Algoritmy a programování I – řešení nebo nástin řešení některých úloh ze cvičení

Obsah

1 Vývojové diagramy

1.1 Jak na vývojový diagram

Cílem této sekce je nacvičit si zápis a zakreslení základních prvků vývojového diagramu. Postupně se dostáváme také k zápisu jednotlivých prvků programového kódu, který budeme používat přímo při programování.

Úkol – zadání:

Podívejte se na web
https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/U%C4%8Debnice/Algoritmus/V%C3%BDvojov%C3%A9_diagramy

Projděte si všechny tam uvedené příklady a okomentujte. Jsou tam ukázky jak správných, tak i chybných diagramů, a to jak vývojových, tak i UML.

Komentář k úkolu:

Projděte si opravdu všechny diagramy na dané stránce – vývojové i UML. Druhý diagram v pořadí nepředstavuje algoritmus – proč? Nemá začátek ani konec, nemůže skončit po konečném počtu kroků, není deterministický (jednoznačný).

Zajímavý je i „Záhadný diagram“. Vyzkoušejte si průchod tímto diagramem na několika dvojicích různých čísel. Příslušný algoritmus postupně odečítá od většího čísla to menší (a výsledek uloží do původního většího), dokud do obou proměnných nedostane totéž číslo. To následně vypíše.

1.2 Jednoduché úlohy

Úkoly – zadání:

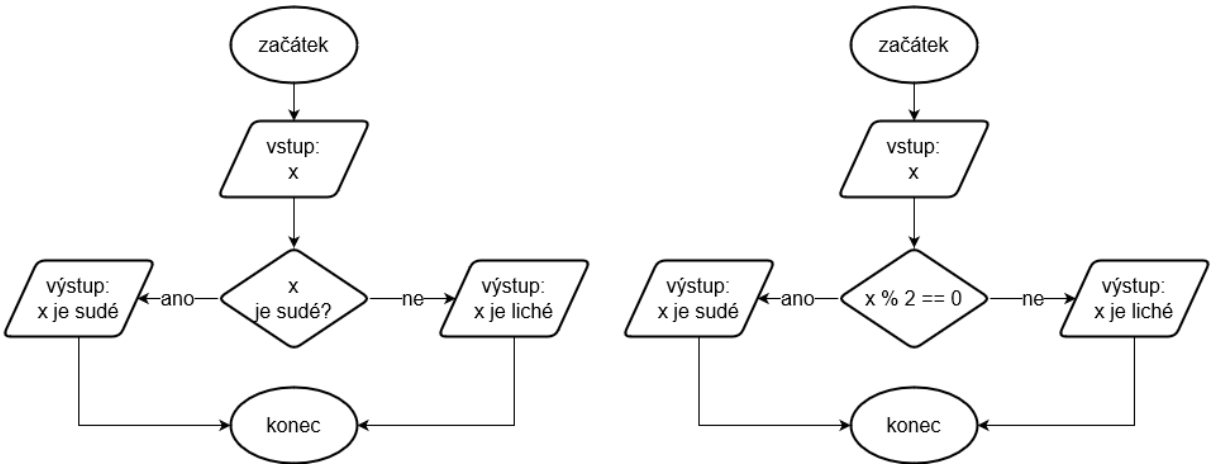
Navrhněte vývojové diagramy, které provádějí následující:

1. Od uživatele načtěte číslo. Zjistěte, zda je sudé nebo liché a vypište zjištěný výsledek.
2. Od uživatele načtěte číslo. Zjistěte, zda je dělitené pěti. Použijte operaci modulo, která vrací výsledek po dělení daným číslem – pro dělení pěti: (x % 5) == 0. Výsledek vypište.
3. Od uživatele načtěte dvě čísla. Vyměňte jejich obsah (můžete použít pomocnou proměnnou) a pak vypište výsledek.
4. Předchozí úkol trochu pozměníme: vymyslete, jak se to dá udělat bez pomocné proměnné. Nápověda: použijte operace sčítání a odčítání, budou to celkem tři operace.
5. Načtěte od uživatele číslo a zjistěte, zda leží v intervalu $(8\,;20\rangle$. Pozor na hranice intervalu.
6. Navrhněte kalkulačku, která bude umět sčítat, odčítat, násobit a dělit dvě čísla zadaná uživatelem. Nezapomeňte na kontrolu dělení nulou. Načtěte nejdřív jedno číslo, pak operátor, podle něj dělte kód postupně několika diamanty, následně načtěte druhé číslo a proveďte operaci.

Nástin řešení některých úloh:

První úloha je jednoduchá – načteme vstup od uživatele, pak rozhodneme (zda je číslo sudé či liché) a podle výsledku rozhodnutí vypíšeme výsledek. Nicméně diagram může mít dvě podoby – buď více abstraktní (do diamantu dáme otázku, zda je sudé) nebo více matematickou. Číslo je totiž sudé, pokud

při celočíselném dělení dvěma je zbytek 0. Pokud chceme získat zbytek po celočíselném dělení, použijeme operátor „modulo“, který se většinou zapisuje symbolem procenta. Pozor na rozdíl v použití znaku „=“: pokud porovnáváme, jsou dvě rovnítka za sebou. Následují obě možnosti, jak diagram zapsat:



Druhý úkol je variací toho, co vidíme výše v druhém diagramu, jen místo čísla 2 by bylo číslo 5 a jiný výstup.

Nyní k úlohám 3 a 4. Třetí je jednoduchá – načteme od uživatele dvě čísla a s pomocí třetí proměnné přehodíme jejich obsah, k čemuž potřebujeme celkem tři přiřazení (předpokládejme, že čísla od uživatele jsou v proměnných x a y):

```
pom = x;
x = y;
y = pom;
```

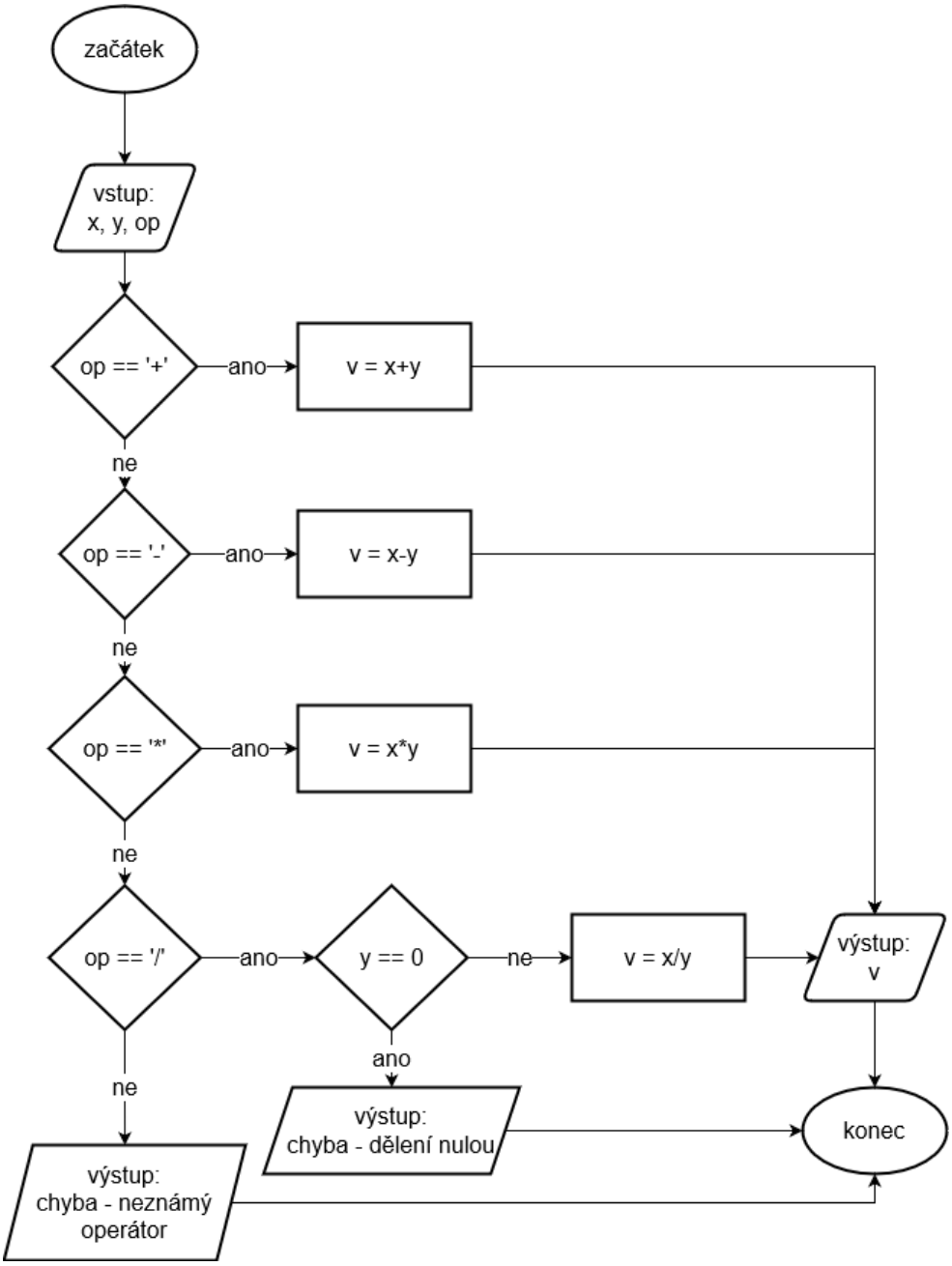
Důležité je hlavně to, aby se žádná z hodnot „neztratila“. Pokud jako první použijete x = y, tak máte smůlu.

V úloze 4 využijeme fakt, že když dvě čísla sečteme, ve výsledku jsou „uschovány“ obě hodnoty. Dokud máme k dispozici jednu z původních hodnot, dokážeme ze součtu určit tu druhou. Následující tabulka ukazuje, jak se postupně bude měnit obsah obou proměnných. Na každém řádku se pro danou operaci využijí hodnoty proměnných z předchozího řádku. Všimněte si, že v jakémkoliv mezikroku výpočtu máme v jedné z proměnných jednu z původních hodnot a v té druhé součet původních hodnot.

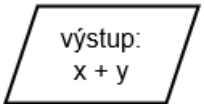
operace	obsah první proměnné po operaci	obsah druhé proměnné po operaci
	X	Y
X = X + Y	X + Y	Y
Y = X – Y	X + Y	(X + Y) – Y = X
X = X – Y	(X + Y) – X = Y	X

Pátý úkol je opět jednoduchý: číslo načtené od uživatele musí být větší než 8 a menší nebo rovno 20. Takže výpočet půjde postupně přes dva diamanty.

Šestý úkol znamená, že si musíme ujasnit, jak poskládat celou řadu rozhodovacích prvků, a také ošetřit případné chyby. Co se týče poskládání, ve vývojovém diagramu si vystačíme s rozhodováním typu ano/ne, třebaže v programovacích jazycích obvykle máme i jiné mechanismy. Co se týče ošetření chyb, tak samozřejmě musíme zamezit dělení nulou, a také případu, že uživatel měl zadat jeden ze čtyř určených operátorů, ale místo toho zadal jiný symbol.



