Anino BELAN

KURZ JAZYKA C

učebný text pre kvartu a kvintu osemročného gymnázia

2. vydanie

BRATISLAVA
2003 – 2011

Obsah

Úvod

Najprv by sa patrilo povedať, čo táto knižka určite nie je. Nie je to referenčná príručka. Nebudú v nej popísané všetky črty jazyka C. Mnohé veci vám ostanú zatajené a nenájdete ich tu. Dozviete sa iba nevyhnutné minimum, aby ste mohli začať pracovať na vlastných projektoch a vytvárať vlastné programy. Dôvod pre to je jednoduchý – človek sa cudzí jazyk nenaučí tak, že by sa naučil naspamäť jeho gramatiku. Naučí sa ho tak, že sa bude pokúšať ten jazyk používať a postupne v jeho používaní získa zručnosť. Až potom môže oceniť jemné nuansy gramatiky. S jazykom C je to rovnaké. Ak nebudete programovať, nič vám nepomôže, že viete, ako vyzerá jeho gramatika. A keď programovať začnete a bude vás to baviť, tak si referenčnú príručku zoženiete.

Táto knižka nie je ani učebnicou programovania. Nedozviete sa v nej, ako písať programy tak, aby sa v nich dalo ľahko orientovať, ako navrhovať štruktúry dát ani iné veci, ktoré by mal skúsený programátor vedieť. Dôvod je podobný ako v predošlom prípade. Na to, aby ste začali programovať to nepotrebujete a isté dobré rady oceníte až potom, keď sa niekoľko hodín márne pokúšate predrať neprehľadnou húštinou kódu, ktorú ste vy sami vytvorili.

A čo teda táto knižka ponúka? Naučíte sa z nej základy jazyka C. Bude to dosť na to, aby ste mohli začať experimentovať a skúšať, čo zvládnete. Bude to dosť dobrý základ pre to, aby ste sa mohli učiť ďalšie veci (ďalší diel je venovaný grafike v Cčku). A bude to dosť na to, aby ste mohli začať rásť ako programátori. K tomu Vám želám veľa šťastia.

Anino

1. lekcia

Totálny začiatok alebo "Hello, world!"

Jazyk C vymysleli Brian W. Kernighan a Denis M. Ritchie – zamestnanci Bellových laboratórií v sedemdesiatych rokoch 20. storočia. Jazyk C bol navrhnutý a implementovaný pod operačným systémom UNIX a takmer celý UNIX bol v C-čku napísaný. (Vyzerá to ako paradox, ale skutočne je to tak.) Odvtedy C-čko prešlo určitým vývojom a existujú viaceré jazyky, ktoré sú od neho odvodené (napríklad C++, Java alebo PHP), ale stále patrí medzi obľúbené a používané jazyky. (V C-čku je napísaných mnoho hier a napr. aj LINUXové jadro.)

Jeho obľúbenosť má viacero dôvodov. Prvým z nich je jeho univerzálnosť – mnohé programy napísané v C-čku môžete preložiť pod takmer ľubovoľným operačným systémom na takmer ľubovoľnom type počítača a budú fungovať. Ďalším dôvodom je, že je to jazyk nízkej úrovne, to znamená, že sa v určitých svojich črtách podobá na strojový kód. To znamená, že síce neobsahuje priame prostriedky na niektoré zložité konštrukcie, ale programy naprogramované v C-čku sú veľmi rýchle (ak sa spravia šikovne). Ak viete jazyk C, budete sa ľahšie učiť jednak assembler (ktorý potrebujú ľudia, ktorí sa zaujímajú viac o hardvérovú stránku počítača), jednak vyššie programovacie jazyky (v ktorých sa nejaké veci dajú robiť pohodlnejšie, ale štruktúru jazyka do veľkej miery prebrali z C-čka). A v neposlednom rade – jazyk C je vymyslený pekne. Dajú sa v ňom písať prehľadné programy, s použitím jednoduchých prostriedkov sa dajú dosiahnuť silné výsledky a je s ním radosť pracovať.

Na rozdiel od BASICu alebo Comenius Loga jazyk C nie je interpretovaný. Nefunguje teda spôsobom, že v nejakom prostredí napíšete program a to prostredie je potom zodpovedné aj za jeho vykonanie. (Napríklad programy napísané v Imagine nemôžete spustiť bez toho, že by ste mali Imagine k dispozícii.) C-čko funguje inak. Program v C-čku sa napíše v ľubovoľnom editore (ľuďom, ktorí robia pod LINUXom odporúčam `kate` alebo `gedit`, tí, čo pracujú pod MS Windows použijú editor prostredia Dev-C++[1]) Keď naprogramujete všetko, čo potrebujete, súbor nahráte na disk a predhodíte ho kompilátoru. Kompilátor je program, ktorý z vášho zdrojového kódu vytvorí program, ktorý sa dá spustiť. Pre spúšťanie takto vytvoreného programu už nepotrebujete ani zdrojový kód ani kompilátor.

Úloha č.1: Vytvorte si priečinok `programy`. Vojdite do tohto priečinku a vytvorte tam súbor `hello.c` s nasledujúcim obsahom (dávajte pozor, aby ste žiaden znak nevynechali a nepridali a aby ste si to uložili, keď skončíte!):

```
#include <stdio.h>

main()
{
    printf("Hello, world!\n");
}
```

Keď ste úspešne zvládli prvú úlohu, môžeme si vysvetliť, čo ste to teda vlastne napísali.

1Dev-C++ si môžete stiahnuť z adresy http://downloads.zoznam.sk/download/dev-c-57

Kompilátor C-čka pozná pomerne málo funkcií. Napríklad aj funkcia `printf`, ktorá slúži na výpis na obrazovku je pre neho neznáma. Je ale nesmierne množstvo funkcií uložených v knižniciach, ktoré môže používať. Aby sme teda mohli funkciu `printf` použiť, musíme kompilátoru povedať, kde sa má dozvedieť, ako tá funkcia vlastne vyzerá. Popis funkcií, ktoré slúžia na vstup a výstup sa nachádza v súbore `stdio.h` (názov `stdio` vznikol z anglického „standard input-output“, nepleťte si ho teda so `studio`). Príkaz `#include <stdio.h>` teda znamená „načítaj súbor `stdio.h`“.

Program v jazyku C môže mať množstvo funkcií (o funkciach ešte budeme hovoriť), ale jednu funkciu vždy mať musí. Je to funkcia `main()`. Funkcia `main` je funkcia, kde program začína a keď táto funkcia dobehne, program sa skončí. Po názve funkcie nasleduje jej telo uzavreté do kučeravých zátvoriek `{` a `}`.

Náš program obsahuje jediný príkaz – príkaz `printf("Hello, world!\n");` Tento príkaz vypíše na obrazovku správu `Hello, world!`. (Znaky `\n` na konci znamenajú „prejdi na nový riadok“.) Všimnite si, že za príkazom je bodkočiarka. Tá sa píše za každý príkaz jazyka C.

Dobre. Program sme napísali. Teraz by sme ho chceli spustiť. Ako na to? Ako sme už spomínali, treba z neho s pomocou programu zvaného kompilátor vyrobiť niečo, čo sa dá spustiť. Najprv ukážeme, ako sa to robí pod Linuxom, pretože tam je dobre vidieť, čo sa vlastne deje. Pod MS Windows to funguje úplne rovnako, len je to ukryté za tlačidlami vývojového prostredia.

V priečinku, v ktorom máte súbor `hello.c` napíšte príkaz

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

Môžu sa stať dve veci. Buď vám to niečo vypíše, alebo to nevypíše nič. Ak to niečo vypíše, je zle. Váš program obsahuje chyby. Pozorne si prečítajte, čo vám to vlastne píše, chyby opravte a skúste to znovu. Ak vám to nič nevypíše, vyhrali ste. Kompilátor vaše programátorské dielo pochopil a preložil. V priečinku vznikol súbor `hello.o`. Tento súbor sa nazýva objektový súbor (object file) a je v ňom skompilovaný program.

Tento program je, žiaľ, zatiaľ bez knižničných funkcií. Takže napríklad v našom prípade vie, že má spustiť funkciu `printf`, ale zatiaľ netuší, čo tá funkcia robí. Knižničné funkcie sa k programu pripoja ďalším spustením kompilátora (táto fáza sa nazýva linkovanie):

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

Tento príkaz zoberie objektový súbor, pripojí k nemu potrebné funkcie z knižníc a výsledok nazve tak, ako mu poviete slovom, ktoré nasleduje hneď za parametrom `-o`.[2]

Teraz teda nastáva ten historický okamih, kedy môžete spustiť prvý program napísaný v C-čku.

Úloha č.2: (zatiaľ iba pre Linuxákov) Skompilujte váš program uvedeným spôsobom a spustite ho. Program sa spúšťa tak, že napíšete `./hello` (Linux z bezpečnostných dôvodov spúšťa iba programy z prednastavených priečinkov. Ak chcete spustiť príkaz z iného priečinku, treba pred neho uviesť cestu. A tá bodka znamená „priečinok, v ktorom sa práve nachádzate“.)

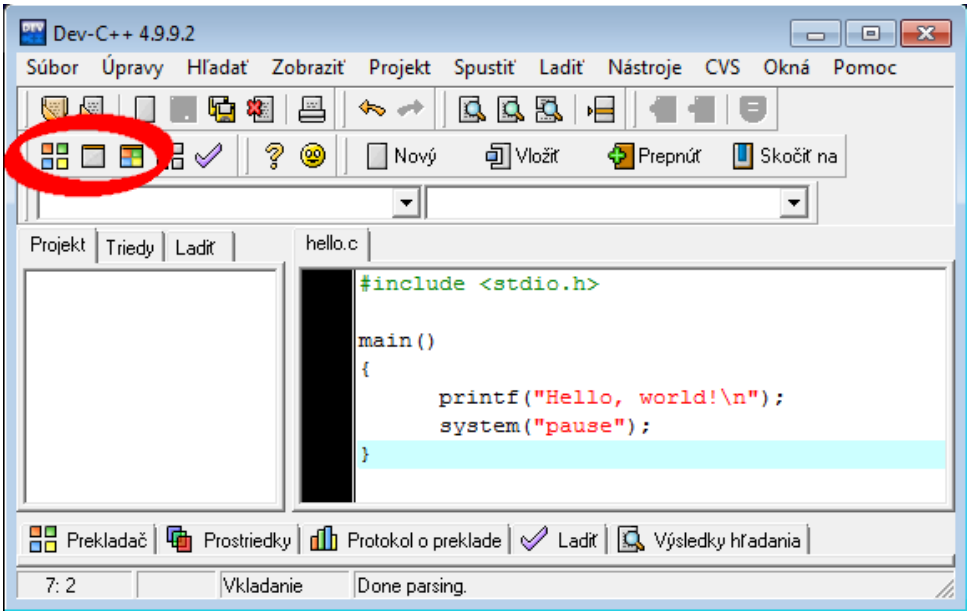
Skôr než sa začneme venovať používateľom MS Windows, ešte dve dobré rady pre Linuxákov (Windowsáci môžu ignorovať):

2 Ak by ste pracovali pod MS Windows, výsledok by sa patrilo nazvať `hello.exe`, aby bolo systému jasné, že je to spustiteľný súbor. Pod Linuxom sa to, či je súbor spustiteľný, rieši inak, takže sa programom prípony nedávajú.

Prvá. Uvedený dvojkrokový postup kompilácie človeku ušetrí veľa času, ak pracuje na veľkom projekte, v ktorom je tak zodvadsať zdrojových súborov. Keď sa zmení iba jeden súbor, netreba zakaždým kompilovať všetky, stačí ten zmenený. Potom sa to v druhom kroku všetko dohromady zlinkuje. Ak ale máte všetko iba v jednom súbore (čo pravdepodobne budete mať až do konca týchto skrípt), tak sa dá všetko skompilovať len na jeden krok. Keď teda chceme skompilovať súbor `hello.c`, použijeme príkaz „`gcc -o hello hello.c`“.

Druhá. Väčšina Linuxov má nainštalovaný program `make`, s pomocou ktorého sa dajú jednoducho kompilovať aj veľké projekty. V prípade veľkých projektov treba vytvoriť pre `make` konfiguračný súbor, ale jednoduchý program ako ten náš sa dá skompilovať aj bez neho. Stačí napísať „`make hello`“. (Pozor! Nepíšte tam „`make hello.c`“. Nebude to fungovať.) Samozrejme, ak sa váš program bude volať napr. `hack.c`, tak napíšete `make hack` a nie `make hello`.

Užívatelia systému MS Windows a Dev-C++ majú oproti Linuxákom jednu výhodu a jednu nevýhodu. Výhoda spočíva v tom, že nemusia písať nič do príkazového riadku v termináli a stačí im stlačiť tlačidlo. Z tých troch tlačidiel vyznačených na obrázku to vľavo spôsobí, že sa program skompiluje (automaticky sa udeje presne to isté, čo sme popisovali pri linuxe), prostredné tlačidlo spôsobí, že sa program spustí a tretie tlačidlo v poradí spôsobí, že sa program najprv skompiluje a potom spustí. Pohoda.



Nevýhoda spočíva v tom, že program nespúšťajú v príkazovom riadku. Náš program ale potrebuje niekam to „Hello, world!“ napísať. Preto Windows otvorí nové okno, vypíše doň čo má a okno zase zavrie. Vzhľadom k tomu, že počítač to zvládne rýchlo, zostane okno na obrazovke iba zlomok sekundy a nikto si nestihne prečítať, čo sa do neho vlastne vypísalo.

Situáciu vyriešite tak, že na koniec programu (tam, kde to vidíte na obrázku) pridáte funkciu

```
system("pause");
```

Funkcia `system` vie spustiť ktorýkoľvek z príkazov operačného systému a príkaz `pause` spôsobí, že sa nebude diať nič, kým nestlačíte klávesu na klávesnici. To znamená, že okno počká, kým si prečítate všetko, čo program vypísal a zavrie sa až potom, keď stlačíte klávesu.

