Algoritmy a programování I – cvičení

Tento text je studijním materiálem do cvičení v předmětu Algoritmy a programování I. Najdete zde jak řešené příklady, tak i úkoly k vyřešení. Zatím není kompletní, bude vytvářen průběžně v semestru.

Obsah

1 VÝVOJOVÉ DIAGRAMY ... 3
1.1 Jak na vývojový diagram ... 3
1.2 Jednoduché úlohy ... 6
1.3 Cykly ... 6
2 PSEUDOKÓD ... 8
3 PROGRAMOVACÍ JAZYKY A NÁSTROJE ... 11
3.1 Programovací jazyky ... 11
3.2 Stručně o nástrojích ... 12
4 ZAČÁTEK ... 16
4.1 Základní tvar programu a hlavní funkce ... 16
4.2 Jednoduchý vstup a výstup ... 16
4.3 Komentování ... 17
4.4 S jakými daty můžeme při programování pracovat ... 17
4.5 Řetězce ... 19
4.6 Jak vyrobit náhodné číslo ... 20
4.7 Zpět ke vstupům a výstupům ... 21
4.7.1 Vstup a výstup tradičně: stdio.h ... 21
4.7.2 Vstup a výstup pomocí streamů: iostream.h ... 22
5 VÝRAZY A PODMÍNKY ... 25
5.1 Konstanty ... 25
5.2 Aritmetické operace ... 26
5.3 Bitové operace ... 28
5.4 Logické operace, výrazy v podmínkách ... 30
6 PODMÍNĚNÉ PROVEDENÍ KÓDU A VĚTVENÍ PROGRAMU ... 33
6.1 Ternární operátor – pro pokročilé ... 33
6.2 Příkaz IF ... 33
6.3 Příkaz SWITCH ... 36
7 CYKLY A POLE ... 39
7.1 Jednoduchý cyklus s podmínkou ... 39

7.2 Složený datový typ pole ... 41
7.3 Cyklus s pevným počtem kroků ... 42
7.4 Přerušení cyklu ... 44
7.5 Vícedimenzionální pole ... 46
8 Složené a uživatelské datové typy ... 50
8.1 Výčtový typ ... 50
8.2 Struktura ... 50
8.3 Union pro určení variant ... 52
8.4 Řetězce ... 53
9 Pointer a dynamická alokace paměti ... 55
9.1 Pracujeme s pointery ... 55
9.2 Pointer na strukturu ... 57
9.3 Dynamická alokace paměti v C++ ... 57
9.4 Jiné možnosti dynamické správy paměti ... 59
9.5 Dynamika na n-tou ... 59
10 Funkce ... 60
10.1 Jak na funkce ... 60
10.2 Volání funkce z funkce a rekurze ... 62
10.3 Parametry volané hodnotou a parametry volané odkazem ... 64
10.4 Globální a lokální proměnné ... 65
11 Soubory a streamy ... 69
11.1 Soubory projektu ... 69
11.2 Soubory jako vstup a výstup ... 70
11.2.1 Práce se soubory objektově podle C++: streamy ... 70
11.2.2 Pohyb v streamu souboru ... 74
11.2.3 Práce se soubory „céčkovým“ stylem ... 75
11.2.4 Jak zpracovat celý soubor ... 76
11.3 Řešení problémů se streamy ... 76
12 Řazení ... 79
Doporučená literatura ... 80

1 Vývojové diagramy

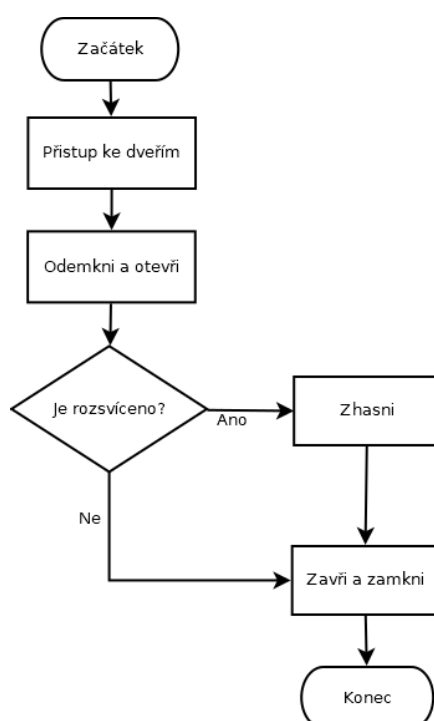
1.1 Jak na vývojový diagram

Vývojový diagram je grafické vyjádření algoritmu. Umožňuje nám stručně a srozumitelně vyjádřit, co má určitý algoritmus provádět. Pomocí vývojového diagramu můžeme zapsat vše, co lze rozdělit do jednotlivých kroků a určit posloupnost těchto kroků.

