Úvod do C++

Ako správne používať tento študijný text: Milý čitateľ, chystáš sa prečítať si kuchárku o základoch programovacieho jazyka C++. Najdôležitejšie pri učení sa nového programovacieho jazyka je poriadne si ho precvičiť. Preto počas čítania narazíš na niekoľko úloh, ktoré ti odporúčame vyriešiť a naprogramovať. Má to dopomôcť k tomu, aby si si všetko lepšie zapamätal a dostal do krvi.

Tento text je napísaný aj pre ľudí, ktorí nikdy program nevideli. Pokiaľ už máš nejaké základy z programovania, môžeš pokojne preskakovať tie časti, ktoré ovládaš.

Obsah:

Program

O čom tu celý čas rozprávame? Čo je to program? Čo je to algoritmus? Čo je to programovací jazyk? Ak si myslíš, že tieto základy už vieš, môžeš túto sekciu preskočiť, inak si ju radšej prečítaj.

Algoritmus je postupnosť niekoľkých dobre definovaných inštrukcií – úkonov, ktorá slúži na vykonanie nejakej úlohy. Napríklad algoritmus na uvarenie čaju môže byť nasledovný:

`Zoberieme pohár, vrecúško čaju a rýchlovarnú kanvicu. Naplníme kanvicu vodou, zapojíme do zásuvky a zapneme. Kým voda nevrie, čakáme. Do pohára vložíme vrecúško čaju a zalejeme vriacou vodou. Kanvicu vypneme a vypojíme zo zásuvky. Počkáme 8 minút a potom vyberieme vrecúško.`

Super, vieme uvariť čaj (teda ak máme kanvicu atď..), ale v tejto chvíli nás zaujímajú počítače. A počítač nevie naplniť kanvicu vodou, ani nevie, čo je kanvica alebo čo je voda. Predošlý algoritmus je ozajstným *algoritmom* len vtedy, ak vieme, čo znamenajú jednotlivé úkony (napríklad čo znamená zobrať pohár).

Keď budeme vytvárať algoritmy pre počítače, tak budeme používať celkom iné inštrukcie – také, ktorým počítač rozumie a vie ich aj vykonávať.

Medzi inštrukcie, ktoré počítač vie vykonávať patria napríklad jednoduché aritmetické operácie (sčítanie, delenie...), čítanie z pamäte, ukladanie do pamäte, ale aj mnoho ďalších.

Avšak aby im dobre rozumel, musia byť zapísané ako hromada núl a jednotiek. A tým zase veľmi nerozumieme my ľudia. Preto existujú **Programovacie jazyky**. Tie sú (väčšinou) dobre čitateľné ľuďmi a navyše si ich vie počítač prepísať do svojich jednotiek a núl.

My sa budeme zaoberať programovacím jazykom C++, ktorý patrí medzi vyššie programovacie jazyky – to znamená, že sa lepšie číta ľuďmi a zvláda toho oveľa viac ako nižšie.

Program je skupina inštrukcií v nejakom programovacom jazyku. A tieto programy obvykle niečo robia, inak by nám boli nanič.

V našom prípade budeme pracovať s programami, ktoré načítajú nejaký vstup a vypíšu nejaký výstup. Napríklad sčítací program môže načítať dve čísla a vypísať ich súčet.

Napríklad toto:

```
int main(){
}
```

je program v jazyku C++, ktorý nerobí vôbec nič. Ale dá sa skompilovať.

Kompilácia - Linux

Kompilácia je prepis programu napísaného v programovacom jazyku do strojového kódu (jednotiek a núl), ktorý dokáže počítač spustiť.

Teraz sa ideme naučiť skompilovať program na Linuxe, vyskúšajte si to:

- Uistíme sa, že máme nainštalovaný kompilátor (napríklad napíšeme `g++ -v` malo by nám to vypísať okrem balastu aj verziu kompilátora)
- Napíšeme program a uložíme ho do súboru `hocico.cpp`
- Otvoríme konzolu a vojdeme do priečinka, kde sa nachádza súbor `hocico.cpp`
- Skompilujeme jedným z nasledovných príkazov:
  - `g++ hocico.cpp` skompiluje program a výsledok uloží do `a.out`
  - `g++ hocico.cpp -o hocico` skompiluje program a výsledok uloží do `hocico`
  - `make hocico` za bežných okolností zavolá `g++ hocico.cpp -o hocico`, ale dá sa nastaviť aj iné správanie
- Ak kompilátor vypísal chybu, skúsime ju opraviť a skompilujeme program znova (nezabudnúť uložiť súbor)
- Spustíme skompilovaný program buď `./a.out` alebo `./hocico`, podľa toho kam sa nám uložil.

Kompilácia - Windows

Kompilácia je prepis programu napísaného v programovacom jazyku do strojového kódu (jednotek a núl), ktorý dokáže počítač spustiť.

Kompilácia vo Windows závisí od prostredia, ktoré používate. Väčšinou stačí stlačit nejakú klávesovú skratku a program sa skompiluje.

Náš prvý program

Napíšme si náš prvý program a vysvetlíme si, čo robí:

```
#include<cstdio>

int main(){
    printf("jesko\n");
}
```

Samotný program v C++ píšeme medzi `int main(){` a `}`. To, čo je tam napísané, sa začne vykonávať pri spustení skompilovaného kódu. `main()` je v skutočnosti funkcia a medzi kučeravými zátvorkami `{` a `}` je jej telo. O funkciách si povieme viac neskôr, teraz nám stačí vedieť, že funkcia je v podstate pomenovaný program (postupnosť príkazov).

V tele funkcie `main()` z príkladu je len jeden príkaz, konkrétne volanie funkcie `printf("jesko\n")`. Za každým príkazom musí nasledovať bodkočiarka.

Do zátvoriek funkcií zasa píšeme *argumenty*, ktoré upresňujú čo má funkcia robiť. Napríklad funkcia `printf` vypisuje text na výstup a do zátvoriek jej píšeme, čo má vypisovať.

Telo funkcie `printf` nevidíme, lebo je napísané v tvz. knižnici `cstdio`. *Knižnica* je súbor, v ktorom sú definované funkcie ale aj všeliaké iné veci. Takéto knižnice nám veľmi zjedodušujú prácu, pretože mnohé veci si potom nemusíme programovať sami.

Ak chceme veci z knižnice používať, treba ju `include`nuť. Na to slúži riadok `#include<cstdio>`, ktorý vidíme v príklade.

`cstdio` (c++ standard input output) je teda názov knižnice, v ktorej je definovaných mnoho užitočných funkcií súvisiacich s načítavaním vstupu a vypisovaním výstupu.

Ako sme si už spomínali, funkcia `printf`, ktorá je v knižnici `cstdio`, slúži na vypisovanie výstupu. A pokiaľ jej do zátvorky vložíme nejaký reťazec znakov v úvodzovkách, tak ich vypíše na obrazovku. V našom príklade má reťazec 6 znakov `'j'`, `'e'`, `'s'`, `'k'`, `'o'` a `'\n'`. Prvých 5 sú obyčajné písmená, posledný je znak konca riadku. (Niektoré znaky (napríklad koniec riadku, alebo `"`) nemôžeme do reťazca napísať priamo, takže ich píšeme pomocou znaku `'\'`. Viac nájdete napríklad na `http://en.cppreference.com/w/cpp/language/escape`)

Cvičenie 1: Skompilujte si vyššie uvedený program. A spustite ho. Čo sa stalo? Čo sa stane, keď z programu vynecháte znak `'\'`? Prečo? Ako by ste vypísali dva riadky so slovom `jesko`?

Cvičenie 2: Čo všetko môžeme z programu vynechať, aby stále robil to isté?

Cvičenie 3: Skúste na obrazovku vypísať reťazec `"jesko"` aj s úvodzovkami.

Riešenie 1: Program vypíše reťazec `jesko`. Keď vynecháme znak `'\'` program vypíše `jeskon` a riadok neukončí. Dvakrát vypísať reťazec `jesko` môžeme dvoma spôsobmi: buď `printf("jesko\njesko\n");` alebo `printf("jesko\n"); printf("jesko\n");`.

Riešenie 2: Niektoré prebytočné medzery a prázdny riadok. Z nasledovného programu už nie je možné nič vynechať bez straty funkčnosti.

```
#include<cstdio>
int main(){printf("jesko\n");}
```

Riešenie 3: Použijeme príkaz `printf("\"jesko\"\n");`

Premenné

Bez toho, aby sme si mohli údaje zapamätávať sa programovať nedá. Na zapamätanie nejakých hodnôt nám slúžia premenné. **Premennú** si môžeme predstaviť ako krabičku, do ktorej vieme všeličo strčiť. Krabičky majú svoje meno a aj svoj typ.