Úloha č.2: (pre Windowsákov): Skompilujte váš program a spustite ho.

Všetky úlohy, ktoré nasledujú, sú nezávislé od operačného systému. To samozrejme neznamená, že všetky veci (napríklad v úlohe č. 3) musia všade dopadnúť rovnako.

Úloha č.3: Odstráňte zo súboru `hello.c` prvý riadok a skúste to skompilovať. Podarí sa kompilácia? Bude to fungovať? Zistite, čo sa stane, ak odstránite `()` za `main`. Bude to fungovať? Čo vám povie kompilátor? Ako sa zmení správanie programu, ak vynecháte `\n` ? Ako sa zmení správanie programu, ak vynecháte `}` na konci programu? Ako sa zmení správanie programu, ak vynecháte bodkočiarku? Ak kompilátor vypíše chybu, pokúste sa porozumieť chybovým hláškam a spätne z nich určiť (podľa popisu a čísla riadku), kde chyba nastala.

Úloha č.4: Urobte program `vizitka.c`, ktorý vypíše vašu vizitku ohraničenú hviezdičkami. Urobte program `srdiecko.c`, ktorý nakreslí srdiečko z hviezdičiek. Urobte program `riadky.c`, ktorý vypíše desať prázdnych riadkov. Urobte program `idea.c`, ktorý vypíše aspoň trojriadkovú životnú múdrosť.

2. lekcia

Čo sú to premenné alebo "Krabičky na číselká a písmenká"

Možnosť ukladať si nejaké hodnoty do pamäte sa ukázala byť výhodná už pri tvorbe prvých kalkulačiek. Kalkulačky, ktoré sa dali programovať, sa často hrdili tým, že „majú až desať pamätí“. Ak človek programuje niečo pre počítač, má k dispozícií jeho operačnú pamäť a môže do nej ukladať čo len chce a potrebuje. (Pre porovnanie – do jedného gigabajtu pamäte počítača sa vmestí 134 217 728 šesnásťciferných čísel. Oproti tým desiatim je to celkom pokrok.)

„Pamäte“, do ktorých sa čísla (alebo iné objekty) pri programovaní ukladajú, sa nazývajú **premenné**. S premennými ste sa už stretli pri programovaní v Imagine.[3]

Na začiatku každej procedúry v jazyku C je treba povedať, aké premenné v nej budete používať a akého sú typu. (Programátori tomu hovoria, že musíte premenné **deklarovať**.) Premenné totiž môžu byť rôznych typov. Do premennej typu **int** sa dá uložiť celé číslo (napr. `3`, `-97` alebo `0`). Do premennej typu **float** sa dá uložiť desatinné číslo (napr. `3.141593` alebo `1.5e15` = $= 1.5 \times 10^{15} =$ `1500000000000000`). Do premennej typu **char** sa dá uložiť jedno písmeno (napr. `'e'` alebo `'U'`).[4] Takže ak uvidíte, že nejaký program v C-čku začína

```
main()
{
    int i,j;
    float f;
    char znak;
...
```

tak viete, že programátor bude v procedúre `main()` používať dve premenné, do ktorých bude ukladať celé čísla (premenné `i` a `j`), jednu premennú, do ktorej bude ukladať desatinné čísla (premennú `f`) a jednu premennú, do ktorej bude ukladať znaky (premennú `znak`).

Konkrétne hodnoty sa do premenných priradzujú s pomocou `=`. Funguje to tak, že počítač sa pozrie, čo sa mu nachádza vpravo od `=`, vyráta hodnotu, akú to má a výsledok vloží do premennej, ktorá je vľavo od `=`. (Dajte pozor na to, aby premenná do ktorej vkladáte hodnotu bola skutočne vľavo od `=`. Ak by ste ju dali na druhú stranu, nebude to fungovať.) Takže v programe

```
main()
{
    int i;

    i = 5;
    i = i + 3;
    i = i - 1;
}
```

sa najprv do premennej `i` vloží číslo 5, potom sa do nej vloží číslo 8 a nakoniec sa do nej vloží číslo 7. (Viete prečo?)

Zápis uvedeného programu sa dá skrátiť. Keď potrebujeme obsah premennej zvýšiť o 3, môžeme miesto `i = i + 3;` písať `i += 3;` Podobne zápis `i -= 7;` zmenší hodnotu premennej

3 Podaktorí si možno ešte matne spomínajú na príkazy `urob` a `urobTu`.

4 Situácia s typmi premenných je trošku komplikovanejšia. Existuje ich ešte niekoľko typov a jazyk C obsahuje prostriedky, ako si ďalšie typy vyrobiť. Na začiatok vám však stačí poznať tieto tri.

`i` o 7. V prípade, že celočíselnú premennú `i` potrebujeme zväčšiť o 1, môžeme použiť superkrátky zápis `i++;` Rovnako `i--;` premennú zmenší o 1. Pokročilejší programátor by teda uvedený program napísal takto:[5]

```
main()
{
    int i;

    i = 5;
    i += 3;
    i--;
}
```

Ak chceme vypísať obsah premennej na obrazovku, slúži na to stará známa z minulej lekcie – funkcia `printf`. Skrýva to ale v sebe drobný zádrheľ. Totiž funkcia `printf` potrebuje ako vstupný parameter nejaký text v úvodzovkách a ten text presne vytlačí. Ak teda napíšete príkaz `printf("i");` program vám nevypíše hodnotu premennej `i`, ale písmenko i. A ak napíšete príkaz `printf(i);` program sa vám nepodarí skompilovať, pretože `printf` ako prvý parameter potrebuje text v úvodzovkách.[6] Táto situácia sa rieši tak, že do toho reťazca môžete vložiť isté špeciálne značky. Keď `printf` bude svoj reťazec tlačiť, všimne si ich a namiesto nich vytlačí veci, ktoré jej zadáte, ako ďalšie parametre. Ak chcete niekde vypísať hodnotu typu `int`, použijete značku `%d`. Ak chcete vypisovať `float`, použijete `%f` a ak chcete vypisovať nejaký `char`, použijete `%c`. Takže ak mám napríklad premennú `i` typu `int` a je v nej uložené číslo 7 a do programu napíšem príkaz

```
printf("%d trpaslikov a %d snehulienka\n", i, i-6);
```

program prejde text, ktorý má `printf` na začiatku. Keď narazí na prvé `%d`, pozrie dozadu a miesto `%d` tam vloží hodnotu nasledujúceho parametra (teda 7). Potom vypisuje textík ďalej až kým nenatrafí na druhé `%d`. Tam sa pozrie, aká je hodnota ďalšieho parametra (pričom môže vykonať aj nejaké výpočty) a vloží ju tam. Program teda vypíše `7 trpaslikov a 1 snehulienka`.

Takže úlohy pre vás:

Úloha č.1: Napíšte do súboru `ludolf.c`, skompilujte a spustite nasledovný program:

```
#include <stdio.h>

main()
{
    float pi;

    pi = 3.1415926535;
    printf("Pi = %f",pi);
}
```

5 Ešte pokročilejší programátor by napísal `main() {int i=7;}` a naprostý C-čkový guru by sa písaním programu, ktorý aj tak nič nerobí už vôbec nezdržiaval.

6 Prvý parameter musí byť reťazec. Neskôr sa dozviete fintu, ako sa dá reťazec používať aj bez úvodzoviek, ale teraz by to miatlo.

Úloha č.2: Napíšte program `pism.c`, v ktorom budete mať premennú `pismk` typu `char`. Najprv do nej vložíte písmeno A (príkazom `pismk = 'A';` – tie apostrofy okolo toho `A` sú dôležité) potom jej obsah zväčšíte o 1 a vypíšete. Čo dostanete? Čo dostanete, ak o 1 zväčšíte premennú v ktorej je uložené písmenko `'Z'`? Čo napíšu na obrazovku príkazy

```
printf("%c %d\n",65,65);
printf("%c %d\n",'C','C');
```

Úloha č.3: Napíšte program, v ktorom budete mať dve premenné typu `int` (mená im vymyslite vy), naplníte ich nejakými hodnotami a potom vypíšete ich súčet a súčin. (Na násobenie sa používa `*`.) Je nutné aby program vypisoval súčet a súčin správne pre ľubovoľné hodnoty tých dvoch premenných.

Úloha č.4 Napíšte program, ktorý vypíše nasledujúcu tabuľku:

```
1 x 9 = 9
2 x 9 = 18
3 x 9 = 27
4 x 9 = 36
5 x 9 = 45
6 x 9 = 54
7 x 9 = 63
8 x 9 = 72
9 x 9 = 81
10 x 9 = 90
```

pričom sa v jeho zdrojovom kóde nachádza znak `9` iba raz a to v riadku, ktorý vyzerá takto:

```
i = 9;
```

3. lekcia

Vstup z terminálu alebo "Načo je tam to & ?"

Programy, ktoré sme doteraz v jazyku C vytvárali, boli pomerne nezdvorilé – ak sa nám ich podarilo skompilovať a spustiť, tak na nás väčšinou niečo vychrlili a tým to pre ne haslo. Po spustení sme ich beh už nemohli nijako ovplyvniť. Dobre vychovaný program by sa ale takto správať nemal. Mal by sa nás aspoň občas na niečo spýtať.

Jednou z možností, ako ovplyvniť beh programu, je vstup z terminálu. Na určitom mieste program zastaví a čaká, kým niečo napíšete. Potom si to uloží do premennej (ktorú musí mať predtým deklarovanú) a ďalej s tým pracuje.

Na vstup z terminálu slúži funkcia `scanf`. Funguje podobne ako funkcia `printf`. Ako prvý argument musí dostať riadiaci reťazec, ktorý jej povie, aké typy premenných bude čítať. Potom nasleduje zoznam premenných, do ktorých sa majú načítané hodnoty uložiť. Ale pozor! Pred každé meno premennej sa musí napísať znak &. V praxi to bude vyzerať takto:

```
#include <stdio.h>

main()
{
  int i,j;

  printf("Ja som mimoriadne nadany program, ktory vie scitat.\n");
  printf("Zadaj cislo: ");
  scanf("%d", &i);
  printf("Este jedno: ");
  scanf("%d", &j);
  printf("%d + %d = %d. To som dobry. Co?\n", i, j, i+j);
}
```

V programe najprv deklarujeme dve premenné `i` a `j`. Potom sa vypíšu nejaké úvodné bláboly a zavolá sa funkcia `scanf`. Jej úvodný reťazec jej povie, že má očakávať celé číslo (ak tam niekto napíše niečo iné, program za seba neručí a asi bude robiť hlúposti). To, čo užívateľ zadá z klávesnice, sa uloží do premennej `i`. Potom sa zas niečo vypíše a očakáva sa vstup ďalšieho celého čísla, ktoré sa uloží do premennej `j`. Nakoniec sa vypíše obsah oboch premenných a ich súčet.

Pozor!!! Vo funkcii `scanf` sa do úvodného reťazca okrem riadiacich znakov (ako napr. `%f`, `%c`, `%d`) nič nepíše. Ak by ste tam niečo napísali, počítač to zmätie, takže do premenných sa vám nemusí nič nenačítať.[7]

Úloha č.1: Napíšte a skompilujte uvedený program. Pochopte, ako funguje. Spustite ho viackrát. Vyskúšajte zadať aj iný vstup, než program očakáva (napr. nejaké písmená).

Patrilo by sa vysvetliť, prečo sa do `printf` píšu premenné normálne a do `scanf` pred ne treba písať `&`. Postupnosť znakov `&i` znamená „miesto, kde sa v pamäti nachádza premenná `i`“. Ak by ste napísali príkaz `printf("%d",&i);` program vám nevypíše, čo sa nachádza v krabičke

7 Za istých okolností sa do riadiaceho reťazca nejaké znaky občas píšu. Ak to ale chcete použiť, najprv si z nejakého iného zdroja naštudujte, na čo je to dobré a ako presne vtedy funkcia `scanf` funguje. Inak spôsobíte chaos.

nazývanej "i", ale adresu v pamäti, kde sa tá krabička na čísla nachádza. Môže to fungovať napríklad takto:

```
#include <stdio.h>

main()
{
  int i;
  i = 5;
  printf("Premenna i je na adrese %x a obsahuje cislo %d\n",&i,i);
}
```

(Riadiaci znak %x vypíše celé číslo, ktoré mu určíte v šestnástkovej sústave, ktorá sa zvyčajne používa na určenie adresy v pamäti.)

Keď používame funkciu printf, ide nám väčšinou o to, aby sme zistili, čo sa v premenných nachádza. Vtedy tam & nedávame. Keď ale používame funkciu scanf, musíme počítaču povedať, kam do pamäte má načítanú hodnotu uložiť. Keď ju chceme uložiť do premennej cis, ako parameter dáme funkcii scanf výraz &cis, ktorý jej povie, kam do pamäte má načítanú hodnotu uložiť – na to miesto, kde sa nachádza premenná cis.

Keď tomu rozumiete, tešte sa. Keď nie, zapamätajte si, že do scanf treba dať pred meno premennej & a tiež to nejako pôjde.

Nasledujúce programy píšte tak, aby sa správali slušne, aby povedali, čo od vás chcú a aby povedali, čo vypisujú.

Úloha č.2: Napíšte program, ktorý načíta dve strany obdĺžnika a vypíše jeho obsah.

Úloha č.3: Napíšte program, ktorý načíta znak a vypíše jeho ASCII kód. (Ak neviete, čo sú ASCII kódy, pozrite si pozorne koniec úlohy 2 z druhej lekcie.)

Úloha č.4: Napíšte program, ktorý načíta číslo, vypíše číslo o jedna väčšie a komentár

```
Haha, vyhral som !!!
```

Úloha č.5: Napíšte program, v ktorom deklarujete tri premenné typu int a vypíšete adresy v pamäti, na ktorých sa tie premenné nachádzajú. Koľko bajtov zaberá jedna premenná typu int?