Příklad:

Představme si, že máme popsat část pracovní náplně nočního hlídače, který večer projde celý areál pracoviště a u všech místností zkontroluje, zda je zhasnuto. Aby diagram nebyl ze začátku moc rozsáhlý, soustředíme se pouze na tu část, která se bude týkat jediné (některé) místnosti. Předpokládejme tedy, že se hlídač nachází před dveřmi kontrolované místnosti.

Na začátku a konci diagramu je mezní značka, podle které poznáme, kde cesta diagramem začíná a kde končí – má tvar oválu. V těchto značkách máme (v našem případě) zapsáno „Začátek" a „Konec".



Obdélníkové značky určují konkrétní činnost, která má být provedena, diamant (kosodélník) určuje větvení (tedy jsou dvě možnosti, jak pokračovat). V diamantu obvykle míváme otázku nebo podmínku, která buď může nebo nemusí být splněna. Mezi jednotlivými prvky vedou šipky, protože i směr průchodu je důležitý.

K čemu je to dobré? Může se to zdát jako hračka pro děti, ale ve skutečnosti se různé druhy diagramů (včetně vývojových) používají i v praxi, třebaže ve firmách se častěji setkáme spíše s UML diagramy. Pokud na větším projektu spolupracuje celý tým, jak jinak chcete přehledně popsat strukturu celého projektu než diagramem?

Jaký nástroj se dá použít pro vytvoření vývojového diagramu?

- Existuje celá řada cloudových nástrojů, mnohé z nich jsou dostupné zdarma. Asi nejznámější je *draw.io* na adrese draw.io (následně jsme přesměrováni na „delší adresu", tak se nelekněte).

Tento nástroj se dá také nainstalovat jako aplikace, odkaz na github najdeme na stránce projektu https://www.diagrams.net/.

- Zdarma dostupný je nástroj *Dia* (najdete na http://dia-installer.de/), jde o aplikaci, kterou si můžeme stáhnout a nainstalovat.
- Zástupcem komerčních nástrojů je *Microsoft Visio* (adresa je dlouhá, ideální je „zeptat se strýčka Googla“ na „MS Visio“). Pro účely procvičení asi bude zbytečné pořizovat si podobný nástroj za peníze.
- Můžeme se „zeptat strýčka Googla“, zadáme „flowchart diagram“ (to je anglicky „vývojový diagram“). Doporučuji přeskočit reklamní odkazy.
- Hodně podobných nástrojů je komerčních s možností vyzkoušení trial verze, nebo alespoň vyžadují registraci. Namátkou můžeme jmenovat Whimsical, Lucidchart, Miro a další.

Nástroje pro tvorbu diagramů bývají také často součástí komplexních balíků, například u produktů Atlassianu se počítá s používáním nástroje Gliffy.

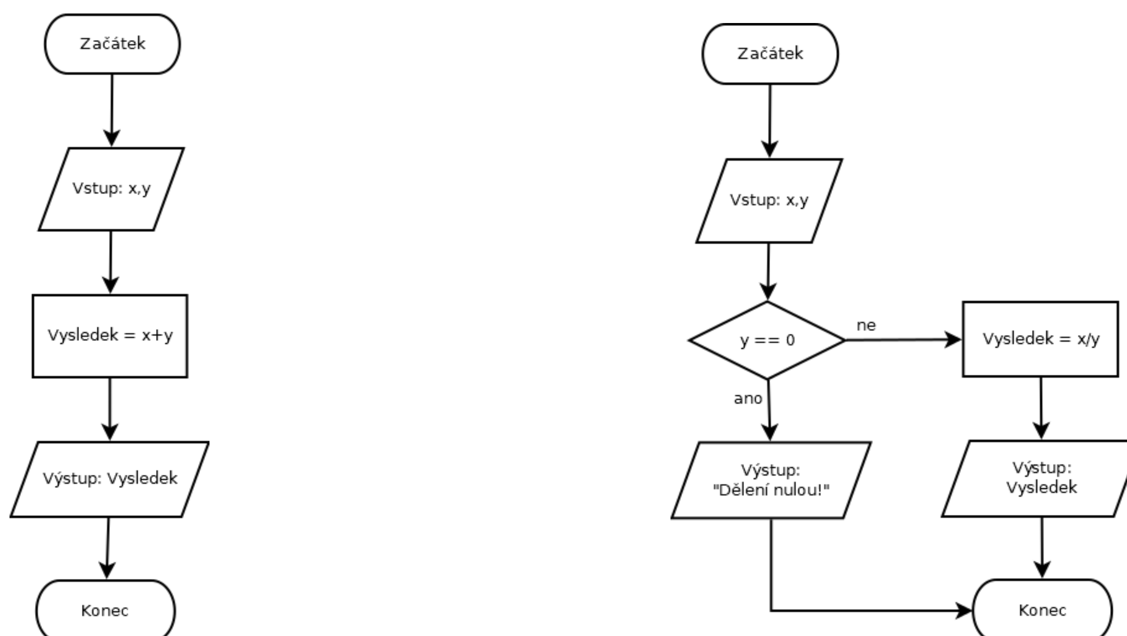
Úkol:

Vyzkoušejte draw.io. Na začátku budete zřejmě dotázáni na to, kam chcete ukládat vytvořené diagramy (zvolte podle svých preferencí), a pak už můžete vytvořit nový nebo otevřít dříve uložený. Vyzkoušejte si práci v tomto nástroji tak, že „nakreslíte“ některý z vývojových algoritmů v této kapitole.

