] = 0;
```

6. Majme dvojrozmerné pole definované príkazmi

```
char **c;
c=(char**)malloc(10*sizeof(char *));
for (i=0;i<10;i++)
  c[i]=(char *)malloc(10*sizeof(char));
```

Ktorými príkazmi ho môžeme vynulovať?

a.
```
for (i=0; i<10; i++)
      c[i] = 0;
for (i=0; i<10; i++)
   for (j=0; j<10; j++)
      c[i][j] = 0;
```

b.
```
for (i=0; i<10; i++)
   for (j=0; j<10; j++)
      c[i][j] = 0;

for (i=0;i<10;i++)
      c[i] = 0;
```

c.
```
char *D;
for (i=0; i<10; i++)
     {D = c[i];
     for (j=0;j<10;j++)
       D[j]=0;

     c[i]=0;}
```

d.
```
char *p;
p=c;
for (i=0; i<10; i++)
   {for (j=0; j<10; j++)
     p[i*10 + j] = 0;
    c[i]=0;}
```

7. Majme príkazy:

```
char ret[4][5];
strcpy(ret[0],"ABCD");
strcpy(ret[1],"BCDA");
strcpy(ret[2],"CDAB");
strcpy(ret[3],"DABC");
```

Výraz `ret[1]` predstavuje:

a. reťazec `BCDA`
b. adresu reťazca `BCDA`
c. znak `A`
d. adresu znaku `A`.

8. Majme príkazy:

```
char ret[4][5];
strcpy(ret[0],"ABCD");
strcpy(ret[1],"BCDA");
strcpy(ret[2],"CDAB");
strcpy(ret[3],"DABC");
```

   Reťazec `DABC` dáme vypísať príkazom:
   a. `printf("%s",ret[3][4]);`
   b. `printf("%p",ret[3][4]);`
   c. `printf("%s",ret[3]);`
   d. `printf("%p",ret[3]);`

9. Majme statické reálne pole `A` rozmerov `2x2`. Chceme vytvoriť procedúru `deter` pre výpočet determinantu, pričom jej argumentom má byť práve pole A. Nasledujúce riadky ukazujú príklady záhlaví funkcie `deter` a príklady volania danej funkcie. Premenná `x` je reálne číslo. Vyberte, ktoré z príkladov sú správne.
   a. `float deter(float A[2][2])` a `x = deter(A[2][2]);`
   b. `float deter(float A)` a `x = deter(A);`
   c. `float deter(float B[2][2])` a `x = deter(A);`
   d. `deter(float A[2][2])` a `x = deter(A);`

10. Pri načítavaní dát zo súboru sa dostaneme do jeho stredu. Chceme sa vrátiť na začiatok bez toho, že by sme uzavreli a znovu otvorili súbor. Nech je `subor` smerník na otvorený súbor. Urobíme to príkazom:
    a. `fseek (*subor,(long)0,SEEK_SET),`
    b. `fseek (*subor,L0,SEEK_SET),`
    c. `fseek (subor,(long)0,SEEK_SET),`
    d. `fseek (subor,SEEK_SET=(long)0).`

11. Sme v operačnom systéme XWindows a v aktuálnom adresári máme obrázok *obr*. Budeme ho v programe upravovať a na záver programu chceme zaradiť príkaz, ktorý zabezpečí jeho zobrazenie. Dá sa to urobiť príkazmi:
    a. `system("xv obr");`
    b. `system(xv obr);`
    c. `system(xv "obr");`
    d. `system(strcat("xv","obr");`

12. Majme dvojrozmerné dynamické pole `A` s definíciou `double **A`, rozmerov `m x n`, ktorému bola korektne pridelená pamäť. Toto pole zaberá v pamäti:
    a. `m x n` bytov,
    b. `m x n * sizeof(double)` bytov,
    c. `m x n * sizeof(8)` bytov,
    d. `m x n * sizeof(double)+ m x sizeof(double*)` bytov.

Kontrolné otázky

1. Uveďte príklady polí smerníkov na rôzne typy. Vymyslite k nim príklady použitia.
2. Vymenujte čo najviac rozdielov medzi statickým a dynamickým poľom.
3. Ako sa uvoľní pamäť dynamického poľa?
4. Ako by ste utriedili pole reťazcov? Musí byť pamäť pre jednotlivé reťazce v poli rovnakej dĺžky? Prečo?
5. Chcete načítať dvojrozmerné pole? Vymenujte spôsoby, akými môžete dvojrozmerné pole načítať a definovať.
6. Ako sa pri načítavaní zo súboru vrátite na jeho začiatok bez toho, aby ste uzavreli a znovu otvorili súbor?
7. Ako sa dá vyvolať vykonanie systémového príkazu priamo z programu?
8. Keď je dvojrozmerné statické pole argumentom funkcie, musí byť uvedený jeho rozmer? V čom je rozdiel oproti jednorozmernému poľu?
9. Ako je to v prípade, keď má byť argumentom funkcie dvojrozmerné dynamické pole?

Cvičenia

1. Zo súboru načítajte maticu reálnych čísel ľubovoľných rozmerov. Nech rozmery matice sú uvedené ako prvý riadok súboru. Vytvorte jednorozmerné pole, v ktorom budú postupne uložené všetky kladné čísla matice, načítavanej po riadkoch,

2. Zo súboru načítajte maticu ľubovoľných rozmerov. Nech rozmery matice sú uvedené ako prvý riadok súboru. Do pamäte uložte transponovanú maticu a dajte ju vypísať.

3. Zo súboru načítajte maticu, jej rozmer je zadaný v súbore pred samotnými dátami, nech matica nie je veľmi veľká. V programe zadajte číslo *n* (do 10) a vytvorte novú maticu, v ktorej sa každý prvok nahradí maticou *nxn*, obsahujúcou hodnoty rovnajúce sa danému prvku. Použite dynamické pole.

4. Majte maticu rozmerov 2^n x 2^n. V tejto matici nech sú uložené hodnoty od 0 do 255, čo si môžeme predstaviť ako napríklad úrovne šedej farby. Zvoľte si nejaké tolerančné kritérium, napr. 3, nazvime ho ε. Postupne deľte maticu na 4 rovnaké podmatice – kvadranty a porovnajte hodnoty v danej podmatici. Ak sú všetky v rámci danej tolerancie, nahraďte všetky hodnoty v kvadrante ich priemerom. Ak je pôvodná matica vhodná, výsledná matica by mala obsahovať menej hodnôt. Skúste vymyslieť úspornú reprezentáciu pre novú maticu.

5. Vymyslite si krátku databázu, ktorá bude obsahovať číselné a textové údaje. Urobte program, ktorý bude pridávať a uberať záznamy z dátabázy a bude ju triediť podľa zadaných položiek.

Zadania

1. Zistite, čo robí nasledujúci program:

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

```
#define  N 3
#define tol 1E-8
double x[N];
double a[N][N+1];



int nieco(double a[N][N+1], double x[N])
{
  int max,i,j,k;
  double tmp,koef;
  for(i=0; i<N; i++)
  {
    max=i;
    for( j=i+1; j<N; j++)
      if(fabs(a[j][i])>fabs(a[max][i]))
        max=j;
    for( k=i; k<N+1; k++)
    {
      tmp=a[i][k];
      a[i][k]=a[max][k];
      a[max][k]=tmp;
    }
    if(fabs(a[i][i])<tol)
      return 0;

    koef=a[i][i];
    for(k=i; k<N+1; k++)
        a[i][k]=a[i][k]/koef;

    for(j=i+1; j<N; j++)
      for(k=N; k>=i; k--)
        a[j][k]=a[j][k]-a[i][k]*a[j][i];
  }

  for(j=N-1; j>=0; j--)
    for(i=j-1; i>=0; i--)
   {
      koef= -a[i][j];
      for(k=j; k<=N; k++)
         a[i][k]+=a[j][k]*koef;
    }


    for (i=0;i<N;i++)
      x[i]=a[i][N];
  return 1;
}

int main()
{ int i,j;
  FILE *f;
  f=fopen("data2","r");
  if (f==NULL) {printf("subor!!!\n"); system("pause");return(1);}
```

```
  for (i=0;i<N;i++)
    for (j=0;j<N+1;j++)
  fscanf(f, "%lf",&a[i][j]);

  if(!nieco(a,x))
    printf("Problem\n");
  else

{
    printf("Vysledok:\n");
    for(i=0;i<N;i++)
      printf("x[%d]= %lf",i,x[i]);
  }
}
```

2. a) Napíšte program pre výpočet determinantu ľubovoľnej (štvorcovej) matice.

   b) Napíšte program pre výpočet inverznej matice k ľubovoľnej regulárnej matice a urobte skúšku vynásobením pôvodnej a inverznej matice.

10 Rozsah platnosti premenných

Hlavné podtémy kapitoly:

- Oblasť platnosti identifikátorov. Globálne a lokálne definície a deklarácie.
- Pamäťové triedy premenných. Triedy `auto`, `extern`, `static` a `register`.
- Pole smerníkov na funkcie, jeho definícia. Volanie funkcie z ponuky – menu.
- Práca s binárnymi súbormi. Zápis do súboru pomocou `fputc`, čítanie s použitím `fgetc`.
- Súbory typu `raw`. Konverzia textových súborov na súbory `raw` pomocou `fread` a `fwrite`.

10.1 Oblasť platnosti identifikátorov

Jazyk C umožňuje viacero možností v stanovení, kde, kedy a aký identifikátor (premenná, funkcia, typ, ...) bude viditeľný (dostupný) a komu. Ďalší popis sa bude sústreďovať na premenné, pričom upozorníme na všetky prípadné súvislosti s funkciami.

10.1.1 Globálne a lokálne premenné

Základná organizácia definícii v programe v ANSI C je:

```
...
globálne definície a deklarácie

definície funkcií
...
hlavný program
```

Globálne definície definujú premenné, ktorých rozsah platnosti je od miesta definície do konca súboru - nie programu (program môže byť napísaný vo viacerých súboroch)! Vezmime si nasledujúci príklad:

```
int i;

funkcia1() {
    ...         /* telo funkcie funkcia1()    */
}

int j;

funkcia2() {
    ...         /* telo funkcie funkcia2()   */
}

funkcia3() {
    ...         /* telo funkcie funkcia3 () */
}
/* koniec suboru  */
```

Premenná `i` bude platná vo všetkých troch funkciách, ale premenná `j` iba vo funkciách `funkcia2()` a `funkcia3()`.

Globálne deklarácie sú deklarácie premenných, ktoré sú definované v iných súboroch (moduloch). Pretože sú tieto deklarácie často špecifikované použitím kľúčového slova `extern`, nazývajú sa tiež často *externé deklarácie*. Pripomeňme, že deklarácie premenné nezavádzajú ani im nepriradzujú pamäťový priestor, ale používajú sa na určenie interpretácie, ktorú dáva jazyk C každému identifikátoru.

Lokálne definície definujú premenné, ktorých rozsah platnosti je od miesta definície do konca funkcie, v ktorej sú definované. Tieto definície sa teda vyskytujú vo vnútri definícii funkcií, napr.:

```
funkcia1() {
    int i;
    ...          /* telo funkcie funkcia1() */
}
```

V C môžu byť niektoré globálne identifikátory prekryté (zatienené) identifikátormi lokálnymi, napr.:

```
int i1, i2;

funkcia1() {
    int i1, j1;
    ...                /* telo funkcie funkcia1()  */
}

int j1, j2;

funkcia2() {
    int i1, j1, k1;
    ...                /* telo funkcie funkcia2() */
}
```

Vo funkcii `funkcia1()` je globálna premenná `i1` prekrytá lokálnou premennou rovnakého mena, t.j. platí hodnota lokálnej premennej. V tejto funkcii môžu byť použité tri premenné - `i2` (globálna) a `i1` a `j1` (lokálne). Rovnako tak vo funkcii `funkcia2()` môžu byť použité dve globálne premenné `(i2` a `j2)` a tri lokálne `(i1, j1, k1)`.

S globálnymi premennými sme sa už stretli, napr. globálnymi premennými boli znakové polia pre načítavanie záhlavia obrázku.

10.2 Pamäťové triedy

Okrem rôznych typov, môžu byť premenné uvedené i v rôznych pamäťových triedach. Tie určujú, v ktorej časti pamäte bude premenná kompilátorom umiestnená, a taktiež, kde všade bude premenná viditeľná. Rozširujú tak možnosti viditeľnosti premenných, ktoré dosiaľ boli len globálne a lokálne.

Jazyk C rozpoznáva tieto pamäťové triedy:

- `auto`
- `extern`
- `static`
- `register`

10.2.1 Trieda `auto`

O týchto premenných sa často hovorí ako o automatických. Je to implicitná (cháp ako prednastavená) pamäťová trieda pre lokálne premenné: ak je premenná definovaná vo vnútri funkcie bez určenia typu pamäťovej triedy, je jej implicitný typ práve typ `auto` a premenná je uložená v zásobníku - statickej lokálnej pamäti. Premenná typu `auto` existuje od vstupu do funkcie a zaniká pri výstupe z funkcie. Pri každom vstupe do funkcie ma náhodnú hodnotu - nie je teda ani implicitne inicializovaná na 0, ani si neponecháva svoju pôvodnú hodnotu medzi dvoma volaniami funkcie.

```
funkcia() {
    auto int i;         /* je to iste ako:       int i;*/
    auto int j = 5;     /* je to iste ako:       int j = 5;*/
    ...
}
```

10.2.2 Trieda `static`

Pre túto pamäťovú triedu neexistuje implicitná definícia, ale kľúčové slovo `static` musí byť pri definícii vždy uvedené. Premenné tejto triedy sú uložené v dátovej oblasti. Pamäťovú triedu `static` využívajú najčastejšie **lokálne premenné** (definované vo vnútri funkcie), ktoré si ponechávajú svoju hodnotu i medzi jednotlivými volaniami tejto funkcie. (To je podstatný rozdiel medzi `static` a `auto`). Tato premenná existuje od prvého volania príslušnej funkcie až do doby ukončenia programu trvalo, nevzniká a nezaniká pri každom volaní funkcie. Poskytuje funkcii súkromný, stály pamäťový priestor, ako lokálna premenná nie je mimo funkcie prístupná. **Externá premenná** triedy `static` je známa vo zvyšku zdrojového súboru, v ktorom bola deklarovaná, ale nie je známa v iných súboroch.

Priklad 1

```
void f(void) {
    int x = 2;
    static int i = 0;

    printf("f bola volana %d-krat, x = %d\n", i, x);
    i++;
    x++;
}
```

Premenná `x` je lokálna automatická premenná a premenná `i` je lokálna statická premenná. Zakaždým, keď je `f()` zavolaná, je miesto pre premennú `x` alokované v inej časti pamäte (zásobníku) a vždy je automaticky inicializované na hodnotu 2. Premenná `i` je inicializovaná na 0 pri prvom vstupu do funkcie `f()` a svoju hodnotu si zachováva medzi jednotlivými volaniami - **inicializácia na nulu sa pri ďalších volaniach funkcie `f()` už nikdy neuskutoční.**

Napríklad pre volanie:

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

bude výstup:

```
f bola volaná 0-krat, x = 2
f bola volaná 1-krat, x = 2
f bola volaná 2-krat, x = 2
```

Poznámky:

- Globálne premenné i funkcie môžu byt označené taktiež ako `static`, čo má ten význam, že sú viditeľné iba v module, v ktorom sú definované.

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

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

  spracujú tak, že statická premenná bude iba `i` a premenná `j` bude mat implicitnú pamäťovú triedu - teda `auto`. Obe premenné budú ale typu `int`. Buď je vhodné si aktuálny stav prekladača overiť na jednoduchom príklade alebo je lepšie použiť implementačne nezávislú definíciu:

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

10.2.3 Trieda `extern`

Je to implicitná pamäťová trieda pre globálne premenné. Tieto premenné sú uložené v dátovej oblasti. Táto pamäťová trieda sa najčastejšie používa pri oddelenom preklade súborov, keď je potrebné, aby dva alebo viac súborov zdieľalo tú istú premennú. Táto premenná je v jednom súbore definovaná ako globálna a vo všetkých ostatných je deklarovaná ako `extern` .

10.2.4 Trieda `register`

Pretože jazyk C je "nízko úrovňový", môže programátor požadovať, aby niektorá premenná nebola uložená v pamäti, ale iba v registri počítača. To má výhodu oveľa rýchlejšieho prístupu k premennej, a teda i rýchlejšieho programu. Označenie premennej ako `register` neviaže túto premennú na určitý konkrétny register počítača - to záleží celkom na prekladači. Definícia premennej ako:

```
register int i;
```

znamená iba, že táto premenná môže byt uložená do registra, ak je nejaký voľný a ak je to z najrôznejších systémových dôvodov možné. Ak označíme teda všetky použite premenné ako `register`, potom skutočne v registroch budú umiestnené iba niektoré. Z toho vyplýva, že ako registrovú premennú je vhodné označiť napríklad premennú jednoduchého cyklu alebo často používaný formálny parameter. **U registrových premenných sa neuskutočňuje žiadna implicitná inicializácia a používajú sa výhradné ako lokálne premenné**.

Poznámka:

- Pre definíciu viac premenných rovnakého typu v pamäťovej triede `register` platí to isté, čo pre `static`, je teda vhodnejšie písať:

- ```
  register int i;register int j;
  ```
  namiesto:
  ```
  register int i, j;
  ```

Príklad 2

Funkcia pre výpis malej násobilky:

```
void nasobilka(register int k) {
    register int i;

    for (i = 1; i <= 10; i++)
        printf("%2d x %d = %2d \n", i, k, i*k);
}
```
```

10.3 Smerníky na funkcie

Doteraz sme si hovorili o smerníkoch, ktoré ukazovali na základný alebo zložený dátový typ. Smerník však môže ukazovať aj na funkciu. Ukážeme si, ako sa taký smerník definuje. Potom zadefinujeme pole smerníkov na funkcie a pomocou neho budeme vyberať funkcionalitu ponúkanú v menu.