4. lekcia
Logika v C-čku
alebo "je rozdiel medzi = a =="

Všetky programy, ktoré sme zatiaľ v C-čku napísali, fungovali (ak vôbec fungovali) tým spôsobom, že vykonávali jeden príkaz za druhým a tak sa postupne prepracovali od začiatku až na koniec. To je síce fajn, ale keby sa týmto spôsobom mal napríklad urobiť program, ktorý sčíta všetky čísla od 1 do 1000, tak napísať postupnosť príkazov `a = 0; a = a + 1; a = a + 2; a = a + 3; ...` až do 1000 by mohlo trvať dlhšie, než keby to mal človek sčítať ručne. A keby ste mali napísať program, ktorý by sa mal najprv spýtať, kde má s tým sčítavaním prestať a až potom sčítavať, tak ten by sa týmto spôsobom nedal urobiť vôbec.[8] Preto je dôležité beh programu nejakým spôsobom riadiť.

Každý programovací jazyk poskytuje prostriedky, ako počítaču povedať, že nejaké príkazy má vykonať len za istých okolností, že niečo má opakovať dovtedy, kým nie je splnená nejaká podmienka alebo že si má vybrať z viacerých možností, ako pokračovať v tom, čo robí. Všetky tieto veci vyžadujú, aby sa dalo zistiť, či sú splnené nejaké podmienky.

Na zápis podmienok sa v jazyku C používajú **logické výrazy**. (Niekedy sa nazývajú aj **booleovské** na pamiatku pána Boolea, ktorý ich študoval z matematického hľadiska.) Logické výrazy môžu nadobúdať dve hodnoty – hodnotu **pravda** (anglicky **true**) a hodnotu **nepravda** (anglicky **false**). Niektoré jazyky (napr. Pascal) majú pre logické výrazy samostatný typ premenných. V C-čku sa však pre logické hodnoty používa typ **int**. Pričom hodnota 0 znamená „nepravda“ a akákoľvek iná hodnota znamená „pravda“.

Logické výrazy môžu testovať, či má alebo nemá premenná nejakú hodnotu, môžu sa v nich porovnávať hodnoty a jednoduché výrazy sa môžu spájať do zložitejších. Na porovnávanie dvoch čísel slúži operátor `==`. (Tie rovnítka tam musia byť dve, pretože jedno rovnítko slúži ako príkaz priradenia.) Takže výraz `cis == 7` nadobúda hodnotu „pravda“, ak je v premennej `cis` hodnota 7 a inak nadobúda hodnotu „nepravda“. Podobne sa dajú použiť znamienka `>`, `>=`, `<`, `<=`, ktoré znamenajú „väčší“, „väčší alebo rovný“, „menší“, „menší alebo rovný“. Na test, či sú dve hodnoty rôzne, sa používa operátor `!=`. Takže výraz `7 > a` nadobúda hodnotu „pravda“ len vtedy, ak je v premennej `a` uložená hodnota menšia ako 7 a výraz `i != j` nadobúda hodnotu „pravda“ iba vtedy, keď sú hodnoty uložené v premenných `i` a `j` rôzne. Ako sú logické výrazy vyhodnocované uvidíte v úlohe:

Úloha č.1: Pozrite si nasledujúci program a na papier si napíšte váš tip, čo bude vypisovať. Potom program skompilujte a spustite. Porovnajte si vaše tipy s realitou. Vysvetlite, prečo to píše to, čo to píše.

```
#include <stdio.h>

main()
{
    int i = 3, j = 4;
```

8 Nie, že by sa nedal napísať program, ktorý to dokáže a pritom každú inštrukciu vykoná presne raz. To s pomocou pomerne jednoduchej matematickej finty ide. Ale nejde to urobiť spôsobom uvedeným vyššie.

```
    printf("%d\n", i == 3);
    printf("%d\n", j >= 6);
    printf("%d\n", j = 5);
    printf("%d\n", i < 6);
    printf("%d\n", j == 4);
}
```

Vaše tipy sa pravdepodobne líšia od reality. Viete prečo?

Niekedy je nutné v jednej podmienke otestovať viacero vecí naraz. Napríklad potrebujete zistiť, či je premenná `i` väčšia ako 3 a menšia ako 17. Každú vec zvlášť viete zistiť jednoducho (`i > 3, i < 17`). Oba testy je však treba nejakým spôsobom spojiť. Ak chcete zistiť či sú splnené obidva naraz, použije sa operátor `&&`. Takže podmienka, ktorá zistí, či je hodnota v premennej `i` súčasne väčšia ako 3 a menšia ako 17 sa napíše takto: `i > 3 && i < 17`.[9]

Ak potrebujeme zistiť, či je splnená aspoň jedna z podmienok, použijeme operáciu `||`.[10] Takže výraz `i > 0 || j > 0` nadobúda hodnotu „nepravda“ jedine vtedy, ak sú hodnoty oboch premenných `i` a `j` menšie alebo rovné ako nula. Stačí, keď je hodnota v jednej z premenných kladná (kľudne môžu byť aj obidve), podmienka nadobudne hodnotu „pravda“.

Ak chceme hodnotu logického výrazu zmeniť na opačnú, používa sa na to operátor `!`.[11] Ak napríklad chceme napísať podmienku „v premennej `znak` nie je veľké písmeno“, (teda podmienku, ktorá nadobudne hodnotu „pravda“ ak v tej premennej veľké písmeno nie je a hodnotu „nepravda“ inak), budeme postupovať nasledovne: Najprv si napíšeme podmienku, ktorá nám zistí, či v premennej `znak` veľké písmeno je. Taká podmienka môže vyzerať napríklad takto:

```
znak >= 'A' && znak <= 'Z'
```

Táto podmienka však funguje presne naopak, než potrebujeme. Ale vďaka operátoru `!` z nej vyrobíme podmienku, ktorú potrebujeme nasledujúcim spôsobom:

```
!(znak >= 'A' && znak <= 'Z')
```

Úloha č.2: Napíšte program, ktorý načíta hodnotu do znakovej premennej `zn` a vypíše `1` ak je to číslica `'0'` až `'9'`. Inak vypíše `0`.

Úloha č.3: Vymyslite podmienku, ktorá nadobudne hodnotu „pravda“ buď vtedy, keď je v premennej `i` hodnota `0` alebo vtedy, keď je tam hodnota `1`. Inokedy nadobudne hodnotu „nepravda“. Vyskúšajte v programe.

Úloha č.4: Vymyslite a oskúšajte podmienku, ktorá otestuje, či je hodnota v premennej `i` párna. (Pomôcka: zvyšok po delení sa v C-čku ráta s pomocou operátora `%`. Takže napr. `8 % 3` je `2` a `9 % 3` je `0`.)

9 Tento operátor sa nazýva „a“, „and“ alebo „logický súčin“.

10 Operátor „alebo“, „or“ alebo „logický súčet“.

11 Tento operátor sa nazýva „negácia“.

5. lekcia

Príkaz if
alebo "zariadime sa podľa okolností"

V lekcii č. 4 sme hovorili o tom, ako vytvárať podmienky a ako ich jazyk C vyhodnocuje. V lekcii č. 5 si ukážeme niektoré spôsoby, ako sa to, či nejaká podmienka je alebo nie je splnená dá použiť a zneužiť pre naše účely.

Ak chceme nejaký kus kódu vykonať iba vtedy, keď je splnená podmienka, používa sa na to príkaz **if** (po slovensky „ak“). Tento príkaz má dva varianty. Jednoduchší funguje napríklad takto:[12]

```
char c;

printf("Zadaj na vstup pismenko A: ");
c = getchar();
if (c != 'A')
{
    printf("Vravel som A !!!\n");
}
```

Za príkazom `if` nasleduje podmienka uzavretá v zátvorkách. (Pozor! Častá začiatočnícka chyba je, že sa na tie zátvorky okolo podmienky zabudne.) Potom nasleduje kus kódu v kučeravých zátvorkách, ktorý sa vykoná len vtedy, keď je podmienka splnená.[13]

Náš program najprv vypíše výzvu, aby sa zadalo písmenko. Potom sa zavolá funkcia `getchar()`. Je to prvá funkcia, ktorá dáva nejaký výsledok, s ktorou sme sa zatiaľ stretli. Výsledkom funkcie `getchar()` je jeden načítaný znak z terminálu.[14] Ten sa vloží do premennej `c`. Za príkazom `if` je podmienka, ktorá je splnená vtedy, keď v premennej `c` nie je `'A'`. Takže ak tam človek nedá `A`, tak mu program vynadá.

Druhý zložitejší variant je kombinácia príkazov `if – else` („ak – inak“). Zase ukážka:

```
int i;

printf("Zadaj cislo: ");
scanf("%d", &i);
if (i % 2 == 0)
    printf("%d je parne cislo\n", i);
else
{
    printf("%d je neparne\n", i);
    printf("cislo.\n");
}
```

12 Aby to vôbec išlo spustiť, nasledujúci kus kódu musí byť umiestnený v nejakej funkcii (napr. `main()`). Aby sa dali použiť funkcie `getchar()` a `printf()`, musí byť načítané `stdio.h` Takéto veci budem pre budúcnosť považovať za samozrejmé, nebudem ich tam z priestorových dôvodov vypisovať a nebudem to zakaždým obkecávať.

13 V prípade, že je v kučeravých zátvorkách iba jeden príkaz, tak sa môžu vynechať. V našej ukážke by teda vôbec nemuseli byť. Ale nie je chyba, ak tam sú. Ak by medzi nimi mal byť viac, než jeden príkaz, tak sú nutné.

14 `c = getchar();` robí to isté, ako `scanf("%c",&c);` Funkcia `getchar` je ale jednoduchšia a s jej pomocou bola naprogramovaná aj funkcia `printf`.

Na začiatku sa po výzve načíta jedno celé číslo. Za príkazom `if` je podmienka, ktorá je splnená vtedy, keď je číslo v premennej `i` párne. (Zvyšok po delení dvoma (`i % 2`) je rovný nule.) V prípade, že tá podmienka splnená je, vykoná sa kus kódu, ktorý nasleduje tesne po podmienke. (Keďže je to len jeden príkaz, mohli sme vynechať `{` a `}`.) V prípade, že podmienka splnená nie je, vykoná sa kód za príkazom `else`. (Keďže sú tam až dva príkazy, sú kučeravé zátvorky nutné!!! Keby ste ich tam nedali, jazyk C by za súčasť príkazu `else` pokladal iba prvý z nasledujúcich príkazov a druhý by vykonal vždy.)

Úloha č.1: Napíšte program podľa druhej ukážky. Vyskúšajte ako funguje. Vynechajte kučeravé zátvorky za `else`, znovu skompilujte a vyskúšajte.

Úloha č.2: Napíšte program, ktorý sa vás spýta, či máte radi programovanie, a podľa vašej odpovede (`A` alebo `N`) vám povie niečo duchaplné.

Úloha č.3: Vylepšite program z úlohy č.2 tak, aby v prípade, že neodpoviete ani `A` ani `N` oznámil, že dostal neočakávanú odpoveď.

Úloha č.4: S pomocou príkazu `if – else` napíšte program, ktorý načíta celé číslo. Ak je toto číslo kladné, tak ho vypíše a inak vypíše nulu.

Riešenie úlohy č. 4 sa dá uľahčiť s pomocou **podmienených výrazov**. Podmienený výraz vyzerá napríklad takto: `i > 0 ? i : 0` Na jeho začiatku je podmienka, potom je otáznik, prvá možnosť, dvojbodka a druhá možnosť. Hodnota výrazu sa zistí jednoducho – pozrie sa, či je podmienka splnená, ak je, vezme sa prvá možnosť a ak nie je, vezme sa druhá. Takže ak je v premennej `i` kladné číslo, hodnota uvedeného výrazu je `i`. A ak je v premennej `i` záporné číslo alebo nula, hodnota výrazu je `0`. Úloha č. 4 sa teda dá riešiť takto:

```
int i;
scanf("%d", &i);
printf("%d\n", i > 0 ? i : 0 );
```

Úloha č.5: Napíšte program, ktorý načíta dve čísla a vypíše to väčšie z nich. Skúste to spraviť s pomocou podmienených výrazov. (Nepovinný variant pre frajerov a guruov: Napíšte program, ktorý načíta tri čísla a vypíše ich od najväčšieho po najmenšie.)

6. lekcia

Cyklus while
alebo "kým to nebude hotové, ostaneš tu"

Pri programovaní sa človek často dostáva do situácie, že potrebuje, aby sa nejaký kus programu opakoval. Vo väčšine počítačových hier sa dookola opakuje postupnosť „zisti si, čo hráč urobil (napr. či stlačil klávesu alebo pohol myšou), vyhodnoť to a prekresli obrazovku“. Program, ktorý tlačí výplatné pásky musí tú istú činnosť zopakovať pre každého zamestnanca. Program na prehrávanie hudby musí opakovať činnosť „prečítaj trochu údajov zo súboru alebo z disku, niečo s tým urob a pošli to do reproduktorov“ až kým nezahrá celú skladbu.

Na programovanie takýchto vecí slúžia **cykly**. Céčko pozná viacero spôsobov, ako cyklus napísať. Jedným z nich je príkaz `while`. Ľudia v angličtine zbehlí vedia, že to znamená „kým“. Kus kódu uzavretý v cykle sa teda bude opakovať, **kým** je splnená podmienka cyklu. Dajú sa tak robiť roztodivné veci – pozrite si nasledujúci príklad. (Znovu – omáčičku ako `include` si napíšte sami.)

```
int i = 1, dokolko;

printf("Ja som mudry pocitac a viem pocitat.\n");
printf("Do kolko mam napocitat? ");
scanf("%d", &dokolko);

while (i <= dokolko)
{
    printf("%d\n", i);
    i++;
}
```

V premennej `i` je na začiatku 1. Vo vnútri cyklu sa vždy vypíše, čo je v `i`-čku a zväčší sa to o 1. To sa opakuje, kým (while) je hodnota v `i`-čku menšia alebo rovná hodnote v premennej `dokolko`. Keď táto podmienka prestane byť splnená, cyklus sa skončí.