Alternativně můžeme místo obdélníků s výpočty rovnou dát výstup výsledku výpočtu, pak bychom vůbec nepotřebovali proměnnou „v“, například pro sčítání:



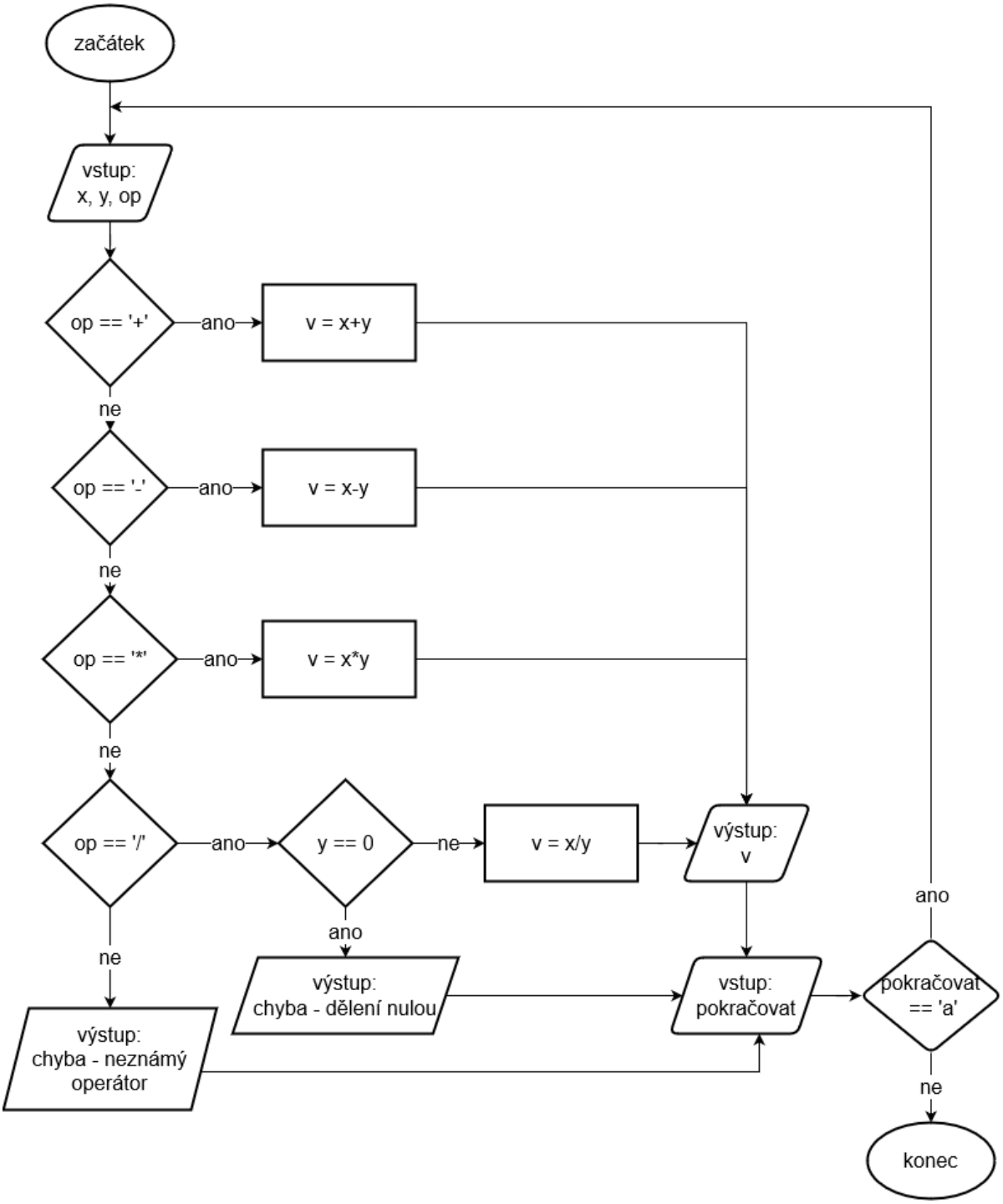
1.3 Cykly

Úkol – zadání:

Upravte příklad s kalkulačkou tak, aby se na konci výpočtu zeptala uživatele, zda chce skončit. Pokud souhlasí (například klepne na tlačítko „Ano“ nebo napíše písmeno „a“), přejděte na konec. Jestli ne, přesuňte se opět k načtení čísel a operátoru a začněte nové kolo.

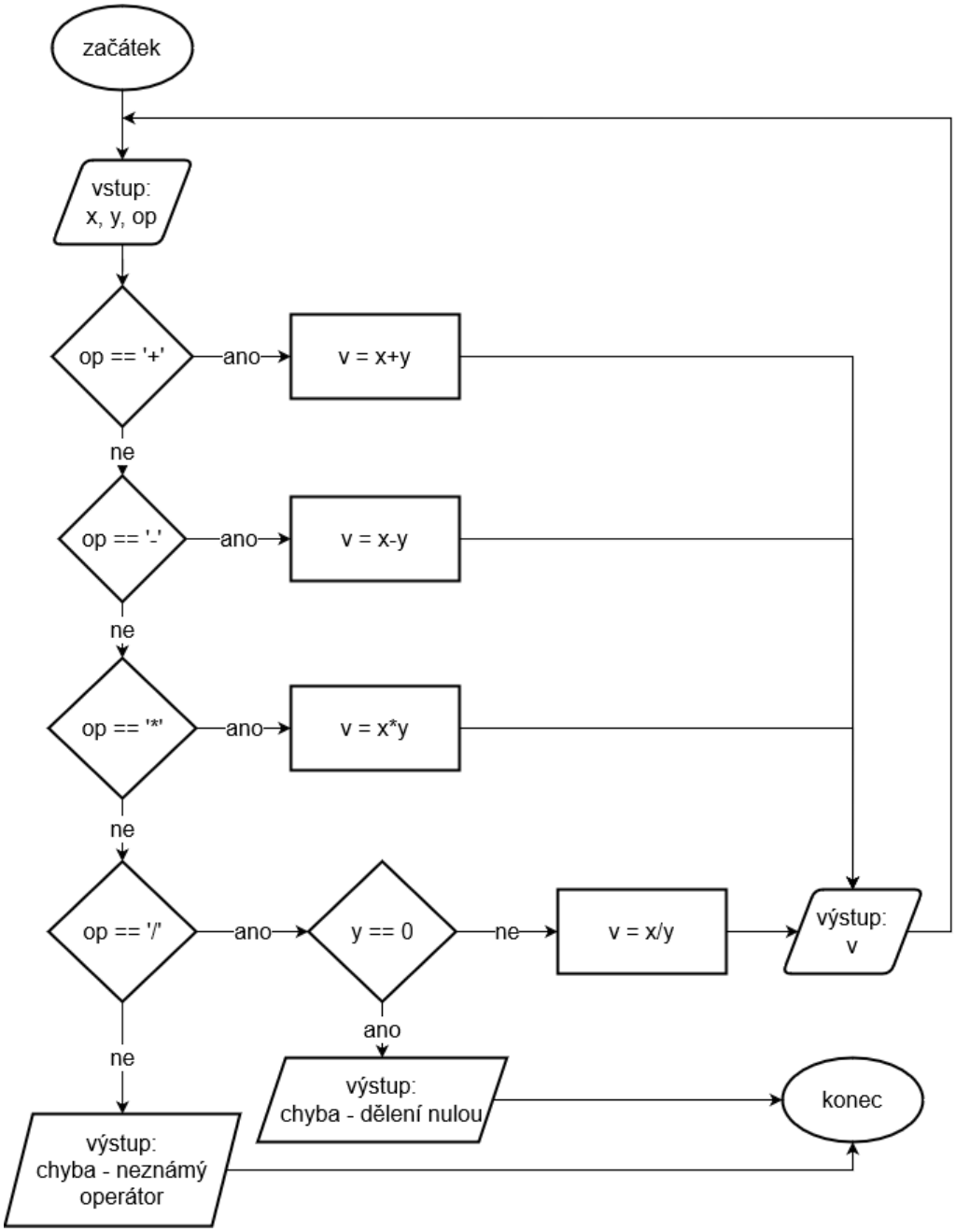
Možnosti řešení:

Nejdřív si ukážeme, jak by to vypadalo přesně podle zadání. Oproti předchozímu úkolu přidáme možnost rozhodnutí po výpisu výsledku nebo chyby.



Od uživatele načteme písmeno, a pokud je to ‘a’, přejdeme opět na začátek. Tím vytvoříme cyklus s podmínkou na konci, což znamená, že tělo cyklu se provede alespoň jednou.

Zadání by mohlo být i trochu pozměněno: pokud uživatel zadá buď chybný operátor (nebo místo něj například jen klepne na Enter) nebo nastane jiná chyba (například operátor je dělení a druhý operand je číslo 0), cyklus končí. Pak by stačilo drobně pozměnit diagram vytvořený na konci předchozí sekce – od výstupu chybových hlášení by vedla cesta do konce, kdežto od výpisu výpočtu by se přecházelo na začátek k zadávání hodnot pro další průchod cyklem.



Úkol – zadání:

Sestrojte vývojový diagram pro výpočet faktoriálu pomocí cyklu. Nejdřív si napište vzorec, určete, co konkrétně se bude provádět opakovaně, jaká bude ukončující podmínka cyklu, jaké proměnné budete potřebovat, jak provést výpočet, jaké jsou hraniční podmínky (kdy do cyklu vůbec nejít). Až budete mít diagram hotový, ověřte si, zda pracuje na různé vstupy korektně (včetně nuly).

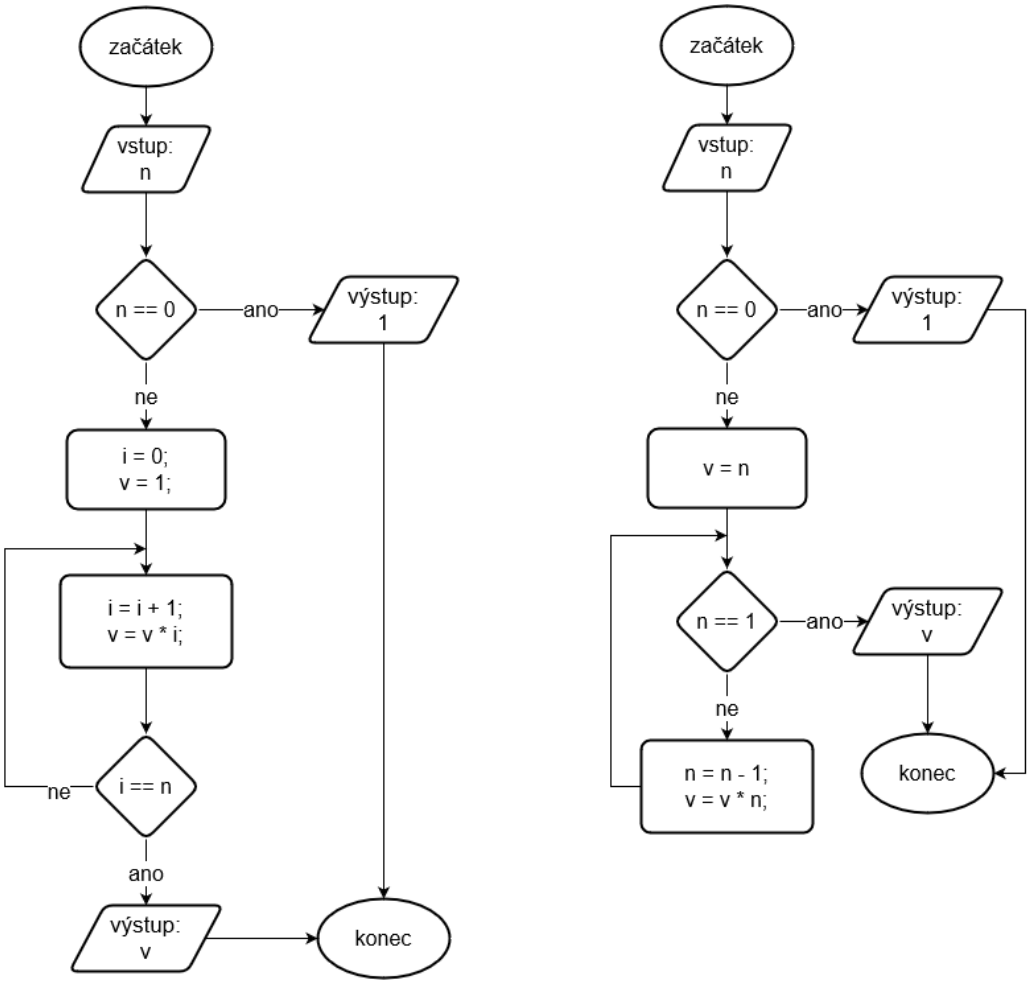
Řešení:

Jak víme, vzorec pro výpočet faktoriálu je následující:

```
N! = N * (N – 1) * (N – 2) * … * 1  =  N * (N – 1)!
0! = 1
```

Tedy násobíme určitý počet čísel (ten počet známe předem, je roven číslu, z něhož počítáme faktoriál), a taky víme, že jde o aritmetickou posloupnost, kde rozdíl mezi sousedy je 1. Existuje více způsobů

řešení. My si ukážeme dva z nich: budeme postupně násobit N čísel s využitím pomocné proměnné, kterou v každém kroku buď zvýšíme o 1 nebo snížíme o 1 (podle toho, z které strany intervalu začneme).