10.3.1 Definícia smerníka na funkciu

Ak si spomenieme napr. na funkciu `fopen()`, vieme, že funkcia môže vracať smerník na nejaký dátový typ. Táto možnosť sa v C využíva pomerne často. Vezmime napr. funkciu:
`char *najdi_adresu_slova(char *s).`
Táto funkcia by hľadala v pamäti od nejakej adresy zadané slovo a vracala by smerník na toto slovo.

Často sa tiež využíva možnosť definovať premennú ako smerník na funkciu vracajúcu nejaký typ. Napríklad smerník `p_fi` ukazujúci na funkciu vracajúcu typ `int` sa dá definovať príkazom

```
int (*p_fi) ();
```

Prázdne zátvorky () pred ukončovacou bodkočiarkou sú nevyhnutné, pretože

```
int (*p_fi);
```

by znamenalo to isté ako: `int *p_fi;` teda že `p_fi` je smerník na `int`, a nie smerník na funkciu vracajúcu `int`.

Zátvorky okolo mena premennej sú nevyhnutné, pretože:

```
int *p_fi();
```

by znamenalo, že tento riadok je deklarácia funkcie pomenovanej `p_fi`, ktorá vracia smerník na `int`.

10.3.2 Priradenie hodnoty smerníku na funkciu

Ak máme definovaný smerník na funkciu `p_fv`:

```
void (*p_fv) ();
```

a funkciu

```
void negat(int pic[100][100])
{
...},
```

potom je možné napísať priradenie:

```
p_fv = negat;                    /* Pozor! žiadny & */
```

ktoré priradí smerníku `p_fv` adresu funkcie `negat()`.

10.3.3 Volanie funkcie

Pomocou smerníku `p_fv`, je potom možné volať funkciu `negat()` ako:

```
(*p_fv)(pic);
```

alebo ako:

```
p_fv(pic);
```

Prvý spôsob bol ako jediný možný v K&R verzii C, a obidva sú možné v ANSI C.

10.4 Pole smerníkov na funkcie

Tak, ako môže byť pole zložené z jednoduchých premenných, zo smerníkov na jednoduché premenné, môžu byť prvky poľa aj smerníky na funkcie. Všetky funkcie musia byť samozrejme rovnakého typu . Aby bola definícia prehľadnejšia, pre smerník na funkciu vytvoríme nový typ. Potom pole smerníkov na ne sa definuje takto:

```
typedef void (* P_FUN) ();    /* definicia smernika na funkciu
                                 vracajucu typ void */
```

Podobne ako pri definícii nového typu predstavujúceho pole určitých rozmerov, aj tu je meno typu dané názvom smerníka, ktorý bol v definícii použitý. Pole smerníkov na funkcie už potom definujeme známym spôsobom:

```
P_FUN funkcie[10];              /* definicia pola 10-tich smernikov*/
```

Toto pole je potom nutné naplniť adresami existujúcich funkcií, čo sa robí úplne rovnako ako pri priraďovaní adresy funkcie do smerníku na funkciu. Možná praktická aplikácia poľa smerníkov na funkcie je program riadený pomocou ponuky (menu). Adresy jednotlivých funkcií uskutočňujúcich príslušné príkazy menu sú uložené v poli a odtiaľ môžu byť priamo volané pomocou indexu. Ak využijeme predchádzajúce definície nového typu P_FUN, potom je možné definovať pole smerníkov na funkcie vrátane jeho inicializácie takto:

```
P_FUN funkcie[] = {file, edit, search, compile, run};
```

kde identifikátory `file, edit, search`, ... sú názvy jednotlivých funkcií s návratovým typom `void`. Volanie funkcie je potom:

```
(* funkcia[1])();
```

alebo len v ANSI C:

```
funkcia[1]();
```

Príklad 3

Majme takúto úlohu. Načítame čiernobiely obrázok a napíšeme preň štyri procedúry, ktoré budú: robiť z obrázku negatív, robiť prahovanie (obrázok zmenia na čiernobiely, kde intenzity menšie ako prahová hodnota budú nahradené nulou a väčšie ako prahová hodnota budú nahradené hodnotou 255), zväčšovať kontrast a ukončia program. Nech sa funkcie nazývajú `prah(),negat(),kontrast()` a `koniec()`. Program vypíše menu a podľa voľby sa spustí príslušná funkcia.

```
negativ    - 1
prahovanie - 2
kontrast   - 3
koniec     - 4
Zadaj volbu:_
```

Návrh riešenia:

```
void negat(int pict[v][s])
{...}
void prah(int pict[v][s])
{...}
void kontrast(int pict[v][s])
{...}
```

```
void koniec(int pict[v][s])
{...}

int  main(int argc, char* argv[]){
int  pic[v][s],i;
typedef void (* P_V)();
P_V menu[4];
menu[0]=negat;
menu[1]=prah;
menu[2]=kontrast;
menu[3]=koniec;
/* to iste P_V menu[4]={negat,prah,kontrast,koniec};*/

...
nacit(pic);     /*nacitanie obrazku*/
printf("negativ   - 1\n");
printf("prahovanie- 2\n");
printf("kontrast  - 3\n");
printf("koniec    - 4\n");

SLUCKA:
    printf("Zadaj volbu:\n");
    scanf("%d",&i);
    if((i>0)&&(i<5)) {i--; (*menu[i])(pic); goto SLUCKA;}
}
```

Program sa môže vykonať neúspešne, ak namiesto číslice omylom zadáme písmeno. Ešte raz sa takouto slučkou budeme zaoberať príklade 6 v 11. kapitole. Podrobnejší príklad používajúci takúto konštrukciu uvedieme v Kapitole 12.

10.5 Práca s binárnymi súbormi

V predchádzajúcich kapitolách sme sa učili pracovať iba s textovými súbormi. *Textový súbor* je postupnosť znakov, kde jeden znak je zapísaný do jedného bytu. **V *binárnych súboroch* sa dáta uchovávajú ako v pamäti počítača.** Textové súbory majú tú výhodu, že je možné si ich obsah kedykoľvek prezrieť, vytvoriť alebo opraviť bežným editorom. Ich nevýhodou ale je, že pre uchovanie rovnakého množstva informácie potrebujú oveľa viac priestoru. Napríklad číslo 255 zaberie v textovom súbore priestor troch bytov, zatiaľ čo v binárnom súbore je potrebný iba 1 byte. Druhou výhodou binárnych súborov je, že sa s nimi pracuje oveľa rýchlejšie než s textovými súbormi. Dôvody sú dva - jednak sú binárne súbory kratšie a jednak pri zápise čísla do textového súboru je nutné previesť jeho konverzie z vnútornej reprezentácie čísla v počítači na textovú podobu, čo je časovo náročné. Tieto konverzie v binárnych súboroch odpadajú, pretože sa do nich zapisuje priamo obsah pamäti po bytoch. Z týchto dôvodov sa binárne súbory v profesionálnych programoch využívajú pomerne často. Najvýhodnejšie je ich použitie pre ukladanie rozmerných údajov - veľkých polí, štruktúr, atď.

Ako sme už spomenuli pri formáte *pgm* obrázku, môže byť tento formát binárny. Ak si otvoríte v textovom editore binárny *pgm* súbor, uvidíte niečo takéto:

```
P5
# CREATOR: XV version 3.10a-jumboFix+Enh of 20040523
440 601
255
UF80)%'.#'**('+/'"#**%#&(&'*)(+/$,)(/('9+*()/31,0*,4513<0/0478526:21
?>37/023579...
```

Všimnite si, že úvodná špecifikácia obrázku v záhlaví je P5, a nie P2, ako to bolo pri formáte ASCII. Textový súbor zaberal na disku 1 073 330 bytov, binárny súbor 264 508 bytov.

10.5.1 Čítanie z binárneho súboru

Jeden byte načítame z binárneho súboru pomocou funkcie `fgetc`, ktorá má funkčný prototyp:

```
int fgetc(FILE *f);
```

Funkcia načíta ďalší byte zo súboru a vráti ho skonvertovaný ako `int`, pričom v prípade `EOF` vráti -1, inak je rozsah hodnôt príslušný typu `unsigned char`.

Pri tejto ale aj pri podobných funkciách je dobré dodržiavať nasledujúce zásady:

- na čítanie znaku zo súboru používajme vždy premennú typu `int`, aby sa správne načítal koniec súboru,
- vždy testujme, či sa funkcie `fopen()` a `fclose()` vykonali správne,
- uzatvárajme súbor okamžite, len čo s ním prestaneme pracovať.

Príklad 4

Načítajme obrázok, ak je daný ako binárny *pgm* súbor.

```
FILE *vstup;
unsigned char  pic[v_max][s_max];
...

vstup=fopen("vstup","rb");
 if(vstup==NULL)
     {
      perror("\nChyba otvorenia vstupu\n");
      exit(-1);}

/*načítanie záhlavia*/
...
/*načítanie dát zo súboru*/
for (i=0;i<v;i++)
    for (j=0;j<s;j++)
       pic[i][j]= (unsigned char)fgetc(vstup);//pretypujeme
...
```

10.5.2 Zápis do binárneho súboru

Jeden byte zapíšeme do binárneho súboru pomocou funkcie `fputc`, ktorá má funkčný prototyp:

```
int fputc(int c, FILE *f);
```

`c` sa skonvertuje na `unsigned char` a zapíše sa do súboru `f`. Ak sa nevyskytne žiadna chyba, ten istý znak, ktorý sa zapísal, sa aj vráti. Ak sa vyskytne chyba, vráti sa `EOF` (preto je návratový typ `int`.

Príklad 5

Zapíšme obrázkové dáta do binárneho súboru.

```
FILE *vystup;
unsigned char  pic[v_max][s_max];
...

vystup=fopen("vystup","wb");
 if(vystup==NULL)
     {
      perror("\nChyba otvorenia vystupu\n");
      exit(-1);}

/*zápis záhlavia*/
...
/*zápis dát do súboru*/
for (i=0;i<v;i++)
    for (j=0;j<s;j++)
       fputc(pic[i][j],vystup);

...
```

V OS Windows je niekedy lepšie záhlavie zapísať tak, že súbor otvoríme s parametrom "**w**", zapíšeme záhlavie, zavrieme a potom znovu otvoríme na zápis intenzít s parametrom "a**b**" – dáta sa pridajú za záhlavie ako binárne.

10.5.3 Všeobecné `raw` súbory

Pod týmto názvom budeme v tejto učebnici chápať súbory, ktoré nemajú žiadne záhlavie a obsahujú iba dáta uložené tak ako v pamäti počítača. Ako už bolo povedané v úvode tejto časti pre binárne súbory, pre svoju úspornosť sa formát *raw* často používa v programátorskej praxi. Mnohé programy robia prevod do tohto typu, napr. program pre prácu s obrázkami IrfanView, ktorý sa dá voľne stiahnuť z Internetu, robí tento prevod po jednoduchom doinštalovaní plug-inov. Obrázkový súbor typu *raw* môžeme získať aj tak, že z binárneho obrázku typu *pgm* jednoducho odstránime záhlavie.

Pri práci s obrázkami bolo špecifické to, že jedna hodnota – intenzita sa zapísala jedným príkazom `fputc`, ktorý vždy zapisoval hodnotu do jedného bytu. To je však len jeden z možných prípadov. Ako formát *raw* môžeme chcieť zapísať dáta, ktoré predstavujú reálne čísla, alebo celé čísla väčšie ako 255. Napríklad predstavme si, že naše dáta sú typu `float` a chceme ich zapísať vo formáte *raw*. Teraz na zápis jednej hodnoty budeme potrebovať 4 byty. Spôsobov, ako to možno urobiť, je niekoľko. Nasledujúci príklad bude z jedného – textového súboru načítavať reálne čísla a bude ich zapisovať vo formáte *raw* do nového súboru, ktorého meno sa bude od pôvodného líšiť príponou *raw*.

Príklad 6

```
char   *pomoc;
float ll;
...
 f1=fopen(meno,"r");      /*povodny subor*/
 strcat(meno,".raw");
 f2=fopen(meno,"wb");      /*vystupny subor*/

...
   for (j=0;j<M;j++)
      {
      fscanf(f1,"%f",&ll);
```

```
      pomoc=(void*)&ll;    /*alebo pomoc=(char*)&ll;*/
      fputc(*pomoc,f2);
      fputc(*(pomoc+1),f2);
      fputc(*(pomoc+2),f2);
      fputc(*(pomoc+3),f2);
      }
fclose(f2);
fclose(f1);
}
```

Iný, pohodlnejší spôsob predstavuje použitie funkcie `fwrite()` s nasledujúcim funkčným prototypom:

`int fwrite(char *odkiaľ, size_t rozmer, size_t počet, FILE *fr);` **kde**

`odkiaľ`	–	adresa pamäte, odkiaľ sa prečíta zapisovaný blok dát
`rozmer`	–	veľkosť jednej položky z bloku dát v bytoch
`počet`	–	počet spracovávaných položiek (nie bytov).

Ak použijeme túto funkciu, príslušná časť cyklu z predchádzajúceho príkladu bude takáto:

```
...
   for (j=0;j<M;j++)
      {
      fscanf(f1,"%f",&ll);
      fwrite((char*)&ll, 4 ,1, f2)
      }
...
```

Podobne, pre načítavanie dát z *raw* súboru môžeme použiť funkciu

`int fread(char *kam, size_t rozmer, size_t počet, FILE *fr);` **kde**

`kam`	–	adresa pamäte, kam sa uloží prečítaný blok dát
`rozmer`	–	veľkosť jednej položky z bloku dát v bytoch

☑ Kontrolný test

1. Sú dané príkazy:

```
int a=3,b=10;
int funkcia1() {
     int i=5, b=1;
     a = a + b + i;
     return a;}
int i=4;
void funkcia2() {
     int i=0;
     b = b  + a + i;}
main(){
      a = funkcia1();
      funkcia2();}
```

Vyberte správne tvrdenie.

a. `a = 9, b = 19`
b. `a = 18, b = 23`

c. `a = 9, b = 23`
d. `a = 18, b = 17`

2. Lokálne definícia definuje premenné, ktorých rozsah platnosti je:
   a. od začiatku do konca hlavného programu
   b. od začiatku do konca súboru, v ktorom sú definované
   c. od miesta definície do konca súboru, v ktorom sú definované
   d. od miesta definície do konca funkcie, v ktorej sú definované.

3. Majme definíciu funkcie:

```
void abc(){
        double x=1;
        static double y=1;
        y = y + x;
        x =  x + 1;
        printf("x=%lf, y=%lf", x ,y); }
```

   a jej volanie príkazom:

```
for (j=0;j<5;j++)
     abc();
```

   Posledný výpis bude hovoriť, že:
   a. `y = 2`
   b. `y = 6`
   c. `y = 5`
   d. `y = 32.`

4. Vyberte nesprávne tvrdenia:
   a. externá premenná triedy `static` je známa v zvyšku zdrojového súboru, v ktorom bola deklarovaná, ale nie je známa v iných súboroch
   b. lokálna statická premenná je inicializovaná pri prvom vstupu do funkcie, v ktorej je definovaná a medzi jednotlivými volaniami si svoju hodnotu zachováva, jej inicializácia sa pri ďalších volaniach funkcie už nevykoná
   c. pamäťová trieda `extern` sa najčastejšie používa pri oddelenom preklade súborov, keď je potrebné, aby dva alebo viac súborov zdieľalo tu istú premennú
   d. registrová premenná sa vždy uloží do registra počítača, môže byť lokálna aj globálna.