Ukážeme si dva jiné příklady, které budou řešit jednoduchou matematickou úlohu.

Příklad:

Oproti předchozímu příkladu zde máme jednu novou značku: kosodélník určený pro vstup a výstup. V obou diagramech je použit pro vstup dvou čísel od uživatele (tedy uživatel je požádán, aby zadal dvě čísla – neřešíme zde způsob, pouze fakt, že se tak má stát) a pak na konci pro výstup, tedy vypíšeme výsledek výpočtu (nebo chybové hlášení).



V prvním případě chceme obě čísla sečíst a výsledek dát na výstup. Sčítání je operace nezáludná, takže žádný problém. V druhém případě však musíme ošetřit případ, že druhé číslo (které má být „pod zlomkovou čárou“) by mělo být nenulové, protože (jak víme) nulou nelze dělit. Pro větvení kódu jsme

použili diamant, dvě rovnítka za sebou značí porovnání. V případě, že druhé číslo je opravdu nula, vypíšeme chybové hlášení a ukončíme program.

Při vytváření vývojového diagramu je třeba dodržovat určitá pravidla:

- Každý algoritmus má právě jeden začátek a jeden konec – obojí označené oválem.
- Každá cesta diagramem vede od začátku do konce a vždy musíme být schopni se jednoznačně rozhodnout, kudy dál => z většiny značek vede právě jedna šipka, kromě rozhodovacího bloku (diamantu), ze kterého vede pro každou možnost (určenou podmínkou) jedna šipka.
- Šipky vedoucí z diamantu musí být vždy jasně popsány, aby bylo jasné, za jakých okolností se kterou cestou vydat.

Při návrhu složitějšího algoritmu můžeme pomocí vývojového diagramu určit základní „kostru“ postupu, kdy jednotlivé kroky jednoduše pojmenujeme, dále v jiném diagramu takové „multikroky“ rozvedeme.

Příklad:

Pro zápis takového multikroku, který míníme jinde rozvést, se používá obdélník se zdvojenými bočnicemi. Na obrázku níže je takto reprezentován krok „Odemkni a otevři“.

Úkol:

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.

1.2 Jednoduché úlohy

V následujících úlohách si procvičíme vývojové diagramy s rozhodováním – diamantem.

Úkoly:

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.

1.3 Cykly

Zaměřme se na poslední příklad z předchozího úkolu. Vytvořili jsme kalkulačku, která od uživatele načte dvě čísla a operátor, vypočte výsledek a skončí. Ale co když budeme chtít, aby svou práci dělala opakovaně?

Úkol:

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.

Z úkolu je zřejmé, jak můžeme postupovat, když tvoříme cyklus. Promyslíme si, jak bude vypadat jeden krok, určíme podmínku, za které se má cyklus opakovat a „uzavřeme kruh“. Vždy platí, že z cyklu musí existovat cesta ven, žádný cyklus by neměl trvat nekonečně dlouho. Ta cesta ven je ukončující podmínka. V předchozím úkolu to byl nesouhlas uživatele s pokračováním.

Úkol:

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).

Úkol:

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

Pro předchozí dva úkoly existují různá řešení, ale velmi pravděpodobně jste pro první z nich zvolili cyklus s podmínkou na začátku (v cyklu nejdřív testujeme a pak něco počítáme) a pro ten druhý cyklus s podmínkou na konci (nejdřív provedeme – hodíme kostkou – a pak testujeme – zda padla 6). Existuje ještě třetí typ cyklu: s daným počtem kroků.

Úkol:

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.

Algoritmus (včetně toho, který zapisujeme vývojovým diagramem) má splňovat tři podmínky: musí být

- hromadný (platný pro různé vstupy, v matematice bychom hovořili o definičním oboru),
- deterministický (v každém kroku jednoznačný),
- konečný (pro jakýkoliv vstup musí být možné projít algoritmem v konečném počtu kroků).

Jak musí vypadat vývojový diagram, aby tyto vlastnosti splňoval?

Úkol:

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.

2 Pseudokód

Pseudokód je dalším způsobem symbolického zápisu algoritmu, tentokrát v řádcích. Jde o to, abychom zjednodušeně zachytili jednotlivé kroky algoritmu bez vazby na konkrétní programovací jazyk nebo bez nutnosti (zatím) zacházet do podrobností.