Premennú s menom si môžeme predstaviť ako krabičku s nálepkou. *Meno* krabičky/premennej slúži na to, aby sme s nimi vedeli pracovať. Napríklad ulož číslo do tamtej premennej"počítač nepochopí, on by rád počul ulož číslo do premennej s menom `a`".

Typ premennej zase určuje, čo do nej môžeme vkladať. Do premennej typu `int` (odvodené od anglického slova integer) môžeme ukladať celé číslo, do premennej typu `string` (z angličtiny sa dá preložiť ako reťazec) zasa môžeme vkladať reťazce znakov (na `string`y potrebujeme knižnicu). Neskôr sa naučíme používať oveľa viac typov premenných, dokonca vyrábať vlastné typy.

Ako premenné používať? Na to, aby sme mohli s premennou niečo robiť, si ju musíme vytvoriť (deklarovať). To sa robí príkazom `<typ-premennej> <meno-premennej>;` napríklad `int cislo;`

Do premenných môžeme vkladať priradením, teda pomocou `=`. Syntax je `<meno-premennej> = <výraz>`, napríklad `a = 4`. O výrazoch si povieme viac neskôr, dôležité zatiaľ je, že výraz musí mať nejakú hodnotu. Napríklad výraz `4` má hodnotu 4. Výraz `a` má hodnotu toho, čo je vnútri premennej/krabičky s menom `a`. Výraz `4 + 7` má hodnotu 11. Pokiaľ v `a` je číslo 5, tak výraz `a + 5` má hodnotu 10.

Výraz musí mať hodnotu preto, lebo práve táto hodnota sa uloží do premennej: Napríklad keď napíšeme `a = 3 + 8;` tak do `a` sa priradí číslo 11.

Cvičenie 4: Skúste určiť, čo robí nasledovný program. Aké hodnoty budú na konci v premenných `a`, `b`, `c`?

```
int main(){
    int a;
    a = 4;
    a = 7;

    int b, c;
    b = a;
    c = a + b;
    a = a + 4;
}
```

Riešenie 4: Najskôr sme vyrobili premennú `a`. Potom sme do nej priradili hodnotu 4. Následne sme do nej priradili hodnotu 7 (stará hodnota 4 sa zahodila), teda premenná bude obsahovať iba číslo 7. Potom sme vyrobili dve nové premenné `b` a `c` (správne, premenných rovnakého typu vieme vyrábať aj viac naraz, stačí ich mená oddeliť čiarkou). Do premennej `b` sa priradila hodnota výrazu `a`, teda 7. Do `c` sa priradí hodnota výrazu `a + b`, čo je hodnotou výrazu `7 + 7` teda 14. Nakoniec sa do `a` priradí `a + 4`, čiže `7 + 4`, čiže 11. V `a, b, c` budú na konci programu postupne hodnoty 11, 7, 14.

Aby sme vedeli lepšie pozorovať, čo sa v programe deje, naučíme sa vypisovať premenné na obrazovku. Bude na to slúžiť príkaz `printf()`, ale teraz bude použitý zložitejšie. Funkcii `printf()` totiž môžeme do zátvoriek napísať viac argumentov ako jeden. Prvý argument totiž nemusí byť obyčajný reťazec, ale môže byť špeciálny formátovací reťazec. Od obyčajného sa líši tým, že obsahuje podivné značky ako `%d` `%lf` `%4d` a mnohé ďalšie. Tieto značky sú pri spracovaní funkciou nahradené hodnotami, ktoré určíme v ďalších argumentoch tejto funkcie.

Vyskúšajte si skompilovať a spustiť nasledovný program.

```
#include<cstdio>
int main(){
    printf("%d + %d = %d\n",4,7,4+7);
}
```

Mal by vypísať reťazec `4 + 7 = 11`. Totiž funkcii `printf` sme dali 4 argumenty (argumenty oddeľujeme čiarkou) reťazec `"%d + %d = %d\n"` a tri výrazy `4`, `7` a `4+7` s hodnotami 4, 7 a 11. Funkcia najprv zoberie prvý argument, ktorý musí byť reťazec, nájde tam všetky výskyty `%<nieco>` a nahradí ich hodnotami ďalších argumentov. `%d` znamená, že argument bude celé číslo, ktoré chceme vypísať v desiatkovej sústave. O iných značkách si povieme viac neskôr, slúžia buď na vypisovanie iných typov objektov (reálne čísla, znaky, reťazce...) alebo na rôzny spôsob ich vypísania (počet desatinných miest, sústava...).

Keďže premenné sú tiež výrazy, tak môžeme vypisovať ich hodnoty takto:

```
#include<cstdio>
int main(){
    int a;
    a = 17;
    printf("premenna a ma hodnotu %d\n",a);
}
```

Alebo bez zbytočných blábolov:

```
#include<cstdio>
int main(){
    int a = 7;
    printf("%d\n",a);
}
```

Všimnite si riadok `int a = 7;`. Ide o skrátený zápis dvoch príkazov `int a; a = 7`. Dá sa to použiť aj pri deklarácií viacerých premenných, teda `int a = 1, b = 2, c = 3;` je skrátený zápis pre `int a = 1; int b = 2; int c = 3;`

Cvičenie 5: Skúste na rôznych miestach programu z cvičenia 1 vypísať hodnoty premenných. Všelijako upravujte program a pozorujte, čo sa deje.

Typy číselných premenných

K číselným premenným treba poznamenať ešte niekoľko dôležitých vecí. Každá číselná premenná a aj každé číslo má obmedzenú veľkosť.

Napríklad premenná typu `int` môže mať v sebe uložené len celé číslo od -2147483648 po 2147483647. Je to preto, že `int` je v pamäti zapísaný ako 32 jednotiek a núl. Teda si vieme zapamätať len 2^{32} rôznych hodnôt. V prípade premennej typu `int` sú to hodnoty od -2^{31} po $2^{31}-1$ (záporných čísel je presne polovica, kladných čísel je o 1 menej, kvôli číslu 0).

Cvičenie 6: Vyskúšajte si spustiť príkaz `printf("%d\n", 2147483647 + 1);`.

Existujú premenné aj s menším rozsahom, napríklad `short int` má rozsah $-32768..32767$ alebo $-2^{15}..2^{15}-1$. `char` má rozsah $-128..127$ resp. $-2^7..2^7-1$. Premenná typu `bool` má len dve hodnoty `false` resp. 0 (nepravda) alebo `true` resp. 1 (pravda).

Väčší rozsah majú zasa premenné `long long`, od $-2^{63}..2^{63}$. Pri ich vypisovaní však treba dávať pozor, namiesto `"%d"` treba písať `"%lld"` na Linuxe alebo `"%I64d"` na Windowse.

Existujú aj typy premenných, ktoré neukladajú záporné čísla. Napríklad `unsigned int` má rozsah $0..2^{32}-1$. Pri ich používaní musíme byť obozretní, lebo sa vám môže stať, že mínus jedna je viac ako nula. (Totiž -1 v bezznamienkovom `int`e bude 4294967295, kvôli tomu, ako sa udržujú hodnoty v pamäti.)

Binárna sústava a reprezentácia dát v pamäti.

Každá hodnota je v pamäti počítača reprezentovaná ako postupnosť núl a jednotiek – bitov. Osem bitov je jeden bajt. Napríklad `int` má veľkosť 4 bajty, čiže 32 bitov. `char` má zase len jeden bajt – 8 bitov. Keď máme v premennej typu `char` uložené číslo 47, tak v pamäti počítača to vyzerá ako 00101111. Rovnaké číslo v premennej typu `int` vyzerá 00000000 00000000 00000000 00101111.

Poznámka, niektoré počítače alebo systémy majú iné poradie bitov v pamäti.

Čísla sú teda v pamäti napísané v dvojkovej (binárnej) sústave. Tá je veľmi podobná našej desiatkovej (decimálnej), takže sa jej netreba báť. Aby sme sa neplietli, tak teraz chvíľu budeme písať sústavu do dolného indexu, lebo napr. $11_2 \neq 11_{10}$. (Alebo $3_{10} \neq 11_{10}$)

Keď v desiatkovej sústave napíšeme 325_{10} myslíme tým tri stovky, dve desiatky a päť jednotiek. V dvojkovej sústave napr. 1101_2 je jedna osmička, jedna štvorka, žiadna dvojka a jedna jednotka. Dokopy $8+4+0+1=13$, teda $1101_2 = 13_{10}$.