5. Majme funkciu s návratovou hodnotou typu `double`. Smerník `fun` na túto funkciu je definovaný príkazom
   a. `double (*fun)();`
   b. `double (*fun);`
   c. `double *fun();`
   d. `double *fun;`

6. Majme smerník `s` na funkciu s návratovou hodnotou typu `double`. Nech `f1` je funkcia s návratovou hodnotou typu `double`, ktorá nemá žiadne parametre a `x` je premenná typu `double`. Napíšte, ktoré príkazy nepredstavujú v ANSI C volanie funkcie `f1`:
   a. `x = f1();`
   b. `s = f1; x = s();`
   c. `s = f1; x = (*s)();`
   d. `s = &f1; x = s();`

7. Napíšte, ktorými príkazmi môžeme definovať typ `SMF` smerníkov na funkcie vracajúce smerník na typ `void`.
   a. `typedef SMF void (* SMF_FUN)() ;`
   b. `typedef SMF void* (* SMF_FUN)() ;`
   c. `typedef void (* SMF) ;`
   d. `typedef void* (* SMF) () ;`

8. Nech je `pole[4]={f1,f2,f3,f4};` pole smerníkov na funkcie vracajúce smerník na typ `void`. Funkciu `f2` vyvoláme príkazmi:
   a. `menu[2]();`
   b. `menu[1]();`
   c. `(*menu[1])();`
   d. `*menu[1]();`

9. Majme binárny obrázok vo formáte *pgm*. Nech í-ta hodnota predstavujúca intenzitu je 128 a je uložená v premennej `value` typu `int`. Vyberte nesprávne tvrdenie:
   a. hodnotu `value` zapíšeme do súboru (smerník `f1`) príkazom `fputc(value,f1);`
   b. do súboru sa zapíše text `1,2,8;`
   c. do súboru sa zapíše byte s binárnou hodnotou `10000000;`
   d. do súboru sa zapíšu tri byty s binár. hodnotami `00000001,00000010,00001000.`

10. Majme súbor s dátami (smerník `f1`) vo formáte `raw`, kde kódované dáta predstavujú typ `double`. Majme premennú `value` typu `double`. Jej hodnotu do súboru zapíšeme príkazom:
    a. `fputc(value,f1);`
    b. `fputc(&value,f1);`
    c. `fwrite((char*)&value,1,8,f1);`
    d. `fwrite((char*)&value,8,1,f1);`

11. Majme súbor s dátami (smerník `f1`) vo formáte `raw`, kde kódované dáta predstavujú typ `short int`. Majme premennú `value` typu `short int`. Vyberte správne tvrdenia:
    a. ľubovoľnú zapísanú hodnotu môžeme získať príkazom `value = fputc(f1);`
    b. ľubovoľnú zapísanú hodnotu môžeme získať príkazom `value = fgetc(f1);`
    c. ľubovoľnú zapísanú hodnotu môžeme získať príkazom `fread((char*)&value,2,1,f1);`
    d. ľubovoľnú zapísanú hodnotu môžeme získať príkazom `fread((char*)&value,1,2,f1);`

12. Vyberte nesprávne tvrdenia:
    a. *raw* súbor je súbor, v ktorom sú dáta uložené ako v pamäti počítača
    b. číslo 1.2 chápané ako `float` zaberie v súbore *raw* 3 byty
    c. číslo 1.2 chápané ako `float` zaberie v súbore *raw* 4 byty
    d. číslo 1.2 chápané ako `float` zaberie v textovom súbore 3 byty

Kontrolné otázky

1. Charakterizujte rozdiel medzi lokálnymi a globálnymi premennými.
2. Charakterizujte rozdiel medzi definíciou a deklaráciou. Aký je rozdiel medzi lokálnou a globálnou deklaráciou?

3. Aká je výhoda registrovej premennej? Musí byť premenná triedy register vždy uložená do registra?
4. Charakterizujte automatickú premennú.
5. Aký je rozdiel medzi automatickou a statickou premennou?
6. Uveďte príklad definície smerníka na funkciu, poľa smerníkov na funkciu a inicializácie poľa smerníkov na funkcie. V akých príkladoch môžeme pole smerníkov na funkciou s výhodou použiť?
7. Aký je rozdiel medzi binárnym *pgm* súborom a textovým *pgm* súborom? Ako môžeme previesť jeden na druhý? Aké funkcie môžeme použiť pre prácu s nimi?
8. Charakterizujte súbory formátu *raw*. Aké funkcie pre prácu s nimi môžeme použiť?

Cvičenia

1. Príklad 9 z predchádzajúcej kapitoly prerobte pomocou globálnych premenných.
2. Prerobte príklad 9 tak, aby aj samotná matica bola globálna premenná a nemusela sa funkcii predávať ako parameter.
3. Majme súbor obsahujúci riadky čísel rôznej dĺžky. Napíšte program, ktorý vypíše čiastkové súčty riadkov, t.j. najprv súčet 1.riadku, potom súčet 1. a 2. a nakoniec súčet 1. až *n*-tého. Na výpočet súčtu použite lokálnu statickú premennú.
4. Podľa vzoru z príkladu 2 nájdite vhodné príklady a prerobte ich s použitím registrových premenných. Zmerajte čas ich vykonávania a porovnajte s pôvodným príkladom.
5. Vytvorte si krátku databázu, ktorá bude obsahovať takéto položky: reťazcovú premennú s priezviskom študenta, celočíselnú premennú predstavujúcu vzdialenosť bydliska od školy v km a reálnu premennú predstavujúcu priemer známok. Napíšte 3 rôzne porovnávacie funkcie (pre každú zložku zvlášť) pre triediacu procedúru `qsort`. Vypíšte menu zobrazujúce kľúče, podľa ktorých sa bude databáza triediť, podobne ako v príklade 3. Podľa voľby použite príslušnú porovnávaciu funkciu. Použite pole smerníkov na funkcie.
6. Napíšte program, ktorý skonvertuje *pgm* obrázok typu ASCII do *pgm* obrázku typu *raw* a naopak.
7. Dorobte príklad 6 tak, aby bol funkčný. Naprogramujte opačnú funkciu, ktorá bude mať na vstupe súbor vo formáte *raw*, kde 4 byty predstavujú čísla typu `float` a načíta ich do poľa reálneho typu.
8. Majme súbor (textový), ktorý obsahuje čísla od 0 do 4000. Skonvertujte ho do *raw* súboru.

Zadania

1. Zistite, čo robí nasledujúci program:

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

#define  M 512
```

```
#define  N 512

char   *meno;
char   *meno_koniec,*meno_zac;
int DH,HH;
/*-----------------------------------------------------*/
void start()
{
 int i,j,k,ll;
 FILE  *f1,*f2;
 char *prem;

 prem=(char *)malloc(5);
 f1=fopen("vystup.raw","w");

 for (k=DH;k<=HH;k++)
 {

       sprintf(prem,"%d",k);
       strcat(meno,prem);
       strcat(meno,meno_koniec);
       f2=fopen(meno,"r");

   if(f2==NULL)
    {
     perror("\nSubor sa neotvoril\n");
     exit(-1);}

   for (j=0;j<N*M;j++)
      {
      ll=fgetc(f2);
      fputc(ll,f1);
      }
      fclose(f2);
}
    fclose(f1);
}

/*_________________________Hlavny program_____________________________/
int main()
{
  meno=(char *)malloc(45);
  meno_koniec=(char *)malloc(15);
  meno_zac=(char *)malloc(30);

  printf("zadaj zac.mena suboru,koniec mena suboru, dh,hh:");
  scanf("%s%s%d%d",meno_zac,meno_koniec,&DH,&HH);
```

```
  start();
  system("pause");
}
```

2. Napíšte program, ktorý z trojrozmerných dát typu *raw* s hodnotami od 0 do 255 urobí *i*-ty vodorovný rez a hneď ho dá vykresliť pomocou programu *xv* (alebo iného vhodného programu pre prácu s obrázkami).

11 Preprocesor programovacieho jazyka C

Hlavné podtémy kapitoly:

- Preprocesor. Makrá bez parametrov a makrá s parametrami. Príkaz `#define`.
- Vkladanie súborov príkazom `#include`. Niektoré preddefinované makrá.
- Podmienený preklad. Riadenie prekladu pomocou hodnoty konštantného výrazu.
- Riadenie prekladu pomocou definície makra. Makrá `getchar`() a `putchar`().

11.1 Preprocesor jazyka C

Preprocesor sme spomínali v prvej kapitole a charakterizovali sme ho ako súčasť prekladača, ktorá upravuje zdrojový súbor tak, aby mal prekladač ľahšiu prácu, napr. vynechá komentáre, zaisťuje správne vloženie hlavičkových (`.h`) súborov, rozvoj makier, atď. Výsledkom jeho práce je znova textový súbor, ten si však môžeme pozrieť iba vtedy, keď vieme spustiť preprocesor samostatne bez kompilátora, inak preprocesor odovzdá výsledky svojej práce priamo svojmu nadriadenému - kompilátoru. Až preprocesorom spracovaný text je vstupom pre samotný prekladač.

Doteraz sme využívali viac-menej nevedomky najčastejšie používaný príkaz preprocesora `#include`. Preprocesor však poskytuje aj ďalšie možnosti, z ktorých najdôležitejšie spomenieme v ďalšom texte. Zatiaľ najdôležitejšie charakteristiky prekladača vypíšeme do nasledujúceho zoznamu:

- spracováva zdrojový text programu pred použitím prekladača,
- uskutočňuje zámenu textu, napr. identifikátorov konštánt za im zodpovedajúce číselné hodnoty,
- vypusti zo zdrojového textu všetky komentáre,
- uskutočňuje podmienený preklad,
- riadok, ktorý je určený pre spracovávanie preprocesorom musí začínať znakom "#" ,
- nekontroluje syntaktickú správnosť programu.

11.1.1 Makro

Pod makrom budeme chápať nejaký jednoduchý text, ktorý sa nahradí nejakým iným textom, ktorý môže predstavovať číslo, inštrukciu alebo skupinu inštrukcií. Makrá môžu byť *bez parametrov*, vtedy sa text (zvyčajne krátky a zrozumiteľný) jednoducho nahradí iným textom, napr. nejakou číselnou konštantou, alebo pevne danou inštrukciou, ktorá nevyžaduje žiadne ďalšie vstupy. Makro *s parametrami* umožňuje text nahrádzať tak, že časť z nahrádzaného textu bude pevne daná a časť z neho bude premenná a bude závisieť od zadaných parametrov.

11.1.2 Príkaz `define` – makro bez parametrov

Medzi makrá bez parametrov patria predovšetkým symbolické konštanty (ďalej bude často používaný výraz konštanta v zmysle symbolickej konštanty), ktoré sa využívajú veľmi často. Ich účel je dvojaký: jednak slovne charakterizujú číslo, ktoré reprezentujú, čím robia program prehľadnejší, jednak sa ľahšie pomocou nich riadi zmena, pozri napríklad kapitolu 4, príklad 3, kde sme túto inštrukciu prvýkrát spomínali pri práci so statickými poliami. Konštanty sú

väčšinou definované na začiatku programu (modulu). Náhrada konštanty skutočnou hodnotou sa nazýva rozvojom (expanziou) makra alebo tiež substitúciou makra.

Pre písanie symbolických konštánt platia nasledujúce pravidlá:

- mena konštánt sa zvyčajne píšu veľkými písmenami,
- meno konštanty je od jej hodnoty oddelené aspoň jednou medzerou,
- za hodnotou môže byť (a mal by byť) komentár,
- nové konštanty môžu využívať už skôr definované konštanty,
- medzi menom konštanty a hodnotou nie je znak = ,
- konštanta sa môže objaviť kdekoľvek v programe s jedinou výnimkou - nemala by byť súčasťou reťazca (medzi úvodzovkami), pretože tam k rozvoju konštanty nedôjde - viď ďalší text,
- konštanta začína platiť od miesta definície a platí až do konca súboru, v ktorom bola definovaná.

Napr.

```
#define N                  1000
#define PI                  3.14
#define DVE_PI           (2 * PI)
#define MOD                   %
#define AND                  &&
#define BEGIN                {
#define END                     }
```

Priklad 1

Použitie symbolickej konštanty pre π.

```
#define DVE_PI  (2*3.14)
main() {
    double r;

    printf("Zadaj polomer : ");
    scanf("%lf",&r);
    printf("Obvod kruhu s polomerom %lf je %lf\n",r,r * DVE_PI);
}
```

Číslo π je v C definované s oveľa väčšou presnosťou. Ako využiť presnú hodnotu si povieme ďalej.

Makro sa nerozvinie, ak je uzatvorené v úvodzovkách.

Napr:

```
#define N 200

main() {
    printf("Matica ná N riadkov");
}
```

vytlačí: `Matica má N riadkov, a nie Matica má 200 riadkov`

V prípade, že chceme konštantu predefinovať, je nutné najprv starú definíciu zrušiť použitím direktívy `#undef`, napr.:

```
#define POCET 10  /*  stara definicia POCET    */
```

```
#undef  POCET     /*  POCET stratila platnost  */
#define POCET 20  /*  nova definicia POCET  */
```

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

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

Ak je text makra dlhší ako riadok, musí byť na konci riadku znak "\", ktorý sa ale do makra nerozvinie - je to iba pomocný znak. Hneď za spätným lomítkom musí nasledovať ENTER.

11.1.3 Makrá s parametrami

Pri riešení programov sa často vyskytne prípad, keď mnohokrát používame nejakú funkciu, ktorá je veľmi krátka, napr. uskutočňuje jednoduchý výpočet nejakej hodnoty. Takúto funkciu je možné samozrejme napísať bez problémov, ale potom niekedy nastáva problém s efektivitou programu. Ak je totiž funkcia veľmi krátka, je niekedy jej "administratíva", t. j. *predávanie parametrov, úschova návratovej adresy, skok do funkcie, navráť z funkcie do miesta volania a výber použitých parametrov*, dlhší než samotný kód funkcie. Jazyk C ponúka možnosť, ako túto administratívu celkom potlačiť použitím makier s parametrami. Ak inštrukcie tvoriace funkciu zmeníme na makro, tak tie sa vložia pri každom „volaní“ funkcie, kde pod funkciou skôr myslíme funkcionalitu ako konštrukciu jazyka C. Program sa predĺži, ale odbúra sa spomínaná „administratíva“. Na programátorovi je, aby vybral, čo je výhodnejšie: alebo bude program kratší, ale pomalší, alebo bude dlhší, ale rýchlejší. Makrá s parametrami sa niekedy nazývajú aj *vkladané funkcie (inline functions)*, pretože, na rozdiel od skutočných funkcií, sa makrá s parametrami nevolajú, ale pred prekladom nahradí preprocesor meno makra konkrétnym textom. Praktické použitie teda je len pre veľmi krátke akcie, kde by administratíva funkcie trvala zrovnateľnú dobu s vlastným výpočtom funkcie.

Syntax makra s parametrami je nasledujúca:

```
#define meno_makra(arg1,...,argN) hodnota_makra
```

Syntax volania makra potom vyzerá takto:

```
meno_makra(par1,...,parN)
```

Medzi menom makra a otvárajúcou okrúhlou zátvorkou "(" nesmie byť medzera. Argumenty by potom boli považované za hodnotu makra.

Priklad 2

Napíšme makro pre výpočet plochy kruhu. Makro bude mať jeden parameter, a to polomer `r`.