Pseudokód sice není nutno zapisovat v konkrétním programovacím jazyce, ale často se v syntaxi (tj. způsobu zápisu jednotlivých částí) uchylujeme k některému „oblíbenému“ programovacímu jazyku, jen prostě některá místa zjednodušujeme.

Příklad:

U vývojových diagramů jsme řešili načtení dvou čísel od uživatele a operaci dělení. Ukážeme si několik možností, jak to zapsat v pseudokódu. První možnost:

```
program dělení
   načti x
   načti y
   jestliže y == 0 tak
      vypiš "Dělení nulou!"
   jinak
      vysl = x / y
      vypiš "výsledek: ", vysl
   konec jestliže
konec programu
```

Druhá možnost:

```
program dělení {
   read x;
   read y;
   if (y == 0) {
      print ("Dělení nulou! ");
   }
   else {
      vysl = x / y;
      print ("Výsledek je ", vysl);
   }
}
```

Třetí možnost:

```
dělení:
   načti od uživatele x a y;
   otestuj y;
   pokud y je 0, vypiš chybu "Dělení nulou";
   jinak proveď dělení a vypiš výsledek;
konec
```

Jak vidíte, pseudokód může být napsán na různých stupních abstrakce – to znamená, že pokud není nutné určité kroky rozepisovat (nebo je v plánu psudokód rozepsat později), tak jsme struční, jasní, výstižní.

Přímo pro zápis pseudokódu zřejmě programy nenajdeme, můžeme si vystačit s běžným textovým editorem nebo třeba čmárat na papír, ale za pseudokód ve skutečnosti můžeme považovat i vývojový diagram nebo schéma v UML. Nicméně v různých sofistikovanějších nástrojích existují možnosti, jak pseudokód zapisovat a zpřehledňovat.

Příklad:

Pokud píšeme knihu, ve které budou úseky kódu či pseudokódu (jestli nemá být výsledkem spustitelný soubor, tak je to celkem jedno), můžeme využít právě tyto nástroje. Například pro systém LaTeX existuje balíček (tj. jakási knihovna) algorithm2e, pomocí kterého můžeme tvořit například takto (neděste se, to nebudete muset dělat):

```
Algorithm 2: Auxiliary functions
function node.changeRootPort(newRootPortID, newPortRole)
begin
    //rootPortID==0 would mean that I am the root
    if (self.rootPortID != 0) then
        self.ports[self.rootPortID].state = blocking;
        self.ports[self.rootPortID].role = newPortRole;
    end
    self.rootPortID = newRootPortID;
    self.ports[newRootPortID].state = blocking;
    self.ports[newRootPortID].role = rootPort;
end

//Obtained info about new root or root path cost changed, do synchronization downwards.
function node.makeSync(sender)
begin
    //Only connected/used ports are processed:
    foreach (port in self.ports) do
        port.state = blocking;
    self.ports[self.rootPortID].state = forwarding;
    self.ports[self.rootPortID].sendBPDU_agree();
    foreach (port in self.ports) do
        if ((port.role == designatedPort) or (port.role == alternatePort)) then
          port.sendBPDU_proposal() ;
end

function node.IAmNewRoot()
begin
    self.rootID = self.BID;
    self.RPcost = 0;
    self.rootPortID = 0;
    foreach (port in self.ports) do
```

Podobné, ale zase o něčem jiném:

```
Algorithm 4: Controller – function
// An object received from the membrane:
function controller.receiveObject(obj)
begin
    switch obj.type do
        case c do
            if (authenticator.check(obj.ID, obj.credentials)) and
             (device^.processMessage(connect, obj.ID, obj.credentials)) then
                components[obj.ID].turnOn();
        case d do
            device^.processMessage(disconnect, obj.ID);
            components[obj.ID].turnOff();
        case s do
            device^.processMessage(obj.ID, obj.topic);
            subscriptionsDB.add(obj.topic, obj.ID);
            if topicsDB.retainSet(obj.topic) then
                membrane^.createObject(d, topicsDB.lastPublisher(obj.topic), obj.ID,
                 obj.topic, 1, topicDB.lastValue(obj.topic));
        case u do
            subscriptionsDB.remove(obj.topic, obj.ID);
            device^.processMessage(obj.ID, obj.topic);
        case p do
            if obj.retain then topicsDB.storeData(obj.topic, obj.data);
            else topicsDB.unsetRetain(obj.topic) ;
    end
end
```

Příklad:

Starořecký matematik Euklides je známý řadou postupů koncipovaných jako algoritmus, jedním z nich je algoritmus výpočtu největšího společného dělitele (NSD) dvou čísel pojmenovaný po jeho tvůrci: Euklidův algoritmus. Jeho zápis běžně bývá dostupný ve formě pseudokódu:

```
vstup: přirozená čísla a, b;
pomocná proměnná: m;

dokud b > 0  prováděj:
   m = a % b;
   a = b;
   b = m;
výsledek = a;
```

Jiný (trochu více „slovní“ způsob zápisu:

```
vstup: přirozená čísla a, b;
pomocná proměnná: m;

dokud b > 0  prováděj:
   ulož do proměnné m zbytek po celočíselném dělení čísel a, b;
   ulož do proměnné a hodnotu b;
   ulož do proměnné b hodnotu m;
vrať hodnotu proměnné a.
```

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.