Výše vidíme dvě z možných řešení. V prvním případě při násobení bereme postupně čísla od 1 do n, v druhém případě jdeme opačným směrem od nejvyšším k jedničce. Druhé řešení se zdá optimálnější, protože potřebujeme o jednu proměnnou méně a v případě jedničky už k násobení nedochází. Je zde ještě jeden rozdíl – všimněte si, že v prvním případě používáme cyklus s podmínkou na konci, v druhém případě cyklus s podmínkou na začátku.

Další možnost: pokud na začátku zjistíme, že uživatel zadal číslo 0, můžeme ho přepsat na 1 (což nezmění výsledek) a „pustit“ do následujícího výpočtu. Pokud bychom pro zbytek použili postup z druhého diagramu, bude cyklus ukončen prakticky hned na začátku a vypíše se výsledek.

V diagramech vidíme, že často používáme drobnou úpravu proměnné typu přičtení, odečtení atd. hodnoty k proměnné, příp. přičtení či odečtení jedničky. Pro tyto operace existuje jednodušší zápis:

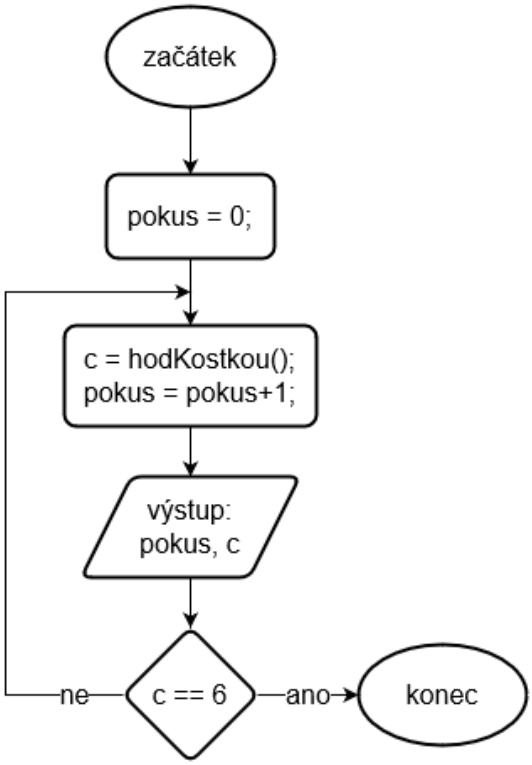
„Běžný“ zápis	**Zkrácený zápis**
`i = i+1`	`i++`
`i = i-1`	`i--`
`v = v * i`	`v *= i`
`m = m + k`	`m += k`

Úkol – zadání:

Simulujte házení kostkou tak dlouho, dokud nepadne 6. Vypisujte jednotlivé hody (vždy číslo pokusu a hozenou hodnotu).

Řešení úkolu:

Protože potřebujeme, aby cyklus proběhl alespoň jednou (hodíme nejméně jednou, jinak bychom tu šestku ani nemohli získat), zvolíme cyklus s podmínkou na konci. Házení kostkou bychom sice mohli ještě rozepsat (vezmi kostku do ruky, protřepej, pusť,…), ale až tak to nebudeme komplikovat a použijeme abstrakci.



Pokusy počítáme v proměnné „pokus“. Je třeba dát pozor na to, abychom tuto proměnnou na začátku inicializovali, tedy dali jí počáteční hodnotu. Tu zvolíme tak, aby „nenabourala“ zbytek výpočtu – na začátku ještě nebyl žádný hod, tedy pro inicializaci použijeme nulu.
Říkáme, že nula je neutrálním prvkem vzhledem ke sčítání (a odčítání). Pokud by šlo o operaci násobení nebo dělení, neutrálním prvkem by bylo číslo jedna.

Úkol – zadání:

Načtěte od uživatele číslo N. Následně načtěte N čísel a vypočtěte z nich průměr. Výsledek vypište. Podle čísla N určete počet kroků (kolik čísel načíst). Číslo N může být jakékoliv nezáporné číslo. Pohlídejte si, aby se algoritmus choval korektně při jakémkoliv vstupu. Nezapomeňte, že nulou nelze dělit.

Řešení úkolu:

Průměr se počítá tak, že všechna čísla sečteme a vydělíme jejich počtem.

K tomuto úkolu jen pár poznámek. Potřebujeme:

- proměnnou, do které načteme počet čísel a proměnnou, do které budeme postupně načítat ta čísla, z nichž se má vypočíst průměr (stačí jedna, načteme do ní číslo, zpracujeme a pak do téže proměnné můžeme načítat další číslo),
- proměnnou, do které postupně budeme přičítat čísla, ze kterých má být vypočten průměr (pozor, je třeba ji inicializovat – na 0, protože ta je neutrální vzhledem ke sčítání),
- proměnnou, přes kterou si pohlídáme, kolikrát už cyklus proběhl (čítač cyklu), tuto proměnnou samozřejmě taky musíme inicializovat.

Je třeba ošetřit případ, že jako počet čísel zadá uživatel 0. Proto hned na začátku otestujeme, co nám zadal, a pokud nulu, „vynadáme“ mu a ukončíme výpočet. Jinak by mohlo dojít k dělení nulou.

Po inicializaci proměnné pro součet a čítače cyklu následuje cyklus, v němž vždy

- načteme číslo od uživatele,
- přičteme do proměnné pro součet,
- čítač zvýšíme o jedničku a zkontrolujeme, jestli už cyklus nemá končit (tady je to variabilní, pohlídejte si, jakou hodnotu při kontrole má čítač mít v prvním průchodu cyklu vzhledem k tomu, kdy ji zvyšujete).

V cyklu jsme všechny vstupy sečetli, zbývá je vydělit počtem vstupů a vypsat výsledek.

Úkol – zadání:

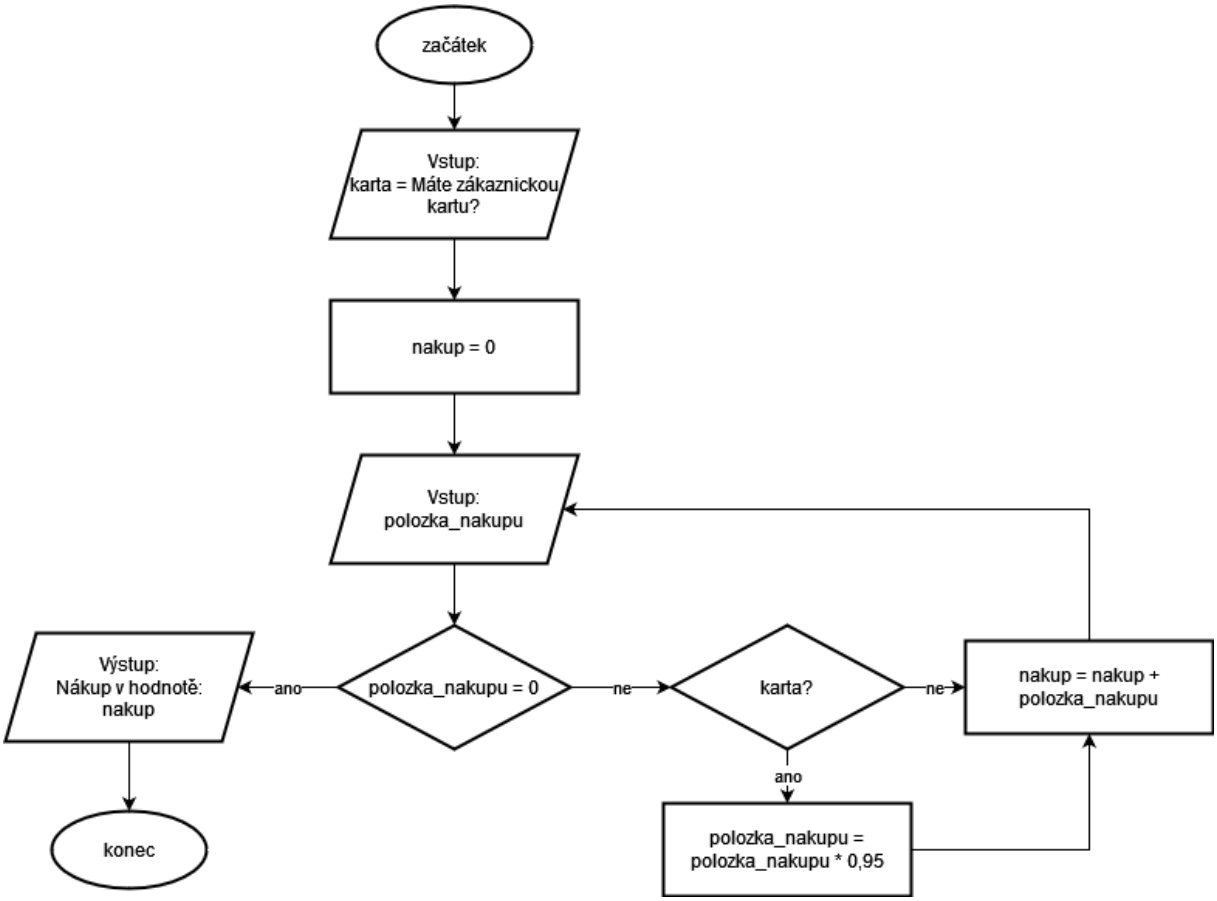
Pracujete u pokladny v obchodě, který při předložení zákaznické karty poskytuje slevu 5 %. Sestavte vývojový diagram, který nejdřív zjistí, zda má klient zákaznickou kartu. Jestliže ano, zapamatuje si to. Následně načítá ceny nakupovaného zboží, pokud má být udělena sleva, tak cenu sníží o danou slevu a následně cenu vypíše. Končí tehdy, když je načtena cena 0.

Řešení úkolu:

Zadání je poměrně volné, řešení by mohlo mít více variant. Taktéž nemáme k dispozici konkrétnější informace o situaci a vybavení – úkoly ze zadání jsou prováděny částečně pokladnou a částečně člověkem, který tu pokladnu obsluhuje, případně čtečkou čárových kódů apod. Následuje jedna z možností řešení – slovní popis a vývojový diagram, který bohužel přetekl na další stranu.