```
#define PI 3.1415
#define plocha(r) PI*r*r
...
o1=plocha(x);
o2=plocha(10);
```

Makro sa rozvinie do príkazu:

```
o1= PI*x*x;
o2= PI*10*10;
```

Odporúčaný vzhľad programov písaných v jazyku C:

- na rozdiel od symbolických konštánt, ktorých mená sa píšu veľkými písmenami, sa mená makier s parametrami píšu malými písmenami, rovnako ako mená funkcií.

Cvičenie 1

Pre cyklus C by sme mohli napísať nasledujúce makro:

```
#define FOR(i,a,b) for(i=a; i<b; i++)
```

Príklad 3

V kapitole 4 sme definovali funkciu, ktorá vráti minimum dvoch čísel. Jej definícia bola takáto:

```
 int min(int a, int b)
{
  if (a <= b)
    return a;
  else
    return b;
}
```

Túto funkciu sme použili pri hľadaní minimálneho prvku poľa. Takéto použitie funkcie robilo síce program (uvedený nižšie) prehľadnejším, ale nebolo veľmi efektívne. Napríklad aj v prípade, že testovaný prvok bol väčší ako minimum, priraďovali sme do minima tú istú hodnotu, ktorá v ňom bola. Okrem toho pribudla administratíva, spomínaná v úvode paragrafu. Makro, ktorým nahradíme funkciu, bude hodnotu minima meniť len v prípade, že sa tá zmenší. Makro tiež program sprehľadní, ale navyše program bude efektívnejší ako v prípade použitia funkcie.

Zmena funkcie na makro bude:

```
#define minim(a,b) {if (b<a) a=b;}
```

Použime funkciu aj makro v nasledujúcom príklade z kapitoly 4, ktorý nájde minimálny prvok poľa.

Príklad 4

```
#define n 10
#define minim(a,b) {if (b<a) a=b;}
#define FOR(i,a,b) for(i=a; i<b;i++)
int min(int a, int b)
{
  if (a<=b)
    return a;
  else
    return b;
}
main()
{
int i,minimum;
int pole[n]={3,2, 5, 7, -1,45,23,4,12,0};
/*spracovanie pola pomocou funkcie*/
minimum=pole[0];
FOR(i,1,n)
minimum=min(minimum,pole[i]);
printf("minimum:%d\n",minimum);
```

```
/*spracovanie pola pomocou makra*/
minimum=pole[0];
FOR(i,1,n)
minim(minimum,pole[i])
printf("minimum:%d\n",minimum);
}
```

Keď sa chceme pozrieť ako sa makro rozvinulo, musíme vedieť preprocesor spustiť samostatne bez kompilátora. Majme program uložený v súbore, ktorý sa volá `subor.c` a použime takýto príkaz:

`gcc -E -o subor.i subor.c`

Výsledkom je textový súbor s menom `subor.i`, ktorý si môžeme prezrieť v ľubovoľnom editore.

Program po spracovaní preprocesorom je takýto:

```
# 1 "subor.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "subor.c"

int min(int a, int b)
{
  if (a<=b)
    return a;
  else
    return b;
}

main()
{
int i,minimum;
int pole[10]={3,2, 5, 7, -1,45,23,4,12,0};

minimum=pole[0];
for (i=1;i<10;i++)
minimum=min(minimum,pole[i]);
printf("minimum:%d\n",minimum);

minimum=pole[0];
for (i=1;i<10;i++)
{if (pole[i]<minimum) minimum=pole[i];}
printf("minimum:%d\n",minimum);
}
```

Všimnite si, že sa vypustili komentáre, rozvinula sa symbolická konštanta `n`, cyklus `for` aj samotné makro `minim`.

11.1.4 Preddefinované makrá

Tak ako je v C veľa preddefinovaných konštánt a funkcií, tak sú v ňom aj preddefinované makrá. Sú to napríklad makrá pre určenie typu znaku, makrá pre konverziu znaku apod. Mnohé z nich sú uložené v rôznych hlavičkových súboroch.

Súbor, v ktorom je uvedené množstvo užitočných makier, je súbor `ctype.h`. Makrá v ňom

definované pracujú so znakmi a delia sa do dvoch skupín:

- Makrá pre určenie typu znaku:
  Tieto makrá začínajú písmenami `is`, napr. `isdigit(c)`, ktoré vracia znak v premennej `c` (nenulovú hodnotu), ak je v `c` znak `číslice`, a inak vracia nulu `(FALSE)`. Nasledujúca tabuľka uvádza niektoré makrá z tohto súboru. Uvedená návratová hodnota zodpovedá kladnému prípadu, čiže prípad, ktorý makro testuje, aj nastal:

vracia	meno	rozsah pouzitia
znak	`isalnum`	číslice a malé a veľké písmena
znak	`isalpha`	malé a veľké písmena
1	`isascii`	ASCII znaky (0 až 127)
znak	`iscntrl`	CTRL znaky (1 až 26)
znak	`isdigit`	číslica
znak	`islower`	malé písmena
1	`isprint`	tlačiteľné znaky (32 až 126)
znak	`ispunct`	interpunkčné znaky(bodka, čiarka, lomítko,...)
znak	`isspace`	biele znaky(medzera, tabulátor, novy riadok, ...)
znak	`isupper`	veľké písmena
znak	`isxdigit`	hexadecimálne číslice('0' - '9', 'A' - 'F', 'a' - 'f')
1	`isgraph`	znak s grafickou podobou (33 až 126)

- Makrá pre konverziu znaku:
  Tieto makrá začínajú písmenami `to`, napr.: `b = tolower(c);` ktoré konvertuje veľké písmeno v `c` na malé; ostatné znaky ponechá nezmenené a parameter `c` ponechá taktiež nezmenený.

`tolower`	konverzia na malé písmena
`toupper`	konverzia na veľké písmena
`toascii`	prevod na ASCII - len najnižších 7 bitov je významných

Ak chceme používať vyššie uvedené makrá, je nutné na začiatku programu použiť okrem príkazu:

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

aj príkaz:

```
#include <ctype.h>
```

U makier s parametrami nie je možné – na rozdiel od funkcií - použiť rekurziu.

V príklade 5 budeme potrebovať preddefinované funkcie (presnejšie by bolo povedať makrá) `putchar()` a `getchar()` s takýmito prototypmi:

```
int  getchar(void)
int  putchar(int znak)
```

Jediným parametrom funkcie `putchar` je znak, návratová hodnota je ten istý znak, ale len v prípade, že všetko prebehlo korektne. V opačnom prípade je návratovou hodnotou `EOF`. Funkcia `getchar` slúži na načítanie jedného znaku zo štandardného vstupu (väčšinou z klávesnice) a vracia celé číslo reprezentujúce jeden znak. V prípade chyby, ale aj načítania konca súboru je vrátená hodnota `EOF`.

Príklad 5

V tomto príklade načítame jeden riadok z klávesnice. Ten istý riadok vypíšeme, ale tak, že malé písmená zmeníme na veľké, pričom ostatné znaky ostanú nezmenené. Funkcia `getchar()` načíta jeden znak z terminálu a funkcia `putchar()` vypíše zadaný znak na terminál. Najprv načítame znaky do textového poľa a spočítame ich. Potom znaky vypíšeme tak, že najprv testujeme, či predstavujú alebo nepredstavujú písmeno. Ak áno, skonvertujeme ho na veľké a vypíšeme. Ostatné znaky jednoducho vypíšeme.

```
#include <stdio.h>
#include <ctype.h>
main(){
char pole[80];
int i=0,n=0;
while (((pole[n]=getchar()) !='\n')&& n<80)  n++;
for (i=0;i<n;i++)
   if (isalpha(pole[i]))
          putchar(toupper(pole[i]));
   else
          putchar(pole[i]);
 }
```

Rozumnejšie sa javí testovať, či písmeno je malé a len v tom prípade urobiť konverziu.

```
#include <stdio.h>
#include <ctype.h>
main(){
int pole[80];
int i=0,n=0;
while ((pole[n]=getchar()) !='\n') n++;
for (i=0;i<n;i++)
  if (islower(pole[i]))
     putchar(toupper(pole[i]));
  else
     putchar(pole[i]);
 }
```

Majme vstupný riadok:

```
asdgfhh\^^&*^&^*(12547asdfghJKIUY
```

Výstupom programu bude napríklad:

```
ASDGFHH\^^&*^&^*(12547ASDFGHJKIUY
```

Príklad 6

Vezmime príklad 3 z predchádzajúcej kapitoly. Časť s menu so zadávaním čísla by sme mohli prerobiť takto:

```
SLUCKA:
    printf("Zadaj volbu:\n");
```

```
c=getchar();              /*nacitanie znaku z klavesnice */
while(getchar()!='\n');   /*vyprazdnenie vstup.  buffra*/
if (isdigit(c)) {         /* je c cislica?*/
    i=c-'0'-1;  /*prevod cislice na cislo -1*/
    if((i>=0)&&(i<4)) {(*menu[i])(pic); goto SLUCKA;}
   else {printf("nespravna cislica\n");goto SLUCKA;}}
else {printf("nespravny znak\n");goto SLUCKA;}
```

Pri zadávaní čísla menu musíme okrem samotného čísla zadať minimálne znak ENTER. Všetky zadané znaky včítane znaku pre koniec riadku sa zapíšu do vstupného buffra. Ďalšie volanie funkcie `getchar` číta z tohto buffra, ktorý sa v momente, keď je z neho prečítaný znak konca riadku, vyprázdni. Preto je rozumné, po načítaní čísla buffer vyprázdniť, lebo ďalšie volanie by pravdepodobne načítalo znak konca riadku.
Po otestovaní, či načítaný znak je číslica, ho skonvertujeme na číslo tak, že od neho odčítame ASCII kód číslice nula, pričom využívame, že číslice sú v ASCII tabuľke za sebou.

11.2 Vkladanie súborov - príkaz `include`

S týmto príkazom preprocesoru sme sa stretli napr. v príkaze:

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

Vkladanie súborov v C sa používa v programoch veľmi často - v podstate vždy. Najčastejšie používaný príkaz pre preprocesor je práve:

```
#include <meno_suboru>
```

alebo

```
#include "meno_suboru"
```

ktoré spôsobia, že kód zo súboru `meno_suboru` bude vložený do zdrojového súboru na miesto, kde sa v ňom nachádza príkaz `#include`. V súbore, ktorý je vkladaný, môžu byť ďalšie príkazy `#include`, ale v takom prípade treba dávať veľký pozor na zacyklenie. Ako už bolo uvedené, príkaz `#include` má dva tvary, ktoré určujú, kde sa má špecifikovaný súbor hľadať:

1) Príkaz:

   ```
   #include "DEFINICIE.H"
   ```

   hľadá súbor DEFINICIE.H v rovnakom adresári, v ktorom leží "volajúci" súbor. Ak ho tam nenájde, je možné, že ho bude ďalej hľadať v ďalších adresároch, čo však už závisí na konkrétnej implementácii. Používa sa pre pracú so súbormi, ktoré sme vytvorili my sami.

2) Príkaz:

   ```
   #include <ctype.h>
   ```

   hľadá súbor `ctype.h` v systémovom adresári. Používa sa pre prácu s už hotovými špeciálnymi súbormi, ktorým sa hovorí *štandardné hlavičkové súbory.*

11.2.1 Vkladané súbory

Súbory, ktoré sa dajú vkladať pomocou príkazu `#include`, môžu byť ľubovoľné textové súbory. Význam má samozrejme vkladanie len zdrojových súborov - s príponou `.c`. Druhá - a oveľa častejšia - možnosť, je vkladanie tzv. hlavičkových súborov - s príponou `.h`. Vkladanie hlavičkových súborov je veľmi užitočný mechanizmus, ako program pozostávajúci z

viacerých súborov udržať čitateľný. Napríklad všetky definície konštánt využívané viacerými súbormi sa uvedú iba raz do súboru typu `.h`, ktorý sa pomocou `#include` pripojí do všetkých súborov, ktoré tieto definície konštánt potrebujú. To má obrovskú výhodu, že prípadná zmena konštánt sa potom uskutoční iba v jednom súbore `.h` a ostatne súbory, využívajúce tieto konštanty, stačí iba preložiť.

S predstaviteľmi týchto súborov - súborom `stdio.h` - sme sa už mnohokrát stretli. Každá implementácia prekladača C má niekoľko desiatok štandardných `.h` súborov a v každom z týchto súborov je popísaná časť funkcií a konštanty zo štandardnej knižnice. To, čo tieto súbory popisujú, je definované ANSI normou, čo má tu veľkú výhodu, že program v C, ktorý pomocou týchto súborov využíva iba štandardnú knižnicu, by mal byť v podstate na 100% prenositeľný medzi najrôznejšími počítačmi a najrôznejšími operačnými systémami. Súbor `stdio.h` obsahuje okrem definícii základných vstupných a výstupných funkcii aj definície rozličných konštánt a typov napr. `getc(), EOF, NULL, FILE,` atď. V `.h` súboroch nie sú uvedené celé zdrojové texty príslušných funkcii, ale iba ich hlavičky (tzv. funkčné prototypy), t. j. opisy, aké má tá ktorá funkcia parametre a aký typ hodnoty vracia. S využitím týchto vedomostí potom môže prekladač zistiť, že je napr. knižničná funkcia pre vstup znaku z klávesnice volaná v programe ako: `getchar(c);` použitá chybne. V súbore `stdio.h` je totiž jej funkčný prototyp uvedený bez parametrov ako: `int getchar()`. Dôležité je, že všetky štandardné `.h` súbory sú normálne textové súbory, ktoré je možné si prezerať ľubovoľným editorom. Z mnohých ďalších štandardných `.h` súborov sa zmienime iba o `math.h` a `mathcalls.h,` ktoré sú využívané na prácu s matematickými funkciami ako sú `sin, cos` atď. a matematickými konštantami.

11.2.1.1 Definícia matematických konštánt

Dá sa nájsť v hlavičkovom súbore `math.h`. Aj keď nebudeme rozumieť všetkému, čo je v hlavičkových súboroch popísané, dajú sa nájsť časti, ktoré sú nám užitočné a ktoré sú zrozumiteľné. Medzi také patrí napríklad definícia konštánt.

```
/* Some useful constants.  */
# define M_E            2.7182818284590452354   /* e */
# define M_LOG2E        1.4426950408889634074   /* log_2 e */
# define M_LOG10E       0.43429448190325182765  /* log_10 e */
# define M_LN2          0.69314718055994530942  /* log_e 2 */
# define M_LN10         2.30258509299404568402  /* log_e 10 */
# define M_PI           3.14159265358979323846  /* pi */
# define M_PI_2         1.57079632679489661923  /* pi/2 */
# define M_PI_4         0.78539816339744830962  /* pi/4 */
# define M_1_PI         0.31830988618379067154  /* 1/pi */
# define M_2_PI         0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI     1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2        1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2      0.70710678118654752440  /* 1/sqrt(2) */
#endif
```

Príklad 7

Príklad 2 z tejto kapitoly by sme teda mohli prerobiť pomocou preddefinovanej konštanty tak, že by počítal presnejšie:

```
#define PI 3.14
#define plocha(r) PI*r*r
main(){
```

```
double o1,o2;
o1=plocha(2);
o2=plocha(3);
printf("o1: %lf, o2: %lf\n",o1,o2); }
```

Výpis programu je:

```
o1: 12.560000, o2: 28.260000
```

```
#include <math.h>
#define plocha(r) M_PI*r*r
main(){
double o1,o2;
o1=plocha(2);
o2=plocha(3);
printf("o1: %lf, o2: %lf\n",o1,o2);
}
```

Výpis programu je:

```
o1: 12.566371, o2: 28.274334
```

Ak v súbore často používame konštantu π, môžeme použiť tento príkaz, ktorým nahradíme systémový názov pre konštantu:

```
#define PI  M_PI
```

11.2.1.1 Definícia matematických funkcií

Dá sa nájsť v hlavičkovom súbore `mathcalls.h`. Z definícií sa dá porozumieť, aká funkcia je zadefinovaná a aké potrebuje vstupy. Vyberme niekoľko príkladov.

```
/* Trigonometric functions.  */

__Mdouble_BEGIN_NAMESPACE
/* Arc cosine of X.  */
__MATHCALL (acos,, (__Mdouble_ __x));
/* Arc sine of X.  */
__MATHCALL (asin,, (__Mdouble_ __x));
/* Arc tangent of X.  */
__MATHCALL (atan,, (__Mdouble_ __x));
/* Arc tangent of Y/X.  */
__MATHCALL (atan2,, (__Mdouble_ __y, __Mdouble_ __x));

/* Cosine of X.  */
__MATHCALL (cos,, (__Mdouble_ __x));
/* Sine of X.  */
__MATHCALL (sin,, (__Mdouble_ __x));
/* Tangent of X.  */
__MATHCALL (tan,, (__Mdouble_ __x));
/* Hyperbolic functions.  */
* Hyperbolic cosine of X.  */
__MATHCALL (cosh,, (__Mdouble_ __x));
/* Hyperbolic sine of X.  */
__MATHCALL (sinh,, (__Mdouble_ __x));
/* Hyperbolic tangent of X.  */
__MATHCALL (tanh,, (__Mdouble_ __x));
__Mdouble_END_NAMESPACE

...
/* Exponential and logarithmic functions.  */
```

```
_Mdouble_BEGIN_NAMESPACE
/* Exponential function of X.  */
__MATHCALL (exp,, (_Mdouble_ __x));

/* Natural logarithm of X.  */
/*/* Base-ten logarithm of X.  */
__MATHCALL (log10,, (_Mdouble_ __x));

/* Break VALUE into integral and fractional parts.  */
__MATHCALL (modf,, (_Mdouble_ __x, _Mdouble_ *__iptr));
_Mdouble_END_NAMESPACE

#ifdef __USE_GNU
/* A function missing in all standards: compute exponent to base
ten.  */
__MATHCALL (exp10,, (_Mdouble_ __x));
/* Another name occasionally used.  */
__MATHCALL (pow10,, (_Mdouble_ __x));
#endif
...
```

Ak nám definícia nie je jasná, môžeme použiť manuál a získať podrobnejšie informácie o funkcii. Napríklad, potrebovali by sme podrobnejší popis funkcie `modf`. Pod operačným systémom linux môžeme použiť príkaz `man` k vyvolaniu manuálu. Po zadaní príkazu `man modf`, sa zobrazí nápoveda, z ktorej vyberáme:

SYNOPSIS

```
#include <math.h>

double modf(double x, double *iptr);
```

DESCRIPTION

The `modf()` function breaks the argument `x` into an integral part and a fractional part, each of which has the same sign as `x`. The integral part is stored in `iptr`.

RETURN VALUE

The `modf()` function returns the fractional part of x.

Príklad 8

Podľa manuálu sa zdá, že funkcia vráti reálnu a celú časť zadaného reálneho čísla. Reálna časť sa vráti ako návratová adresa a celá časť sa vráti do smerníka, ktorý je druhým argumentom funkcie. Použime dve preddefinované konštanty, π a *e*.