Úkol:

Na stránce

https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/U%C4%8Debnice/Algoritmus/Slovn%C3%AD_popis_pracovn%C3%ADho_postupu

jsou ukázky popisu pracovních postupů a doporučení, jak se takové popisy mají dělat. Zaměřte se na dva postupy násobení čísel. Fungují? V čem se liší? Chybí v druhém z těchto postupů něco? Dá se něco vylepšit?

3 Programovací jazyky a nástroje

3.1 Programovací jazyky

Existují různé programovací jazyky, mezi nimiž panuje více či méně určitá příbuznost. Hodně známých programovacích jazyků vychází z jazyka C, například C++, C#, Java. V programovacím jazyce tvoříme obvykle zdrojový kód, což je text obsahující příkazy v určité smysluplné struktuře.

Programovací jazyky jsou buď *překládané* nebo *interpretované*. V případě překládaného jazyka je cílem vytvořit spustitelný soubor (například .exe ve Windows), v případě interpretovaného jazyka se spustitelný soubor nevytváří a účelem je kód přímo provést.

Při programování se musíme držet určité *syntaxe*, to znamená, že konkrétní kroky postupu (kód není nic jiného než popis postupu) zapisujeme určitým konkrétním způsobem – pokud toto pravidlo porušíme, program nepůjde zpracovat (přeložit, provést).

Většina programovacích jazyků má syntaxi odvozenou z programovacího jazyka C, včetně jazyků C++, C#, Java, Python. Znamená to, že hodně programových konstrukcí (tj. součástí kódu) zapisujeme v těchto jazycích stejně, ale neznamená to, že všechny. Následují ukázky kódu v jazycích C a C++.

Ukázka kódu v jazyku C, soubor svetc.c

```
/* **************************************************
Tento program načte od uživatele číslo a pak ho vypíše.
***************************************************** */
#include <stdio.h>

int main(void) {
   int vstup;
   printf("Zadej číslo: ");
   scanf("%d", &vstup);                  // Tady načteme číslo od uživatele.
   printf("Zadáno číslo %d.\n", vstup);   // Tady vypisujeme načtené číslo.
   return 0;
}
```

Jeden z odvozených jazyků je C++, podíváme se na jeho syntaxi:

Ukázka kódu v jazyku C++, soubor svetcpp.cpp

```
/* **************************************************
Tento program načte od uživatele číslo a pak ho vypíše.
***************************************************** */
#include <iostream>
using namespace std;

int main(void) {
   int vstup;
   cout << "Zadej číslo: ";        // Tady načteme číslo od uživatele.
   cin >> vstup;                   // Tady vypisujeme načtené číslo.
   cout << "Zadáno číslo " << vstup << endl;
}
```

Když tyto dvě ukázky srovnáme, zjistíme, že něco je stejné a něco je jiné. V obou případech máme na začátku víceřádkový komentář (uzavřený do lomítek+hvězdiček – pozor na pořadí), u dvou

z následujících řádků máme lokální komentáře začínající dvěma lomítky a platné do konce řádku. Komentáře pro samotný program nejsou důležité, ale mohou být důležité pro programátora.

Následuje řádek `#include`, za nímž v každém z kódů najdeme něco jiného. Jde o načtení statické knihovny pro vstupy a výstupy, v každém z jazyků obvykle používáme jinou. Pro C++ (a především knihovnu iostream) nastavujeme obor názvů. V knihovně iostream je totiž více oborů názvů používajících různé příkazy. Kdybychom na začátku neurčili obor názvů, museli bychom určovat u každého příkazu z této knihovny, v našem případě:

```
std::cout << "Zadej číslo: ";
std::cin >> vstup;
std::cout << "Zadáno číslo " << vstup << std::endl;
```

Což by nám zbytečně znepřehledňovalo kód.

Knihovna stdio.h, kterou pro vstupy a výstupy používáme obvykle v jazyce C, má pro běžné vstupy a výstupy funkce `printf()` a `scanf()`. Knihovna iostream, kterou pro tyto účely máme v C++, používá objekty `cout` a `cin`. V používání těchto funkcí/objektů se, jak vidíme, oba jazyky liší.

3.2 Stručně o nástrojích

Aby mělo programování smysl (tj. abychom vytvořili použitelný program), potřebujeme překladač. Překladač je nástroj, který dokáže ze zdrojového kódu vytvořit spustitelný soubor, nebo v případě interpretovaného jazyka tento kód rovnou provést. První typ překladače se nazývá kompilátor, druhý interpret.