Nejdřív požádáme o informaci, zda má zákazník kartu, uložíme do proměnné karta a pak spustíme cyklus načítání cen. Uvnitř cyklu nejdřív ověříme, jestli nebyla zadána nula (pak vypíšeme celkovou cenu nákupu a ukončíme program), a pokud je cena nenulová, přičteme buď celou nebo sníženou (v případě, že zákazník má kartu) k ceně nákupu.

Všimněte si, že celková cena nákupu byla inicializována na 0 (tedy počáteční hodnota nákupu je 0).


začátek
Vstup:
karta = Máte zákaznickou kartu?
nakup = 0
Vstup:
polozka_nakupu
polozka_nakupu = 0
ano
Výstup:
Nákup v hodnotě:
nakup
konec
ne
karta?
ano
polozka_nakupu =
polozka_nakupu * 0,95
ne
nakup = nakup +
polozka_nakupu

2 Pseudokód

Úkol:

Pokuste se v pseudokódu (s vhodnou mírou abstrakce) napsat postup, jehož část byla naznačena v prvním ukázkovém vývojovém diagramu – procházení nočního hlídače areálem a kontrola, zda někdo nezapomněl zhasnout. Zamyslete se nad tím, jak by vypadalo procházení celého patra s jednou chodbou, na které je řada dveří. Pak se to pokuste rozšířit na celou budovu s tím, že nevíte předem, kolik má pater (můžete například použít konstrukci

```
pokud jsem v nejvyšším patře, tak …, jinak …
```

Úkol:

V sekci o cyklech je úkol o házení kostkou. Tento úkol přepište do pseudokódu.

3 Programovací jazyky a nástroje

Budeme začínat s jednoduchými programy, pro které nám stačí online překladače. Projděte si ty, na nějž najdete odkazy ve studijním materiálu. Vyberte si ten, který vám nejlépe vyhovuje. Je o webové aplikace, kód je většinou jen interpretován, což zatím nevadí. Čeho si všímat:

- jak se pracuje s editorem, zda je přehledný (barevné vysvícení syntaxe apod.), kde najdeme tlačítko pro spuštění (většinou Run nebo Execute), kde nám spuštěný program bude vypisovat výstup,
- jaké další možnosti aplikace nabízí, například jestli existuje možnost krokování programu (může být například tlačítko „Step“), zda můžeme vytvořený zdrojový kód taky uložit do souboru (nicméně pokud ne, stačí text zkopírovat a uložit „ručně“),
- jak reaguje, zda správně načítá vstupy (to uvidíme později).

Pokud chcete a máte možnost, můžete si nainstalovat vlastní vývojové prostředí.

4 Začátek

Úkol – zadání: vyzkoušejte – jaký dostanete výsledek?

```
int x = 25, y = 30;
double vysl;
vysl = x/y;          NEBO       vysl = (double) x/y;
cout << vysl;                   cout << vysl;
```

Vysvětlení: pokud jste vyzkoušeli (klidně v online editoru), narazili jste na problém nesourodosti datových typů, přesněji automatického přetypování. Pokud v C/C++ dělíte dvě celá čísla, překladač automaticky předpokládá, že budete chtít jako výsledek taky celé číslo. Proto je třeba výsledek ručně přetypovat na double.

Úkol – zadání: vyzkoušejte:

```
const int PI = 3.14159;
cout << PI;
```

Co je špatně? Ano, dávejte si pozor, jaký datový typ použijete. Opravte a znovu přeložte.

Vysvětlení: ano, v deklaraci konstanty je celočíselný datový typ místo float nebo jiného pro plovoucí řádovou čárku. Překladač opět automaticky přetypovává, aby číslo bylo možné uložit do celočíselného datového typu.

Úkol - zadání:

Napište program, který si od uživatele vyžádá postupně jeho jméno, příjmení, ulici, číslo popisné, město a PSČ (na konec každé výzvy k zadání prvku dejte tabulátor), a pak po dvou vynechaných řádcích vypište všechny tyto údaje ve vhodné formě jako na dopisní obálku (na první řádek jméno a příjmení, na druhý ulici a číslo, na třetí PSČ a město).

Pozor – při načítání do datového typu string pomocí cin je problém s mezerami. Pro jistotu volte název ulice bez mezer. Taktéž si pohlídejte, aby mezi dvěma prvky na tomtéž řádku byla mezera.

Řešení:

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

int main()
{
   string jmeno, prijmeni, ulice, mesto, psc;
   short cislo;

   cout << "Zadej jmeno:\t";     cin >> jmeno;            // načtení vstupních
   cout << "Zadej prijmeni:\t";  cin >> prijmeni;         // hodnot
   cout << "Zadej ulici:\t";     cin >> ulice;
   cout << "Zadej cislo popisne:\t"; cin >> cislo;
   cout << "Zadej mesto:\t";     cin >> mesto;
   cout << "Zadej PSC:\t";       cin >> psc;

   cout << endl << endl;
```

```
   cout << jmeno << ' ' << prijmeni << endl;        // výpis ve formě adresy
   cout << ulice << ' ' << cislo << endl;           // na dopisu
   cout << psc << ' ' << mesto;

   return 0;
}
```

Úkol – zadání:

Totéž jako v příkladu naprogramujte s použitím knihovny stdio.h. Funkce pro náhodná čísla budou fungovat stejně, a poslední příkaz (počkání na stisk jakékoliv klávesy) bude následující:

```
getchar();
```

Řešení:

```
#include <stdio.h>
int main()
{
   srand(time(0));
   printf("%d\n", rand());
   getchar();
}
```

5 Výrazy a podmínky

5.1 Konstanty

Úkol:

Kódy barev se často zapisují v šestnáctkové soustavě, protože taková čísla lépe „napasují“ do stanoveného počtu bitů: hexadecimální číslo 0xFF je horní hranicí toho, co lze v šestnáctkové soustavě vměstnat do dvou číslic, a zároveň po přeložení do binární soustavy maximálně využijeme 8 bitů (binárně to je 11111111).

Předpokládejme, že píšeme program, který má vygenerovat HTML kód (ano, kód může generovat, tedy vytvářet, jiný kód). Chceme, aby kromě jiného na výstupu bylo:

```
<font color="#xxxxxx">yyyyyyyyy</font>
```

Místo xxxxxx bude číselné označení barvy v hexadecimálním tvaru, místo yyyyyyyyy bude řetězec, který se tou barvou vypíše. Pro určení červené, zelené a modré složky barvy použijeme proměnnou, protože tutéž barvu chceme využít na více místech v kódu, a zároveň si chceme ponechat možnost ji ovlivňovat bez toho, abychom museli kvůli změně barvy procházet celý kód.

Daný úsek kódu může vypadat třeba takto:

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

int main() {
   int cervena=0x1a, zelena=0x1a, modra=0x8f;
   string hlaska = "Setting text colors";

   cout << "<font color=\"";
   cout << '#' << hex << cervena << zelena << modra;
   cout << "\">";
   cout << hlaska << "</font>";
}
```

Všimněte si, že i pro výstup je použita hexadecimální forma (manipulátor). Dále se podívejte na místa, kde se na výstup vypisují uvozovky. Jak zajistíme, aby se na výstup opravdu zapsal symbol uvozovky a aby to přímo v našem programu nebylo bráno jako konec řetězce? Vyzkoušejte. Výstup můžete buď uložit do souboru s příponou .html a zobrazit ve webovém prohlížeči, nebo můžete vyzkoušet v online editoru HTML kódu, například https://www.w3schools.com/html/tryit.asp?filename=tryhtml_intro

(funguje podobně jako online editory kódu v C++, včetně tlačítka Run).

Vysvětlení: Uvedený kód vygeneruje tento řetězec:

```
<font color="#1a1a8f">Setting text colors</font>
```

Vyzkoušejte i jiné barvy podle vlastního vkusu (pro jednotlivé základní barvy je minimum 0x00, maximum 0xff). Pokud všechny tři nastavíte na nulu, dostanete černou. Čím větší hodnota, tím jasnější barva a tím více se blížíte bílé.

Uvozovky: Pokud chcete uvnitř řetězce (vymezeného uvozovkymi) napsat znak uvozovka, musíte před něj dát zpětné lomítko, tedy použít escape sekvenci: `\"`

5.2 Aritmetické operace

Úkol – zadání:

Test, zda je číslo sudé či liché, se dá provést třeba tak, že provedeme bitový součin s číslem 1.

Napište kód, který načte číslo od uživatele a pak pomocí logického součinu uloží do další proměnné číslo 0, pokud uživatel zadal sudé číslo, a 1, pokud zadal liché. Výsledek vypište.

Řešení:

Možná trochu moc „upovídaný“ program může být třeba takový:

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

int main()
{
   unsigned int vstup, vystup;
   cout << "Zadej cislo: ";
   cin >> vstup;
   vystup = vstup & 1;
   cout << "Liche ci sude cislo?\n\t0...sude\n\t1...liche" << endl;
   cout << "Vysledek pro cislo " << vstup << ": "  << vystup;
}
```

Všimněte si využití escape sekvencí \n pro přechod na nový řádek a \t pro tabulátor. Výsledkem je něco na způsob tabulky, výstup programu může vypadat třeba takto (pro číslo 56):

```
Zadej cislo: 85
Liche ci sude cislo?
     0...sude
     1...liche
Vysledek pro cislo 85: 1
```

Úkol – zadání:

Podívejte se na následující kód:

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

int main() {
   unsigned int x = 1;
   for (int i = 0; i<10; i++)          // předbíháme: provede se pro i od 1 do 9
      cout << (x << i) << ' ';
}
```

Pokuste se tuto „záplavu symbolů menší“ rozkódovat, případně vyzkoušejte. Co je výstupem?

Řešení:

Zaměřme se na řádek

```
cout << (x << i) << ' ';
```

Je jasné, že první dvojsymbol `<<` zajišťuje výstup na objekt `cout`. U toho druhého vidíme, že před ním není objekt `cout`, ale číselná proměnná `x`, před kterou je závorka, tedy zde půjde o binární operátor

pro bitový posun vlevo (proměnnou `x` upravujeme, proměnná `i` určuje, o kolik míst se mají bity posouvat). Třetí výskyt dvojsymbolu je opět určen výstupu `cout`.

Závorky jsou zde nutné, bez nich by se prostě vypsal obsah obou proměnných.

6 Podmíněné provedení kódu a větvení programu

6.1 Ternární operátor

Příklad a úkol:

Pokud je proměnná větší než nula, snížíme její hodnotu o 1, jinak vypíšeme chybovou hlášku:

```
x > 0 ? x-- : cout << "chyba, už jsme na nule";
```

Do proměnné `max` chceme načíst tu z hodnot `x,y`, která je větší:

```
max = (x > y) ? x : y;
```

Ternární operátor se moc nepoužívá, ale v jednom případě má určitě smysl – když potřebujeme provést konverzi znaku na malá písmena (nebo velká, to by bylo podobně):

```
c  =  (c >= 'A' && c <= 'Z')  ?  c + ('a'-'A')  :  c;
```

To znamená, že nejdřív zjistíme, zda je ve znakové proměnné velké písmeno (to je tehdy, když spadá do rozsahu od velkého A do velkého Z). Pokud ano, tak provedeme konverzi na malé písmeno, jinak se vrací původní hodnota. Všimněte si, že výsledek se načte do původní proměnné, a to v případě obou „větví“. Pokud by v proměnné bylo původně něco jiného než písmeno (třeba číslice nebo některý jiný „nepísmenný“ znak), skočíme taktéž do druhé větve a konverze se neprovede.