```
#include <math.h>

main(){
double pi_r,pi_c,e_r,e_c;
pi_r=modf(M_PI,&pi_c);
e_r=modf(M_E,&e_c);
printf("pi_r: %lf, pi_c: %lf\n",pi_r,pi_c);
printf("e_r: %lf, e_c: %lf\n",e_r,e_c);
}
```

Výpis programu je (nezabudnite prilinkovať matematickú knižnicu):

```
pir: 0.141593, pii: 3.000000
er: 0.718282, ei: 2.000000
```

Viac o hlavičkových súboroch a v nich definovaných konštantách, makrách a funkciách sa dá nájsť napríklad v knihe: Pavel Herout: Učebnica jazyka C, 2. diel[].

11.3 Podmienený preklad

V mnohých prípadoch sa zložitejšie programy píšu tak, že obsahujú ladiacu časť. To sú najčastejšie pomocné výpisy, ktoré majú uľahčiť ladenie, ale môžu to byť napr. i funkcie, ktoré dozerajú na medze polí atď. Tieto časti sa do programov dávajú, i keď máme k dispozícii výkonný debugger. Je dobrým zvykom počítať už pri návrhu programu s tým, že ho bude nutné ladiť a už pri vytváraní programu tieto časti do programu zaraďovať. Po odladení programu však nastáva typicky problém - ako tieto ladiace časti, ktoré vypisujú už nepotrebné (a teda nevhodné) informácie a zdržujú vykonávanie programu, z odladeného programu odstrániť. Najjednoduchším riešením je pomocou editoru prejsť celý program a ladiace časti jednoducho vymazať. Toto riešenie však so sebou prináša niektoré úskalia, napríklad že vymažeme i to, čo sme vymazať nemali alebo že ladiace časti boli nejakým spôsobom potrebné i pre skutočný program - najčastejšie sa jedná o odovzdávané premenné definované v ladiacej časti a využívané i v programe. Použitie podmieneného prekladu je však širšie, používa sa pri tvorbe veľkých systémov, zvyčajne keď je preklad od niečoho závislý, napríklad od nastavenia nejakej systémovej premennej.

V jazyku C sa tento problém rieši tak, že pomocou príkazu preprocesoru môžeme určiť, ktoré časti programu sa majú prekladať podmienene. To znamená, že všetky ladiace časti už pri vytváraní programu označíme ako podmienené a pri ladení ich prekladáme a po odladení ich neprekladáme. Ladiace časti programu sú tak trvalou súčasťou zdrojového súboru. Preprocesor potom na náš jeden príkaz všetky tieto časti vypusti sám, ale ak to bude niekedy v budúcnosti opäť potrebné, potom ich jedným príkazom opäť do programu zaradí.

Podmienený preklad je riadený dvoma spôsobmi, ktoré budú popísané v nasledujúcich dvoch paragrafoch.

11.3.1 Riadenie prekladu hodnotou konštantného výrazu

V tomto prípade je preklad riadený konštantným výrazom, čo môže byť číslo, symbolická konštanta alebo i podmienený výraz vytvorený z týchto možnosti. Má túto syntax:

```
#if konstantny_vyraz
    cast_1
#else
    cast_2
#endif
```

Prekladač bude túto časť programu spracovávať tak, že ak je hodnota `konstantny_vyraz` rovná 0 `(FALSE)`, prekladá sa iba `cast_2` a v opačnom prípade (nenulová hodnota - `TRUE)` sa prekladá iba `cast_1`. Časť `#else` a `cast_2` môžu byť vynechané.

11.3.2 Riadenie prekladu definíciou makra

Podmienený preklad riadený hodnotou konštantného výrazu - pozri predchádzajúci odstavec - je silný nástroj, ale oveľa častejšie sa pre riadenie podmieneného prekladu používa jeho jednoduchšia verzia, ktorá je závislá iba na tom, či bola určitá symbolická konštanta definovaná alebo nie. Nebudeme tu uvádzať syntax, ktorá je veľmi podobná syntaxi predchádzajúcej a uvedieme hneď modifikovaný príklad použitia z predchádzajúceho odstavca.

```
#ifdef KONST
     cast_1
#else
    cast_2
#endif
```

V porovnaní s predchádzajúcim prípadom je hlavná zmena v tom, že ak chceme používať časť 1 , napíšeme príkaz

```
#define KONST
```

a ak chceme používať časť 2 , napíšeme príkaz

```
#undef  KONST
```

alebo jednoduchšie symbolickú konštantu KONST vôbec nedefinujeme.

Príklad 9

V tomto príklade sa vrátime k príkladu 4 z tejto kapitoly. V prípade, že bude definovaná konštanta CAS, dáme vypísať čas, za ktorý bolo zistené minimum s použitím funkcie a s použitím makra. Veľkosť poľa je kvôli porovnaniu predefinovaná a do poľa sú vygenerované náhodné čísla.

```
#include <time.h>
#define n 10000000
#define CAS 1
#define minim(a,b) {if (b<a) a=b;}
#define FOR(i,a,b) for(i=a; i<b;i++)

int min(int a, int b)
{
  if (a<=b)
    return a;
  else
    return b;
}
main()
{
int i,minimum;
int pole[n];
FOR(i,1,n) pole[i]=rand()%500000-500000;
clock_t zac,koniec;

/*spracovanie pola pomocou funkcie*/
zac= clock();
minimum=pole[0];
FOR(i,1,n)
minimum=min(minimum,pole[i]);
printf("minimum:%d\n",minimum);

#ifdef CAS
koniec=clock();
printf("S funkciou to trvalo :%lf [sec]\n",
     (koniec-zac)/(double)CLOCKS_PER_SEC);
#endif
```

```
/*spracovanie pola pomocou makra*/
Zac = clock();
minimum=pole[0];
FOR(i,1,n)
minim(minimum,pole[i])
printf("minimum:%d\n",minimum);

#ifdef CAS
koniec=clock();
printf("S makrom to trvalo :%lf [sec]\n",
     (koniec-zac)/(double)CLOCKS_PER_SEC);
#endif
}
```

Súbor po spracovaní preprocesorom vyzerá takto:

```
int min(int a, int b)
{
  if (a<=b)
    return a;
  else
    return b;
}

main()
{
int i,minimum;
int pole[10000000];
for(i=1; i<10000000;i++) pole[i]=random()%500000-250000;
clock_t zac,koniec;

zac= clock();
minimum=pole[0];
for(i=1; i<10000000;i++)
minimum=min(minimum,pole[i]);
printf("minimum:%d\n",minimum);

koniec=clock();
printf("S funkciou to trvalo :%lf [sec]\n",
     (koniec-zac)/(double)10000001);

zac= clock();
minimum=pole[0];
for(i=1; i<10000000;i++)
{if (pole[i]<minimum) minimum=pole[i];}
printf("minimum:%d\n",minimum);

koniec=clock();
printf("S makrom to trvalo :%lf [sec]\n",
     (koniec-zac)/(double)10000001);
}
```

Príklad výpisu programu:

```
minimum:-250000
S funkciou to trvalo :0.250000 [sec]
minimum:-250000
S makrom to trvalo :0.060000 [sec]
```

Ak konštantu `CAS` nebudeme definovať, teda na začiatok programu napíšeme

```
#undef CAS,
```

súbor po spracovaní preprocesorom bude vyzerať takto:

```
int min(int a, int b)
{
  if (a<=b)
    return a;
  else
    return b;
}

main()
{
int i,minimum;
int pole[10000000];
for(i=1; i<10000000;i++) pole[i]=random()%500000-500000;
clock_t zac,koniec;
zac= clock();
minimum=pole[0];
for(i=1; i<10000000;i++)
minimum=min(minimum,pole[i]);
printf("minimum:%d\n",minimum);
# 37 "podmienka.c"
zac= clock();
minimum=pole[0];
for(i=1; i<10000000;i++)
{if (pole[i]<minimum) minimum=pole[i];}
printf("minimum:%d\n",minimum);
}
```

Program vypísal iba minimá.

V oboch prípadoch boli zo súboru vynechané definície, ktoré boli zaradené pri vložení štandardných hlavičkových súborov.

Na záver zosumarizujme, že je dobré si uvedomiť, že :

- všetky parametre v definícii makra s parametrami by mali byť uzavreté do zátvoriek,
- každú použitú konštantu definujme ako symbolickú, a to hneď na začiatku programu - zlepšuje to prenositeľnosť programu a jeho čitateľnosť,
- používajme makra s parametrami pre skrytie dlhých, opakujúcich sa a komplikovaných výrazov - významne to zlepšuje čitateľnosť programu,
- používajme podmienenú kompiláciu pre vynechanie ladiacich časti programu,
- ak hlási prekladač nejakú chybu, na ktorú nemôžeme prísť, je vhodne prezrieť si súbor po spracovaní preprocesorom. Je možné, že chyba bude v rozvoji niektorého makra.

Zoznam konštrukcii, ktoré rozpoznáva preprocesor jazyka C:

definovanie symbolickej konštanty	`#define KONST`
zrušenie definície konštanty	`#undef KONST`
definovanie makra	`#define meno text_rozvoja`

podmienený preklad textu v závislosti na hodnote konštantného výrazu rozvoji makra	`#if konstantny_vyraz`
	`  cast_1`
	`#else`
	`  cast_2`
	`#endif`

riadenie prekladu definíciou makra	`#ifdef KONST`
	`  cast_1`
	`#else`
	`  cast_2`
	`#endif`

vloženie textu zo špecifikovaného súboru v adresári užívateľa	`#include "filename"`
vloženie textu zo špecifikovaného súboru v systémovom adresári	`#include <filename>`

☑ Kontrolný test

1. Vyberte správne tvrdenia:
   a. úlohou preprocesora je robiť kontrolu syntaxe,
   b. preprocesor uskutočňuje zámenu textu, napr. identifikátorov konštánt za im zodpovedajúce číselné hodnoty,
   c. riadok určený preprocesoru začína znakom !,
   d. preproceso upraví vzhľad všetkých komentárov.

2. Vyberte správne tvrdenie:
   a. medzi makrami s parametrami a funkciami nie je žiaden rozdiel
   b. rozvoj makra predstavuje vloženie textu podľa definície makra na miesto, kde sa makro vyskytuje
   c. inštrukcie funkcie v C sa vložia vždy na miesto, kde je funkcia zavolaná
   d. v prípade, že funkcia aj makro sú si zodpovedajúco zostavené, program používajúci makrá je dlhší ako program používajúci funkciu,.

3. Po spracovaní preprocesorom získame:
   a. textový súbor
   b. objektový súbor
   c. zlinkovaný súbor
   d. binárny súbor.

4. Majme nasledujúce definície makier bez parametrov:

```
#define v1x 0.8
#define v2x 0.1
#define v3x 0.3
#define suma v1x + v2x + v3x
```

a premennú `y` typu `double`. Výsledkom priradenia `y=suma/3 bude:`

a. `y = 0.4`
b. `y = 0.04`
c. `y = 0.1`
d. `y = 1.`

5. Majme nasledujúcu definíciu makra s parametrami:

```
#define polynom(x)  (x*x + 2*x + 2)
```

a príkazy:

```
int a = 1, b;
b = polynom(a+1);
```

Honodta v `b` bude rovná:

a. `b = 8`
b. `b = 10`
c. `b = 7`
d. `b = 9.`

6. Vyberte správne definované makro:

a. `#define mod(a,b)  a%b`
b. `#define mod (a,b)  a%b`
c. `#define vypis(typ,a) printf(„vysledok vypoctu: %typ",a)`
d. `#define vyp(a,b) printf("vysl.%d. kroku vypoctu:%d\n",\`
`a,b)`

7. Majme definíciu typu `int z = 'a'.` Vyberte nesprávne tvrdenie:

a. `putchar(z)` vypíše znak uložený v `z` na obrazovku
b. `putchar(z)` vypíše na obrazovku znak `z`
c. `putchar('z')` vypíše na obrazovku znak `z`
d. `putchar(97)` vypíše na obrazovku znak `a.`

8. Chceme použiť preddefinovanú matematickú konštantu pre odmocninu z 2. Vyberte správne tvrdenie:

a. netreba vložiť žiadny hlavičkový súbor, stačí prilinkovať matematickú knižnicu
b. treba vložiť `"math.h"`
c. treba vložiť `<math.h>`
d. treba vložiť `<mathcalls.h>` .

9. V ASCII tabuľke idú malé písmená za veľkými. Vyberte správne tvrdenia. Malé písmeno v `z` môžeme zmeniť na veľké príkazom:

a. `z = toupper(z);`

b. `z = z + 'A'-'a';`
c. `z = z + 'a'-'A';`
d. `z = tolower(z).`

10. Majme definíciu `int pole[30].` Vyberte správne tvrdenie. Príkaz:
```
while (((pole[n]=getchar())!='\n') && (n<20)) n++;
```
načítavajúci znaky z klávesnice :
a. načíta do `pole[30]` 20 znakov
b. načíta do `pole[30]` maximálne 20 znakov
c. načítané znaky budú vždy ukončené znakom konca riadku
d. v prípade, že `n` bolo maximálne 19, znaky budú ukončené znakom konca riadku.

11. Majme príkazy :
```
main()
{int x,y=0;
  #if STAV
    x=10;
  #else
    x=100;
  #endif
  while (x>0)
    {x--;y+=2;}
    }
```
Vyberte správne tvrdenia:
a. ak `STAV` získame definíciou `#define STAV 1,` do programu sa vloží riadok `x=10;` a `y` bude rovné 20
b. ak `STAV` získame definíciou `#define STAV 0,` do programu sa vloží riadok `x=100;` a `y` bude rovné 198
c. ak `STAV` získame definíciou `int STAV = 1,` do programu sa vloží riadok `x=10;` a `y` bude rovné 18
d. ak `STAV` získame definíciou `int STAV = 0,` do programu sa vloží riadok `x=100;` a `y` bude rovné 200.

12. Majme príkazy :
```
main()
{int x,y;
#ifdef STAV
   x=15; y=4;
 #else
   x=23; y=2;
#endif
 while (x>0)
  x=x/y;
}
```

Vyberte správne tvrdenia:

a. ak `STAV` získame definíciou `#define STAV 1`, do programu sa vloží riadok `x=15;` a `y =4;` a výsledok bude `x=0; y=4;`

b. ak `STAV` získame definíciou `#undefine STAV`, do programu sa vloží riadok `x=23;` a `y =2;` a výsledok bude `x=0; y=2;`

c. ak `STAV` získame definíciou `#undef STAV`, do programu sa vloží riadok `x=23;` a `y=2;` a výsledok bude `x=0; y=2;`

d. výsledok závisí od premennej `STAV` a od toho, či je kladná alebo záporná.

Kontrolné otázky

1. Charakterizujte rozdiel medzi funkciou a makrom.
2. Vymenujte úlohy preprocesora.
3. Akým spôsobom sa používajú makrá s parametrami? Ako sa makro rozdelí na konci riadku?
4. Vymenujte niekoľko preddefinovaných makier a uveďte príklady ich použitia.
5. Aký je rozdiel medzi `putchar()` a `fputc()`? Aký je rozdiel medzi `getchar()` a `fgetc()`?
6. Aký je princíp podmieneného prekladu? Kedy ho využívame?
7. Čím sa líši preklad riadený definíciou makra a definíciou konštanty?
8. Akým spôsobom môžeme použiť preddefinované matematické konštanty?
9. Kde bývajú uložené štandardné hlavičkové súbory?
10. Ako zistíme, či sa makro správne rozvinulo? Ako získame text programu spracovaného preprocesorom?

Cvičenia

1. V príklade 4 zvoľte väčšie polia, ich hodnoty môžete nastaviť pomocou generátora náhodných čísel. Odmerajte čas.
2. V príklade 6 zvoľte pri prevode číslice na číslo vhodné makro.
3. Napíšte makro, ktorého vstupom bude dĺžka odvesny rovnostranného pravouhlého trojuholníka a výstupom dĺžka prepony. Použite konštantu pre odmocninu 2, dajte si vypísať rozvoj makra.
4. Napíšte program, ktorý načíta riadok z klávesnice a vypíše, koľko v ňom bolo číslic, malých a veľkých písmen.
5. V rovine sú dané dva body A a B dvojicou svojich súradníc. Napíšte, aký uhol zviera priamka, ktorá je nimi určená s kladnou časťou osi `x`. Nájdite vhodnú matematickú funkciu a použite ju.
6. Zadefinujte makro `swap(x,y)`, ktoré navzájom vymení dva argumenty typu `int`.
7. Napíšte makro `je_z_int(x,a,b)` na testovanie, či je číslo `x` z intervalu (a,b). Napíšte príklad použitia tohto makra.
8. Napíšte program, ktorého vstupom bude celočíselné pole dĺžky 10 zadané z klávesnice a výstupom rozdiel maxima a minima. Napíšte program, ktorý bude riadený konštantou POMOC tak, že keď táto bude zadefinovaná, bude sa vypisovať aj hodnota minima a maxima a keď nebude, bude sa vypisovať iba samotný rozdiel.

Zadania

1. Majme súbor `defs.h` uložený v tom istom adresári ako súbor s nasledujúcim programom. Zistite, čo daný program robí.

Súbor `defs.h` obsahuje tieto inštrukcie:

```
#define vyp
#define MOD %
#define FOR(i,a,b) for(i=a; i<b;i++)
#define OD 10
#define DO 200
#define n 30
#define minim(a,b) {if (b<a) a=b;}
#define maxim(a,b) {if (b>a) a=b;}
```

Súbor s programom obsahuje inštrukcie:

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

main()
{int i;
 double k,q,MIN, MAX;
 double pole[n];
 MIN=MAX=pole[0];
 FOR(i,0,n)
    pole[i]= rand() MOD 20;
 FOR(i,0,n)
     { minim(MIN,pole[i])
       maxim(MAX,pole[i]) }

 k=(DO-OD)/(MAX-MIN);
 q=OD-k*MIN;

 #ifdef vyp
   printf("k %lf\n",k);
   printf("q %lf\n",q);
   FOR(i,0,n)
      printf("%lf\n",pole[i]);
 #endif

 FOR(i,0,n)
   pole[i]=(int)(k*pole[i]+q);

 #ifdef vyp
   printf("MIN %lf\n",MIN);
   printf("MAX %lf\n",MAX);
   FOR(i,0,n)
     printf("%lf\n",pole[i]);
 #endif
   system("pause");
}
```

2. Zadefinujte 2 celočíselné polia `u1` a `u2` rôznej dĺžky. Z klávesnice do nich načítajte čísla. Napíšte makro, ktoré bude mať 2 argumenty: prvý bude predstavovať možný index do prvého poľa, druhý možný index do druhého poľa a úlohou makra bude zabezpečiť výmenu prvkov s danými indexmi. Porozmýšľajte, či by nebolo vhodné doplniť do makra ešte jeden argument. Napíšte program, ktorý vygeneruje dvojicu celých náhodných čísel v takom intervale, že predstavujú možný index `i1` do prvého a `i2` do druhého poľa. V prípade, že `u1[i1]` je menšie (násobkom, deliteľom, druhou mocninou) ako `u2[i2]`, vymeňte ich a dajte vypísať číslo predstavujúce počet výmen. Snažte sa použiť čo najviac makier.

12 Zhrnutie

Hlavné podtémy kapitoly:

- Práca s dlhými programami. Oddelený preklad súborov.
- Zadanie vzorovej úlohy: program pre filtrovanie obrázkov.
- Opísanie potrebných funkcií a ich rozdelenie do skupín.
- Definovanie globálnych premenných a vytvorenie potrebných makier, vytvorenie hlavičkového súboru.
- Stanovenie stratégie pre zdieľanie obrázkových dát a ďalších potrebných údajov.

12.1 Oddelený preklad súborov

Oddelený preklad súborov využijeme pri práci s dlhými programami, prípadne keď viacero ľudí robí spolu. Oddelený preklad súborov by nemal byť zamieňaný s vkladaním súborov. Ak je program rozdelený do viacerých súborov, ktoré sa pomocou `#include` vložia do jedného súboru, vznikne po preklade iba jeden `.OBJ` súbor. *Oddelený* preklad súborov však znamená, že sa každý súbor preloží zvlášť - vznikne teda niekoľko `.OBJ` súborov, a tie sa spoja do jedného programu až pomocou zostavovacieho programu (linkeru). Na prvý pohľad to vyzerá ako zbytočne zložitejšie, ale v praxi je to jediný spôsob, ako rozumne zvládnuť prácu s veľkými programami. Tento spôsob prekladu podporujú aj moderné kompilátory. Čím je totiž väčší prekladaný súbor, tým viac narastajú kompilátoru požiadavky na rôzne administratívne informácie, napr. tabuľky identifikátorov, atď. To môže viesť až ku stavu, kedy sa pri preklade obsadí všetka operačná pamäť a preklad sa v horšom prípade neúspešné ukončí a v lepšom prípade extrémne spomalí. Spomalenie je spôsobené tzv. *swapovaním*, t. j. stavom, kedy je procesor nútený odkladať momentálne nepotrebné úseky operačnej pamäte na disk.

Pri oddelenom preklade sa každý súbor samostatne preloží do `.OBJ` súboru. Výhodou potom je, že sa pri opakovanom preklade (najčastejšie pri ladení) prekladá iba ten súbor, ktorý bol skutočne menený. Tento spôsob prekladu sa odporúča používať, pretože nás automaticky núti rozdeliť problém do viacerých menších časti. To síce spočiatku zaberie určité množstvo času, ale pri ladení veľkého programu sa to vyplatí.

Prvé, čo je nutné urobiť pri oddelenom preklade súborov, je zamyslieť sa nad rozdelením problému na viac, na sebe čo možno najmenej závislých častí. Niekedy takéto rozdelenie vyplýva priamo z podstaty problému, niekedy je potrebné sa nad ním zamyslieť, pretože pri nevhodnom rozdelení programu do viacerých súborov si pravdepodobne viac problémov prirobíme, než ušetríme. **Druhým krokom** je, že po rozdelení je nutné podrobne stanoviť *interfejs* (rozhranie), t.j. spôsoby komunikácie oddelených častí navzájom. Obyčajne sa za vhodnejší spôsob komunikácie považuje komunikácia pomocou volaní funkcií druhého modulu než pomocou spoločných globálnych premenných, viditeľných vo všetkých moduloch.

12.2 Opísanie problému

Majme úlohu z príkladu 3 z kapitoly 10. Mierne ju upravíme. Záverečnú slučku na spracovanie menu upravíme, výber z ponuky upravíme podľa kapitoly 11 a samotnú ponuku rozšírime o tri ďalšie obrazové funkcie na zobrazenie hrán, zašumenie obrázku šumom typu *salt and pepper* a filtráciu obrázku pomocou mediánu. V úlohe načítame čiernobiely obrázok a napíšeme preň sedem procedúr, ktoré budú vykonávať nasledovné činnosti:

- robiť z obrázku negatív (funkcia `negat`),
- prahovať (obrázok zmenia na čiernobiely, kde intenzity menšie ako prahová hodnota budú nahradené nulou a väčšie ako prahová hodnota budú nahradené hodnotou 255, funkcia `prah`),
- zväčšovať kontrast (funkcia `kontrast`),
- zobrazovať hrany (funkcia `hrany`),
- zašumia obrázku šumom typu *salt and pepper* (funkcia `salt_pepper`),
- odšumia obrázok pomocou mediánového filtra (funkcia `median`),
- a ukončia program (funkcia `koniec`).

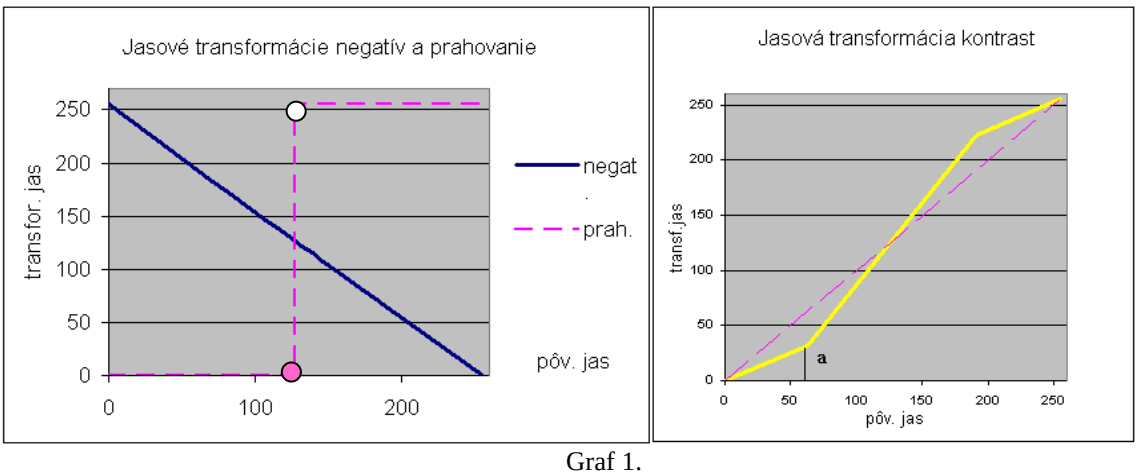
Program vypíše ponuku a podľa voľby sa spustí príslušná funkcia. Po vykonaní obrazovej transformácie v závislosti na zvolenej funkcii sa obrázok zobrazí, v tomto programe v okne aplikácie `xv`. Po uzavretí okna bude program pokračovať opätovným vypísaním ponuky.

```
negativ    - 1
prahovanie- 2
kontrast   - 3
hrany      - 4
zasumenie  - 5
median     - 6
reset      - 7
koniec     - 8
Zadaj volbu:_
```

Najprv si jednotlivé funkcie vizuálne opíšme a napíšme im prislúchajúci kód.

12.2.1 Funkcie `negativ`, `prahovanie` (thresholding) a `kontrast`

Tieto funkcie patria medzi typické transformácie jasovej stupnice (svetlosti, tmavosti obrázku). Môžeme ich opísať nasledujúcimi grafmi:

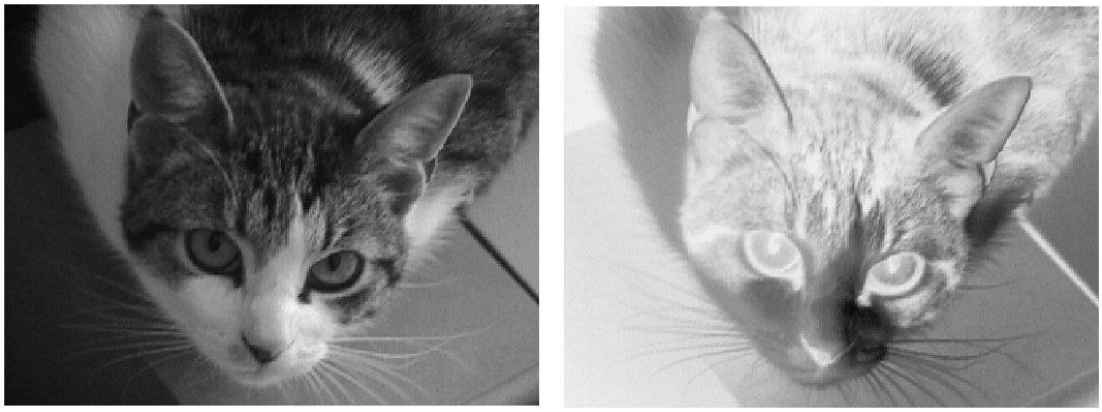


Graf 1.

Jasová transformácia $t_1(x)$ pre negatív je (pozri Obr.1 a Graf 1):

$$t_1(x)=255-x;$$ kde x je intenzita (0 -čierna, 255 biela)

Funkcia `prah` využíva jasovú transformáciu $t_2(x)$ (pozri Obr.2 a Graf 1): je doplnená o výpočet mediánovej hodnoty a umožnenie prahovania podľa mediánu.



Obr. 1
Vľavo: mačka Čima, pôvodný obrázok. Vpravo: negatív.

Jasová transformácia $t_2(x)$ pre prahovanie je:

$$t_2(x)=\begin{cases} 0 & pre\ x < prah \\ 255 & pre\ x \geq prah \end{cases}$$

Obr. 2
Vľavo: prah=120. Vpravo: prah=89 (mediánová hodnota).

Zmenu kontrastu budeme riadiť lomenou čiarou, popísanou na Grafe 1 vpravo. Čiara sa láme v $x=64$ a $x=192$. Z grafu by mal byť zrejmý význam hodnoty a, aj to, že pre $a<64$ sa kontrast zvyšuje, pre $a>64$ sa kontrast znižuje a pre $a=64$ dostávame identickú funkciu, čiže obrázok sa nezmení. (Je to jen jedna z mnohých možností)
Jasovú transformáciu $t_3(x)$ (pozri Obr.3 a Graf 1 vpravo) pre zvyšovanie a znižovanie kontrastu môžu predstavovať napríklad nasledujúce vzťahy pre lomenú čiaru:

$$t_3(x)=\begin{cases} \frac{a}{64}x & pre\ 0 \leq x < 64 \\ \frac{128-a}{64}(x-64)+a & pre\ 64 \leq x < 192 \\ \frac{a}{64}(x-255)+255 & pre\ 192 \leq x < 256 \end{cases}$$

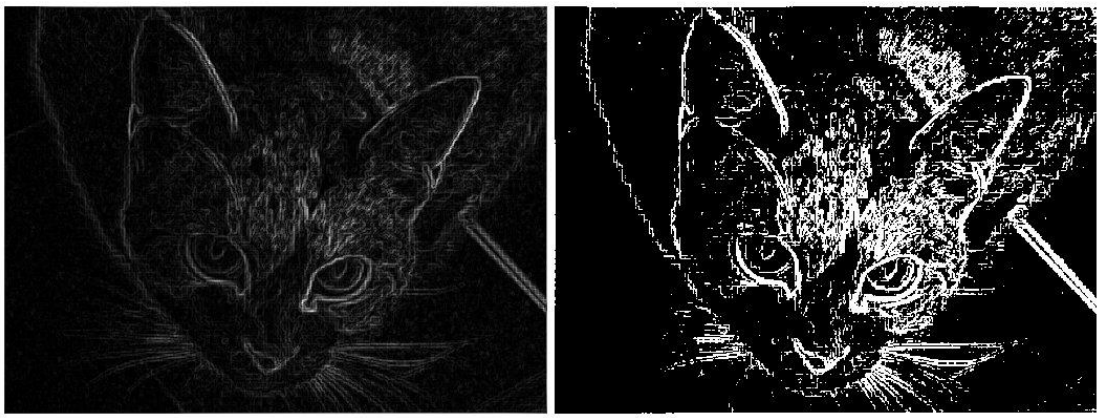
Obr. 3
Zmena kontrastu. Vľavo: a=32, zvýšenie kontrastu. Vpravo: a=96, zníženie kontrastu.