Matematicky zapísané, $325_{10} = 3 \cdot 100 + 2 \cdot 10 + 5 \cdot 1$. Ak chceme napísať 325_{10} v dvojkovej sústave, musíme použiť namiesto mocnín desiatky (1, 10, 100, 1000, ...) mocniny dvojky (1, 2, 4, 8, 16, ...). Teda $325 = 1 \cdot 256 + 0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$. Preto $325_{10} = 101000101_2$.

Opačný prevod sme už zvládli vyššie, napríklad ak nás zaujíma hodnota 101111_2, stačí jednoducho sčítať $1 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 0 \cdot 16 + 1 \cdot 32 = 47_{10}$.

Cvičenie 7: Preveďte z desiatkovej sústavy do dvojkovej nasledovné čísla 31, 200, 1024.

Cvičenie 8: Preveďte z dvojkovej sústavy do desiatkovej čísla 10101, 100000000, 1100110011.

Cvičenie 9: Sčítajte v binárnej sústave čísla 110110 a 1111101. Následne ich preveďte do desiatkovej sústavy a sčítajte v desiatkovej sústave. Výsledok preveďte naspäť do dvojkovej a výsledok overte.

Riešenie 7: $31_{10} = 11111_2$. $200_{10} = 11001000$. $1024_{10} = 10000000000$, lebo $1024 = 2^{10}$.

Riešenie 8: $10101_2 = 21_{10}$. $100000000_2 = 256$. $1100110011_2 = 819_{10}$

Riešenie 9: V binárnej sústave sčítavame rovnako ako v desiatkovej, napíšeme si čísla pod seba, sčitavame zľava doprava a pripočítavame prípadné zvyšky.

```
    110110     =     54
+  1111101     =  + 125
   -------          ---
  10110011     =    179
```

Kladné čísla sú v pamäti reprezentované práve v dvojkovej sústave. Keď je číslo príliš veľké, jednoducho sa zahodia prvé bity – ukážeme si to na príklade. Premenná typu `unsigned char` má jeden bajt (8 bitov) a teda je schopná reprezentovať čísla od 0 po 255. Nie je žiadným prekvapením, že 5 bude zapamätané ako 00000101, či 255 ako 11111111. Čo sa stane, keď ku 255 pripočítame 1? Dostaneme $11111111 + 1 = 100000000$. Keďže si môžeme zapamätať len 8 bitov, musíme prvú jednotku zahodiť. Dostaneme 00000000, čo je 0.

Preto pri premennej unsigned char $255 + 1 = 0$ alebo $100 \cdot 100 = 16$.

Ale ako je to s premennými ktoré podporujú aj záporné čísla? Kladné čísla sa reprezentujú tak ako predtým, napríklad v premennej typu `char` $0 = 00000000$ až $127 = 01111111$.

Čo sa stane, keď od nuly odpočítame 1? $00000000 - 1 = ...11111111111111$ (nekonečné čislo) ale nás zaujíma len posledných 8 bitov, čo je 11111111. Takže $-1 = 11111111$. Keď ďalej odpočítavame 1 dostávame $-2 = 11111110$, $-3 = 11111101$ ... $-128 = 10000000$.

Super je, že sa nám zachovali logické vlastnosti, napríklad že $-1 + 1 = 0$, $-50 + 40 = -10$, $-31 + 100 = 69$... bez toho, aby sme museli pri práci s binárnym čislom rozlišovať, či je číslo záporné alebo kladné. Jednoducho sčítame čísla pod sebou a ak sa výsledok nezmestí, tak zahodíme prvé bity.

Napr. $-1 + (-1) = 11111111 + 11111111 = 1011111110 \rightarrow 11111110 = -2$.

Tento spôsob reprezentovania záporných čísel sa nazýva aj dvojkový doplnkový kód.

Vtip: "Ľudí môžeme rozdeliť do 10 skupín – tých, ktorí rozumejú binárnej sústave a tých, ktorí jej nerozumejú."

Pretypovanie

Sčítavať, odčítavať, násobiť, priraďovať, porovnávať...sa dajú len premenné a čísla rovnakých typov. (Všimnite si, že aj čísla (napríklad číslo 45) majú svoje typy.)

Pokiaľ sú pri operácii typy rôzne, treba ich pretypovať a naštastie v C++ sa mnohé pretypovanie robí automaticky. Napríklad medzi číslami. `int a = 47; char c = a;` nie je žiaden veľký problém. (V tajnosti prebehne jedno pretypovanie, v príkaze `c = a`, sa a vyhodnotí ako `(int)47`, pretypuje sa na `(char)47` a následne sa priradí i `c = (char)47`.

Pokiaľ pretypujeme z menších premenných do väčších, číslo sa doplní nulami zľava. `(char)19` = 00010011 → 00000000000000000000000000010011 = `(int)19`

Z väščích do menších, sa zasa zahadzujú prvé bity – **pozor** pri tomto pretypovaní sa môže číslo zmeniť. `(int)200` = 00000000000000000000000011001000 → 11001000 = `(char)(-56)` Tento jav sa volá **pretečenie**.

Bežné čísla, ktoré napíšete v programe majú typ `int`. Teda aj pri `char a = 20;` nastáva pretypovanie z `int` do `char`.

Keď niekde napíšete číslo 123 456 789 123, tak vám pravdepodobne kompilátor vypíše varovanie, lebo dané číslo je typu `int`, ale nezmestí sa do jeho rozsahu. Treba písať `123465789123LL` alebo `12345678912311`.

Pri operáciach ako sčítanie, násobenie... na rôznych typoch sa najprv obe čísla pretypujú na všeobecnejší typ, až potom sa spočítajú.

Napríklad ak `char a = 20;` a `int b = 20;` tak výsledkom `a * b` bude 400.

Treba si dávať **pozor** na nasledovnú vec.

Cvičenie 10: Skopírujte, skompilujte a spustite nasledovný program. (Pokiaľ používate windows, použite miesto `%lld` `%I64d`)

```
#include<cstdio>
int main(){
    ll a = 1000000*1000000;
    printf("d = %lld\n",d);
}
```

Čo vypísal? Prečo? Ako by ste ho opravili, aby naozaj vypísal 1 000 000 000 000?

Riešenie 10: Vynásobili sme dva `int`y, keďže majú rovnaký typ, nič sa nepretypuje. Keďže výsledok je príliš veľký, nastáva pretečenie a výsledkom násobenia bude `int` −727 379 968. Následne sa vykoná priradenie `a = -727379968`, ale správna hodnota už je nenávratne preč. Opraviť to môžeme tak, že zmeníme `1000000` na `1000000LL`. (Stačí zmeniť jedno.)

Pretypovanie bežne funguje automaticky, ale vieme ho spôsobovať aj ručne. Syntax je `(<typ>)<hodnota>` alebo `<typ>(hodnota)` alebo použijeme obe dvojice zátvoriek.

Cvičenie 11: Aké budu hodnoty v `a`, `b` po nasledovných priradeniach?

```
int a = 1234, b;
b = (char)a;
b = char(a);
b = char(a)*a;
```

Riešenie 11: Hodnota `a` bude vždy 1234, pretypovanie nemení premenné. `b` bude mať hodnoty -46, -46 a -56764 (teda $-46 \cdot 1234$).

Poznámka, `bool` nie je *číselný* typ. Teda `bool(int(4))` nie je 0 hoci posledný bit v zápise 4 je 0. Pre `bool` platí, že `bool(0)` je `false` (resp. 0), všetky ostatné čísla sa zmenia na `true` (resp. 1).

Ku číslam sa hodí spomenúť ešte dve číselné operácie, delenie `/` a zvyšok po delení `%`.

Keď pracujeme s celými číslami, tak chceme aby aj podiel bol celé číslo. Preto sa výsledok vždy zaokrúhľuje smerom k nule. $3/2$ nie je 1.5 ale $3/2 = 1$, $11/3 = 3$, $14/7 = 2$, $-15/3 = -5$, $-100/3 = -33\ldots$

Zvyšok po delení určite poznáte so základnej školy, dá sa definovať ako $a\%b = a - (a/b) \cdot b$. Napríklad $47\%10 = 47 - 47/10 \cdot 10 = 47 - 4 \cdot 10 = 47 - 40 = 7$, $-5\%3 = -2$.