Podobne ako pri príkaze `if` musí byť podmienka v zátvorke a **nepíše** sa za ňou bodkočiarka. Kučeravé zátvorky označujú kus kódu, ktorý sa má opakovať a ak sa má opakovať len jeden príkaz, tak tam byť nemusia.[15]

Úloha č.1: Napíšte to a skompilujte. Čo to spraví, keď tomu poviete aby rátal, do 20? Skončí pri 20 alebo pri 19? Čo to spraví, keď tomu poviete, aby rátal do -7 ?

Ako môžete vidieť z výsledkov úlohy č.1 (teda konkrétne z tej mínus sedmičky), podmienka cyklu `while` sa vyhodnotí ešte pred tým, než program vôbec do cyklu vstúpi. Ak nie je splnená hneď na začiatku, telo cyklu sa nevykoná vôbec. To sa nám ale nie vždy hodí. Niekedy by sme boli radi, aby sa podmienka neoverovala na začiatku, ale až na konci cyklu. Skrátka aby cyklus aspoň raz prebehol a až na konci sa zisťovalo, či sa to má znovu opakovať.

---

15 Poznámka pre guruov: ten cyklus sa teda dal napísať aj

```
while (i <= dokolko)
    printf("%d\n", i++);
```

Aby sme také niečo vedeli urobiť, musíme v programe nejakým kľúčovým slovom vyznačiť, kde ten cyklus začína. Samotným slovom `while` to nepôjde – to sa bude vyskytovať až na konci cyklu pri podmienke. Na začiatku takéhoto cyklu bude stáť slovíčko `do` (po anglicky „rob“). Ako to funguje, si môžete pozrieť na nasledujúcom príklade:

```
char pis;

printf("Napis A!\n");
do
{
    pis = getchar();
    if (pis != 'A')
        printf("Hovoril som A!!! Takze znovu: Napis A!\n");
} while (pis != 'A');

printf("Hura, naozaj je to A!\n");
```

Tento program načíta do premennej `pis` znak z klávesnice. Ak to nie je `A`, vypíše varovanie. Túto činnosť bude opakovať dovtedy, kým v tej premennej `pis` bude niečo iné než `A`.

Oproti predošlému variantu `while` je tu ešte jeden rozdiel – keďže celý cyklus podmienkou za `while` končí, **píše** sa za ňou bodkočiarka!!!

Takže úlohy:

Úloha č.2: Napíšte program, ktorý napíše za trest stokrát `Nebudem zhadzovat operacny system.`

Úloha č.3: Napíšte program, ktorý do premennej `poc` načíta celé číslo z klávesnice a vypočíta súčet `1 + 2 + 3 + ... + poc`. (Spraví sa to jednoduchou úpravou prvej úlohy – pridajte tam ďalšiu premennú, na začiatku ju nezabudnite vynulovať a v každom behu cyklu k nej pripočítajte riadiacu premennú cyklu.)

Úloha č.4: Napíšte program, ktorý vypíše do tabuľky ASCII kódy všetkých veľkých písmen. Funkcia `printf` sa v ňom môže nachádzať maximálne dvakrát. (Tí, čo zabudli, čo sú ASCII kódy, nech sa opäť pozrú na koniec úlohy č. 2 z druhej lekcie.)

Úloha č.5: (nepovinná, pre guruov) Upravte druhý vzorový program tak, aby si to `A` pýtal maximálne trikrát a ak sa človek trikrát netrafí, tak vypíše nejaký komentár psychického stavu jedinca za klávesnicou a skončí.

7. lekcia

Cyklus for alebo "tri v jednom"

Cyklus `while`, s ktorým ste sa zoznámili v predošlej lekcii je skvelý a univerzálny nástroj. Čo sa cyklov týka, všetko sa dá urobiť cez neho. Keď potrebujeme opakovať nejakú postupnosť príkazov určitý počet krát alebo keď potrebujeme čosi opakovať dovtedy, kým je splnená nejaká podmienka, vždy sa to s pomocou cyklu `while` a nejakých premenných dá uhrať. Vyzerá to tak, ako by sme už ďalšie typy cyklov nepotrebovali.

Napriek tomu je tu cyklus `for`. Totiž ľudia, ktorí C-čko vymýšľali si všimli, že väčšina cyklov funguje tak, že na začiatku sa spraví nejaká príprava, potom cyklus beží kým je splnená nejaká podmienka. Pritom sa v každom behu cyklu udeje nejaká malá zmena. No a rozhodli sa, že všetky tieto veci napchajú do jedného príkazu.

Cyklus `for` sa používa hlavne vtedy, keď dopredu vieme, koľkokrát sa má cyklus vykonať, ale dá sa s úspechom použiť aj inde. Jeho použitie ukážeme na programe, ktorý napočíta do desať:

```
int i;

for ( i = 1; i <= 10; i++ )
{
    printf("%d\n", i);
}
```

Za príkazom `for` nasleduje zátvorka v ktorej sa nachádzajú tri chlieviky oddelené **bodkočiarkami**. V prvom chlieviku je napísané, čo sa má spraviť ešte predtým, ako sa cyklus začne vykonávať. (Do premennej `i` sa hodí jednička.) V druhom chlieviku je podmienka. Cyklus bude bežať dovtedy, kým bude táto podmienka splnená. (Teda kým bude `i` menšie alebo rovné desiatim.) Vždy na začiatku nového behu cyklu sa podmienka skontroluje, a keď prestane platiť, cyklus sa skončí a začnú sa vykonávať príkazy, ktoré v programe nasledujú za ním. V treťom chlieviku je zmena, ktorá sa vykoná po každom prebehnutí cyklu. (Premenná `i` sa zväčší o 1.) Za touto „veľkou“ zátvorkou **nenasleduje** bodkočiarka ale rovno kučeravé zátvorky s príkazmi, ktoré sa majú opakovať. (Ak je príkaz len jeden, môžeme si kučeravé zátvorky odpustiť.)

Uvedený program robí presne to isté, ako program

```
int i;

i = 1;
while ( i <= 10 )
{
    printf("%d\n", i);
    i++;
}
```

Oba tieto programy sú z hľadiska počítača naprosto rovnaké. Z oboch sa pri kompilácii vygenerujú presne tie isté strojové inštrukcie. Ale zápis cez `for` je v tomto prípade kratší a prehľadnejší.

Iný príklad použitia cyklu `for` môžete vidieť tu:

```
int i,j,velkost;

printf("Zadaj velkost:");
scanf("%d", &velkost);
for ( i = 1; i <= velkost; i++ )
{
     for ( j = 1; j <= i; j++ )
     {
         putchar('*');
     }
     putchar('\n');
}
```

Úloha č.1: Prídite na to, čo robí uvedený program. Potom (až potom!!!) to napíšte, skompilujte a vyskúšajte. (Funkcia `putchar` vypíše na monitor jediný znak, ktorý jej dáte ako parameter. `putchar('*');` je jednoduchší ekvivalent príkazu `printf("*");`)

Úloha č.2: S pomocou cyklu `for` napíšte program, ktorý vypíše všetky nepárne čísla od 1 do 99.

Úloha č.3: S pomocou cyklu `for` napíšte program, ktorý vypíše všetky čísla od 1 do 100 a ku každému napíše, či je párne, alebo nepárne.

Úloha č.4: Napíšte program, ktorý s pomocou `scanf` načíta číslo a potom urobí štvorček z hviezdičiek danej veľkosti. Napr. ak užívateľ zadá číslo 3, výsledok bude

```
***
***
***
```

Úloha č.5: (Nepovinná, pre machrov.) Napíšte program, ktorý načíta číslo a potom spraví symetrický trojuholník danej veľkosti. V každom riadku tak bude treba najprv vypísať správny počet medzier a až potom správny počet hviezdičiek. Pre vstup 4 to bude vyzerať takto:

```
   *
  ***
 *****
*******
```

8. lekcia

Príkazy break, continue a switch alebo "obušok z cyklu von"

Sú situácie, v ktorých je vhodné cyklus ukončiť. (Najmä vtedy, keď chceme, aby program vôbec skončil.) Bežne sa to robí tak, že sa zvolí správna podmienka cyklu. Keď podmienka prestane platiť, cyklus sa prestane cykliť a program pokračuje za ním. Občas sa ale stane, že kdesi vo vnútri cyklu zistíme, že musíme cyklus ukončiť okamžite. A práve na to slúži príkaz `break`.

V prípade, že použijeme `break`, program ihneď opustí cyklus, v ktorom sa nachádza a pokračuje až za ním. Ak sa program práve nachádza vo viacerých cykloch (vo vnútri jedného cyklu môže byť ďalší!), tak vyskočí z toho najvnútornejšieho. Použitie príkazu `break` si môžete pozrieť v nasledujúcej ukážke:

```
int i = 0;

for ( ; 1 ; )
{
    putchar('*');
    if ( i == 10 )
        break;
    i++;
}
putchar('\n');
```

Cyklus `for` nachádzajúci sa v ukážke je evidentne cyklus nekonečný. Nič sa v ňom nezačína ani nemení a v podmienke má napevno nastavené `1` čiže „pravda“. Za normálnych okolností by takýto cyklus bežal až do vypnutia počítača či zhodenia programu. Ale vďaka tomu, že hodnota premennej `i` sa v každom priebehu cyklu zväčší o 1 a keď nadobudne hodnotu `10`, zavolá sa `break`, program šťastne skončí. Mimochodom – koľko hviezdičiek vypíše? Prečo práve toľko?

Ďalší zaujímavý príkaz sa nazýva `continue`. Používa sa v trochu odlišnej situácii: Predstavte si, že píšete nejaký dlhý cyklus. A kdesi uprostred cyklu zistíte, že potrebujete, aby sa toto „kolo“ cyklu skončilo a okamžite začalo ďalšie.[16] A práve vtedy vám príkaz `continue` príde vhod. Pozrite sa na príklad:

```
int i;

for( i = 0; i <= 100; i++ )
{
    if (i % 2 == 0)
        continue;
    printf("%d\n",i);
}
```

V cykle `for` nadobúda premenná `i` postupne všetky hodnoty od `0` po `100`. Keďže sa ale pri všetkých párnych hodnotách zavolá príkaz `continue`, k výpisu sa program dostane len pri nepárnych hodnotách.

16 Pozor! Teraz z cyklu nevyskakujete. Budete sa v ňom točiť ďalej, potrebujete iba zrušiť aktuálne kolo.

Posledný dnešný príkaz je príkaz `switch`. Je to príkaz prepínač. Opäť sa pozrite na ukážku:

```
int i;

printf("Napis cislo 1, 2 alebo 3: ");
scanf("%d", &i);
switch (i)
{
case 1:
    printf("Napisal si jednicku.\n");
    break;
case 2:
    printf("Napisal si dvojku.\n");
    break;
case 3:
    printf("Napisal si trojku.\n");
    break;
default:
    printf("Napisal si nejaku blbost.\n");
}
```

Za samotným príkazom `switch` je v zátvorkách niečo typu `char` alebo `int`. (Teda napríklad `float` tam byť nesmie.) Potom nasledujú kučeravé zátvorky v ktorých sú jednotlivé možnosti, uvedené slovíčkom `case` a ukončené dvojbodkou. Ak je hodnota výrazu v zátvorke niektorou z možností, vykonajú sa všetky príkazy za touto možnosťou. V prípade, že žiadna z možností nenastane, vykoná sa možnosť `default`. (V prípade, že tam túto možnosť nedáte, nevykoná sa nič.)

Úloha č.1: Napíšte, pochopte a vyskúšajte prvé dva programy.

Úloha č.2: Napíšte tretí program, ale vynechajte z neho všetky príkazy `break`. Aký bude výstup, keď na vstup napíšete `3`? A keď napíšete `1`?

Úloha č.3: S pomocou príkazu `switch` a skúseností z predošlej úlohy napíšte program, ktorý si vypýta číslo, v prípade, že je to `1`, `2` alebo `3` napíše `obstojne cislo`, ak je to `4` alebo `5`, napíše `ujde to` a ak je to nejaké iné číslo, napíše `humus`.

Úloha č.4: Napíšte program, ktorý zistí a vypíše najmenší násobok čísla `31` väčší ako `10000`. Použite nekonečný cyklus (v ktorom sa nejaká premenná vždy zväčší o `31`) a príkaz `break`.

Úloha č.5: (nepovinná, pre guruov) Fibonacciho postupnosť je postupnosť čísel `1, 1, 2, 3, 5, 8, 13,...`, každý ďalší člen je súčtom dvoch predchádzajúcich. Napíšte program, ktorý vypíše všetky členy Fibonacciho postupnosti menšie než `10000`.

9. lekcia

Všetky doterajšie príkazy alebo "ukážte, či ste frajeri"

Riadiace štruktúry jazyka C máme šťastne za sebou. Aby ste si vyskúšali, akí ste dobrí, čaká vás sada úloh. Niektoré sú ľahšie, iné ťažšie. Môžete ich riešiť v ľubovoľnom poradí.

Úloha č.1: Napíšte program, ktorý načíta **desatinné** číslo a vypíše jeho tretiu mocninu. (Tretia mocnina čísla 6 je 6.6.6 = 216)

Úloha č.2: Napíšte program, ktorý načíta tri čísla a vypíše prostredné (podľa veľkosti) z nich.

Úloha č.3: Napíšte program, ktorý načíta číslo. Ak je toto číslo `6`, vypíše `Dobré ráno.` Ak je toto číslo `12`, vypíše `Dobry den.` Ak je toto číslo `19`, vypíše `Dobry vecer.` Vo všetkých ostatných prípadoch vypíše `Dovidenia.`

Úloha č.4: Napíšte program, ktorý načíta číslo a patričný počet krát vypíše `Chcem jeden bod.`

Úloha č.5: Napíšte program, ktorý načíta číslo a nakreslí trojuholník danej veľkosti. Príklad pre vstup 3:

```
*
**
***
**
*
```

Úloha č.6: Napíšte program, ktorý číta znaky z klávesnice až kým nenájde `'X'`. Potom vypíše, koľko z prečítaných znakov boli znaky `'A'`.