Pozor, překladač není ten program, ve kterém tvoříme kód. K tomu máme *editory*, které nicméně mohou být propojeny s některým překladačem (nebo nemusejí).

Vývojové prostředí (IDE = Integrated Development Environment) je komplexní nástroj, který obsahuje překladač, editor zdrojového kódu a další nástroje, například na tvorbu grafického rozhraní aplikace.

Programovací jazyk je jedna věc, vývojové prostředí druhá. V tomtéž programovacím jazyce můžeme programovat ve více různých prostředích, a naopak v jednom vývojovém prostředí může být možné používat několik různých programovacích jazyků.

Pokud budeme programovat v C a C++, můžeme používat například některý z těchto produktů:

- C++ Builder Community Edition od společnosti Embarcadero, můžete používat zdarma do doby, než začnete programováním vydělávat víc než 5000 dolarů ročně nebo budete v týmu s více než 5 lidmi (včetně vás),
- Microsoft Visual Studio – komerční nástroj, který se při instalaci „nahrne“ na disk C:, až ve vyšších edicích lze jednoduše vytvářet grafické rozhraní aplikace, v community edici vytváříme spíše konzolové aplikace (bez grafického rozhraní),
- Visual Studio Code není komplexní IDE, při vývoji v C/C++ je to pouze editor, obvykle se propojuje s překladačem Mingw pocházejícím z Linuxu (ve Windows běží ve virtuálu), je pro nekomerční použití zdarma,
- Dev-C++ je vývojové prostředí dostupné zdarma, instalace zahrnuje překladač Mingw,
- v Linuxu můžeme použít překladač gcc (pro jazyk C), případně g++ (pro C++), píšeme v jakémkoliv textovém editoru pracujícím s čistým textem. Mingw je odnož programu gcc.

Kromě „instalovatelných“ nástrojů existují také online nástroje, vpodstatě webové aplikace nabízející editor, překladač a případně něco dalšího. Online editory obvykle běží ve virtuálu a používají překladač gcc.

Odkazy:

- C++ Builder Community Edition: https://www.embarcadero.com/products/cbuilder/starter
- Visual Studio Code: https://code.visualstudio.com/Download
- Visual Studio Community Edition: https://visualstudio.microsoft.com/cs/vs/community/
- Dev-C++: informace najdete na https://www.bloodshed.net/ , ke stažení na https://sourceforge.net/projects/orwelldevcpp/

- OnlineGDB: https://www.onlinegdb.com/online_c++_compiler
- CPP.sh: https://cpp.sh/
- JDoodle: https://www.jdoodle.com/online-compiler-c++/
- Programiz: https://www.programiz.com/cpp-programming/online-compiler/

Můžeme vyzkoušet různé možnosti. V učebně je ve VirtualBoxu Linux a v něm najdeme překladač gcc, tedy můžeme zkusit i ten:

Úkol:

V učebně je ve Virtual Boxu nainstalován Linux. Podle pokynů vyučujícího spusťte Virtual Box a v něm najděte Linux Mint, spusťte. Počkejte, až systém najede. Přihlašovací jméno i heslo jsou nastaveny na „student“ (bez uvozovek).

V menu (na horním panelu) *Aplikace – Příslušenství* najdete textový editor. Spouští se buď odsud, nebo třeba příkazem xed (jako parametr mu můžeme předat název souboru, který chceme otevřít). Tento textový editor sice pracuje s čistým textem podobně jako ve Windows program Poznámkový blok, na rozdíl od něj ale dokáže vysvicovat syntaxi, tedy pokud ví, že jde o kód v určitém programovacím jazyce, pak různými barvami zobrazí různé části kódu.

Do souboru přepište obsah souboru `svetc.c`, který je v předchozí sekci jako ukázka kódu v jazyce C. Soubor uložte (nezapomeňte na to, že má mít příponu .c, a to malým písmenem – pozor, rozlišují se malá a velká písmena). Rozlišování malých a velkých písmen platí i v kódu. Uložit můžete i průběžně. Právě jste vytvořili *zdrojový soubor programu*, neboli soubor se zdrojovým kódem.

Poklepáním na ikonu „Domov student“ se dostanete do správce souborů, kde si můžete ověřit, kam konkrétně se váš soubor uložil a zda má správnou příponu.

Otevřete si terminál. To provedete buď tak, že tuto položku najdete v menu Aplikace – Systémové nástroje, nebo můžete použít ikonu s černým obdélníkem na spodním panelu (dole na obrazovce). Předpokládejme, že se terminál otevře na tom místě, kde máte vytvořený soubor svetc.c. Napište tento příkaz:

```
gcc svetc.c –o svetc
```

Tímto příkazem váš soubor se zdrojovým kódem přeložíte, právě jste použili překladač jazyka C. Pokud jste v kódu udělali chybu, bude vám patřičně vynadáno. Pokud jste chybu neudělali, pak platí „žádná zpráva dobrá zpráva“. Nicméně klidně můžete vyzkoušet i spáchání chyby.