Podobně konverze z malého na velké písmeno:

```
char c;
…  // nějaký kód pracující s proměnnou c
c  =  (c >= 'a' && c <= 'z')  ?  c - ('a'-'A')  :  c;
```

Proč to funguje? Nezapomeňte, že datový typ char je vlastně číslo. Můžete vyzkoušet:

```
cout << c << (int)c;
```

Špatně by bylo následující:

```
c  =  (c >= 'A' && c <= 'Z')  ?  c  :  c - ('a'-'A');
```

Proč je to špatně? V jakých případech by se to chovalo jinak než bychom čekali?

Řešení – vysvětlení:

Vedle vidíme ASCII tabulku pro kódování Windows 1250. Odvodíme si, co je výsledkem rozdílu

```
'a'-'A'
```

ASCII hodnota malého 'a' je 97, ASCII hodnota velkého 'A' je 65. Takže výsledkem rozdílu je `97-65=32`. Pokud bychom tedy přičetli takto získané číslo 32 například k proměnné obsahující písmeno 'M' (které má ASCII hodnotu 77), přičtením čísla 32 bychom dostali číslo 109 a po převodu této ASCII hodnoty na znak by šlo o písmeno 'm'. Takže přičtením rozdílu mezi „áčky“ k velkému písmenu provedeme převod na malé písmeno. A naopak.

Nejdřív se podíveme na to, co je označeno jako správné:

```
c  =  (c >= 'a' && c <= 'z')  ?  c - ('a'-'A')  :  c;
```

Otestujeme obsah proměnné c. Pokud jde o malé písmeno, provede se příkaz mezi otazníkem a dvojtečkou, tedy od původní ASCII

hodnoty se odečte (jak už víme) číslo 32, čímž provedeme převod na velké písmeno. Část příkazu za dvojtečkou určuje, co se má vrátit v případě neplatnosti podmínky (tj. nejde o malé písmeno) – vrátíme tutéž hodnotu, jaká byla v proměnné předtím, tedy neprovede se žádná změna.

A teď to, co je označeno jako chybné:

```
c  =  (c >= 'A' && c <= 'Z')  ?  c  :  c - ('a'-'A');
```

Pokud je v proměnné `c` velké písmeno, neprovede se žádná změna, jinak se odečte číslo 32. Pokud by v naší proměnné bylo původně malé písmeno, provede se konverze na velké. Ano, ale co kdyby to vůbec nebylo písmeno, ale třeba číslice, tečka, čárka, zavináč, otazník,…?

Takže už víme, kde je chyba: konverze by se provedla i v případě, že znak vůbec není písmeno, tento příkaz má nečekaný vedlejší efekt.

6.2 Příkaz IF

Úkol – zadání:

Načtěte od uživatele dvě čísla. To druhé otestujte, a pokud je to nula, vypište chybové hlášení „nulou nelze dělit“. Jinak vypište číslo vzniklé vydělením těchto dvou čísel.

Řešení:

Kratší řešení je následující – vytvoříme dvě proměnné, načteme do nich hodnoty od uživatele a pak otestujeme druhou proměnnou. Použijeme příkaz `if` s oběma větvemi (tj. také `else`), přičemž v každé větvi je jen jeden příkaz, tedy složených závorek netřeba:

```
#include <iostream>
using namespace std;
int main() {
   int a,b;
   cout << "a = "; cin >> a;
   cout << "b = "; cin >> b;
   if (b == 0)
      cerr << "Nulou nelze dělit!";
   else
      cout << "a/b = " << (double)a/b;
}
```

Všimněte si, že pro chybový výstup jsme použili objekt `cerr`.

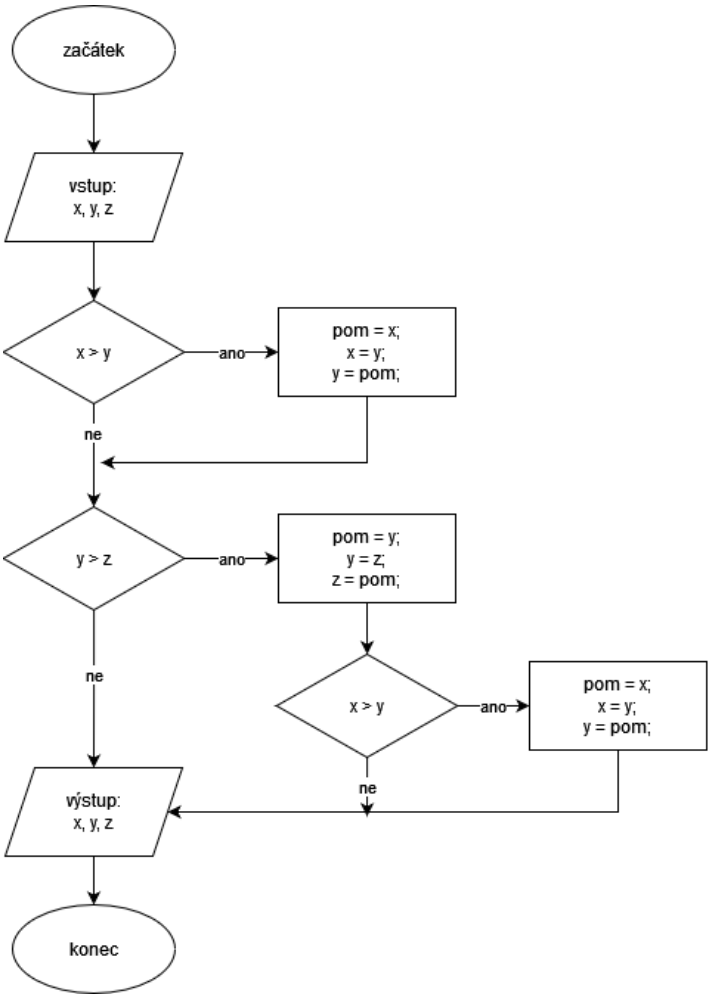
Jiné řešení:

```
#include <iostream>
using namespace std;
int main() {
   int a,b;
   cout << "a = "; cin >> a;
   cout << "b = "; cin >> b;
   if (b == 0) {
      cerr << "Nulou nelze dělit!";
      return 0;
   }
   double vysl = (double) a/b;
   cout << "a/b = " << vysl;
}
```

V čem je rozdíl? Předně jsme odstranili větev `else`. Místo toho jsme do první větve přidali příkaz, kterým končí program. Pak opravdu větev `else` nemá význam, protože následující kód se stejně provede jen tehdy, když podmínka příkazu `if` je vyhodnocena jako `false`. Ovšem teď máme v první větvi dva příkazy, tedy je nutné tuto větev uzavřít do složených závorek. Další změnou je použití proměnné pro výstup, ale v pořádku by bylo i řešení bez proměnné (umístit výpočet přímo do určení výstupu).

Úkol – zadání:

Napište program, který od uživatele načte tři čísla a pak je vypíše v pořadí podle velikosti od nejmenšího. Postupujte podle následujícího vývojového diagramu. Nejdřív si ujasněte, na kterých místech bude `if` s větví `else` a kde bez větve `else`, případně jestli vůbec bude tato větev použita.



Řešení:

Pokud půjdeme přesně podle vývojového diagramu, nebudeme větev `else` vůbec potřebovat:

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

int main()
{
   int x, y, z, pom;
   cout << "Zadej tri cisla:\n";
```

```
   cout << "x = "; cin >> x;
   cout << "y = "; cin >> y;
   cout << "z = "; cin >> z;

   if (x > y) {
      pom = x;
      x = y;
      y = pom;
   }

   if (y > z) {
      pom = y;
      y = z;
      z = pom;
      if (x > y) {
         pom = x;
         x = y;
         y = pom;
      }
   }
   cout << endl << "Byla nactena tato cisla:" << endl;
   cout << x << "\t" << y << "\t" << z;
}
```

Úkol – zadání:

Napište program, který bude řešit lineární rovnici A*X + B = 0. Načtěte od uživatele konstanty A a B, ošetřete případ, kdy A je či není nula (první případ se samozřejmě ještě bude větvit na dva podpřípady – B je buď 0 nebo nenulová hodnota, podle toho máme buď nekonečně mnoho řešení nebo žádné řešení). Může pomoci nakreslení vývojového diagramu.

Řešení:

V tomto příkladu je třeba dát si pozor na vnořování podmínek vedoucích k nekonečnu nebo naopak žádnému řešení.

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

int main()
{
    float a, b;
    cout << "Reseni rovnice A*X + B = 0" << endl;
    cout << "Zadej parametry rovnice." << endl;
    cout << "A = "; cin >> a;
    cout << "B = "; cin >> b;

    if (a == 0) {
        if (b == 0)
            cout << "Rovnice ma nekonecne mnoho reseni.";
        else

            cout << "Rovnice nema reseni.";
    }
    else {
        cout << "X = ";
        cout << -b/a;
```

```
    }
}
```

Zde na nás číhá jistý problém, který si programátor někdy nemusí uvědomit: pokud jako datový typ načítaných čísel zvolíme `int`, pak na konci ve skutečnosti provádíme celočíselné dělení. Přetypování se budeme věnovat později, takže jen upozornění – poslední příkaz by pak vypadal následovně:

```
cout << - (float)b / a;
```

Mezery jsou přidány kvůli přehlednosti. U proměnné `b` jsme explicitně určili, že se s hodnotou má zacházet jako s racionálním číslem, tedy výsledek bude racionální číslo, ne `int`.

7 Cykly a pole

7.1 Jednoduchý cyklus s podmínkou

Úkol – zadání:

Srovnejte tyto úseky kódu (případně také přeložte a spusťte).

```
int i = 0;
while (i < 5) {
   cout << i++ << endl;
}
```

```
int i = 0;
do {
      cout << i++ << endl;
} while (i < 5);
```

```
int i = 0;
while (i < 5) {
   cout << ++i << endl;
}
```

```
int i = 0;
do {
      cout << ++i << endl;
} while (i < 5);
```

Uvnitř všech cyklů máme vždy jen jeden příkaz, tak by tam složené závorky vlastně ani nemusely být, ale je dobré si u cyklů zvyknout raději je psát.

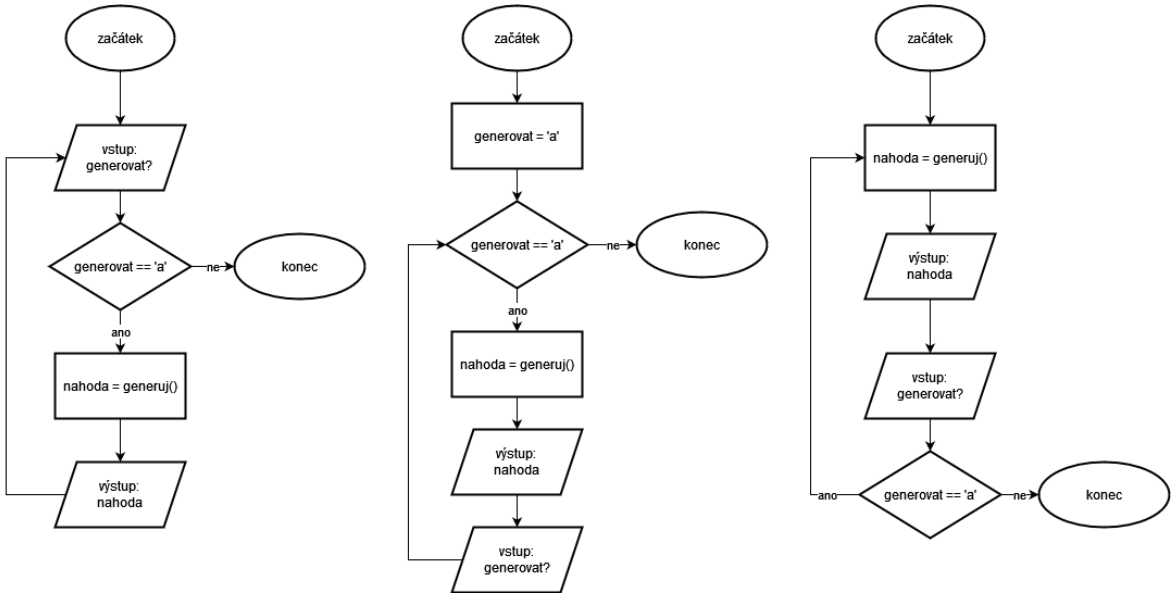
Konstrukci `i++` už známe z vývojových diagramů, tento příkaz přičte k proměnné `i` jedničku. Ale co se stane, když plusy dáme před proměnnou?