Úloha č.7: Napíšte program, ktorý načíta znak z klávesnice. Ak tento znak nie je písmeno, tak ho vypíše, ak je to malé písmeno, vypíše k nemu zodpovedajúce veľké písmeno a ak je to veľké písmeno, vypíše k nemu zodpovedajúce malé písmeno.

Úloha č.8: Napíšte program, ktorý načíta dve celé čísla a vypíše všetky párne čísla, ktoré ležia medzi nimi. (Pre vstup `3` a `8` vypíše čísla `4`, `6` a `8`. Pre vstup `6` a `3` vypíše čísla `4` a `6`. Pre vstup `3` a `3` nevypíše nič.)

Úloha č.9: Napíšte program, ktorý vypočíta faktoriál zo zadaného čísla. (Pre vstup 4 bude výsledok 1.2.3.4 = 24)

Úloha č.10: Napíšte program, ktorý načíta 20 čísel a vypíše, koľko z nich bolo od `5` do `10`.

Úloha č.11: Napíšte program, ktorý pozostáva iba z dvoch príkazov (omáčička a deklarácia premenných sa nerátajú) a ktorý načíta číslo a v prípade, že je to číslo párne, vypíše z neho polovicu a inak vypíše jeho trojnásobok zväčšený o 1.

Úloha č.12: Napíšte program, ktorý načíta číslo a opakuje s ním operáciu z predošlej úlohy, až kým nedostane jedničku. Čísla priebežne vypisuje. (Teda pre vstup `3` vypíše čísla `3, 10, 5, 16, 8, 4, 2, 1`.) Mimochodom – myslíte, že takýto program vždy skončí?

Úloha č.13: (Nepovinná, pre guruov.) Napíšte program, ktorý načíta číslo a spraví patričný počet sústredných štvorcov. Príklad pre vstup `3`:

```
***********
*         *
* ******* *
* *     * *
* * *** * *
* * * * * *
* * *** * *
* *     * *
* ******* *
*         *
***********
```

10. lekcia
Súbory
alebo "prečítaj mi, čo je na disku"

Všetky programy, ktoré sme doteraz urobili, čítali svoj vstup z klávesnice a výsledky vypisovali na terminál. Pri jednoduchších programoch s tým vystačíme. Ale v prípade, že by sme museli na vstup zadávať päťsto čísel, prepísať kapitolu nejakej knižky alebo z klávesnice zadať obrázok, nastali by problémy. A práve pre to, aby sa podobné veci robili ľahšie, ľudia vymysleli súbory.

Čokoľvek si ukladáte na disk, USB kľúčik alebo DVD, ukladá sa tam v súboroch. A v tejto lekcii sa dozviete niektoré základné finty, ako vaše programy môžu súbory otvárať a čítať z nich alebo do nich zapisovať.

Pre prácu so súbormi používa C-čko štruktúru `FILE`. Ak chcete nejaký súbor otvoriť, musíte si najprv zadeklarovať premennú typu `FILE *`. (Načo je tam tá hviezdička sa dozviete v dvanástej lekcii. V tejto súvislosti ale rozhodne neznamená „krát“.) Potom na túto premennú nalepíte nejaký konkrétny súbor z disku. Pri tom nalepovaní musíte povedať, akým spôsobom sa má súbor otvoriť. Sú v podstate tri základné možnosti. Súbor môžete otvoriť na čítanie (v tom prípade už musí na disku existovať), môžete ho otvoriť na zápis (vtedy existovať nemusí, vytvorí sa nový; ak súbor s takým menom pred tým existoval, tak sa zmaže a nejeden programátor sa potom diví...) a môžete ho otvoriť na zápis na koniec (vtedy sa existujúci súbor nezmaže a môžete k nemu niečo pridať.) Keď je súbor šťastne otvorený, môžete z neho čítať alebo do neho zapisovať funkciami podobnými na tie, ktoré už poznáte. Keď prácu so súborom skončíte, je dobré súbor za sebou zavrieť, aby sa previedla synchronizácia a mohli sa uvoľniť systémové zdroje (po slovensky: aby sa to, čo sa má do súboru zapísať, skutočne zapísalo a neostalo trčať v pamäti a aby si systém nemusel pamätať, že tamten súbor je ešte stále otvorený.) V praxi to vyzerá nasledovne:

```
FILE *f;
int c;

f = fopen("pokus.txt","r");
while ((c = getc(f)) != EOF)
    putchar(c);

fclose(f);
```

Na začiatku sme deklarovali premennú `f` typu `FILE *` a premennú `c` typu `int`. Potom sme s pomocou funkcie `fopen` k premennej `f` priplácli súbor `pokus.txt`. Druhý parameter `"r"` funkcie `fopen` hovorí, že súbor otvárame na čítanie (po anglicky *read*). Keby sme ho otvárali na zápis, bolo by tam `"w"` (ako *write*) a keby sme ho otvárali na pripájanie, bolo by tam `"a"` (pripojiť je *append*).

Potom nasleduje cyklus v ktorom sa opakovane volá funkcia `getc(f)` – slúži na to isté ako funkcia `getchar()` s tým rozdielom, že načíta jeden znak zo súboru priplácnutému k `f`. Ak sa náhodou už prečítal celý súbor, funkcia vráti hodnotu `EOF`. (`EOF` je číslo závislé na operačnom systéme. Väčšinou je to –1.[17]) Takže ten cyklus bude znak po znaku prepisovať celý súbor na

17 Práve z toho dôvodu bola premenná `c` deklarovaná ako `int`. Premenná typu `char` môže nadobúdať iba hodnoty od `0` do `255` a tá `-1` by sa do nej nedala priradiť.

obrazovku, až kým nepríde na jeho koniec. Keď cyklus skončí, súbor sa zavrie príkazom `fclose(f)`.

Úloha č.1: Vytvorte si editorom súbor `pokus.txt` a niečo do neho napíšte. Potom do iného súboru napíšte, skompilujte a spustite uvedený program.

Ak chcete čítať zo súboru priviazanému k `f` znaky, použijete príkaz `getc` (ako v predošlej ukážke). Ak chcete zo súboru čítať čísla, použijete funkciu `fscanf`. Funkcia `fscanf` pracuje rovnako ako funkcia `scanf`, s tým malým rozdielom, že ako prvý parameter treba uviesť, z ktorého súboru sa číta. Teda na načítanie celého čísla zo súboru priviazanému k `f` do premennej `i` slúži príkaz `fscanf(f,"%d",&i)`. Ak chcete zo súboru čítať desatinné čísla, samozrejme miesto `%d` napíšete do riadiaceho reťazca `%f`.

Na zápis do súboru slúži funkcia `putc`, ktorá do súboru vloží jeden znak (napr. `putc('A',f)` – pozor, tu je prvý parameter znak, ktorý idete písať a až druhý je smerník na súbor). Druhá možnosť je funkcia `fprintf`. Táto funguje rovnako ako `printf`, len jej treba povedať, do ktorého súboru má písať, napr. `fprintf(f,"Premenna i ma hodnotu %d.",i)` zapíše do súboru priradenému k `f` hodnotu premennej `i` aj s pokecom.

V nasledujúcej ukážke skopírujeme súbor `pokus.txt` do súboru `kopia.txt` a popri tom spočítame, koľkokrát sa v ňom vyskytne písmeno `k`.

```
FILE *fr, *fw;
int c, pocet = 0;

fr = fopen("pokus.txt","r");
fw = fopen("kopia.txt","w");

while ((c = getc(fr)) != EOF)
{
    fprintf(fw,"%c",c);
    if (c == 'k')
        pocet++;
}

fclose(fr);
fclose(fw);
printf("Pismenko k tam bolo %d krat.\n",pocet);
```

Úloha č.2: Pochopte, skompilujte a vyskúšajte druhú ukážku.

Úloha č.3: Napíšte program, ktorý zo súboru `cisla.txt` prečíta tri celé čísla a vypíše ich súčet.

Úloha č.4: Napíšte program, ktorý zistí počet riadkov súboru `pokus.txt`. (Pomôcka: na konci každého riadku okrem posledného sa nachádza znak `'\n'`.)

Úloha č.5: Napíšte program, ktorý skopíruje súbor `pokus.txt` do súboru `kopia.txt`, ale všetky písmená `'A'` pri tom vynechá.

Na záver ešte jedna užitočná finta. Sú tri súbory, ktoré si otvára každý program automaticky. Je to štandardný vstup (meno súboru je `stdin`), štandardný výstup (súbor `stdout`) a štandardný chybový výstup (súbor `stderr`). Príkaz `printf("Hello, world\n");` môžeme teda rovnako dobre napísať ako `fprintf(stdout,"Hello, world\n");` a podobne príkaz `scanf("%d",&i);` robí presne to isté, ako `fscanf(stdin,"%d",&i);` No a teraz tá finta: Keď

píšete program, ktorý má čítať iba z jedného súboru a nechce sa vám všade písať `fscanf`, môžete použiť funkciu `freopen`. Keď program vykoná funkciu `freopen("data.txt", "r", stdin);` všetky príkazy `scanf` a `getchar`, ktoré štandardne čítajú z konzoly, budú namiesto toho čítať zo súboru `data.txt`. Podobne ak zadáte príkaz `freopen("vystup.txt", "w", stdout);` všetko, čo vypíšete s pomocou `printf` alebo `putchar` sa nebude vypisovať na konzolu, ale zapíše sa do súboru `vystup.txt`.

11. lekcia
Funkcie
alebo "rozdeľuj a panuj"

Programy, ktoré majú robiť niečo čo len trochu komplikovanejšie, majú jednu zákernú vlastnosť – zvyknú pomerne rýchlo narastať. Nie je vzácnosť, ak má zdrojový kód programu niekoľko megabajtov kódu. Skúste si na chvíľu predstaviť, že by bol celý ten program obsiahnutý v jedinej funkcii (konkrétne v `main`). Bol by to jeden neuveriteľný chaos. Hľadať logickú chybu v takej spúste textu by bola práca pre masového samovraha. Keby sa mala tá istá postupnosť príkazov vykonať na sto rôznych miestach programu, musel by ju programátor napísať stokrát. A vôbec, taký program by bol ako jeden obrovský dinosaurus. Dinosaury majú sklon vymierať a veľké funkcie majú sklon nefungovať. A tak ľudia vymysleli spôsob, ako činnosť jednej obrovskej funkcie rozdeliť medzi viacero menších.

Hlavný dôvod pre použitie funkcií je ono rímske príslovie „divide et impera“. Ak si celý problém rozdelím na viacero menších častí, môžem si každú zvlášť naprogramovať a odladiť a v hlavnom programe budem volať už iba funkcie, ktoré som si urobil a o ktorých viem, že fungujú.

V nasledujúcej ukážke máme okrem funkcie `main` ešte jednu:

```
#include <stdio.h>

int delitel(int a)
{
  int i;

  if (a == 1)
    return 1;
  i = 2;
  while (a % i != 0)
    i++;
  return i;
}

main()
{
  int i,j;
  printf("Najmensi netrivialny delitel %d je %d.\n",12,delitel(12));
  printf("Najmensi netrivialny delitel %d je %d.\n",35,delitel(35));

  i = 120;
  printf("%d = ",i);
  while (i > 1)
  {
    j = delitel(i);
    printf("%d",j);
    if (i > j)
      printf(".");
    i = i/j;
  }
  printf("\n");
}
```

Funkcia `delitel` vie nájsť najmenšieho netriviálneho deliteľa čísla, ktoré jej zadáte. Funguje takto:

Najprv sa pozrie, či ste jej náhodou nezadali jedničku. V tom prípade vyhlási, že jej výsledok je 1. Na to slúži to slovo `return`. Akonáhle funkcia vykoná príkaz `return`, jej činnosť sa končí a beh programu sa vracia tam, odkiaľ bola funkcia volaná. Situáciu, že funkcia bola volaná s hodnotou 1, máme teda vybavenú a viac sa o ňu nemusíme starať.

Ak je zadaná hodnota väčšia, funkcia začne od dvojky skúšať, či momentálna hodnota uložená v premennej `i` náhodou zadané číslo nedelí. Ak nie, zväčší `i` o jedna a skúša ďalej, až kým sa nepošťastí. (V krajnom prípade sa pošťastí, keď bude v `i` hodnota, ktorú funkcia dostala ako parameter – každé číslo delí samé seba.) Keď funkcia nejakého deliteľa nájde, opäť ho vráti ako výsledok s pomocou príkazu `return`.

Vo funkcii `main` teraz môžeme našu novú funkciu používať podľa potreby. Najprv ju použijeme len tak nasucho a vypíšeme delitele 12 a 35. Potom si napíšeme programček na rozklad na prvočísla. V cykle vždy nájdeme deliteľa, vypíšeme ho, číslo ním vydelíme a to opakujeme, až kým sa nedostaneme k jednotke.

Funkcie môžu (a nemusia) mať parametre[18], môžu (a nemusia) vyrobiť nejaký výsledok a môžete (nemusíte) z jednej funkcie volať ďalšiu. Ilustruje to nasledujúci príklad: [19]

```
#include <stdio.h>

/* Toto je uradnik, ktory rata obsah kruhu */
float uradnik(float r)
{
    float s;
    s = 3.1415926 * r * r;
    return( s );
}

/* Toto je sef. */
void sef(float p)
{
    printf("Obsah kruhu s polomerom %f je %f\n",p,uradnik(p));
    printf("Vypocital sef.\n");
}

/* Toto je hlavna funkcia. */
int main()
{
    sef(2.71);
    return(0);
}
```

Ako prvá je definovaná funkcia `uradnik`, ktorá vie vyrátať obsah kruhu s daným polomerom (podľa vzorca $S = \pi r^2$). Nadpis `float` pred jej menom znamená, že táto funkcia vyrobí nejaký výsledok a že tento výsledok bude typu `float`. Tesne za menom funkcie musia nasledovať zátvorky. V nich má funkcia `uradnik` napísané, že vyžaduje jeden vstupný parameter a že tento

18 Keď ale parametre nemajú, treba za ne dať prázdne zátvorky, aby bolo zrejmé, že sa jedná o funkciu. Tak, ako ste to zvyknutí robiť vo funkcii `main()`.