Alternatívny výstup

V C++ sa dá na výstup vypisovať aj inak ako cez `printf`. Pokiaľ includneme knižnicu iostream, môžeme vypisovať pomocou streamu `std::cout`. Používa sa veľmi jednoducho, hoci vo vnútri je schovanej veľa mágie.

```
#include<iostream>
int main(){
    std::cout << "Ahoj svet\n";
    int a = 4;
    std::cout << "a = " << a << endl;
}
```

Nevýhodou streamov je, že sú o máličko pomalšie ako príkazy z `cstdio`. Bežne to nepostrehnete, ale pokiaľ by ste chceli načítať/vypísať niekoľkostotisíc prvkov, odporúčame použiť `cstdio`.

Jednou z výhod je, že sa nemusíte starať o typy a formátovacie značky `%lld`. Preto vám odporúčame používať streamy, ak programujete na Windowse a vypisujete `long long`y.

Pokiaľ nechcete stále písať pred každým príkazom `std::`, stačí pridať riadok `using namespace std;`

```
#include<iostream>
using namespace std;
int main(){
    cout << "Ahoj svet\n";
    int a = 4;
    cout << "a = " << a << endl;
}
```

Cvičenie 12: Vezmite nejaký program, kde ste viackrát použili `printf` a prepíšte ho na streamy.

Načítavanie vstupu

Doteraz všetky programy boli relatívne nanič, pretože pri každom spustení vypisovali to isté. Namiesto programov typu "vypíš prvých 10 nepárnych čísel" by sme radi mali "načítaj číslo n a vypíš prvých n nepárnych čísel".

Takže potrebujeme vedieť načítavať vstup.

Na načítavanie sa dá použiť funkcia `scanf()`, ktorá sa syntaxou veľmi podobá `printf()`. Prvý parameter je opäť formátovací reťazec, ale ďalšie už nie sú výrazy, ale smerníky na premenné. Teda ak chceme načítať jedno číslo do premennej a, napíšeme `scanf("%d", &a);`

Prečo tam je `&a` miesto `a`? Dôvod je veľmi jednoduchý, `a` je výraz, ktorého hodnota je hodnota `a`, napríklad číslo 7. A zavolaním `scanf("%d", 7);` ťažko budeme vedieť, kam máme uložiť načítanú hodnotu. (Vieme ukladať len do premenných, nie do hodnôt). Keď však napíšeme `&a`, tak výsledkom výrazu nebude hodnota `a`, ale miesto, kde je `a` uložené v pamäti počítača. Funkcia teda načíta číslo a uloží ho na to miesto v pamäti, kde bola predtým hodnota `a`.

Vyskúšajte si skompilovať a spustiť tento program:

```
#include<cstdio>
int main(){
    int a, b;
    scanf("%d %d", &a, &b);
    printf("%d\n", a+b);
}
```

Po spustení program čaká, kým mu napíšete dve čísla napr. `105 21`. Po zadaní čísel musíte stlačiť enter, inak sa k programu nedostanú.

Pokiaľ sa vám `scanf` nepáči, dajú sa použiť streamy.

```
#include<iostream>
using namespace std;
int main(){
    int a, b;
    cin >> a >> b;
    cout << a + b << endl;
}
```

Časom sa naučíme načítavať aj iné veci ako čísla.

Zoznámenie sa s testovačom

Bez poriadneho precvičovania sa človek programovať nenaučí, preto máme pre vás pripravené množstvo úloh.

Ako riešiť úlohy?

1. prečítame si zadanie
2. napíšeme program, ktorý rieši úlohu
3. vyskúšame si program u nás na počítači (či ho kompilátor skompiluje, či dáva správne výstupy aspoň na ukážkových vstupoch)
4. submitneme (odovzdáme) **zdrojový kód** cez internet na stránke testovača
5. počkáme, kým testovač otestuje náš program
6. ak je celková odpoveť OK, tešíme sa a ideme riešiť ďalší príklad
7. inak opravíme chybu a pokračujeme bodom 3

Vyriešte prvé štyri úlohy v prvej sade: **cpp01(Ahoj JeŠko)**, **cpp02(Číslo)**, **cpp03(Zámena)**, **cpp04(ObdĺžnikO)**.

Pokračovanie C++

Ako správne používať tento študijný text: Milý čitateľ, chystáš sa prečítať si kuchárku o základoch programovacieho jazyka C++. Najdôležitejšie pri učení sa nového programovacieho jazyka je poriadne si ho precvičiť. Preto počas čítania narazíš na niekoľko úloh, ktoré ti odporúčame vyriešiť a naprogramovať. Má to dopomôcť k tomu, aby si si všetko lepšie zapamätal a dostal do krvi.

Tento text je napísaný aj pre ľudí, ktorí nikdy program nevideli. Pokiaľ už máš nejaké základy z programovania, môžeš pokojne preskakovať tie časti, ktoré ovládaš.

Obsah:

Komentáre

Často z dôvodu väčšej prehľadnosti alebo zrozumiteľnosti píšeme do kódov komentáre, čiže text, ktorý nie je určený pre kompilátor a počítač, ale je určený pre ľudí, ktorí si kód pozerajú.

Z hľadiska funkčnosti programu sa správa, ako keby tam ani nebol. My si pomocou nich budeme občas niečo vysvetľovať. Jednoriadkové komentáre píšeme pomocou `//`. Všetko, čo je za touto dvojicou znakov až do konca riadku sa ignoruje.

Viacriadkové komentáre uzatvárame medzi `/*` a `*/`.

```
\\ toto je jednoriadkovy komentar
int main(){
    /*
        viac riadkovy komentar moze vyzerat napriklad takto.
        Do vnutra mozeme napisat vsetko okrem
        hviezdicky a lomitka, ktore komentar ukoncuju
    */

    int x = 0; // do x priradime 0
}
```

Podmienky alebo rozhodnutia

Bez toho, aby sa program vedel rozhodovať, by sme ďaleko nezašli. Vo väčších aplikáciach je dôležité interagovať s užívateľom – `ak` užívateľ stlačí toto tlačidlo, spravím toto, `inak` nerobím nič a podobne.

V našich programoch môžeme chcieť, aby sa program správal odlišne, keď je v nejakej premennej párne číslo a odlišne, keď je tam nepárne.

Na rozhodovanie v jazyku C++ slúži príkaz `if` (po slovensky 'ak') so syntaxou `if (<podmienka>) <príkaz-t>` alebo `if (<podmienka>) <príkaz-t> else <príkaz-f>` ('else' = 'inak'). Jeho správanie je veľmi intuitívne, pokiaľ je `<podmienka>` pravdivá, vykoná sa `<príkaz-t>`. Pokiaľ je podmienka nepravdivá, tak vykoná to, čo je za `else` alebo sa nevykoná nič, ak tam slovíčko `else` nie je.

Príklady použitia:

```
int x = 1;

if (x == 7)
    printf("x je sedem\n");
else
    printf("x nie je sedem\n");

if (x > 0)
    printf("x je kladne cislo\n");

if (x > 0)
    if (x < 2)
        printf("x je jedna\n");
```

Pokiaľ chceme vykonať viac príkazov, stačí miesto `<príkaz>` napísať `{ <príkaz-1>; <príkaz-2>; ... <príkaz-n>; }` takto:

```
int x = 1;
if (x != 7){
    printf("x nie je sedem\n");
    x = 7;
    printf("preto som zmenil x na sedem\n");
} else {
    printf("x nie je sedem\n");
    printf("takze nemusim nic robit\n");
}
```

Čo všetko môže byť podmienka? Podmienkou môže byť ľubovoľný výraz, ktorý má hodnotu typu bool (v prípade, že výraz nevracia hodnotu typu bool, pretypuje sa). Keď má výraz hodnotu true, považujeme ho za pravdivý, false je nepravdivý.

Poznáme mnoho operátorov, ktoré vracajú true/false.

Napríklad `==`, `!=`, `<=`, `>=`, `<` a `>` patria medzi binárne operátory (t.j. dávame ich medzi dva výrazy, napr. `(a + 7) <= (b * 2)`). Správanie je opäť intuitívne, `a < b` je true vtedy, keď `a` je menej ako `b`. `!=` znamená nerovná sa, čiže `a != 0` vráti `true`, keď `a` nie je 0. Pozor, nesmieme si zameniť `a == b` a `a = b`. Prvé z toho je výraz, ktorý vráti `true`, v prípade, že `a` má rovnakú hodnotu ako `b`. Druhá vec je normálne priradenie, takže do `a` vloží hodnotu `b`-čka, a následne vráti novú hodnotu `a`. Preto `if (a = 7)` vráti `true` bez ohľadu na pôvodnú hodnotu `a` a navyše zmení hodnotu `a` na sedem. Na toto si treba dávať fakt veľký pozor.