Vysvětlení: kámen úrazu je v posloupnosti operací. V jazyce C a C++ totiž každá konstrukce „něco vrací“, i když to na první pohled tak nevypadá.

- Příkaz `i++` (kromě toho, že hodnotu proměnné `i` zvýší o 1, tedy iteruje) vrací číslo uložené v `i` (tedy vypíše na obrazovku), a to ještě před iterací. Tedy nejdřív vrátí hodnotu a pak až iteruje.
- Příkaz `++i` dělá téměř totéž, ale v opačném pořadí. Nejdřív iteruje a pak až vrátí hodnotu proměnné `i` (tedy v našem případě vypíše na obrazovku).

Úkol – zadání:

Pomocí cyklu s podmínkou napíšeme program, který generuje náhodná čísla tak dlouho, dokud bude uživatel chtít. Program skončí, když na dotaz „Generovat další?“ uživatel zadá něco jiného než „a“.



Je několik možností, jak tento úkol provést. Předně – chceme, aby bylo vygenerováno alespoň jedno náhodné číslo? Kdy se budeme uživatele ptát, jestli má být vygenerováno další? Na tom záleží, jestli použijeme cyklus s podmínkou na začátku nebo cyklus s podmínkou na konci. Následují tři vývojové diagramy, z nichž první dva mají podmínku na začátku, třetí má podmínku na konci. Srovnejte první a druhý: který z nich se bude lépe přepisovat do programového kódu? Proč?

Zvolte si jedno z řešení a to naprogramujte. Nezapomeňte, že generátor náhodných čísel je třeba inicializovat (jednou, tedy ještě před cyklem). To ve vývojových diagramech není, protože jsou jen abstrakcí postupu, nemusejí nutně popisovat podrobnosti.

Řešení:

Všechny tři vývojové diagramy mají stejný výsledek – jsou v cyklu generována náhodná čísla a uživatel je pokaždé dotázán, jestli chce pokračovat. Ve všech třech případech proběhne cyklus alespoň jednou.

- V prvním případě jde vpodstatě o cyklus s podmínkou na začátku, ale pro účely naprogramování by potřeboval trochu upravit (ta podmínka je posunuta poněkud dovnitř).
- V druhém případě nemusíme nit měnit, vývojový diagram se dá přímo přepsat na cyklus s podmínkou na začátku.
- V třetím případě jde o cyklus s podmínkou na konci.

Třetí případ je nejoptimálnější, protože vyžaduje nejméně kroků a nejméně kódu. Tak jako tak má cyklus proběhnout vždy nejméně jednou, pro což je cyklus s podmínkou na konci přirozenější.

Oproti vývojovému diagramu je třeba ještě něco přidat (vývojový diagram neobsahuje některé podrobnosti): musíme inicializovat generátor náhodných čísel funkcí srand(…), a také deklarovat potřebné proměnné. Určitě budeme potřebovat proměnnou pro načtení znaku od uživatele, a pak můžeme (nemusíme) mít proměnnou pro načtení náhodného čísla. Zde je varianta, kdy se této proměnné vyhneme – pouze vypíšeme výsledek, který nám vrátí funkce pro generování náhodných čísel:

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

int main()
{
   char g;
   srand(time(0));
   do {
       cout << rand() << endl;  /* jinak bychom výsledek rand() uložili
                                   do proměnné a pak vypsali obsah té proměnné */
       cout << "Pokracovat? ";
       cin >> g;
       cout << endl;
   } while (g == 'a');
}
```

Úkol – zadání:

Napište program, který bude pro celé nezáporné číslo počítat jeho ciferný součet – číslo bude načteno od uživatele. Využijte faktu, že překladač při dělení dvou celých čísel bez přetypování vrací opět celé číslo, tedy provádí celočíselné dělení.

Použijte cyklus s podmínkou na začátku, nejdřív sestavte vývojový diagram. Algoritmus můžete odvodit následovně:

- Nejdřív si promyslete, jak získat hodnotu poslední číslice v čísle. Víme už, jak dosáhnout celočíselného dělení (prostě to nepřetypujeme na double), budeme dělit deseti. Stačí po dělení deseti výsledek vynásobit deseti a odečíst od původního čísla. Získáme hodnotu poslední číslice. Čímž máme návod pro část těla cyklu ve vývojovém diagramu.
- Potřebujeme zjištěný rozdíl (tj. hodnotu poslední číslice) přičíst k výslednému součtu (což znamená, že tento součet musel být někde inicializován).
- Dále sestavte zbytek cyklu. Potřebujeme podmínku a potřebujeme z konce cyklu navázat na jeho začátek, tedy připravit číslo pro další dělení deseti (neboli z čísla odstranit poslední čislici, posunout se v desetinných místech).
- Otestujte, naprogramujte (pro číslo od uživatele použijte co největší celočíselný datový typ), pak otestujte program na různých vstupech.

Řešení:

Nejdřív si promyslíme, jak získat poslední číslici čísla. Původní číslo máme uloženo v proměnné `cislo`, použijeme pomocnou proměnnou `prom` a součet budeme kumulovat v proměnné `soucet`. Jak bylo napovězeno, použijeme celočíselné dělení deseti:

```
pom = cislo / 10;
cislice = cislo – pom*10
soucet += cislice;        // totéž jako soucet = soucet + (cislo – pom*10);
```

To bylo jen pro ilustraci. Ve skutečnosti je celkem zbytečné mít speciálně proměnnou pro uložení číslice, můžeme to zkrátit:

```
pom = cislo / 10;
soucet += cislo – pom*10;        // totéž jako soucet = soucet + (cislo – pom*10);
```

Pokud je například hodnota proměnné `cislo` 258, pak do proměnné `pom` načteme 258/10 = 25 (celočíselné dělení) a do proměnné `soucet` přičteme hodnotu 258 – (25*10) = 258-250 = 8.

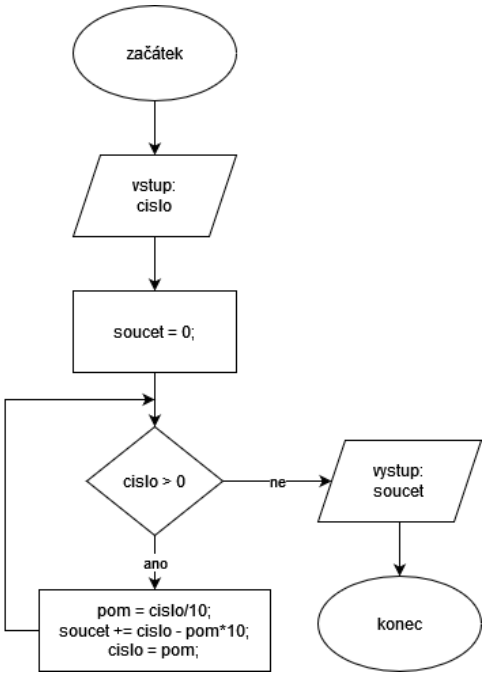
Takže už víme, jak získat poslední číslici, teď bychom potřebovali předposlední. Důležitou vlastností kódu je znovupoužitelnost, proto jednoduše použijeme ten postup, který jsme vymysleli pro poslední číslici, jen z předposlední číslice uděláme poslední (to jsme vlastně už udělali, v proměnné pom).

```
cislo = pom;
```

Zbývá ujasnit si jednu věc – kdy má cyklus skončit? Zřejmě tehdy, kdy nám „dojdou“ číslice, tedy zpracovávané číslo bude 0. To tedy bude podmínka cyklu: pokud `cislo > 0`, pokračuj.

Musí platit, že hodnota podmínky se během cyklu mění, abychom nevyrobili nekonečnou smyčku, tedy pokud použijeme proměnnou `cislo` v podmínce, musí být tato proměnná v těle cyklu změněna. Což platí.

Vedle vidíme kompletní vývojový diagram. Otestujeme, jak se bude algoritmus chovat pro různé vstupy.



Nejdřív „krajní hodnota“ – nula. Pokud uživatel dá na vstup hodnotu 0, pak hned na začátku cyklu neprojdeme přes podmínku a jsme odkloněni na výstup. Vypíše se hodnota proměnné `soucet`, která byla nastavena na 0, tedy vše v pořádku.

Dále jednociferné číslo, například 7. Projde přes podmínku a provede se tento výpočet:

```
pom = 7/10 = 0
soucet += 7 – 0*10        =>     soucet = 0 + 7 = 7
cislo = 0
```

Na začátku dalšího okruhu nebude splněna podmínka cyklu, vypíše se výsledek 7 a program je ukončen. Což je taky správně.

Můžete vyzkoušet i na víceciferném čísle, vždy by se algoritmus měl chovat správně.

Teď kód.

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

int main()
{
  long cislo, pom;
  int soucet = 0;

  cout << "Zadejte číslo: ";
  cin >> cislo;

  while (cislo > 0) {
      pom = cislo/10;
      soucet += cislo - pom*10;
      cislo = pom;
  }
  cout << "Ciferný součet: " << soucet;
}
```

7.2 Složený datový typ pole

Úkol – zadání:

Vytvořte program, ve kterém bude pole 30 prvků typu `long`. Pomocí cyklu naplňte pole náhodnými čísly (pozor na horní hranici – poslední index bude 29). Pak pomocí druhého cyklu vypište všechny prvky pole, každý prvek na nový řádek. Budete potřebovat pomocnou proměnnou, kterou postupně budete zvyšovat o 1. Nezapomeňte ji inicializovat.

Pokud si nejste jisti horní hranicí, vyzkoušejte nejdřív na krátkém poli (například s pěti prvky).

Řešení:

Bude to celkem jednoduchý program, ve kterém potřebujeme dva cykly.

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

int main()
{
   const int delka = 30;
   long nahodna[delka];
```

```
   int i = 0;

   srand(time(0));
   while (i < delka) {
      nahodna[i] = rand();
      i++;
   }

   i = 0;          // pozor, toto je nutné, aby tam nezůstalo číslo 30
   while (i < delka) {
      cout << nahodna[i] << endl;
      i++;
   }
 }
```

Pokud bychom chtěli náhodná čísla nějak ohraničit, například mít v poli pouze čísla o hodnotě menší než 10 000, příkaz uvnitř prvního cyklu by byl takový:

```
nahodna[i] = rand() % 10000;
```

Zbytek po dělení zadaným číslem je totiž vždy menší než dané číslo, zde 10 000.

Další modifikace by mohla být ohledně výstupu – 30 řádků je celkem hodně. Mohli bychom například vypsat vždy 10 čísel na jeden řádek, oddělených tabulátory. Využili bychom buď trojici cyklů while, nebo jednoduchý trik: pokud by pomocný index byl dělitelný 10 a různý od 0, zařádkovali bychom.