19 Tie texty medzi znakmi /* a */ sú komentáre. Kompilátor ich ignoruje a kvôli funkčnosti tam vôbec nemusia byť. Ale písať komentáre k programom je životná nutnosť. Pretože inak sa vo vašich programoch nebude nikto vyznať (po nejakom čase ani vy sami).

parameter musí byť typu `float`. Teda funkcia `uradnik` sa používa napríklad takto: `obsah = uradnik(5.5);`

Funkcia `uradnik` má jednu súkromnú premennú (ktorú žiadna iná funkcia nevidí) menom `s`. Úradník vyráta obsah a hodí ho do `s`. Príkaz `return` funkciu ukončí a vráti ako výsledok to, čo je v premennej `s`. Teda po vykonaní príkazu `obsah = uradnik(5.5);` sa do premennej `obsah` vloží hodnota `95.033176`.

Druhá funkcia – `sef` je typu `void`. Znamená to, že žiaden výsledok nevracia. (Keby sme ju potrebovali predčasne ukončiť, použili by sme `return();`) Vstupný parameter je zase typu `float`. Šéf pre zadaný vstup zavolá služby úradníka, vypíše výsledok a pochváli sa, že to on je ten múdry.

Funkcia `main` nemá žiaden vstup. Prekvapivo je však deklarovaná ako funkcia typu `int`. Doteraz sme do programov nepísali, akého je typu. Ak sa ale tento údaj vynechá, kompilátor automaticky predpokladá, že funkcia bude typu `int`. Podľa návratovej hodnoty funkcie `main` sa zvykne rozoznávať, či program dobehol v poriadku alebo s chybou. Nula znamená, že je všetko O.K. Ak program vráti nenulovú hodnotu, znamená to, že chceme dať vedieť, že sa niekde stala chyba.

Ak chcete použiť nejakú funkciu, máte dve možnosti. Buďto ju zadefinujete pred miestom použitia, alebo až potom. V druhom prípade ale nastáva problém. Keď kompilátor číta program, narazí na použitie funkcie, pričom netuší, čo to má byť za funkciu a vypíše chybu. Preto musíte povedať dopredu, ako tá funkcia vyzerá – musíte ju deklarovať aby kompilátor pri použití vedel, že je tá funkcia v poriadku. To znamená, že napíšete typ, meno a formát vstupných parametrov a za to všetko dáte bodkočiarku. V praxi to vyzerá napríklad takto (nezabudnite pridať `include`):

```
void strasidlo(int i);  /* tu je deklaracia */

main()
{
    strasidlo(5);
}

void strasidlo(int i)   /* tu je definicia */
{
    int j;
    for(j = 0; j < i; j++)
        printf("Bubu.\n");
}
```

Úloha č.1: Pochopte, napíšte a odladťe všetky tri uvedené príklady.

Úloha č.2: Napíšte funkciu, ktorá vyráta druhú mocninu daného čísla. (Druhá mocnina čísla 5 je $5 \times 5 = 25$.) S pomocou tejto funkcie vypíšte druhé mocniny čísel od 1 do 20.

Úloha č.3: Napíšte funkciu s jedným vstupným parametrom typu `int`, ktorá vypíše do riadku patričný počet hviezdičiek a prejde na nový riadok. S pomocou tejto funkcie potom nakreslite hviezdičkový trojuholník so stranou 5.

Úloha č.4: Napíšte funkciu `existuje()` typu int, ktorá vráti hodnotu `1`, ak existuje súbor `POKUS.TXT` a inak vráti `0`. (Pokúste sa otvoriť súbor na čítanie. Ak funkcia `fopen` vráti hodnotu `NULL`, tak súbor neexistuje, inak existuje. Ak súbor úspešne otvorite, nezabudnite ho vo funkcii potom aj zavrieť.)

12. lekcia

Smerníky
alebo "lovenie v pamäti"

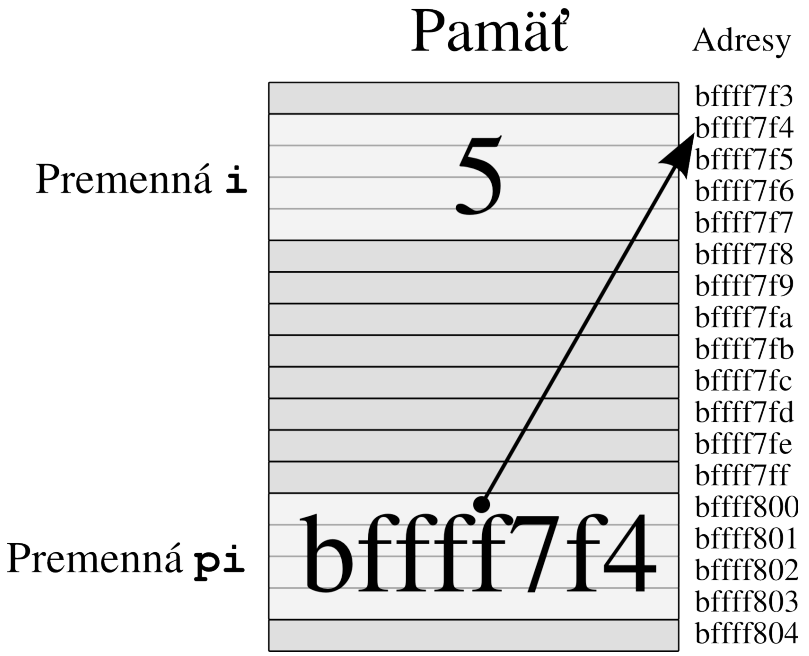
Smerníky sú „duša a srdce“ jazyka C. Iba ten, kto im rozumie a vie ich využívať, si môže dovoliť tvrdiť, že jazyk C ovláda.

Smerník je premenná, do ktorej sa neukladá číslo ale adresa v pamäti. Smerník je ako šípka, ktorá niekam do pamäte ukazuje. Okrem toho, kam smerník ukazuje, je väčšinou treba povedať aj to, aký typ premennej tam môžete očakávať. Takže sú smerníky, ktoré sú typu `int` pretože ukazujú na premenné typu `int`, sú smerníky typu `float`, typu `char` atď.

Ako to funguje? Pozrite si nasledujúci príklad. Nech funkcia `main()` vyzerá takto:

```
int i;
int *pi;
i = 5;
pi = &i;
```

Prvý a druhý riadok sa líšia len nepatrne – jedinou hviezdičkou. V prvom riadku je deklarovaná premenná `i` typu `int`. Pri deklarácii sa premennej vyhradí v pamäti miesto (veľké 4 bajty). Druhý riadok vygeneruje smerník. Tá hviezdička teraz neznamená „krát“ ale dáva kompilátoru vedieť, že ide o smerník. Do premennej `pi` (meno sme zvolili podľa anglického "pointer to integer" – smerník na celé číslo.[20]) môžeme uložiť adresu v pamäti, na ktorej sa nachádza nejaký `int`. Môžeme do neho uložiť napríklad adresu, na ktorej sa nachádza premenná `i`. To sa skutočne udeje v poslednom riadku. (Primpomeňme, že `&i` znamená „miesto v pamäti, kde je uložená premenná `i`“. A to miesto si teraz pamätáme v premennej `pi`.) Po vykonaní tejto sekvencie je situácia asi takáto:



20 Je dobrým zvykom dávať premenným, ktoré sú smerníkmi názvy začínajúce na p.

Na obrázku vidíte kus pamäti, jednotlivé bajty sú očíslované (v šestnástkovej sústave)[21]. Keď sme deklarovali premennú `i`, počítač nám pre ňu niekde vyhradil miesto – konkrétne štyri bajty na adrese `bffff7f4`.[22] Keď sme deklarovali premennú pi, počítač nám vyhradil ďalšie štyri bajty – tie, ktoré začínajú na adrese `bffff800`. Potom sme do premennej `i` vložili 5 a do premennej `pi` sme vložili adresu, na ktorej je premenná `i`. V premennej `pi` bude teda hodnota `bffff7f4`.

So smerníkmi sme sa už dvakrát stretli, aj keď sme ich vyslovene nespomínali. Prvýkrát, keď sme používali funkciu `scanf`, ktorá potrebuje poznať adresu v pamäti, kam má načítať hodnotu. Výraz `&i`, ktorý v takej situácii používame je vlastne smerník ukazujúci na premennú `i`. Druhýkrát sme sa so smerníkmi stretli, keď sme sa učili pracovať so súbormi. Zápis `FILE *f;` ktorý sme používali, deklaroval smerník na štruktúru typu `FILE`.

Ak poznáme premennú, adresu na ktorej sa nachádza vieme zistiť. Slúži na to znak `&` pred menom premennej. Čo ale robiť v prípade, keď je situácia opačná? Predstavte si, že z kusu programu uvedeného hore zavoláme nejakú funkciu, ktorej odovzdáme iba hodnotu `pi`. Môže si táto funkcia zistiť, akú hodnotu má premenná `i` aj keď o jej existencii vôbec nevie a pozná iba jej adresu? Odpoveď je áno. Slúži na to znak `*`. Ak by sme teda napísali `printf("%d\n", *pi);` na obrazovku by sa vypísalo `5`. Deklarácia `int *pi;` sa totiž dá čítať dvoma rôznymi spôsobmi: prvý je „`pi` je smerník na `int`“ a druhý je „`*pi` je `int`“. S výrazom `*pi` sa dá pracovať ako s obyčajnou premennou typu `int`. Môžem napríklad napísať `*pi = 7;` a do miesta v pamäti kam ukazuje smerník `pi` sa vloží číslo 7.

Nuž, rečí o smerníkoch bolo veľa ale zatiaľ sme ešte nepovedali, na čo sú také smerníky vlastne dobré. Na veľa vecí. Dobre sa s ich pomocou manipuluje s veľkými štruktúrami. Napríklad štruktúra `FILE` zaberá (v gcc pod linuxom) 148 bajtov. A je oveľa jednoduchšie pracovať so smerníkom na ňu (ktorý má 4 bajty) než s celým tým hebedom. S pomocou smerníkov sa dá priamo pristupovať do pamäte. O ďalších výhodách (napr. možnosť dynamického prideľovania pamäte) si povieme neskôr. Teraz si pozrite nasledujúci príklad. Máme tam dve funkcie `swap1` a `swap2`, ktorých úlohou je vymeniť hodnoty v dvoch premenných typu `int`.

```
#include<stdio.h>

void swap1(int a, int b)
{
    int c;

    c = a;
    a = b;
    b = c;
}


void swap2(int *pa, int *pb)
{
    int c;

    c = *pa;
    *pa = *pb;
    *pb = c;
}
```

21 Čísla v šestnástkovej sústave fungujú rovnako ako v desiatkovej, iba majú ešte nejaké cifry (konkrétne cifry `a`, `b`, `c`, `d`, `e` a `f`) navyše.

22 Miesto v pamäti prideľuje operačný systém. Môže sa stať, že vám nabudúce pridelí nejakú inú adresu.

```
main()
{
    int i = 3, j = 5;
    swap1(i,j);
    printf("i = %d, j = %d\n",i,j);
    swap2(&i,&j);
    printf("i = %d, j = %d\n",i,j);
}
```

Úloha č.1: Napíšte, skompilujte, pozrite, čo to robí.

Vyzerá to tak, že funkcia `swap1` hodnoty neprehodí, zatiaľ čo funkcia `swap2` áno. Prečo je to tak? Dôvod je jednoduchý: každá funkcia má svoje vlastné premenné s ktorými vie robiť a do súkromných premenných ostatných funkcií nevidí. Takže keď sa volá prvá funkcia, do jej súkromných premenných `a` a `b` sa zapíšu hodnoty z `i` a `j`, potom sa obsah premenných `a` a `b` medzi sebou vymení, ale premenné `i` a `j` to už nijako nezasiahne. Funkcii `swap2` však prezradíme, kde sa premenné `i` a `j` nachádzajú v pamäti. Adresa premennej `i` sa uloží do `pa` a adresa premennej `j` do `pb`. Funkcia `swap2` siahne do pamäte na určené miesta a hodnoty, ktoré tam nájde vymení.

Predošlý odstavec a program povyše si čítajte až dovtedy, kým to nepochopíte. Je to dôležité.

A na záver – ako vždy – úlohy:

Úloha č.2: Napíšte procedúru `dva`, ktorá dostane na vstupe smerník na `int` a na to miesto v pamäti, kam smerník ukazuje, vloží číslo 2. Napíšte hlavný program, v ktorom takúto funkciu použijete.

Úloha č.3 Napíšte procedúru `plus`, ktorá dostane na vstupe smerník na `int` a premennú, na ktorú smerník ukazuje, zväčší o 1. Použite v hlavnom programe.

Úloha č.4 Napíšte procedúru `sucet(int a, int b, int *vys)` ktorá do premennej na ktorú ukazuje `vys` vloží súčet `a + b`.

Dobrá rada: Nemusíte to celé zakaždým pchať do nového súboru. Tri funkcie a `main` sa v jednom súbore znesú (pokiaľ tie funkcie majú rôzne mená), zavolajte z `main`u všetky tri funkcie a ušetríte si robotu.

13. lekcia
Polia
alebo "čo kto zasadí, to zožne"

Vo všetkých prípadoch, ktoré sa doteraz vyskytli, sme pracne museli všetky premenné deklarovať jednu po druhej. Našťastie sme tých premenných zatiaľ nemuseli používať zvlášť veľa. Predstavte si ale, že by sme potrebovali deklarovať tisíc premenných typu `int`. Napísať tisíc deklarácií by bolo náročné na čas a vymyslieť tisíc názvov pre premenné náročné na prehľadnosť. A preto ľudia vymysleli polia.