Potom poznáme aj logické operátory, `&&`(a zároveň), `||`(alebo) a `!`(nie je pravda, že). Ktoré sa správajú opäť intuitívne. `(a == 7) && (b < 3)` je `true` práve vtedy, keď `a` má hodnotu 7 a `b` je menej ako 3. `(a == 3) || (b < 7)` je pravda, keď platí aspoň jedna z podmienok `(a == 3)`, `(b < 7)`. `!` robí z `false` `true` a z `true` `false`, takže napr. `!(a < 3)` je ekvivalentné s `a >= 3`. Zaujímavé je, že `||` a `&&` sa správajú šetrne, napríklad ak máme `a || b` a vieme, že `a` je true, tak nemusíme vyhodnocovať `b` ('pravda alebo hocičo' je stále pravda). A v C++ sa to teda ani vyhodnocovať nebude. Preto zatiaľ čo `(true && a = 5)` a `(false || a = 5)` zmenia hodnotu `a` na 5, `(false && a = 5)` a `(true || a = 5)` ju nechajú na pokoji (príkaz a = 5 sa nevykoná).

Tiež si netreba mýliť `&&` s `&`(bitwise and) a `||` s `|`(bitwise or). Dvojité verzie pracujú s bool-mi, teda najprv sa pokúsia pretypovať operandy na booly a potom vyrobia nový bool. `(10 || 6)` je true. Jednoité verzie rozbijú číslo na bity, s každým bitom spravia operáciu zvlášť (0 = false, 1 = true) a výsledné bity poskladajú naspať do čísla. $(10|6) = (1010_2|110_2) = 1110_2 = 14$. Bitová verzia `!` je `~`.

Treba si dávať pozor na prioritu (precedenciu) operátorov (tak ako násobenie má prednosť pri sčítaní, aj ostatné operátory majú medzi sebou nejakú hierarchiu). A aby toho nebolo málo, niektoré veci sa vyhodnocujú sprava doľava a niektoré zľava doprava. Kompletný prehľad nájdete napríklad na `http://en.cppreference.com/w/cpp/language/operator\_precedenceu` Dôležité je, že ak si nie ste istí, tak zátvorkujte, inak sa to správa relatívne rozumne.

Vyriešte v testovači úlohy **cpp05(Rovnaké čísla)**, **cpp06(Akéže je znamienko?)**, **cpp07(Najmenší)**, **cpp08(Prostredný)**, **cpp09(Priemer)**, **cpp10(Zvyšok po delení 7)** a **cpp11(Nenajväčší)**.

Cvičenie 13: Akú pravdivostnú hodnotu má výraz `( (!7 == false) && (0 | (a = 5)) <= 4 )`?

Cvičenie 14: Akú hodnotu má `((23 | 32) & 30)`?

Riešenie 13: `( (!7 == false) && (0 | (a = 5)) <= 4 )` → `( ( false == false) && (0 | 5) <= 4 )` → `( ( true ) && (5) <= 4 )` → `( true && false )` → `false`

Riešenie 14: $((23|32)\&30) = ((10111_2|100000_2)\&11110_2) = (110111_2\&11110_2) = 10110_2 = 22$

Cykly alebo opakovanie

Ako by ste 10 krát vypísali ähoj"? Spravili by ste to takto?

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

A čo keby ste chceli vypísať ähoj"tisíckrát, alebo milionkrát?

Nebojte sa, aj na toto programátori mysleli a vymysleli cykly. Najjednoduchší cyklus je `while` cyklus. Jeho syntax je `while(<podmienka>) <príkaz>` prípadne namiesto `<príkazu>` môžeme napísať viacero príkazov zabalených v `{}`.

Tento cyklus funguje veľmi jednoducho - dokým je podmienka splnená, opakuje príkaz. Vysvetlíme si to na príklade.

```
int i = 0;
while(i < 10){
    printf("ahoj\n");
    i = i+1;
}
```

Najprv nastavíme `i` na 1. Následne budeme vykonávať vnútro `{}` pokiaľ `i < 10`. Keďže vo vnútri `{}` zväčšíme `i` vždy o 1, celý cyklus sa zopakuje 10 krát, až kým sa nestane, že `i == 10` a teda `!(i < 10)` a teda cyklus skončí. Preto aj tento program vypíše ähoj"10 krát. Prerobiť ho, aby vypísal ähoj"tisíckrát by nám nerobilo žiaden problém však?

Ešte poznamenajme, že miesto `i = i+1` vieme písať `i += 1` alebo dokonca `i++`, či `++i`.

Vyššie uvedená konštrukcia – nainicializujeme premennú, pokým premenná nie je nejaká, robíme niečo je natoľko častá, že jej programátori vymysleli nový názov. `for(<príkaz1>; <podmienka>; <príkaz2>) <príkaz-telo-cyklu>` najprv vykoná `<príkaz1>`, následne kým platí podmienka vykonáva `<príkaz-telo-cyklu>` a `<príkaz2>`.

Vyššie uvedený program s `while`om teda vieme prepísať na

```
for(int i = 0; i<10; ++i)
    printf("ahoj\n");
```

Samozrejme, premennú i môžeme využiť aj v tele cyklu. Nasledovný program vypíše čísla 0 až 7, každé na zvlášť riadku.

```
for(int i = 0; i<8; ++i)
    printf("%d\n", i);
```

Cvičenie 15: Napíšte program, ktorý vypíše prvých 50 nepárnych čísel (1, 3, 5 ...) Zvyšok po delení dvoma (paritu) vieme zistiť napríklad operátorom `%`. `a % b` vráti zvyšok `a` po delení `b`. Napríklad `47 % 10 == 7`, `10 % 2 == 0`. Pozor treba dávať na záporné čísla, `-13 % 5 == -3`.

Riešenie 15: Predvedieme si tri spôsoby.

```
for(int i = 0; i<100; ++i)
    if (i%2 == 1)
        printf("%d\n", i);

for(int i = 0; i<50; ++i)
    printf("%d\n", 2*i + 1);

for(int i = 1; i<100; i+=2)
    printf("%d\n", i);
```

Pri cykloch si treba dávať pozor, keďže sa môžu zacykliť a potom váš program bude bežať donekonečna (kým ho nezabijete). Príklady nekonečných cyklov sú napr. `while(true);` `for(int i = 10; i>0; ++i);` ale aj `while(a<7);` (lebo premenná a sa v cykle nemení).

Cykly a podmienky sa samozrejme dajú do seba ľubovoľne veľakrát vkladať, napríklad ako v nasledujúcom nezmyselnom programe.

```
int a = 7;
while(a<70) {
    if (a == 2) {
        a = 31;
    } else {
        for (int i = 0; i<a; ++i) {
            while(i<a) ++i;
        }
    }
}
```

Keď už teda ľahkovážne pcháme všelikam zátvorky `{}`, zíde sa nám vedieť jedna vec o premenných – majú obmedzenú platnosť. Každá premenná existuje len v rámci `{}` v ktorých bola vyrobená. Potom proste zanikne.

```
int a = 0; // toto je globalna premenna, ta existuje stale a vsade

int main(){
    int b = 0; // tato premenna existuje iba vnutri main()

    if (true) {
        int c = 1;
    }
    c = 7; // !! chyba !! c prestalo existovat na predoslom riadku
    if (true) {
        c = 7; // !! stale chyba !!
    }
    int c = 1;
    if (true) {
        c = 7; // OK
        int b = 3; /*
         Vyrobit novu premennu b vam sice kompilator dovoli
         ale fakt to nerobte (nenazyvajte dve premenne
         vo vnorenych blokoch rovnako). Niekedy sa to moze spravat
         cudne.
        */
    }

    for(int i; i<a; ++i){
        i+=1; // OK
    }
    i = 7; // !! zasa chyba !! premenna i existovala len v ramci for-cyklu
}
```

Vyriešte v testovači úlohy **cpp12(Cykly sú super)**, **cpp13(Pásik)**, **cpp14(Obdĺžnik)**, **cpp15(Trojuholník)**, **cpp16(Pyramída)**, **cpp17(Najväčší)**, **cpp18(Fibonacci)**, **cpp19(Zväčši)**.