```
while (i < delka) {
   nahodna[i] = rand() % 10000;
   i++;
}

i = 0;
while (i < delka) {
   if (i  && !(i % 10))
      cout << endl;
   cout << nahodna[i] << "\t";
   i++;
}
```

Všimněte si podmínky určující, kdy se má zařádkovat. mohli bychom ji zapsat trochu přehledněji:

```
if ((i != 0) && (i % 10 == 0))
```

Významově je to totéž.

Úkol – zadání:

Modifikace předchozího úkolu: Po naplnění pole náhodnými čísly *ke každému druhému prvku* přičtěte číslo 200. Pro to také použijte cyklus, přičemž budete pracovat jen s každým druhým prvkem – indexy 1, 3, 5, … Takže v každém kroku budete zvyšovat pomocnou proměnnou ne o 1, ale o 2, například:

```
i += 2
```

Následně opět výsledné pole vypište.

Řešení:

Použijeme základní variantu předchozího příkladu, za první cyklus while přidáme ještě jeden cyklus:

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

```
int main()
{
   const int delka = 30;
   long nahodna[delka];
   int i = 0;

   srand(time(0));
   while (i < delka) {
      nahodna[i] = rand();
      i++;
   }

   i = 1;
   while (i < delka) {
      nahodna[i] += 200;
      i += 2;
   }

   i = 0;          // pozor, toto je nutné, aby tam nezůstalo číslo 30
   while (i < delka) {
      cout << nahodna[i] << endl;
      i++;
   }
 }
```

Jiná možnost by byla zakomponovat přičítání do prvního cyklu, přičemž bychom testovali, zda je momentálně index sudý či lichý. Pokud lichý, přičetli bychom 200.

7.3 Cyklus s pevným počtem kroků

Úkol – zadání:

Úkoly s poli z předchozí sekce přepište – místo cyklu s podmínkou použijte cyklus `for`.

Řešení:

První z těchto úkolů má následující zadání:

Vytvořte program, ve kterém bude pole 30 prvků typu `long`. Pomocí cyklu naplňte pole náhodnými čísly (pozor na horní hranici – poslední index bude 29). Pak pomocí druhého cyklu vypište všechny prvky pole, každý prvek na nový řádek. Budete potřebovat pomocnou proměnnou, kterou postupně budete zvyšovat o 1. Nezapomeňte ji inicializovat. Pokud si nejste jisti horní hranicí, vyzkoušejte nejdřív na krátkém poli (například s pěti prvky).

Řešení bude podobné jako bylo u původního úkolu, jenom přepíšeme cykly na FOR:

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

int main()
{
   const int delka = 30;
   long nahodna[delka];

   srand(time(0));
   for (int i = 0; i < delka; i++)
      nahodna[i] = rand();
```

```
   for (int i = 0; i < delka; i++)
      cout << nahodna[i] << endl;
}
```

Jak vidíte, kód je dokonce trochu kratší než při použití cyklů WHILE.

Následuje další zadání:

Modifikace předchozího úkolu: Po naplnění pole náhodnými čísly *ke každému druhému prvku* přičtěte číslo 200. Pro to také použijte cyklus, přičemž budete pracovat jen s každým druhým prvkem – indexy 1, 3, 5, … Takže v každém kroku budete zvyšovat pomocnou proměnnou ne o 1, ale o 2, například:

```
i += 2
```

Následně opět výsledné pole vypište.

Řešení s použitím cyklů FOR:

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

int main()
{
   const int delka = 30;
   long nahodna[delka];

   srand(time(0));
   for (int i = 0; i < delka; i++) {
      nahodna[i] = rand();
      if (i % 2 == 1)
         nahodna[i] += 200;
   }

   for (int i = 0; i < delka; i++)
      cout << nahodna[i] << endl;
}
```

První dva původní cykly jsou shrnuty do jediného: načítáme čísla do pole a zároveň všechna lichá zvyšujeme o 200.

Úkol – zadání:

Vytvořte pole s 5 prvky typu `int`. Od uživatele načtěte prvky tohoto pole a následně zjistěte (a vypište) součet a průměr těchto hodnot.

Řešení:

Ve funkci main() bude následující kód:

```
int pole[5], soucet = 0;

for (int i = 0; i < 5; i++) {
   cout << "Zadejte " << i << ". číslo: ";
   cin >> pole[i];
   soucet += pole[i];
}
cout << endl << "Soucet je " << soucet;
cout << endl << "Prumer je " << soucet/5;
```

Úkol – zadání:

Prozkoumejte následující kód:

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

int main() {
   for (int i=4; i<12; i++) {
      cout << i*10 << "\t\t";
      for (int j=0; j<=9; j++)
         cout << (char)(i*10+j) << '\t';
      cout << endl;
   }
}
```

K čemu tento kód slouží? Pokud si nejste jisti, vyzkoušejte a podívejte se na výstup.

Všimněte si, že symboly tabulátoru `\t` jsou nejdřív uzavřeny do uvozovek a v dalším příkazu do apostrofů. Proč? Můžeme v obou případech použít uvozovky nebo v obou případech apostrofy?

Vysvětlení:

Hlavní cyklus FOR probíhá postupně přes čísla 4, 5, 6,…,11. Na konci hlavního cyklu je `cout << endl`, tedy v každém průběhu je vypsán jeden řádek (něčeho). Na začátku řádku je číslo 40, 50, 60, atd., vždy desetinásobek řídicí proměnné `i`. Pak je tedy dvojice tabulátorů, aby byla vytvořena dost velká mezera.

Následně pro všechny číslice od 0 do 9 se postupně vypisují znaky, jejichž ASCII hodnoty jsou vždy desetinásobek čísla `i` plus daná číslice, tedy například pro první řádek to budou znaky s ASCII hodnotami 40, 41, 42,…49, jednotlivé znaky jsou odděleny tabulátorem.

Takže když to shrneme, dostaneme podstatnou část první poloviny ASCII tabulky. Začátek je přeskočen, protože tam se stejně nacházejí netisknutelné znaky. Kdybychom chtěli druhou polovinu, zvýšili bychom hranici pro proměnnou `i` a ošetřili bychom „horní konec“ – ASCII znaky jsou jen do hodnoty 255.

Co se týče tabulátorů: v obou případech můžeme použít uvozovky, ale apostrofy jen tehdy, když je mezi nimi jen jeden znak (třeba ten tabulátor).

Úkol – zadání:

Využijte cyklus FOR pro výpočet faktoriálu čísla, které načtete od užiivatele.

Řešení:

Faktoriál má vzorec

`factorial(0) = 1;`
`factorial(n) = n*factorial(n-1)` pro každé celé n>0.

Tento vzorec je typicky rekurzivní (tj. pro výpočet funkce využíváme i funkci samotnou), ale rekurzi budeme brát až o pár kapitol dál. Tedy si vystačíme s cykly a vzorec trochu upravíme:

`factorial(0) = 1;`
`factorial(n) = n * (n-1) * ... * 1` pro každé celé n>0. S tím si už poradíme.

Takže načteme vstup od uživatele a budeme násobit. Je třeba si pohlídat inicializaci proměnné pro výsledek (taky její datový typ, ať se do ní výsledek vejde i pro velká čísla v základu faktoriálu), a dále spodní hranici (faktoriál čísla 0).

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

int main()
{
   unsigned int zaklad;
   long vysledek = 1;

   cout << "Zadej cislo, ktere bude zakladem faktorialu: ";
   cin >> zaklad;

   for (int i = zaklad; i > 0; i--)
      vysledek *= i;
   cout << "Vysledek: " << vysledek;
}
```

Všimněte si, že v cyklu FOR jdeme „pozpátku“. Je to jednodušší, snadněji si pohlídáme nejnižší číslo. Také si všimněte, že jsme proměnnou pro výsledek inicializovali na 1, protože budeme násobit. Číslo 0 by nám „otrávilo“ jakýkoliv výsledek, vždy by vyšla nula, navíc faktoriál pro nejmenší možný parametr je rovný jedné.

Jako datový typ výsledku jsme použili `long`, velikost tohoto datového typu je totiž 8 B (tedy 8*8=64 bitů), což by mělo stačit.

7.4 Přerušení cyklu

Úkol – zadání:

Rozepište předchozí příklad tak, aby bylo možné zadávat i další operace (násobení, dělení, případně modulo). U dělení nezapomeňte ošetřit dělení nulou, například takto:

```
if (y == 0) {
   cout << "Dělení nulou\n";
   continue;       // předpokládáme, že jsme uvnitř cyklu, skočíme na jeho začátek
}
```

Kdybychom chtěli při výskytu dělení nulou ukončit cyklus, stačilo by zde místo `continue` napsat `break`?

Řešení:

Ten „předchozí příklad“ obsahuje kód jednoduché kalkulačky provádějící sčítání a odčítání, a to tak dlouho, dokud uživatel ještě chce. Kalkulačka je uvnitř cyklu `do-while(1)`. Cyklus končí příkazem `break` nebo příkazem `return`. Přidání dalších operací je jednoduché, stačí se orientovat podle existujících. Je třeba „obohatit“ dvě místa programu: zobrazení menu pro uživatele a pak samotný výpočet.

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

int main() {
   int menu = 0, x, y;
   double vysl;
```

```
   do {
      cout << "1: sčítání\n";
      cout << "2: odčítání\n";
      cout << "3: násobení\n";
      cout << "4: dělení\n";
      cout << "jinak: konec\n";
      cin >> menu;
      if (menu < 1 || menu > 4)  {
         cout << "konec";
         break;
      }
      cout << "x = ";                    // požádáme o hodnotu do proměnné x
      cin >> x;                          // načteno x od uživatele
      cout << "y = ";                    // požádáme o y
      cin >> y;                          // načteno y od uživatele
      switch (menu) {
         case 1: vysl = x+y; break;
         case 2: vysl = x-y; break;
         case 3: vysl = x*y; break;
         case 4:
            if(y == 0) {
               cout << "Dělení nulou!!!";
               continue;
            }
            vysl = (double)x/y;
            break;
         default:
            cout << "chybná operace";
            return 1;                    // něco se nepovedlo, ukončíme program
      }
      cout << "Výsledek: " << vysl << endl << endl;
   } while (1);      // nekonečný cyklus, podmínka je vždy TRUE
}
```

Všimněte si, že jsme pro výsledek použili datový typ `double`. V případě sčítání, odčítání a násobení by to nebylo nutné (mohl by být taky celočíselný, jen by snad bylo vhodné vybrat datový typ zabírající více místa v paměti, třeba `long`), ale u dělení budeme chtít racionální číslo (leda by bylo v zadání celočíselné dělení). Taky si všimněte přetypování při výsledku.

U dělení je také nutné provést kontrolu, zda dělitel není nula. Pokud ano, vynadáme uživateli a ukončíme aktuální průběh cyklu (ne celý program ani celý cyklus, vrátíme se na začátek).

Teď k otázce na konci zadání: v tomto řešení jsme použili příkaz `continue`, abychom po chybném zadání umožnili uživateli „vyčistit stůl“ a začít znovu. Vracíme se na začátek cyklu `do-while`. Kdybychom na tomto místě použili příkaz `break`, ukončili bychom tak větev `case 4` a přesunuli se za `switch`, což by v tomto případě znamenalo, že by se vypsal chybný výsledek (a pak bychom přešli na začátek cyklu jako při použití `continue`). Takže zde musí být opravdu `continue`.

7.5 Vícedimenzionální pole

Úkol – zadání:

Vytvořte čtvercové dvoudimenzionální pole (čtvercovou matici) o délce řádku/sloupce 10, pro číslo 10 však použijte konstantu (a tu používejte jak při deklaraci pole, tak i v cyklech při zpracování), prvky typu `int`. Vymyslete, jak oddělit zpracování prvků matice nacházejících se nad a pod diagonálou.