Vytvořili jste spustitelný soubor svetc. Nemá příponu, protože v Linuxu spustitelné soubory příponu nemívají. Program spustíte takto:

```
./svetc
```

Návod:

Pokud zjistíte, že program gcc není nainstalován (v učebně je), jednoduše ho nainstalujete takto:

```
sudo apt install gcc
```

Podobně se dá nainstalovat i překladač jazyka C++:

```
sudo apt install g++
```

Kód v C++ (například ten, který je ve výše uvedené ukázce kódu v C++) bychom přeložili takto:

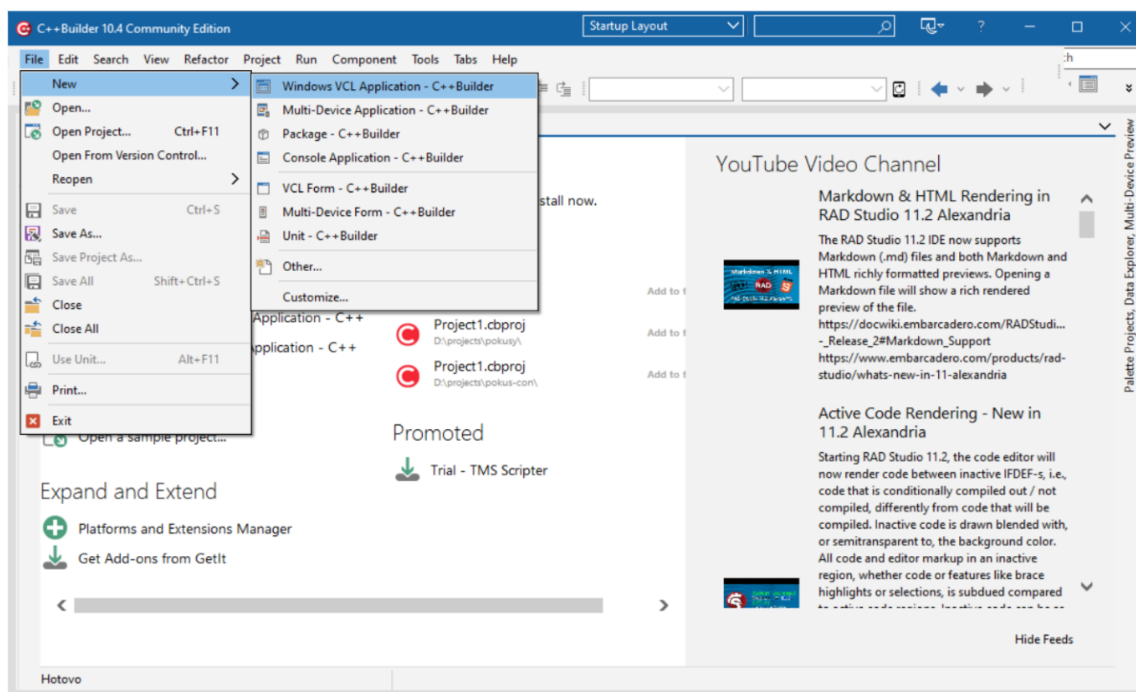
```
g++ svetcpp.cpp -o svetcpp
```

V případě produktu typu IDE (tj. vývojové prostředí) nepracujeme pouze s jedním či dvěma soubory, ale s tzv. projektem, což je sada souborů pro různé účely. Navíc můžeme mít na výběr mezi několika typy aplikace, kterou chceme vytvořit, tedy si při vytváření nového projektu určujeme, zda chceme:

- konzolovou aplikaci – nebude mít grafické rozhraní, s uživatelem se bude bavit textově,
- aplikaci s oknem (window application), případně konkrétní druh okenní aplikace, může být aplikace s jedním či více okny, může to být MDI aplikace (jedno hlavní okno a dále vnořená „child“ okna), formulářová aplikace (to je totéž jako okenní – okno se navrhuje pomocí formuláře),…
- knihovna (např. pro některou aplikaci ve Windows můžeme naprogramovat DLL knihovnu).