Polia

Sú situácie, kde si jednoducho neporadíme s obmedzeným množstvom premenných. (Vedeli by ste načítať n čísel zo vstupu a následne ich vypísať v opačnom poradí? A čo príklad **zcpp20(Zväčši v. 2)**?) To sa s obmedzeným množstvom číselných premenných nedá, pretože ak je na vstupe viac čísel ako máme premenných, tak si ich nedokážeme zapamätať na neskorší výpis.

Aj keby sme mali napríklad na vstupe 20 čísel, tak je veľmi nepraktické vyrobiť si premenné `a0` až `a19`, do ktorých si vstup načítame.

Preto existujú polia. Pole je ako keby skupina premenných rovnakého typu naukladaných za sebou. V súvislosti s nimi používame hranaté zátvorky `[]`. Pole intov dĺžky 20 vyrobíme napríklad takto `int pole[20];` Tento príkaz nám vyrobí 20 `int`ových premenných, ku ktorým vieme pristupovať `pole[0]`, `pole[1]`, `pole[2]`, ... `pole[19]`.

Pozrite si príklad použitia v nasledovnom programe:

```
#include<cstdio>
int main(){
    int a[47];
    a[0] = 0;
    a[1] = 1;
    for(int i = 2; i<30; ++i){
        a[i] = a[i-1] + a[i-2];
    }
    for(int i = 0; i<30; ++i)
        printf("%d ", a[i]);
    printf("\n");
}
```

Na čo si treba dávať pozor?

Keď pristupujeme mimo poľa, napríklad `int b[40]; b[53] = 7; b[-1] = 3; int a = b[40];` ... dejú sa zlé veci. Niekedy program hneď spadne a vyhlási chybu, inokedy pokračuje ďalej a robí čudné veci, premenným sa záhadne zmení hodnota vykonávajú sa `if`y, ktoré by nemali atď. Preto si vždy dávajte veľký pozor, keď používate polia, aby ste nepristupovali mimo nich. Pozor treba dávať aj na to, že `int b[40];` tvoria premenné `b[0]` až `b[39]` (teda ich je 40, ale premenná `b[40]` tam nie je).

Pri deklarácii poľa musí byť v `[]` konštanta. **Nesmiete** spraviť `int a = 7; int b[a];` hoci vám to kompilátor pravdepodobne dovolí, môžu sa diať zlé a nepredvídateľné veci. Pokiaľ chcete pole so závislou veľkosťou, použite `int *b = new b[a];` ale reálne toto nebudete potrebovať. V drvivej väčšine úloh máte zadané obmedzenia, napríklad "na vstupe je číslo n, $1 \leq n \leq 1000$". Vtedy, keď budete potrebovať pole veľkosti n, tak si môžete natvrdo vyrobiť pole veľkosti 1000 a netrápiť sa.

Odporúčame však vyrábať troška väčšie polia, napríklad keď $n < 10000$ tak pole by mohlo mať veľkosť 10047. Menej ako 1% nárast nič nepokazí, a trocha pomôže, ak náhodou siahame o pár políčok ďalej, ako by

sme čakali.

Pamäť počítača má prekvapivo obmedzenú veľkosť. Jeden int zaberá 4 bajty, takže pole `int velkepole[1000000];` zaberá približne 1MB. Bežne na sútažiach máte pamäťový limit obmedzený na 64 MB, 256 MB, alebo 1024MB, tak na to treba myslieť.

Navyše na bežných počítačoch je limitovaná veľkosť lokálnych premenných (tie sú vo vnútri `main()` alebo iných funkcií) na 8MB, takže to tiež môže robiť problémy, keď si doma budete spúšťať programy s veľkými pamäťovými nárokmi (na našom testovači by tento limit nemal byť).

Pole je v podstate zase len premenná a z premenných vieme robiť polia, takže aj z polí vieme robiť polia. Pole polí (dvojrozmerné pole) vieme vyrobiť napríklad takto `int A[1000][1000];` Takto vieme ísť aj ďalej, len treba zasa dávať pozor na pamäť `int A[100][100][100][10][10];` zaberá cca 400MB.

Ak chceme, tak môžeme polia vyplniť už pri deklarácii, napríklad `int a[5] = {1, 2, 3, 4, 5};` Dokonca vtedy si kompilátor dokáže sám pozrieť počet prvkov, takže stačí napísať. `int a[] = {1, 2, 3, 4, 5};`

Pri viacrozmerných poliach môžeme nenapísať len prvé číslo, napr.

`int a[][2][1] = { {{1}, {2}}, {{3}, {4}}, {{5}, {6}} };`

Toto vieme robiť len pri deklarácii, takže `int a[2]; a = {1, 2};` nefunguje.

Vyriešte v testovači úlohy **cpp20(Zväčši v. 2)**, **cpp21(Súčet menších)**, **cpp22(Počet najmenších)**, **cpp23(Rozdiel)**, **cpp24(Súčtová pyramída)**, **cpp25(Výmena)** a **cpp26(Otočenie)**.

Ďalšie časti C++

Ako správne používať tento študijný text: Milý čitateľ, chystáš sa prečítať si kuchárku o základoch programovacieho jazyka C++. Najdôležitejšie pri učení sa nového programovacieho jazyka je poriadne si ho precvičiť. Preto počas čítania narazíš na niekoľko úloh, ktoré ti odporúčame vyriešiť a naprogramovať. Má to dopomôcť k tomu, aby si si všetko lepšie zapamätal a dostal do krvi.

Tento text je napísaný aj pre ľudí, ktorí nikdy program nevideli. Pokiaľ už máš nejaké základy z programovania, môžeš pokojne preskakovať tie časti, ktoré ovládaš.

Obsah:

Funkcie

Ako neprogramovať jednu vec viackráť

Doteraz, keď sme v programovaní chceli niečo zopakovať viackrát, tak sme použili cyklus. Niekedy však potrebujeme danú vec robiť na rôznych miestach. Napríklad vypisovať čísla 1 až 10 v rôznych častiach programu.

```
int main(){
    // radi by sme napisali desat cisel na zaciatku programu
    for(int i = 0; i<10; ++i)
        printf("%d␣", i+1);
    printf("\n");

    int n;
    scanf("%d", &n);
    if (n < 0) {
        // potom raz v pripade, ze uzivatel zada zaporne cislo
        for(int i = 0; i<10; ++i)
            printf("%d␣", i+1);
        printf("\n");
    }

    for(int j = 0; j<n; ++j) {
        // alebo n krat, ked uzivatel zada kladne n
        for(int i = 0; i<10; ++i)
            printf("%d␣", i+1);
        printf("\n");
    }
}
```

Čo je na vyššie uvedenom programe zle?

Veľakrát používa rovnaký kód. A to máme ešte štastie, že opakujeme len trikrát a opakujúci sa kód je dlhý len tri riadky. Aj tak, keď si zmyslíme, že nechceme vypisovať 10 čísel ale 11, tak to treba všade prepísať. Fuj.

Teraz si predstavme, že robíme niečo podstatne zložitejšie a na 200 miestach by sme mali rovnaký 10 riadkový blok. Keby sme tam chceli niečo zmeniť, tak sa zbláznime. A určite pri nejakom prepise spravíme chybu. A nemáme šancu tú chybu nájsť.

Proste copy-pastovanie kódu je zlo a chceme sa mu vyhnúť, lebo tvorí neprehľadné a chybné programy. Ale ako sa mu vyhneme?

Celkom dobrý spôsob je zabaliť kus kódu do funkcie – nejako si ten kus kódu nazvať. Funkciu môžeme chápať ako samostatný program. Proste postupnosť príkazov, ktorá niečo robí. Koniecconcov všetky naše doterajšie programy boli len funkcia `int main()`.

Funkciu vyrobíme tým, že ju zadeklarujeme v tvare `<typ> <meno>(<argumenty>){<príkazy>}` Spúšťame (voláme, po anglicky call) funkcie `<meno>(<hodnoty-argumentov>)`

Podivný program zhora vieme prepísať napríklad takto:

```
// nasledujuca funkcia je typu void, vola sa vypis_desat_cisel a nema ziadne argumenty

void vypis_desat_cisel(){
    for(int i = 0; i<10; ++i)
        printf("%d ", i+1);
    printf("\n");
}

// s funkciou main uz sme sa stretli
int main(){
    // zavolame funkciu
    vypis_desat_cisel()

    int n;
    scanf("%d", &n);
    if (n < 0)
        vypis_desat_cisel()

    for(int j = 0; j<n; ++j)
        vypis_desat_cisel()
}
```

Dostali sme oveľa prehľadnejší kód a ak chceme napríklad zmeniť počet vypisovaných čísel, stačí to spraviť na jednom mieste.