Do prvků na diagonále a pod ní načtěte nulu, do prvků nad diagonálou načtěte náhodné číslo menší než 100. Nápověda:

- Konstantu vytvořte takto: `const int N = 10;`
- Pokud používáte vnořené cykly `for`, můžete řídicí proměnnou z vnějšího cyklu použít i ve vnitřním cyklu.
- S náhodnými čísly už jsme pracovali, příslušné funkce najdete o pár stránek výše.
- Podmínka „menší než 100“ se dá zajistit použitím zbytku po celočíselném dělením „modulo“, která se zapisuje symbolem procenta: `x = y % 100`

Řešení:

Vytvoříme konstantu N a nastavíme ji na hodnotu 10. S pomocí této konstanty deklarujeme dvoudimenzionální pole, které bude plnit roli čtvercové matice.

```
const int N = 10;
int matice[N][N];
```

Teď naplníme matici čísly podle zadání. Nad diagonálou budou náhodná čísla, zbytek matice bude nulový. První souřadnice v matici znamená řádek, druhá je sloupec.

Prvky diagonály mají obě souřadnice stejné, takže diagonála je tvořena právě prvky `matice[0][0]`, `matice[1][1]`, `matice[2][2]`, atd. Prvky nad diagonálou budou mít druhou souřadnici vyšší než tu první, prvky pod diagonálou budou mít naopak vyšší první souřadnici.

```
srand(time(0));
for (int i = 0; i < N; i++) {    // řádky matice
   for (int j = 0; j <= i; j++)  // matice[i][i] je prvek na diagonále,
      matice[i][j] = 0;          //     tedy zde jsou prvky pod diagonálou a na ní
   for (int j = i+1; j < N; j++)
      matice[i][j] = rand() % 100;      // prvky nad diagonálou
}
```

Celou matici teď vypíšeme, ať si ověříme, jestli jsme se „trefili“:

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++)
      cout << matice[i][j] << '\t';
   cout << endl;
}
```

Příklad s otázkami:

Nejdřív zde máme deklaraci s inicializací pro pole o třech řádcích, v každém řádku dvě čísla:

```
int pole1[][2] = { {1, 2}, {3, 4}, {5, 6} }; // plná deklarace by byla pole[3][2]
```

Následuje třídimenzionální pole:

```
int pole2[][2][3] = { { {0,0,0}, {1,-1,1} }, { {10,9,8}, {-10,-9,-8} }};
```

Jaká je chybějící dimenze tohoto druhého pole?

```
cout << pole1[0][2];            // proč je to špatně?
cout << pole1[0][1];            // které číslo se vypíše?
cout << pole2[0][1][2];         // které číslo se vypíše?
```

Vysvětlení:

U prvního pole je chybějící dimenze 3, protože ve složených závorkách máme tři dvojice (vpodstatě tři dvouprvková jednoduchá pole, jde tedy o matici se třemi řádky, v každém řádku dvě čísla.

U druhého pole jsou tři dimenze, z toho ta první znamená počet „vnitřních“ dvoudimenzionálních polí, což znamená 2. Pokud se neorientujete, označte si, která složená levá závorka patří ke které pravé.

Následují tři příkazy pro výstup prvků prvního a druhého pole:

```
cout << pole1[0][2];            // proč je to špatně?
```

Druhý parametr je chybný, protože na tom místě mohou být pouze hodnoty 0 nebo 1. Číslo v deklaraci berte jako počet, ovšem jdeme od 0, nikoliv od 1.

```
cout << pole1[0][1];            // které číslo se vypíše?
```

Vypíše se číslo 2, to odpovídá zadaným souřadnicím. První řádek, druhý sloupec.

```
cout << pole2[0][1][2];         // které číslo se vypíše?
```

Označme si barevně, o který prvek jde:

```
int pole2[][2][3] = { { {0,0,0}, {1,-1,1} }, { {10,9,8}, {-10,-9,-8} }};
```

Jde o první prvek z vnějších závorek, druhý z „prostředních“ závorek, třetí z „vnitřní“ trojice, tedy 1.

Příklad s úkolem:

Do 2D pole o 5 řádcích a 6 sloupcích načteme náhodná čísla menší než 1000, následně od uživatele načteme číslo a pak v poli vynulujeme všechny pozice, kde je číslo menší než to, které zadal uživatel.

Pak se uživatele zeptáme, který řádek chce vypsat (zkontrolujeme, zda zadal číslo od 0 do 4, pokud ne, tak ukončíme program), a požadovaný řádek vypíšeme.

```
cont int R = 5, S = 6;          // pro počet řádků a sloupců použijeme konstantu
cont int operace_modulo = 1000;
int matice[R][S];
int porovnani, radek;

srand(time(0));
for (int i = 0; i < R; i++)     // fáze naplnění matice čísly
   for (int j = 0; j < S; j++)
      matice[i][j] = rand() % operace_modulo; // zbytek po dělení číslem 1000

cout << "Zadej číslo menší než " << operace_modulo << ": ";
cin >> porovnani;

if ((porovnani < 0) || (porovnani >= operace_modulo)) {
   cout << "Číslo je mimo rozsah, konec programu. ";
   return 1;
}

for (int i = 0; i < R; i++)     // vynulujeme vše, co je menší než zadané číslo
   for (int j = 0; j < S; j++)
      if (matice[i][j] < porovnani)
         matice[i][j] = 0;

cout << "Indexy řádků jsou mezi 0 a " << (R-1) << ".\n";
cout << "Který řádek mám vypsat? ";
cin >> radek;
```

```
if ((radek < 0) || (radek >= R)) {
   cout << "Mimo rozsah. ";
   return 2;       // všimněte si, že předtím byl parametr pro return 1, teď 2
}

for (int i = 0; i < R; i++)            // vypíšeme vybraný řádek
   cout << matice[radek][i] << '\t';
```

V příkladu je chyba (ke konci). Pokuste se ji najít. Pokud se vám to nepodaří, zkopírujte kód a odlaďte. Nejde o syntaktickou chybu, překladač vás neupozorní.

Řešení:

Chyba je na předposledním řádku: v posledním cyklu FOR jsme jako hranici použili R místo S, tedy vypíšeme o jedno číslo méně. Zpracováváme jeden řádek, jeho délka odpovídá počtu sloupců, nikoliv řádků.

Úkol pro ty, kdo potřebují procvičit matematiku – zadání:

Deklarujte dvě matice – jednu o dimenzích 2 a 3 (řádky sloupce), druhou o dimenzích 3 a 2. Do obou načtěte vstupy od uživatele (tj. načtete dvakrát 6 čísel).

Vypočtěte součin těchto dvou matic a vypište výsledek (to bude samozřejmě také matice). Můžete postupovat jednou z těchto cest, podle vlastního výběru:

- buď deklarujte třetí matici (pozor na počet řádků a sloupců), do ní vypočtěte jednotlivé prvky a pak vypište,
- nebo vypisujte výsledky už během výpočtu, v tom případě nepotřebujete třetí matici.

Vzorec (pokud si z matematiky nepamatujete):

Pokud označíme původní matice A(R1, S1) a B(R2, S2), pak prvek se souřadnicí [rad,sloup] vypočteme tak, že vezmeme z první matice řádek číslo rad a z druhé matice sloupec číslo sloup, vynásobíme je mezi sebou (tj. první prvek řádku s prvním prvkem sloupce plus druhý prvek řádku s druhým prvkem sloupce atd.):

$$(A \,.\, B)[rad, sloup] = \sum_{k=0}^{S1-1} A\,[rad, k] \,.\, B\,[k, sloup]$$

Mějte neustále na paměti, že řádky i sloupce se počítají od nuly.

Řešení:

Pro přehlednost nejdřív provedeme výpočet a potom teprve vypíšeme výslednou matici.

```
int maticeA[2][3], maticeB[3][2], vysledek[2][2];

// Nejdřív načteme první matici – po řádcích (dva řádky, v každém tři prvky):
cout << "Zadej matici A:\n";
for (int i=0; i<2; i++) {
   for (int j=0; j<3; j++) {
      cout << '[' << i << "][" << j << "]= ";
      cin >> maticeA[i][j];
   }
   cout << endl;
}
```

```
// Teď druhá matice (tři řádky, v každém dva prvky):
cout << "Zadej matici B:\n";
for (int i=0; i<3; i++) {
   for (int j=0; j<2; j++) {
      cout << '[' << i << "][" << j << "]= ";
      cin >> maticeB[i][j];
   }
   cout << endl;
}

// Provedeme výpočet podle vzorce (násobíme 1. řádek A s 1. sloupcem B, atd.):
for (int rad = 0; rad < 2; rad++)
   for (int sloup = 0; sloup < 2; sloup++) {
      vysledek[rad][sloup] = 0;
      for (int k = 0; k < 3; k++)
         vysledek[rad][sloup] += maticeA[rad][k] * maticeB[k][sloup];
   }

// Výpis výsledné matice:
for (int i=0; i<2; i++) {
   for (int j=0; j<2; j++)
      cout << vysledek[i][j] << "\t";
   cout << endl;
}
```

8 Složené a uživatelské datové typy

Úkol – zadání:

Vytvořte datový typ struktury pro zvířata (auta, rostliny, filmové postavy, postavy z počítačové hry, atd. podle svých zájmů), ale tak, aby součástí struktury byly různé vnitřní datové typy (číslo celé, případně racionální, řetězec, výčtový datový typ, může být pole). Zapojte fantazii.

Vytvořte dvě proměnné tohoto datového typu. První z nich inicializujte při deklaraci, prvky z druhé nechte načíst od uživatele. Následně vypište obsah obou těchto proměnných (postupně po položkách).

Řešení:

Zvolíme třeba domácí zvířata. Každé bude mít evidováno jméno jako řetězec, pak typ pomocí výčtového typu, a pak věk jako celé nezáporné číslo:

```
enum TYP_ZVIRETE { kocka, pes, ptak };

struct ZVIRE {
   string jmeno;
   TYP_ZVIRETE typ;
   unsigned int vek;
} ;

// Dvě proměnné daného typu, z nichž první hned inicializujeme:
ZVIRE domaci1 = { "Micka", kocka, 4 } , domaci2;
int pom;

// Načteme druhé zvířátko:
cout << "Zadej jmeno zvirete: ";
cin >> domaci2.jmeno;
cout << "Zadej cislo 1, 2 nebo 3 (1 = kocka, 2 = pes, 3 = ptak): ";
cin >> pom;
switch(pom) {
   case 1: domaci2.typ = kocka; break;
   case 2: domaci2.typ = pes; break;
   case 3:
   default: domaci2.typ = ptak;
}
cout << "Zadej vek zvirete: ";
cin >> domaci2.vek;

// A jdeme vypisovat. Nejdřív to inicializované, pak to načtené od uživatele:
cout << endl << endl;
cout << "Prvni zviratko:\n";
cout << "\tJmeno: " << domaci1.jmeno;
cout << "\n\tTyp: ";
switch(domaci1.typ) {
   case kocka: cout << "kocka"; break;
   case pes: cout << "pes"; break;
   case ptak: cout << "ptak";
}
cout << "\n\tVek: " << domaci1.vek;

cout << endl << endl;
cout << "Druhe zviratko:\n";
cout << "\tJmeno: " << domaci2.jmeno;
cout << "\n\tTyp: ";
```

```
switch(domaci2.typ) {
   case kocka: cout << "kocka"; break;
   case pes: cout << "pes"; break;
   case ptak: cout << "ptak";
}
cout << "\n\tVek: " << domaci2.vek;
```