12.2.2 Funkcia `hrany`

V tejto funkcii (Obr.4) budeme v každom bode počítať normu gradientu. Gradient, predstavujúci pre spojitú dvojrozmernú funkciu vektor derivácií v smere `x` a v smere `y`, budeme aproximovať výrazmi *ux* a *uy*, opísanými ďalej, ktoré budú počítané v každom bode. Majme dvojrozmerné pole `u` a nastavme sa do jeho prvku s indexom `i` a `j`. Potom pre každý bod, okrem okrajových, ktoré pre zjednodušenie necháme nezmenené, budeme počítať zmenu v smere `x`, zmenu v smere `y` a normu vektora *(ux,uy)* takto:

```
ux = (u[i][j+1] - u[i][j-1])/2.0;
uy = (u[i+1][j] - u[i-1][j])/2.0;
u[i][j]=sqrt(ux*ux + uy*uy);
```

Tam, kde bude zmena v intenzite (jase) veľká, čo býva predovšetkým na hranách, dostaneme vyššie, teda bledšie hodnoty, tam, kde je zmena malá, dostaneme nižšie hodnoty. Výsledkom budú obrysy objektu na obrázku. Podľa toho, aké sú hrany kontrastné, budú obrysy tmavšie alebo bledšie. Aj keď budú obrysy bledšie, stále môžu byť zle viditeľné, preto škálu intenzít môžeme roztiahnuť medzi hodnoty 0 a 255 . V tomto algoritme len vypočítame maximálnu intenzitu v poli noriem gradientov a každú hodnotu prenásobíme číslom *255/max*. Potom môžeme pre zvýraznenie hrán použiť prahovanie.



Obr. 4
Vľavo: vypočítanie obrysov pomocou funkcie `hrany`.
Vpravo: použite prahovanie na obrázok vľavo. Hodnota prahu bola nastavená na 30.

12.2.3 Funkcia `salt_pepper`

Pomocou tejto funkcie (Obr.5 vľavo) obrázok „pokazíme“ tak, že na 20% jeho pozícií vygenerujeme náhodnú hodnotu v rozmedzí 0 až 255.

12.2.3 Funkcia `median`

Túto funkciu (Obr.5 vpravo) sme spomínali ako príklad algoritmu v prvej kapitole.
Majme obrázok, ktorý sa zašumil šumom *salt and pepper*, napr. 20%-ným. Na „opravenie“ obrázku použijeme algoritmus, založený na vypočítaní mediánu, t.j. prostrednej hodnoty v usporiadanom súbore. **Medián $\mathbf{M_e}$**, t.j. stredná alebo prostredná hodnota v poradí (v zmysle usporiadania) je počítaná podľa vzorca

$$\mathbf{M_e} = \begin{cases} x_{k+1} & , \quad ak \quad n = 2k+1 \\ \dfrac{x_k + x_{k+1}}{2} & , \quad ak \quad n = 2k \end{cases}$$

Jeden spôsob, ako ľahko nájsť medián, je použiť funkciu pre usporiadanie `qsort`. Pre každý bod obrázku – pixel – nájdeme jeho priamych susedov, t.j. pixle, ktoré ho obklopujú. Pre týchto 8 susedov a pixel samotný nájdeme medián ako piatu hodnotu v usporiadaní. Potom pôvodnú hodnotu obrázku nahradíme mediánom. Ak je v „hladkom“ obrázku, t.j. obrázku so spojitými prechodmi nejaká hodnota „uletená“, t.j. ide o šum, dostane sa na začiatok alebo koniec usporiadania. Táto hodnota určite nebude mediánom, a preto bude nahradená hodnotou bližšou okolitým pixlom.

Obr. 5
Vľavo: mačka Čima zašumená 20%-ným šumom *salt and pepper*.
Vpravo: Obrázok je odšumený použitím mediánového filtra.

12.3 Rozdelenie programu do viacerých súborov

Najprv si musíme rozmyslieť, ako budeme pracovať s obrázkom a ako rozdelíme jednotlivé funkcie do súborov. Tento krát si zadefinujeme obrázok ako statické pole typu `unsigned char`. Rozmer poľa budú udávať konštanty `v_max` a `s_max`. Skutočný rozmer načítaného obrázku bude uložený v globálnych premenných `s` a `v`.
Pole nebude globálne, budeme ho odovzdávať jednotlivým funkciám ako parameter. Pretože názov typu je dlhý, zadefinujeme si nový typ `pole`

```
typedef unsigned char pole [v_max][s_max];
```

pomocou ktorého budeme potom definovať obrázok, jeho prípadné kópie a odovzdávať parametre funkciám.

Hlavný program spolu s funkciou, ktorá program ukončí, bude uložený v súbore `main.c`. Funkcie rozdelíme do troch skupín:

1. vstupno-výstupné funkcie `zapis` a `nacit`, zabezpečujúce načítanie obrázku zo súboru a jeho zápis a funkcia `reset` budú v súbore `vv.c`,
2. funkcie `prah`, `negat` a `kontrast` budú v súbore `fun1.c`,
3. funkcie `salt_pepper`, `hrany` a `medián` budú v súbore `fun2.c`. Do tohto súboru umiestnime aj funkciu `sort_cisla` pre triedenie, ktorú však bude využívať aj funkcia `prah` zo súboru `fun1`.

Definíciu konštánt, typu `pole` a pre precvičenie aj makra `FOR` napíšeme do vlastného hlavičkového súboru `defin.h`, ktorý bude uložený v aktuálnom adresári. Pripomeňme, že pod **definíciou** premennej sa myslí príkaz, ktorý zavedie premennú daného typu. Stanoví jej typ a identifikátor. **Deklarácia** je príkaz, ktorý iba udáva typ premennej a jej meno a nepridelíuje žiadnu pamäť. Deklarácia funkcie iba hovorí, že existuje funkcia s uvedeným menom, danými parametrami a návratovou hodnotou. Vlastné telo funkcie v momente deklarácie ešte nemusí byť známe.

Napokon uvádzame jednotlivé súbory. Niektoré procedúry sú zjednodušené kvôli prehľadnosti. K pochopeniu kódu by mali stačiť predchádzajúce kapitoly. Podotýkame, že rozdelenie do súborov, rozdelenie premenných na globálne a lokálne, by samozrejme mohlo byť urobené aj iným spôsobom.

12.3.1 Hlavičkový súbor `defin.h`

V hlavičkovom súbore sú definované konštanty pre rozmer poľa, pre precvičenie je v ňom zadefinované makro pre pohodlnejšie písanie cyklu `for` a makro, ktoré dá zobraziť obrázok pomocou vhodnej aplikácie, tu `xv`. Pokiaľ zmeníme aplikácie, stačí zmenu urobiť v makre hlavičkového súboru. Všimnite si, ako sa rozdelí definícia makra do viacerých riadkov. Za spätným lomítkom musí nasledovať ENTER.

```
#define s_max  2000
#define v_max  2000

#define FOR(i,dh,hh) for(i=dh;i<hh;i++)
#define zobraz(meno_s) strcpy(pom2,"xv ");\
                       system(strcat(pom2,meno_s))
typedef unsigned char pole [v_max][s_max];
```

12.3.2 Súbor `main.c`

Z funkcií, ktoré sú volané pomocou menu, sú s výnimkou funkcie `koniec` všetky ostatné funkcie definované v iných súboroch. V tomto súbore ich musíme deklarovať.

```
/*hlavny subor pre pracu s obrazkami
  autor 007
  datum vytvorenia: 20.decembra 2005*/

#include <stdio.h>    /*znakove pole pre vytvorenie system. reťazca*/
#include "defin.h"

extern char pom2[];
extern void prah(int *);
extern void negat(int *);
```

```
extern void kontrast(int *);
extern void hrany(int *);
extern void salt_pepper(int *);
extern void median(int *);
extern void reset(int *);

char* meno_r,*meno_w;
/*_____________________________________*/
void koniec(pole pict)
{
    exit(1);
}
/*_____________________________________*/
int  main(int argc, char* argv[]){

char c;
int  i,j;
pole pic;
typedef void (* P_F)();
P_F menu[]={negat,prah,kontrast,hrany,
            salt_pepper,median,reset,koniec};

 if(argc != 3)
    {printf("wrong number of arguments");exit(1);}

meno_r=argv[1];     /*zapamatanie mena vstupneho suboru */
meno_w=argv[2];     /*zapamatanie mena vystupneho suboru */

nacit(pic);         /*nacitanie obrazkovych dat*/
zobraz(meno_r);     /*zobrazenie nacitanych dat*/

SLUCKA:
printf("negativ   - 1\n");
printf("prahovanie- 2\n");
printf("kontrast  - 3\n");
printf("hrany     - 4\n");
printf("zasumenie - 5\n");
printf("median    - 6\n");
printf("reset     - 7\n");
printf("koniec    - 8\n");

printf("Zadaj volbu:\n");
c=getchar();                    /*nacitanie znaku z klavesnice*/

while (getchar()!='\n');        /*vyprazdnenie vstupneho buffra*/
    if (isdigit(c)) {           /* je c cislica?*/
        i=c-'0'-1;              /*prevod cislice na cislo-1*/
        if((i>=0)&&(i<9)) {(*menu[i])(pic); goto SLUCKA;}
       else {printf("nespravna cislica\n");goto SLUCKA;}}
    else {printf("nespravny znak\n");goto SLUCKA;}
}
```

12.3.3 Súbor `vv.c`

V procedúrach `nacit`, `zapis` a `reset` budeme otvárať obrázkový súbor na čítanie a na zápis. Pri otváraní súboru budú tieto procedúry potrebovať vedieť meno otváraného súboru. Toto meno bolo získané v hlavnom programe a my teraz potrebujeme rozšíriť platnosť premennej, ktorá naň ukazuje aj do tohto, prípadne ostatných súborov. Premenné `meno_r` a `meno_w` definujeme v súbore `main.c` ako globálne a tam ich aj nastavíme. V ostatných súboroch, kde sú potrebné, ich potom deklarujeme ako `extern`.

```
/*subor pre vstupno vystupne operacie a resetovanie
  autor 007
  datum vytvorenia 20.decemra 2005*/

#include <stdio.h>
#include "defin.h"

/*externe premenne*/
extern char *meno_w,*meno_r;

/*definicia globalnych premennych*/
char pom1[5];
char pom2[80]="";
char pom3[80]="";
int a1,a2,a3,s,v;

/*definicia procedur*/

/*_______procedura pre nacitanie obrazku zo suboru_________*/
void nacit(pole pict)
{
 int i,j;
 FILE *vstup;

 /*otvorenie vstupneho obrazku na citanie*/
 vstup=fopen(meno_r,"r");
 if (vstup==NULL) {perror("chyba vstupu");exit(1);}

 /*nacitanie zahlavia*/
 fgets(pom1,5,vstup);
 do{
    strcpy(pom3,pom2);
    fgets(pom2,100, vstup);}
 while(pom2[0]=='#');
 sscanf(pom2,"%d %d",&a1,&a2);
 fscanf(vstup,"%d",&a3);
 s=a1;
 v=a2;

 /*nacitanie samotnych dat v podobe textoveho suboru*/
 for (i=0;i<v;i++)
    for (j=0;j<s;j++)
fscanf(vstup,"%d",&pict[i][j]);
fclose(vstup);
}
/*__________procedura pre zapis do suboru_____________*/
```

```
void zapis(pole pict)
{
int i,j;
FILE *vystup;

 /*otvorenie vystupneho suboru na zapis*/
 vystup=fopen(meno_w,"w");
    if (vystup==NULL) {perror("chyba");exit(1);}

/*zapisanie zahlavia*/
fprintf(vystup,"%s",pom1);
if (!strlen(pom3)) fprintf(vystup,"%s",pom3);
fprintf(vystup,"%d %d\n%d\n",a1,a2,a3);

/*zapisanie dat ako textoveho suboru*/
 for (i=0;i<v;i++)
   { for (j=0;j<s;j++)
        fprintf(vystup,"%d ",pict[i][j]);
     fprintf(vystup,"\n");
   }
fclose(vystup);
}


/*_________znovunacitanie  dat a ich zobrazenie____*/
void reset(pole pict)
{
nacit(pict);
zobraz(meno_r);
}
```

12.3.4 Súbor `fun1.c`

Funkcie definované v tomto a nasledujúcom súbore potrebujú skutočné rozmery obrázku definované v súbore `vv.c` ako globálne. Tu ich budeme deklarovať ako externé. Súčasne použijeme globálnu premennú `pom2[]`, definovanú v súbore `vv.c`, na vytvorenie príkazového reťazca na zobrazenie obrázku. Premenná `pom2` je po použití v zapisovacej a čítacej procedúre „voľná“, lebo jej hodnota sa nikde nevyužíva. Tu si treba uvedomiť, že nemôžeme robiť medián z celého poľa, lebo skutočný obrázok zaberá z neho iba časť. Preto vytvoríme pomocné (jednorozmerné) pole, ktoré zaberie iba toľko, čo skutočný obrázok a to potom dáme triediť. Z utriedeného poľa vyberieme prostredný prvok (opäť výpočet zjednodušíme, v skutočnosti to bude prostredný prvok, alebo vedľajší). Pozri aj kapitolu 1, filtrovanie mediánom.

```
/*subor pre 1.skupinu obrazkovych operacii: negativ, prah a kontrast
  autor 007
  datum vytvorenia 20.decemra 2005*/

#include "defin.h"

/*externe premenne*/
extern char* meno_w;
extern char pom2[];
extern int s,v;
extern int sort_cisla(const void*,const void* );
```

```
/*_____________procedura pre prahovanie_____________*/
void prah(pole pict)
{
 int prah;

 int i,j,n,l;
 unsigned char  *p;
 double x;

 n=v*s;
 p=(unsigned char *)malloc(n*sizeof(int));
 printf("Zadaj prah, -1 je pre median:");
 scanf("%d",&prah);
 while (getchar()!='\n');   /*vyprazdnenie vstupneho buffra*/

 /* zistenie medianu*/
 if (prah==-1)
   {  l=0;
     for (i=0;i<v;i++)
       for (j=0;j<s;j++)
          {p[l]=pict[i][j];l++;}

     qsort((void*)p,n,sizeof(unsigned char),sort_cisla);
     prah=p[n/2];
     free((void *)p);
     printf("prahova hodnota je %d\n",prah);
   }

/* samotne prahovanie*/
for (i=0;i<v;i++)
    for (j=0;j<s;j++)
        {x=pict[i][j];
         if (x<prah) pict[i][j]=0;
         else  pict[i][j]=255;
          }

zapis(pict);
zobraz(meno_w);
}
/*_____procedura pre negativ_____*/
void negat(pole pict)
{
 int i,j;
 printf("negat\n");
 for (i=0;i<v;i++)
    for (j=0;j<s;j++)
        pict[i][j]=255-pict[i][j];
zapis(pict);
zobraz(meno_w);
}

/*_____procedura pre kontrast_____*/
void kontrast(pole pict)
{
 int i,j;
```

```
 double a,x;
 printf("Zadaj a:");
 scanf("%lf",&a);
 while (getchar()!='\n');    /*vyprazdnenie vstupneho buffra*/
 for (i=0;i<v;i++)
    for (j=0;j<s;j++)
        {x=pict[i][j];
         if (x<64) pict[i][j]=x*a/64;
         else if (x<192)   pict[i][j]=(128-a)/64.0*(x-64)+a;
              else pict[i][j]=(x-256)*a/64.0+256;
      }

zapis(pict);
zobraz(meno_w);
}
```

12.3.5 Súbor `fun2.c`

```
/*subor pre 2.skupinu obrazkovych operacii: hrany, zasumenie a
medianovy filter
autor 007
datum vytvorenia 20.decemra 2005*/

#include "defin.h"
extern char* meno_w;
extern char pom2[];
extern int s,v;

 /*              procedura  vymen pre qsort                 */
 int sort_cisla(const void* prve,const void* druhe)
 {
 int cislo1,cislo2;
 cislo1=*((unsigned char *)(prve));
 cislo2=*((unsigned char *)(druhe));

 if (cislo1<cislo2) return (-1);
 if (cislo1>cislo2) return (1);
 return 0;
 }

/*                  procedura pre zasumenie                 */
void salt_pepper(pole pict)
{
 int i,j,k,x;
   x=(v*s)*0.2;
   for (k=0;k<x;k++)
             { j=rand()%(s_max);
               i=rand()%(v_max) ;
               pict[i][j]=rand()%256;
              }
zapis(pict);
zobraz(meno_w);
}
```

```
/*_____procedura pre hrany pomocou normy gradientu_________*/
void hrany(pole pict)
{
 int i,j;
 pole u;    /*pomocne pole*/
 double max=0,pomer,ux,uy;
 for (i=1;i<v-1;i++)
    for (j=1;j<s-1;j++)
      {ux=(pict[i][j+1]-pict[i][j-1])/2.0;
       uy=(pict[i+1][j]-pict[i-1][j])/2.0;/*vypocet gradientu
                                            (ux,uy)*/
       u[i][j]=sqrt(ux*ux+uy*uy);         /* vypocet normy
                                            gradientu*/
       if (u[i][j]>max) max=u[i][j];       /*hladanie maximalnej
                                          normy grad.*/
      }
pomer=255/max;                            /*cislo, ktorym sa obrazok
                                           preskaluje*/
printf("%lf %lf\n",max,pomer);
for (i=1;i<v-1;i++)
    for (j=1;j<s-1;j++)
         pict[i][j]=(int)(u[i][j]*pomer);;
zapis(pict);
zobraz(meno_w);
}
/*_____________porcedura pre medianovy filter___________*/
void median(pole pict)
{
 int i,j,ii,jj,l;
 unsigned char  okolie[9];
 pole u;  /*pomocne pole*/

 double max=0,pomer,ux,uy;
 for (i=1;i<v-1;i++)
    for (j=1;j<s-1;j++)
      {l=0;
        for (ii=i-1;ii<=i+1;ii++)
           /*stanovenie okolitych susedov*/
           for (jj=j-1;jj<=j+1;jj++)
                {okolie[l]=pict[ii][jj];
                 l++;}
       /*utriedenia a vyber prostredneho prvku*/
       qsort((void*)okolie,9,sizeof(unsigned char),sort_cisla);
       u[i][j]=okolie[5];
            }
  for (i=1;i<v-1;i++)
      for (j=1;j<s-1;j++)
      pict[i][j]=u[i][j];
zapis(pict);
zobraz(meno_w);
}
```

12. 4 Spôsob prekladu

Najprv preložíme súbory, v ktorých sú funkcie, do objektových súborov príkazmi:

```
gcc -c vv.c
```

```
gcc -c fun1.c
gcc -c fun2.c
```

Nakoniec tieto objektové súbory prilinkujeme k súboru `main.c` príkazom:

```
gcc fun1.o vv.o main.c -o program -lm
```

Program sa spúšťa príkazom:

```
program meno_obr vystup
```

`meno_obr` musí byť existujúci súbor , predstavujúci obrázok v textovom *pgm* formáte, meno výstupného súboru môže byť ľubovoľné.