Čo je to `void` a vôbec ten typ?

Funkcia môže vracať nejakú hodnotu a *ten* typ je typ hodnoty, akú funkcia vracia. `void` znamená, že funkcia nevracia nič. Funkcia, ktorá niečo vracia, môže byť napríklad funkcia `osem()` z nasledujúceho príkladu. Isto si všimnete, že na vrátenie hodnoty slúži príkaz `return`. Príkaz `return` zároveň skončí vykonávanie funkcie, preto všetko, čo je za ním, sa už nevykoná.

```
int osem(){
    return 8;
}

int devat(){
    return 9;
    printf("tento prikaz sa nevykona\n");
}

void nic_nerob(){
    return; // pri void funkcii nemame co vratit, tak piseme return; bez hodnoty
    printf("ani tento\n");
}

int main(){
    // navratove hotnoty mozeme vyuzivat vo vyrazoch:

    printf("funkcia osem vracia %d\n", osem());
    int a = devat();
    printf("a je %d\n", a);

    // V jazyku C sa musel na konci main() pisat aj tento riadok:
    return 0;
    // V C++ pokial nic nevratime, vrati sa nejaka defaultna hodnota, napr. 0,
    // takze ten riadok pisat nemusime
}
```

Zatiaľ tie funkcie nie sú veľmi úžasné, lebo robia stále to isté (asi ako programy bez vstupu). A stále nevieme, čo sú to tie argumenty.

Tak poďme obe veci vyriešiť zároveň. Argumenty sú akoby vstup pre funkciu. Už sme sa s tým stretli, keď sme napríklad funkcii `printf()` do zátvoriek zadávali, čo má vypísať. Analogicky namiesto funkcie `vypis_desat_cisel()` môžeme použiť všeobecnejšiu funkciu:

```
void vypis_cisla(int pocet){
    for(int i = 0; i<pocet; ++i)
        printf("%d ", i+1);
    printf("\n");
}

int main(){
    vypis_cisla(10);

    // takto vieme pouzivat funkciu na rozne pocty vypisovanych cisel
    vypis_cisla(10 + 7);
    int n;
    scanf("%d", n);
    vypis_cisla(n);
}
```

Funkcia môže mať ľubovoľný počet argumetov, pri deklarácii(vyrábaní) funkcie ich jednoducho vymenujeme medzi zátvorky `()` v rovnakom tvare, ako pri deklarácii premenných, oddelené čiarkami. Následne ich môžeme používať ďalej vo funkcii, ako nové premenné (t.j. ich môžeme napríklad aj meniť).

Pri volaní funkcie píšeme medzi zátvorky hodnoty alebo výrazy, samozrejme musí sa zachovať typ.

```
int scitaj(int a, int b, int c){
    // prehodÄme hodnoty premennych, len tak pre srandu
    int pomocna = a;
    a = b;
    b = c;
    c = pomocna;
    return a+b+c;
}
int main(){
    int a = 5;
    printf("%d\n", scitaj(1+7, 4*a, a)
}
```

Veľmi užitočná vlasnosť funkcií je rekurzívne volanie, teda, že funkcia sa može volať opakovne sama seba. Toto vieme využiť napríklad aj na počítanie všeliakých zaujímavých vecí bez toho aby sme museli nejak významne premýšľať.

Fibonacciho postupnosť, $\{F_n\}$, je veľmi zaujímavá číselná postupnosť, definovaná takto: $F_0 = 0$, $F_1 = 1$, ak $i > 1$ potom$F_i = F_{i-1} + F_{i-2}$. Bez toho aby sme tejto postupnosti hlbšie porozumeli, môžeme ju zapísať ako funkciu.

```
int F(int n){
    if (n < 2)
        return n;
    return F(n-1) + F(n-2);
}
```

Zjavne `F(n)`$= F_n$. Nie je to super? Jediný problém tejto funkcie je, že nie je veľmi rýchla, na vašich počítačoch sa rozumne rýchlo zvládajú čísla pre $n \leq 40$. Neskôr sa naučíme, ako sa tomuto neduhu pri funkciách vyhnúť.

Funkcia sa môžu volať aj navzájom, **nie** však takto:

```
void a(){
    b(); // ! chyba ! funkcia b nie je deklarovana.
}
void b(){
    a();
}
```

Volať funkciu môžeme až po deklarácií, ale naštastie môžeme deklarovať funkciu pred definovaním (napísaním tela) takto:

```
void b();

void a(){
    b();
}
void b(){
    a();
}
```

Samozrejme tieto funkcie `a`, `b` nemajú veľký úžitok, lebo po zavolaní jednej z funkcií sa program zacyklí až sa časom minie pamäť do ktorej sa zapísujú zavolané funkcie a program spadne.

Cvičenie 16: Napíšte funkciu, ktorá počíta $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ bez použitia cyklov.

Riešenie 16:

```
int faktorial(int n){
    if (n < 2) return 1;
    return n*faktorial(n-1);
}
```

Vyriešte v testovači úlohu **cpp27(Tlačítka)**.

Znaky

V C++ okrem čísel vieme pracovať aj so znakmi. Znaky píšeme medzi jednoité úvodzovky, napríklad `'a'` `'z'` `'F'` `'7'` `'.'` `'#'` `'-'`... Niektoré znaky treba písať pomocou \, ako sme si už spomínali, napríklad `'\n'` alebo `'\''`.

Čo sa týka premenných, najlepšie je používať typ char, ktorý je v skutočnosti číselná premenná s rozsahom -128 až 127. Každé z týchto čísel má priradený nejaký znak podľa ASCII `http://sk.wikipedia.org/wiki/ASCII`.

Napríklad 64 je `'A'`, ale takéto veci si nemusíme pamätať, lebo so znakmi sa dajú robiť matematické operácie ako s číslami. Napríklad `'a'+1 == 'b'`.

Ak chceme znak vypísať ako znak, tak použijeme značku `%c`.

Cvičenie 17: Skúste si spustiť nasledovné programy a porovnajte ich správanie:

```
#include<cstdio>

int main(){
    char a;
    scanf("%c", &a);
    printf("<%c>\n", a);
}
```

```
#include<iostream>
using namespace std;

int main(){
    char a;
    cin >> a;
    cout << "<" << a << ">" << endl;
}
```

Riešenie 17: Prvý program načítava všetky riadky, teda aj biele znaky (medzery, tabulátory, konce riadkov). Druhý program biele znaky preskakuje. V prípade, že by sme v prvom programe napísali miesto `scanf("%c", &a);` `scanf(" %c", &a);` mal by sa správať rovnako ako druhý.

Pole znakov, teda `char meno[]`, môžeme nazývať reťazec. Napríklad `"jesko"` je reťazec a zároveň je to pole `{'j', 'e', 's', 'k', 'o', '\0'}`. `'\0'` je znak s hodnotou 0 a v C++ sa ním ukončujú všetky reťazce.

Všetky funkcie, ktoré robia niečo s reťazcami, končia čítanie reťazca na nulových znakoch. Preto aj príkaz `printf("a\0hoj");` vypíše len `"a"`.

Jeden zo spôsobov ako vyrábať nami definované reťazce je napr. `char retazec[] = "lubovolny text";`

Cvičenie 18: Ako jedným príkazom skrátime reťazec `char str[] = "dlhy retazec";` na dĺžku 3?

Riešenie 18: `str[3] = 0;` alebo `str[3] = '\0';`.

Načítavať a vypisovať reťazce vieme pomocou značky `%s` s tým rozdielom, že pri načítavaní nepíšeme `&`.

```
#include<cstdio>

int main(){
    char a[100];
    scanf("%s", a);
    printf("<%s>\n", a);
}
```

```
#include<iostream>
using namespace std;

int main(){
    char a[100];
    cin >> a;
    cout << "<" << a << ">" << endl;
}
```

Ako si iste všimneme po spustení programov, reťazce sa načítavajú "po slovách", vždy len po najbližší biely znak (medzera, koniec riadku, tabulátor).

Pri reťazcoch (ale aj iných poliach) a operátoroch si treba dávať pozor. Keď `A` a `B` sú polia, tak `A<B` **ne**vráti `true` v prípade, že A obsahuje menšie čísla. Rovnako aj `A == B` **ne**testuje, či polia obsahujú rovnaké čísla. Ani `A + B` **ne**sčíta obsahy polí ani `A = B` **ne**prekopíruje čísla z jedného do druhého. Jednoducho operátory robia s poľami celkom odlišné veci.