Pole tisícich premenných typu `int` sa deklaruje ľahko. Stačí do deklarácií napísať `int a[1000];` a máme 1000 `int`-ov k dispozícii. Premenné máme očíslované od 0 do 999 (číslovanie polí v jazyku C začína vždy od 0). Keď chceme napríklad do premennej číslo 28 vložiť hodnotu 9, spravíme to príkazom `a[28] = 9;` (Samozrejme, pole sa vôbec nemusí nazývať `a`. Voľba názvu je na vás.)

V nasledujúcej ukážke môžete vidieť, ako sa také pole používa:

```
int a[20];
int i;

for(i = 0; i < 20; i++)
  a[i] = i % 2 == 0 ? i : 0;

for(i = 0; i < 20; i++)
  printf("%d ",a[i]);
printf("\n");
```

Vyrobili sme si pole dvadsiatich premenných typu `int`. Mohli by sme postupne do jednotlivých buniek niečo vkladať, ale keďže je pole dosť dlhé, použili sme cyklus. Riadiaca premenná cyklu určuje, do ktorého prvku poľa budeme niečo vkladať, preto sme veci zariadili tak, aby postupne nadobudla všetky hodnoty od 0 do 19. Do každej položky poľa sme vložili nejakú hodnotu (viete zistiť, čo ten použitý podmienený výraz `i % 2 == 0 ? i : 0` vlastne robí?) V ďalšom cykle sme hodnoty, ktoré sme do poľa napchali, vypísali do jedného riadku.

Použitie polí si ukážeme na trochu zložitejšom algoritme. Ide o známy algoritmus „Eratostenovo sito“, s pomocou ktorého nájdeme všetky prvočísla menšie ako tisíc. Bude fungovať takto: Spravíme si pole `int`-ov, ktoré bude mať 1000 prvkov. Ak bude na danom čísle uložená jednotka, znamená to, že číslo je prvočíslo. Ak tam bude nula, znamená to, že prvočíslo nie je. Tento stav dosiahneme nasledovne: Najprv celé pole naplníme jednotkami. Do premenných č. 0 a 1 vložíme nuly (0 ani 1 nie sú prvočísla). Potom prebehneme celé pole od začiatku do konca a keď niektoré číslo vyzerá byť prvočíslo (teda je na ňom nastavená jednotka), tak všetkým jeho násobkom nastavíme, že prvočísla nie sú – na patričné miesta vložíme nuly. Naprogramované to vyzerá takto (nezabudnite pridať štandardnú omáčičku):

```
int pole[1000];
int i, k;

for( i = 0; i < 1000; i++)
    pole[i] = 1;

pole[0] = 0;
pole[1] = 0;

for( i = 0; i < 1000; i++)
{
    /* ak je to prvocislo, vypisat a nasobky vynulovat */
    if (pole[i] == 1)
    {
        printf("%d\n",i);
        /* najmensi nasobok i-cka je dvojnasobok, preto od
           neho zacneme nulovat. Budeme pridavat i a nulovat
           dalsie nasobky kym je index mensi ako 1000 */
        for( k = 2*i; k < 1000; k = k + i )
            pole[k] = 0;
    }
}
```

Úloha č.1: Pochopte, napíšte, skompilujte. Zvlášť dobre si pozrite ten cyklus, ktorý je riadený premennou `k` a ktorý nuluje násobky nájdeného prvočísla. (To prvočíslo je `i`. Vieme to z toho, že `pole[i]` je jednotka a teda žiaden jeho predchodca prvočíselnosť `i`-čka nespochybnil.)

Polia napodiv veľmi úzko súvisia so smerníkmi. Keď programu poviem, že budem používať pole `pole[1000]`, tak sa mi v pamäti nevytvorí miesto pre toto pole len tak náhodne. Všetky premenné sú v pamäti uložené pekne za sebou. A premenná `pole` (bez hranatých zátvoriek) je smerník, ktorý ukazuje, kde táto oblasť začína. Takže ak spravím niečo takéto:

```
int a[10];
*a = 5;
printf("%d\n",a[0]);
```

program mi vypíše `5`. Výraz `*a` znamená presne to isté ako `a[0]`. Aj k ďalším prvkom poľa môžeme pristupovať takýmto smerníkovým spôsobom. `a[1]` je to isté ako `*(a+1)` teda „vec, ktorá sa nachádza v pamäti na adrese `a+1`“.[23]

To, že prístup k poliam sa v podstate deje cez smerníky, skrýva v sebe výhody aj nevýhody. Medzi výhody patrí, že smerníky môžu polia v istých situáciach nahrádzať, ako si to môžete všimnúť na nasledujúcom príklade:

```
int a[10], b[10], *c;

a[2] = 2; b[2] = 3;
c = a;
printf("%d\n",c[2]);
c = b;
printf("%d\n",c[2]);
```

Smerník `c` tu raz ukazuje na začiatok poľa `a`, raz na začiatok poľa `b`, takže `c[2]` bude zakaždým niečo iné (raz to bude tretí prvok z `a`, raz tretí prvok z `b`).

23 Ono je to vymyslené šikovne. Keď sa vyhodnocuje `*(a+3)` a `a` je smerník na `int`, k adrese `a` sa nepripočíta 3, ale 12 – teda „trikrát veľkosť `int`“. Ak by to bolo pole `char`-ov, pripočítala by sa trikrát veľkosť `char`.

Medzi nevýhody patrí napríklad to, že C-čko vám nekontroluje dĺžku poľa. Aj keď má pole `c` iba 10 prvkov, môžete použiť výraz `c[13]`. Program sa skrátka pozrie v pamäti o 13 ďalej, než ukazuje smerník `c` a niečo prečíta/zapíše. Pri troche šťastia vám program spadne (pod linuxom s hláškou „Segmentation fault“ alebo „Chyba segmentácie“) a vy budete vedieť, že v pamäti siahate niekam, kam nemáte. Ničmenej oveľa častejšie budete iba žasnúť, prečo sa vám mení nejaká iná premenná a celé to blbne.

Úloha č.2: Napíšte program, v ktorom si urobíte desaťprvkové pole, v cykle do neho vložíte čísla od 1 do 10 a v ďalšom cykle obsah poľa vypíšete.

Úloha č.3: Napíšte procedúru, ktorá dostane na vstupe smerník na desaťprvkové pole typu `int` a ako hodnotu vráti najväčší prvok toho poľa. Vyskúšajte, či funguje s poľom z úlohy 2.

Úloha č.4: Naplňte dvadsaťprvkové pole prvkami Fibonacciho postupnosti `1,1,2,3,5,8,...` (To je tá, kde sa ďalší člen rovná súčtu predošlých dvoch.) Použite pri tom smerníkový zápis – hranaté zátvorky smiete použiť len v deklarácii poľa.

14. lekcia
Dynamická alokácia alebo "Mega pamäte, prosím"

Polia sú veľmi užitočný nástroj. V tej forme, v akej ich zatiaľ poznáme ale majú jednu drobnú slabinu – dopredu treba vedieť, aké budú veľké. Keď určujeme ich veľkosť, nemôžeme používať premenné, musí tam byť nejaké dopredu známe číslo. Všetkým premenným a poliam, ktoré sú klasicky deklarované, sa vytvorí miesto v pamäti hneď na začiatku. A potom už žiadna premenná nepribudne a žiadne pole sa nemá šancu zväčšiť či zmenšiť. Pamäť je ale dosť veľká a niekedy až počas behu programu zistíme, že by sme z nej ešte kúsok potrebovali.

Samozrejme – dá sa to. Ale treba dávať na nejaké veci pozor. Keď si necháme dočasne nejakú pamäť prideliť, použijeme ju a už pre nás nie je potrebná, musíme ju zas uvoľniť. Keby sme to totiž nespravili, pamäť by nám ostala pridelená až dovtedy, kým by program nedobehol. A ostatné programy, ktoré v systéme bežia by ju nemohli použiť.

Takže ako sa to robí. Ak chceme používať funkcie, ktoré pracujú s pamäťou, musíme najprv načítať jeden nový hlavičkový súbor – `stdlib.h`. Spravíme to rovnako ako s jeho kolegom: `#include <stdlib.h>`. (Keď okrem toho ešte chcete písať na terminál, musíte samozrejme pridať aj `#include <stdio.h>`.) Teraz môžete používať funkciu `malloc` (z anglického memory allocation – alokácia pamäte). Funkcii zadáte parameter, koľko bajtov pamäte chcete a ona vám vráti smerník na začiatok bloku pamäte (alebo vráti nulový smerník `NULL`, ak pamäť nie je k dispozícii). Takže ak by sme zrazu potrebovali pole 100 celých čísel, zariadime si to takto:

```
int *smernik;
smernik = (int *) malloc(100 * sizeof(int));
smernik[23] = 145;
```

V tom, čo sme tu predviedli sú dve podivné veci. Prvá je to `(int *)` pred funkciou `malloc`. Na čo to tam je? Totiž funkcia `malloc` nevráti smerník na `int`, ale „smerník všeobecný“ teda smerník na `void`. No a kompilátor si pamätá, ktorý smerník na čo ukazuje a keby sme chceli tento všeobecný smerník priradiť do premennej `smernik`, kompilátor by nám vynadal, že to nemôžeme priradiť, lebo ten druhý smerník je iného typu. Typ smerníku sa ale dá meniť. Na to slúži vec zvaná **pretypovanie**, ktorú vidíte vyššie. Skrátka sa pred smerník napíše jeho nový typ a kompilátor potom to priradenie zožerie.

Ďalšia novinka je to `sizeof(int)`. Ako sme už povedali, funkcia `malloc` nám vyhradí toľko bajtov, koľko si zažiadame. Keby sme napísali iba `malloc(100)`, dostali by sme 100 bajtov. Problém je v tom, že jeden `int` zaberie viac než jeden bajt (v závislosti od kompilátora to bývajú väčšinou 2 alebo 4 bajty). A na to, aby sme zistili koľko je to vo vašom systéme, slúži direktíva `sizeof`, ktorá vie zistiť, koľko bajtov nejaký dátový typ zaberá.

Akonáhle vieme, že už takto pridelenú pamäť nebudeme používať, musíme ju uvoľniť. Spravíme to takto:

```
free((void *) smernik);
```

Funkcia `free` musí na vstupe dostať smerník na začiatok pridelenej pamäte (z toho vyplýva, že by sme `smernik` počas používania nemali ničím prepísať, lebo by sme už nemali odkiaľ zistiť,

ktorú časť pamäte uvoľňujeme). Funkcia `free` chce na vstupe smerník typu `void` a tak sme ten náš museli pretypovať.

Nasledujúci program zistí koľko megabajtov pamäte je nám systém ochotný dať k dispozícii. Spraví to tak, že si bude alokovať nejakú pamäť dovtedy, kým mu ju operačný systém bude ochotný poskytnúť a bude si počítať, koľkokrát sa mu to podarilo. (Pamäť v ňom hanebne neuvoľňujeme a začiatky blokov si nepamätáme – tentokrát to zveríme operačnému systému.)

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

main()
{
    int i = 0;

    while(malloc(1000000) != NULL)
        i++;

    printf("K dispozicii je %d MB pamate\n",i);
}
```

Takže úlohy:

Úloha č.1: Pochopte, napíšte a skompilujte predchádzajúcu ukážku.

Úloha č.2: Napíšte program, ktorý vypíše, koľko bajtov pri použití vášho kompilátoru zaberajú premenné typu `int`, `float`, `char`, `(int *)`, `(float *)` a `(char *)`.

Úloha č.3: Do premennej `velkost` typu `int` načítajte hodnotu z terminálu, nechajte si prideliť pole `int`ov danej veľkosti (nezabudnite si dopredu vytvoriť smerník tak, ako je to v prvej ukážke), naplňte ho na striedačku nulami a jedničkami, vypíšte ho a uvoľnite pamäť.

Úloha č.4: Napíšte program, ktorý alokuje pole `int`ov veľkosti 100, spomedzi čísel, ktoré v ňom budú (nejaké náhodné bláboly z pamäte) nájde a vypíše najmenšie číslo a pamäť znova uvoľní. (Variant pre guruov: program nájde najmenšie kladné číslo. Ak žiadne kladné nie je, napíše to.)

15. lekcia
Reťazce
alebo "ukecané polia"

Čo je to pole už vieme. Naučili sme sa, ako si vyrobiť statické pole, ktorého dĺžku pozná program od začiatku aj ako si nechať prideliť kus pamäte s pomocou funkcie `malloc`. Poďme sa teraz podrobnejšie pozrieť na jeden konkrétny typ polí. Ide o pole premenných typu `char`. Do takéhoto poľa možno uložiť veľa písmeniek. A veľa písmeniek môže utvoriť slovo alebo dokonca aj vetu. A polia, do ktorých môžeme uložiť nejaký text sa nazývajú reťazce.

Prvý problém je, ako nejaký text do poľa dostať. Jedna z možností je dať ho tam v momente, keď pole deklarujeme. Môže to vyzerať takto:

```
char pole[10] = "Ahoj";
```

Tento príkaz vyrobí pole desiatich `char`ov. Inicializácia však na počudovanie nezmení prvé štyri z nich, ale prvých päť. Podľa očakávania prebehne niečo ako `pole[0]='A'; pole[1]='h'; pole[2]='o'; pole[3]='j';` Okrem toho sa ale na koniec zapíše špeciálny ukončovací znak `'\0'`, takže sa ešte uskutoční niečo ako `pole[4] = '\0';`. Nulový znak signalizuje, kde reťazec končí. Vďaka nemu funkcie, ktoré s reťazcami pracujú vedia, že ostatné prvky poľa už do reťazca nepatria.

Ďalšia možnosť, ako dostať text do poľa je takáto:

```
char pole[] = "Ahoj";
```

V tomto prípade sme nechali na kompilátore, aby sám určil správnu dĺžku poľa. Dĺžka bude 5 a už sa až do konca behu programu nezmení. (Prečo dĺžka poľa nebude 4?)

Zatiaľ to vyzeralo všetko jednoducho a príjemne. Teraz prídu problémy. Čo robiť, ak máme už deklarované pole a potrebujeme ho naplniť až neskôr? Ak sa pokúsime o niečo takéto:

```
char pole[10];
pole = "Ahoj";
```

kompilátor nám vynadá, pretože premenná `pole` je smerník, ktorý ukazuje nemeniteľne na začiatok poľa a my sa ju pokúšame presvedčiť, aby ukazovala niekde inde. Máme dve možnosti. Buď tam to `"Ahoj"` napchať po písmenkách, alebo použiť funkciu. Prvá možnosť je únavná a zdĺhavá (zvlášť pri dlhších reťazcoch) a preto odporúčame druhú. Keď chceme používať funkcie na prácu s reťazcami, potrebujeme `include`-ovať ďalší súbor – `string.h`. Na kopírovanie jedného reťazca do iného slúži funkcia `strncpy`. Použije sa nasledujúcim spôsobom:

```
strncpy(pole,"Ahoj",9);
pole[9] = '\0';
```

Funkcia má tri parametre. Prvý je smerník do pamäte, kam sa má reťazec ukladať. Druhý parameter je reťazec, ktorý sa má skopírovať. Tretí parameter je maximálny počet znakov, ktoré sa skopírujú. V uvedenom prípade, kedy poznáme dĺžku kopírovaného reťazca sa zdá byť

zbytočný. Ak by sme ale kopírovali reťazec uložený v nejakom inom poli, celkom ľahko by sa mohlo prihodiť, že by bol pridlhý a do nášho poľa by sa nezmestil. A tak by sa prepísalo miesto v pamäti za poľom. Dĺžku sme obmedzili na 9 znakov aby nám ostalo ešte jedno miesto na ukončovací znak, ktorý sa v prípade pretečenia nepridáva. (Mimochodom – jedna známa metóda na prienik do operačných systémov je založená práve na využití programátorskej chyby, že sa nekontroluje pretečenie dĺžky reťazca.)

Na výpis reťazca slúži stará dobrá funkcia `printf`. Riadiaci znak je ale nový. Ak chcem vypisovať reťazec, urobím to takto:

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

Ak chcem reťazec načítať z klávesnice, použijem funkciu `fgets`. Spravím to takto:

```
fgets(pole,9,stdin);
```

Funkcia `fgets` prečíta jeden riadok zo súboru určeného tretím parametrom (ak ako súbor zadám `stdin`, bude čítať zo štandardného vstupu) a vloží ho do reťazca určeného prvým parametrom. Číslo v strede mi opäť hovorí, koľko maximálne znakov sa môže prečítať.

Pre vaše potešenie je tu ešte niekoľko funkcií, ktoré pracujú s reťazcami: Funkcia `strlen` zistí dĺžku reťazca. Teda `strlen("Ahoj")` bude 4. Funkcia `strcmp` vie porovnať dva reťazce podľa abecedy. Ak je prvý menší výsledok je záporné číslo, ak je väčší, kladné a ak sú reťazce rovnaké, výsledok je 0. Teda

- `strcmp("Adam","Eva")` je -4, (nie je dôležité, že práve -4, hlavné je, že je to záporné)
- `strcmp("Eva","Adam")` je `4` a
- `strcmp("Adam","Adam")` je 0.

Funkcia `strncat(pole,"te",4)` skopíruje na koniec reťazca `pole` maximálne 4 znaky z druhého reťazca. Ak tam teda predtým bolo `"Ahoj"`, bude tam teraz `"Ahojte"`. [24]

Funkcií pracujúcich s reťazcami je ešte viac, ale zatiaľ vystačíte s týmito. Záujemcovia nájdu ďalšie detaily na internete.

Úloha č.1: Deklarujte si reťazec dlhý 20 znakov, načítajte doňho niečo z klávesnice a vypíšte, akú to má dĺžku.

Úloha č.2: Načítajte dva reťazce z klávesnice a vypíšte ten, ktorý je podľa abecedy skôr.

Úloha č.3: Načítajte z klávesnice reťazec a zistite, koľkokrát sa v ňom vyskytuje písmeno `'a'`.

Úloha č.4: Deklarujte si jeden reťazec dĺžky 40 a dva dĺžky 20. Do dvoch menších reťazcov načítajte hodnoty z klávesnice, do veľkého vložte reťazec, ktorý vznikne spojením tých malých a výsledok vypíšte. (Pozor! Budete musieť použiť funkciu `strncpy` aj funkciu `strncat`.)

24 V uvedenom príklade sa opäť zdá byť zbytočné uvádzať maximálny počet kopírovaných znakov. Ale ak by sme obsah kopírovaného reťazca nepoznali, mohlo by opäť dôjsť k pretečeniu.

16. lekcia
Typy a štruktúry alebo "urob si svoj typ"

Doteraz sme sa stretli s niekoľkými dátovými typmi. Boli to typy `int`, `float` a `char`, smerníky na ne a polia. Existuje ešte niekoľko ďalších základných typov a o niektorých z nich si možno v budúcnosti niečo povieme. Dnes sa však budeme zaoberať tým, ako si vytvoriť typy nové.

Nový typ sa definuje pomocou slova `typedef`. Robí sa tak obvykle mimo procedúr, najlepšie v hlavičkovom súbore, ale to ešte neviete čo je. Predstavte si, že chcem definovať typ `retazec` ako pole 100 charov. Spraví sa to nasledovne:

```
typedef char retazec[100];
```

Teraz už môžem typ `retazec` veselo používať v mojom programe:

```
main()
{
    retazec a,b;

    strncpy(a, "Ja som strasna ropucha.\n", 99);
    strncpy(b, "A ja som ihlicnaty lekvar.\n", 99);
    printf("%s%s", a , b );
}
```

Za slovom `typedef` nasleduje deklarácia rovnaká ako deklarácia premennej (v tomto prípade poľa `char`ov dĺžky 100), akurát že použité meno `retazec` nie je meno premennej, ale meno nového typu. V procedúre `main` sme si vyrobili dve premenné typu `retazec`. Obe sú plnohodnotné polia typu `char` a mohli by sme kľudne spraviť napríklad priradenie `b[25] = 'k';` v našej ukážke však polia používame ako reťazce.

Takže nový typ by sme si už urobiť vedeli. Zatiaľ nám to ale môže slúžiť len na to, že si niektoré známe typy premenujeme. Predstavte si ale, že by sme potrebovali spraviť úplne nový typ. Napríklad premennú, do ktorej by sme vedeli uložiť meno, priezvisko, vek, váhu a výšku človeka. Takéto premenné sa vytvárajú pomocou príkazu `struct`. Vyzerá to napríklad takto:

```
struct clovek
{
    char meno[20];
    char priezvisko[40];
    int vek;
    float vaha;
    float vyska;
} jano;


main()
{
    strncpy(jano.meno,"Fero",19);
    strncpy(jano.priezvisko,"Horny",39);
    jano.vek = 13;
    jano.vaha = 51.7;
    jano.vyska = 162.5;
```

```
        printf("%s %s Vek:%d Vaha:%f Vyska:%f\n",
            jano.meno, jano.priezvisko, jano.vek,
            jano.vaha, jano.vyska);
    }
```

Ako ste si mohli všimnúť, k jednotlivým zložkám takejto zloženej premennej pristupujeme s pomocou bodky. Ak sa chceme dostať k položke `vaha` premennej `jano`, napíšeme `jano.vaha`.

Deklarácia štruktúrovanej premennej sa dá pekne spojiť s príkazom `typedef`. Môžete si napríklad zadefinovať typ `bod`, ktorý bude obsahovať súradnice bodu v rovine nasledujúcim spôsobom:

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

a potom si urobiť pole bodov takto:

```
main()
{
    BOD bodiky[3] = {{1 , 1},
                     {3 , 3},
                     {7 , 5}};

    bodiky[0].x = 30;
    bodiky[0].y = 20;
    printf("Bod 0 ma suradnice [%d,%d]\n",
        bodiky[0].x, bodiky[0].y);
}
```

K jednotlivým položkám štruktúry sa opäť pristupuje s pomocou bodky. Celé pole môžem naplniť (ale len pri deklarácii) spôsobom, ktorý je vidieť vyššie.

V prípade, že poznáme smerník na nejakú štruktúru, môžeme k jednotlivým položkám pristupovať cez tento smerník. V takom prípade sa nepoužíva bodka ale šípka `->` zložená zo znakov „mínus“ a „väčšie“. Keďže premenná `bodiky` je smerník, ktorý ukazuje na prvú položku poľa, môžeme predošlý program napísať trochu iným spôsobom:

```
main()
{
    BOD bodiky[3] = {{1 , 1},
                     {3 , 3},
                     {7 , 5}};

    bodiky -> x = 30;
    bodiky -> y = 20;
    printf("Bod 0 ma suradnice [%d,%d]\n",
        bodiky[0].x, bodiky -> y);
}
```

Keďže sa pri volaní funkcií častejšie ako parameter používa smerník na štruktúru, než celá štruktúra, je tento druhý spôsob používaný relatívne často.

Úloha č.1: Vyskúšajte oba varianty (aj prístup cez bodky, aj prístup cez šipky) príkladu s poľom bodov. Ako by ste cez šípky pristupovali k druhému prvku poľa?

Úloha č.2: Vyrobte si dátový typ `AUTO`, ktorý bude obsahovať typ, poznávaciu značku a počet najazdených kilometrov. Vytvorte pole troch áut, naplňte ho hneď pri deklarácii a vypíšte údaje o druhom z nich.

Úloha č.3: Vytvorte štruktúru `struct pokus`, ktorá v sebe bude obsahovať jedno celé číslo a jednu položku typu `struct pokus`. Čo vám na to povie kompilátor? Čo vám povie, ak bude štruktúra `struct pokus` v sebe obsahovať jedno celé číslo a jednu položku typu „smerník na `struct pokus`“?

Úloha č.4: (Pre guruov.) Vyrobte si typ `OVOCIE`, ktorý bude obsahovať názov a cenu za kilo. Vytvorte si päťprvkové pole typu `OVOCIE`, napíšte program, ktorý z klávesnice doňho načíta údaje a potom cenník vypíše na obrazovku. (Varovanie, ktoré oceníte až keď to vyskúšate naprogramovať: Keď načítate zo štandardného vstupu číslo s pomocou `scanf`, prechod na nový riadok (znak `\n`), ktorý po tom čísle nasleduje, ešte nikto nenačítal. Ak sa ho nezbavíte, nasledujúci reťazec, ktorý prečítate, bude prázdny.)

17. lekcia
Komplet úplné opakovanie alebo "máme céčko v malíčku"

Takže to by sme mali. Či veríte alebo nie, základy jazyka C máte za sebou. Je síce niekoľko detailov o ktorých zatiaľ reč nebola, ale tie spomenieme, keď ich budeme potrebovať alebo si ich naštudujete v knižke, keď ich budete potrebovať vy. (Nedá mi znova nespomenúť skvelú knižku od pána Pavla Herouta „Učebnice jazyka C“.) Vzhľadom k tomu, že sa skutočne jedná o základy, nasledujúce úlohy majú preveriť, nakoľko sú pevné.

Úloha č.1: Napíšte program, ktorý zo súboru `cislo.txt` prečíta reálne číslo (typ `float`) a do súboru `dvojnas.txt` zapíše jeho dvojnásobok.

Úloha č.2: Napíšte program, ktorý skopíruje súbor `vzor.txt` do súboru `kopia.txt` a pri tom všetky malé písmená zmení na veľké.

Úloha č.3: Napíšte program, ktorý bude obsahovať funkciu `void hviezdicky(int a)`. Funkcia má vypísať `a` hviezdičiek vedľa seba a potom prejsť na nový riadok. Túto funkciu sedemkrát zavolajte z funkcie `main` s parametrami 1,3,5,7,5,3 a 1.

Úloha č.4: Napíšte funkciu, ktorá vypíše riadok `Srdecny pozdrav z funkcie.` Zavolajte túto funkciu z hlavného programu päťtisíckrát.

Úloha č.5: Napíšte funkciu `sucin(int a, int b, int *vys)` ktorá do premennej na ktorú ukazuje `vys` vloží súčin `a × b`.

Úloha č.6: Vytvorte desaťprvkové pole premenných typu `int` a naplňte ho prvými desiatimi nepárnymi číslami.

Úloha č.7: Napíšte funkciu, ktorá ako parameter dostane smerník na prvý prvok desaťprvkového poľa premenných typu `int` a vypíše hodnoty všetkých premenných, ktoré sú deliteľné tromi. Vyskúšajte jej funkčnosť na poli vyrobenom v predošlom príklade.

Úloha č.8: Napíšte program, ktorý s pomocou funkcie `scanf` načíta z klávesnice celé číslo, dynamicky alokuje pole `int`ov určenej dĺžky a naplní ho prirodzenými číslami 1,2,3,...

Úloha č.9: Napíšte program, ktorý sa človeka opýta ako sa volá, načíta jeho meno z klávesnice a potom ho pozdraví. Ak na vstupe dostane napr. `Cyprian`, napíše `Ahoj Cyprian!!!`

Úloha č.10: Vyrobte si štruktúru `pbod` a definujte typ `PBOD`, ktorý bude reprezentovať bod v priestore. (Štruktúra sa teda bude skladať z troch `int`ov `x`, `y` a `z`.) Deklarujte štvorprvkové pole `PBOD`ov, inicializujte ho pri deklarácii a vypíšte jeho druhý prvok po zložkách na terminál.

Úloha č.11: Napíšte program, ktorý zistí, či je zadané číslo prvočíslo.

Úloha č.12: Napíšte program, ktorý vypíše čísla od 1 do 100 ale miesto každého čísla deliteľného tromi a každého čísla, v ktorého zápise sa nachádza trojka vypíše hviezdičku. (Rozhodnúť o tom, či bude vypísané číslo alebo hviezdička musí program. Riešenia, ktoré na terminál iba vychrlia reťazce dopredu vyrobené programátorom nebudú uznané.)

Úloha č.13: Napíšte program, ktorý vypíše všetky možnosti, ktorými sa môžu z deviatich čísel vyžrebovať štyri rôzne.