12.5 Statické globálne premenné a funkcie

Keby sme spolupracovali na programovaní väčšieho projektu s kolegami, mohlo by sa stať, že by dochádzalo ku kolízii názvov. Vtedy, vcelku často, používame pre globálne premenné v oddelene prekladanom programe pamäťovú triedu `static`. Statické globálne premenné majú tú výhodu, že sú viditeľné (dostupné) iba v súbore (module), v ktorom boli definované, ale nie v ostatných súboroch. (Nepomôže ani i keby boli v iných súboroch deklarované ako `extern`.) Rovnaký spôsob (pamäťová trieda `static`) sa používa pre označenie funkcií, napr.:

```
static int fun(float a)
```

Má to rovnaký význam, ako u statických globálnych premenných, t. j. `static` funkcia je dostupná iba vo vlastnom súbore a nie v ostatných súboroch. Označenie `static` ako globálnych premenných, tak i funkcií je veľmi výhodné pravé v prípade, keď na veľkom programe pracuje viac programátorov naraz. Tí sa potom nemusia obávať, že zvolili rovnaké mená pre identifikátory ako ich kolegovia. Na pomenovaní globálnych funkcií a premenných zaisťujúcich interfejs sa ľahko dohovorí, a všetky ostatné funkcie a globálne premenné označia ako `static`. Potom nemôže dôjsť ku kolízii mien, pretože každý "vidí" len tie svoje.

Poznámka:

- Často sa pre označenie lokálnych "funkcií" využíva makro:

  ```
  #define LOCAL    static
  ```

Príklad 1
Súbor A.C obsahuje:

```
/* zaciatok suboru A.C */
static int x;           /* definicia  */
extern double f;        /* deklaracia */

static double fun(double x) {
    return(x);
}

int funkcia(void) {
    return((int) fun(f) + x)
}
/* koniec suboru A.C */
```

Súbor B.C obsahuje:

```
/* zaciatok subor B.C */
#include <stdio.h>
```

```
extern int x;                    /* deklaracia       */
extern int funkcia(void);        /* funkcny prototyp */
double f;                        /* definicia        */

static void fun(void) {
    printf("%d\n", funkcia());
}

main() {
    x = 3;
    f = 3.5;
    fun();
}
/* koniec suboru B.C */
```

Preklad týchto súborov prebehne úspešne, ale pri zostavovaní bude linker hlásiť chybu, že nie je definovaná premenná x. Tá bola totiž definovaná ako `static` v A.C a deklarácia:

```
extern int x;
```

v B.C jej pamäťovú triedu nezmení. Po vynechaní `static` v definícii x v A.C bude program pracovať správne a nebude zmätený tým, že sú tam dve funkcie rovnako pomenovane `fun()`, ale rôznych typov. Funkcia `fun()` v subore A.C nemá nič spoločného s `fun()` z B.C a každá je viditeľná (dostupná, volateľná) iba zo súboru, v ktorom sú definované.

Dodatok

Základom jazyka C opisovanom v tejto učebnici bola americká národná norma ANSI C, ktorá bola prijatá v roku 1988. V r.1999 bola prijatá verzia označená ako ISO/IEC 9899:1999 a v roku 2001 boli k tejto verzii dodané opravy. Táto verzia by mala byť platná v súčasnej dobe, avšak platnosť normy je jedna vec a podpora výrobcov kompilátorov je druhá vec. V dodatku spomenieme iba najzákladnejšie zmeny a odvoláme sa na 23. kapitolu knihy, pre záujemcov o detailnejší výklad odkážeme na [7]. Podotýkajme, že P.Herout v [2] odporúča nepoužívať žiadnu z noviniek v prípade, že chceme programy prenášať medzi rôznymi platformami, pretože tieto novinky nemusia byť ešte všeobecne implementované. V opačnom prípade, ak je kompilátor s novinkami k dispozícii, nie je dôvod ich nevyužiť.

Medzi najdôležitejšie novinky patrí:

- bol zaradený jednoriadkový komentár inšpirovaný C++, napr.
  ```
  // tento riadok predstavuje komentár
  ```
- definície premenných a kód sa môžu ľubovoľne miešať, teda definície premenných nemusia byť pred príkazmi
- zrušilo sa pravidlo, že ak nie je návratový typ funkcie deklarovaný, je automaticky typu `integer`, vždy ho treba uviesť
- dĺžka **lokálneho** statického poľa môže byť premenná, teda nemusí byť známa v dobre prekladu
- funkcie môžu byť vnárané, teda definícia funkcie môže obsahovať definíciu inej funkcie.

ASCII tabuľka (American Standard Code for Information Interchange).

Kódy 0-31 sú riadiace kódy s často špecifickou interpretáciou.
Význam kódov 128 - 255 závisí na použitom type písma.

kód	význam znaku	kód	význam znaku
9	tabulátor	13	návrat vozíku - Carriage Return (CR)
10	posun o riadok - Line Feed (LF)	27	Escape
12	posun o stránku - Form Feed (FF)	32	medzera

kód	znak	kód	znak	kód	znak	kód	znak	kód	znak	kód	znak	kód	znak	kód	znak
32		60	<	88	X	116	t	144		172	¬	200	Č	228	ä
33	!	61	=	89	Y	117	u	145	‘	173		201	É	229	Í
34	"	62	>	90	Z	118	v	146	’	174	®	202	Ę	230	ć
35	#	63	?	91	[	119	w	147	“	175	Ż	203	Ë	231	ç
36	$	64	@	92	\	120	x	148	”	176	°	204	Ě	232	č
37	%	65	A	93	]	121	y	149	•	177	±	205	Í	233	é
38	&	66	B	94	^	122	z	150	–	178	˛	206	Î	234	ę
39	'	67	C	95	_	123	{	151	—	179	ł	207	Ď	235	ë
40	(	68	D	96	`	124	|	152		180	´	208	Đ	236	ě
41	)	69	E	97	a	125	}	153	™	181	µ	209	Ń	237	í
42	*	70	F	98	b	126	~	154	š	182	¶	210	Ň	238	î
43	+	71	G	99	c	127		155	›	183	·	211	Ó	239	ď
44	,	72	H	100	d	128	€	156	ś	184	¸	212	Ô	240	đ
45	-	73	I	101	e	129		157	ť	185	ą	213	Ő	241	ń
46	.	74	J	102	f	130	,	158	ž	186	ş	214	Ö	242	ň
47	/	75	K	103	g	131		159	ź	187	»	215	×	243	ó
48	0	76	L	104	h	132	„	160		188	Ľ	216	Ř	244	ô
49	1	77	M	105	i	133	…	161	ˇ	189	˝	217	Ů	245	ő
50	2	78	N	106	j	134	†	162	˘	190	ľ	218	Ú	246	ö
51	3	79	O	107	k	135	‡	163	Ł	191	ż	219	Ű	247	÷
52	4	80	P	108	l	136		164	¤	192	Ŕ	220	Ü	248	ř
53	5	81	Q	109	m	137	‰	165	Ą	193	Á	221	Ý	249	ů
54	6	82	R	110	n	138	Š	166	¦	194	Â	222	Ţ	250	ú
55	7	83	S	111	o	139	‹	167	§	195	Ă	223	ß	251	ű
56	8	84	T	112	p	140	Ś	168	¨	196	Ä	224	ŕ	252	ü
57	9	85	U	113	q	141	Ť	169	©	197	Ĺ	225	á	253	ý
58	:	86	V	114	r	142	Ž	170	Ş	198	Ć	226	â	254	ţ
59	;	87	W	115	s	143	Ź	171	«	199	Ç	227	ă	255	˙

Priorita operátorov

Operátor umiestnený vyššie v zozname je vyhodnotený skôr.

Priorita	Operá-tor	Význam operátora	Príklad	Asociatívnosť

1	()
	[]
	->>
	.
	++
	--
		zoskupovanie operátorov
		prístup do poľa
		prístup do štruktúry pomocou smerníka
		prístup k prvku štruktúry
		zvýšenie o 1 po vykonaní operácie
		zníženie o 1 po vykonaní operácie
			(a + b) / 4;
			array[4] = 2;
			ptr->vek = 34;
			obj.vek = 34;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
				zľava doprava

2	!
	~
	++
	--
	-
	+
	*
	&
	(typ)
	sizeof
		lLogická negácia
		bitový doplnok
		zvýšenie o 1 pred vykonaním operácie
		zníženie o 1 pred vykonaním operácie
		unárne mínus
		unárne plus
		dereferencia
		asdresa z
		pretypovanie na daný typ
		vráti veľkosť v bytoch
			if(!done) ...
			príznak = ~príznak;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			i = -1;
			i = +1;
			data = *ptr;
			addresa = &obj;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				sprava doľava

3	->* .*	prístup k členu prístup k členu	ptr->*var = 24; obj.*var = 24;	zľava doprava

4	*
	/
	%
		násobenie
		delenie
		modulo
			i = 2 * 4;
			f = 10 / 3;
			rem = 4 % 3;
				zľava doprava

5	+ -	sčítanie odčítanie	int i = 2 + 3; int i = 5 - 1;	zľava doprava
6	<< >>	bitový posun doľava bitový posun doprava	i = 33 << 1; i = 33 >> 1;	zľava doprava

7	<
	<=
	>
	>=
		porovnanie menší ako
		porovnanie menší alebo rovný ako
		porovnanieväčší ako
		porovnanie väčší alebo rovný ako
			if(i < 42) ...
			if(i <= 42) ...
			if(i > 42) ...
			if(i >= 42) ...
				zľava doprava

8	== !=	porovnanie „sa rovná“ porovnanie „nerovná sa“	if(i == 42) ... if(i != 42) ...	zľava doprava
9	&	logický súčin po bitoch	príznaky = príznaky & 42;	zľava doprava
10	^	neekvivalencia po bitoch	príznaky = príznaky ^ 42;	zľava doprava
11	|	logický súčet po bitoch (normal)	príznaky = príznaky | 42;	zľava doprava
12	&&	logický súčin (AND)	if(conditionA && conditionB) ...	zľava doprava
13	||	logický súčet (OR)	if(conditionA || conditionB) ...	zľava doprava
14	? :	ternárny operátor pre if-then-else	i = (a > b) ? a : b;	sprava doľava

15	=
	+=
	-=
	*=
		priradenie
		pričítanie a priradenie
		odčítanie a priradenie
		násobenie a priradenie
			a = b;
			a += 3;
			b -= 4;
			a *= 5;
				sprava doľava

	/=
	%=
	&=
	^=
	|=
	<<=
	>>=
		delenie a priradenie
		Modulo a priradenie
		logický bitový súčin a priradenie
		bitová neekvivalencia a priradenie
		logický bitový súčin a priradenie
		bitový posun doľava a priradenie
		bitový posun doprava a priradenie
			a /= 2;
			a %= 3;
			príznaky &= nové_ príznaky;
			príznaky ^= nové_príznaky;
			príznaky |= nové_ príznaky;
			príznaky <<= 2;
			príznaky >>= 2;	

16	,	sekvenčný vyhodnocovací operátor	for(i = 0, j = 0; i < 10; i++, j++) ...	zľava doprava

Výsledky testov

Správne odpovede testu z kapitoly 1:
1.B 2.A 3.C 4.D 5.B 6.B 7.A 8.C 9.C 10.C 11.B 12.D

Správne odpovede testu z kapitoly 2:
1.C, 2.D, 3.D ,4.C, 5.C, 6.B, 7.C ,8.C ,9.AB, 10.C, 11.C, 12.AB

Správne odpovede testu z kapitoly 3:
1.C, 2.BD, 3.A, 4.B, 5.B, 6.C, 7.D, 8.D, 9.BC, 10.C, 11.B, 12.B

Správne odpovede testu z kapitoly 4:
1.C, 2.CD, 3.BC, 4.B, 5.A, 6.B, 7.B, 8.D, 9.B, 10.A, 11.B,12.BC

Správne odpovede testu z kapitoly 5:
1.CD, 2.D, 3.D, 4.D, 5.ACD, 6.D,7.BC,8.B,9.C,10.AB,11.AB,12.AB

Správne odpovede testu testu z kapitoly 6:
1.ABD, 2.B, 3.A, 4.D, 5.ACD, 6.CD, 7.B, 8.CD, 9.D, 10.A, 11.D, 12.BC

Správne odpovede testu kapitoly 7:
1.BCD, 2.D, 3.CD ,4.D, 5.AB, 6.A, 7.B, 8.D, 9.D, 10.BD, 11.BD, 12.D

Správne odpovede testu kapitoly 8:
1.B, 2.AD, 3.D, 4.D, 5.B, 6.C, 7.D, 8.B, 9.B, 10.A, 11.BD, 12.AC

Správne odpovede testu kapitoly 9:
1.A, 2.B, 3.B, 4.C, 5.B, 6.BCD, 7.B, 8.C, 9.C, 10.C, 11.A., 12. D

Správne odpovede testu kapitoly 10:
1.A, 2.D, 3.B, 4.D, 5.A, 6.D ,7.D ,8.BD, 9.D, 10.D, 11.C, 12.B

Správne odpovede testu kapitoly 11:
1.B, 2.BD, 3.A, 4.C, 5.A, 6.AD, 7.B, 8.C, 9.AB, 10.BD, 11.A, 12 .AC

[1.] HEROUT, P.: *Učebnica jazyka C,* Nakladatelství KOPP, České Budějovice, 1997, ISBN 80-8528-21-9

[2.] HEROUT, P.: *Učebnica jazyka C, 2. díl,* Nakladatelství KOPP, České Budějovice, 2004, ISBN 80-7231-221-4

[3.] KERNIGHAN B., RITCHIE, D.: *Programovací jazyk C,* ALFA, Bratislava, 1989, ISBN 80-05-0154-1

[4.] KAČMÁŘ, D.: *Jazyk C, učebnice pro střední a vysoké školy,* Computer Press, Praha, 2001, ISBN 80-7226-295-5

[5.] KADLEC, V.: *Učíme se programovat v jazyce C,* Computer Press, Praha, 2002, ISBN 80-7226-715-9

[6.] WRÓBLEWSKI, P.: *Algoritmy, Datové struktury a programovací techniky,* Computer Press, Brno, 2004, ISBN 80-251-0343-9

[7.] VIRIUS, M.: *Cečko na přelomu tisíciletí,* Chip, 9/02,10/02,11/02

Odporúčané www-stránky, zaoberajúce sa C/C++ (apríl 2006):

www.cppreference.com
www.cplusplus.com
cplus.about.com
www.knihy.cpress.cz/K0701 (zdrojový kód k príkladom v [5.])
www.knihy.cpress.cz/K0871 (zdrojový kód k príkladom v [6.])
www.uk.tnuni.sk/top_linux/pre_linux/pocitace/c/c_in.php
www.didaktika.input.sk/04jazykc.html
pf.ku.sk/lajciak/index.php?id=jazykc/jazykc
www.fiit.stuba.sk/~mark/prog_c/priklady.html

OBSAH