Je to spôsobené tým, že hodnota premenných `A` a `B` sú smerníky do pamäte a operátory pracujú len s týmito smerníkmi.

Reálne čísla

Okrem celých čísel máme v C++ aj reálne. Pre ne máme typ `double`, načítavame a vypisujeme ich pomocou značky `%lf`, pri výpise vieme ovplyvniť počet vypísaných desatinným miest `%<pocet>lf` ako vidíme v príklade:

```
int main(){
    double a;
    scanf("%lf", &a);
    printf("%2lf\n", a);
    cin >> a;
    cout << a << endl;
}
```

Keď chceme napísať nejakú reálnočíselnú hodnotu, môžeme buď pretypovať `double(7)`, alebo napísať desatinnú bodku `7.0`.

Pri práci s reálnymi číslami si musíme dávať obrovský pozor na presnosť. Napríklad `(5.1 - 3.1 - 2.0 == 0.0)` vráti `false`. Počas výpočtov totiž dochádza k drobným zaokúhľovaniam.

Ilustrovať si to vieme nasledovným príkladom: majme kalkulačku, ktorá počíta v desiatkovej sústave a má presnosť 3 desatinné miesta. V matematike $1-1/3-1/3-1/3 = 0$, ale v kalkulačke $1.000-0.333-0.333-0.333 = 0.001 \neq 0.000$.

V skutočnosti však double nemá presnosť na nejaký počet desatinných miest, ale na nejaký počet cifier. Keby naša kalkulačka mala presnosť na 4 cifry, tak 543210 by sa zaokrúhlilo na 543200. Takže keď používame doubley, tak nastávajú nepresnosti aj v celých číslach.

Podobne napríklad vďaka tomu vieme nájst nenulové a, b také, že $a + b = a$. Napríklad $1000 + 0.01 = 1000$. Takisto výsledok $a + b + c$ môže závisieť od poradia sčiťovania. $(1000 + 0.4) + 0.4 = 1000 + 0.4 = 1000$, ale $1000 + (0.4 + 0.4) = 1000 + 0.8 = 1001$.

`double` ukladá 52 binárnych cifier, teda približne 15 desiatkových. Pri väčšom počte operácií presnosť výsledku klesá.

Najväčšie číslo, aké je `double` schopný reprezentovať je približne 10^{300}.

Dôležité je nepoužívať `a == b` ale miesto toho napríklad `abs(a-b)<1e-8`. `1e-8` je zhruba $0.1^8 = 0.00000001$. `abs` je funkcia, ktorá vracia absolútnu hodnotu (odstráni znamienko) a nachádza sa v knižnici `cmath`.

Matematika

V knižnici `cmath` sa nachádza aj zopár ďalších zaujímavých funkcií. Napríklad `sqrt(x)` $= \sqrt{x}$, `exp(x)` $= e^x$, `log(x)` je prirodzený logaritmus x, pozná aj goniometrické funkcie `sin(x)`, `cos(x)` a mnoho ďalších, ktoré nájdete na `http://www.cplusplus.com/reference/cmath/`. Všetky pracujú s reálnymi číslami.

```
#include<cstdio>
#include<cmath>

int main(){
    double d;
    scanf("%lf", &d);
    printf("%lf\n", sqrt(d));
}
```

Vlastné typy premenných

V C++ si môžte vyrábať vlastné typy premenných. Napríklad, ak vás nebaví písať veľakrát `long long`, ale radšej by ste písali `ll`:

```
typedef long long ll;
int main(){
    ll a;
}
```

Alebo ste zistili, že existujú dvojice – tie nájdete v knižnici `algorihtm` a volajú sa `pair`. Dvojica `int`ov je `pair<int, int>`, trojica `char`u `int`u a `char`u je napríklad `pair<char, pair<int, char> >`. A to už je celkom dlhý výraz na písanie, takže je fajn použiť:

```
typedef pair<int, int> pii;
typedef pair<int, char> pic;
typedef pair<char, pic> pcic;

int main(){
    // takto sa dvojice vyrabaju
    pii dvainty = pii(7, 8);

    // alebo takto
    pic intaznak = make_pair(10, 'a');

    // da sa aj kombinovat
    pcic troica = make_pair('b', pic(11, 'c'));

    // takto sa pristupuje ku prvkom
    dvainty.second = intaznak.first;
    troica.second.first = dvainty.second;
    intaznak = troica.second;
```

```
    //teraz si predstavte, ake by to bolo neprehladne bez typedefu
}
```

Dajú sa vyrábať aj zložitejšie štruktúry pomocou `struct`. Tu je jednoduchý príklad, keď chcete vedieť viac, trocha pogooglite. Všimnite si, že za definíciu `struct` sa píše bodkočiarka.

```
struct Stvorica{
    int a, b, c, d;
    char e, f;
};

int main(){
    Stvorica s;
    s.a = 7;
    s.b = s.a + 8;
    s.e = s.f = '.';
}
```

Makrá a preprocesing

Pred tým, ako sa váš program skompiluje, robí sa časť nazývaná preprocesing. V tomto štádiu, sa dejú všeliaké podivnosti s riadkami, ktoré začínajú znakom `#`. Týmto znakom sa píšu príkazy pre preprocesor.

Z jedným z takýchto príkazov sme sa už stretli, volá sa `#include`. Tento príkaz nakopíruje do náško kódu obsah hlavičky knižnice – teda napr. deklarácie funkcií (ako napr `printf` v `cstdio`, či `cout` v `iostreame`), nie však definície, tie tam netreba. Pre nás tento príkaz nemá nejaký hlbší význam.

Ďalším príkazom je `#define`. Tento príkaz sa dá použiť napríklad na definovanie konštánt. Toto môže mať výhodu prehľadnejšieho kódu a zároveň, keď sa rozhodneme zmeniť hodnotu, stačí nám zmeniť ju na jednom mieste. To čo vyrobíme príkazom `#define` nazývame makro.

```
// preprocesor odstrani aj komentrare
#define MAXN 1047
#define INF 1023456789
#define eps 1e-8

int main(){
    int A[MAXN];
    for(int i = 0; i<MAXN; ++i)
        A[i] = INF;
}
```

Tento program sa po preprocessingu zmení na

```
int main(){
    int A[1047];
    for(int i = 0; i<1047; ++i)
        A[i] = 1023456789;
}
```

a až potom sa kompiluje.

Príkaz `#define` vieme použiť aj na definíciu makier s parametrami. Tieto nám takisto môžu zjednodušiť prácu. Treba si však dávať pozor pri práci s nimi, ako uvidíme na príklade makier `rozdiel1`, `rozdiel2` a `rozdiel3`. Skôr než si pozriete zvyšok kódu, zamyslite sa, prečo sú makrá `rozdiel1` a `rozdiel2` zlé.

```
#define rozdiel1(a,b) a-b
#define rozdiel2(a,b) (a-b)
#define rozdiel3(a,b) ((a)-(b))
//namiesto predoslych makier sa daju pouzit funkcie
//oproti funkciam ma makro vyhodu, ze funguje bez ohladu na typ premennej

//nasledovnu konstrukciu uz funkciou spravime len tazko
#define FOR(i, n) for(int i = 0; i<(n); ++i)

int main(){
    int a = 0, b = 0, c = 0, d = 0;
    // pre toto je makro rozdiel1 zle:
    // namiesto (a-b)*c budeme mat a - b*c;
    a = rozdiel1(a, b)*c;

    // pre toto je makro rozdiel2 zle:
    // namiesto a-(b+c) budeme mat a - b + c;
    a = rozdiel2(a, b+c);

    // konecne spravne makro:
    a = rozdiel3(a, b+c)*d;

    int celkovyrozdiel = 0;
    FOR(i,n) FOR(j, n) {
        celkovyrozdiel = rozdiel3(i,j);
    }
    // na zamyslenie: aka bude hodnota celkovyrozdiel teraz?
}
```

Poslendé, čo si o preprocesingu ukážeme, sú `ifdefy`. Jednoducho sa vďaka ním dajú vypínať kusy kódu. Napríklad vypínateľné debugovacie výpisy:

```
#include<iostream>
using namespace std;

// skus zakomentovat nasledujuci riadok, co sa zmeni?
#define DEBUG

#ifdef DEBUG
#define DBG(x) cout << #x << ": " << (x) << endl;
#else
#define DBG(x)
#endif

int main(){
    int a = 7;
    DBG(a);
    DBG(a+7);
}
```