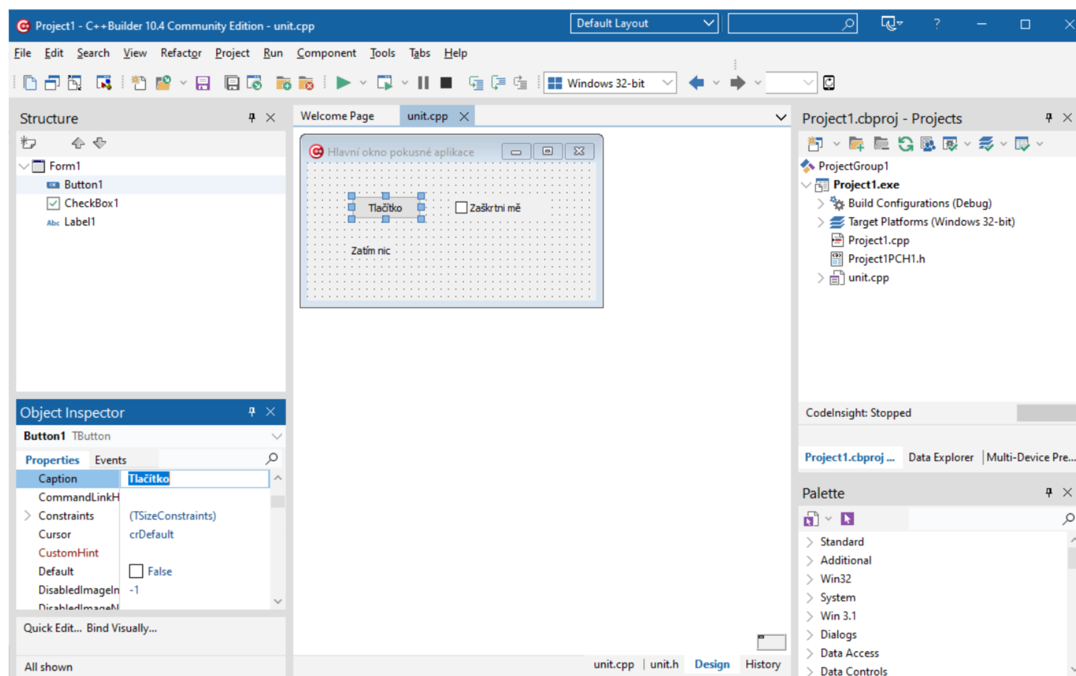
Úkol:

Teď se podíváme na C++ Builder od společnosti Embarcadero. Po spuštění si můžeme vybrat typ aplikace, kteoru vytvoříme. Nejčastěji budeme volit konzolovou aplikaci nebo Windows VCL aplikaci. Konzolová aplikace běží jen v „textovém okně“, kdežto Windows VCL aplikace má grafické rozhraní, které budeme navrhovat v režimu Designer přetahováním myší a doplňováním vlastností.



V tomto vývojovém prostředí pracujeme s projektem, nikoliv pouhým jedním souborem. Do projektu patří jak soubory se zdrojovým kódem, tak i další pomocné soubory, včetně těch s popisem grafického rozhraní vytvářené aplikace.

Předpokládejme, že jsme si vybrali Windows VCL Application. Prostředí je celkem přehledné, jak vidíme na obrázku níže: dole je možné přepínat mezi soubory projektu a režimem návrhu grafického prostředí.



Při přesunu do režimu Design máme uprostřed formulář pro návrh okna aplikace a v okolních podoknech různé pomocné struktury a nástroje. Prvky uživatelského rozhraní máme vpravo dole, vlastnosti vložených prvků (Object inspector) vlevo dole.

Podle pokynů vyučujícího se pusťte do průzkumu prostředí a vytvoření jednoduché aplikace. V Object inspectoru jsou dvě záložky – jedna pro vlastnosti prvku, druhá pro události související s vybraným objektem. Naše aplikace bude reagovat na stisknutí tlačítka a klepnutí myší na zaškrtávací pole. Stačí v Object inspectoru při výběru konkrétního prvku prostředí na kartě Events poklepat myší do prázdného pole u požadované události a pak v automaticky vytvořené funkci přidat kód. V našem případě do funkce pro tlačítko vepíšeme:

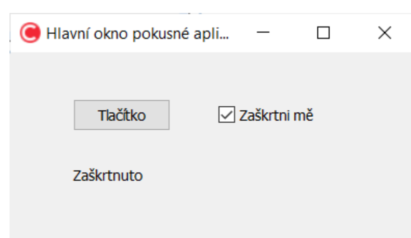
```
Label1->Caption = "Stisknuto tlačítko";
```

A do funkce pro zaškrtávací pole vepíšeme:

```
if(CheckBox1->Checked) {
      Label1->Caption = "Zaškrtnuto";
   }
   else Label1->Caption = "Nezaškrtnuto";
}
```

Až budeme hotovi, pošleme náš projekt překladači. Buď použijeme položku menu Run, nebo zelený trojúhelník v panelu nástrojů pod menu. Projděte si různé možnosti s vyučujícím.

Po spuštění by aplikace měla vypadat nějak takto (tedy po zaškrtnutí zaškrtávacího pole):



4 Začátek

4.1 Základní tvar programu a hlavní funkce

Jednoduchý program v jazyce C++, který bude vypisovat řetězec „Hello World“, může vypadat třeba takto:

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

int main()
{
    cout << "Hello World";
    return 0;
}
```

Všimněte si, že nejdřív načítáme knihovny (v našem případě jednu knihovnu), a to pomocí direktivy #include, k načtení knihovny iostream ještě přidáváme stanovení jmenného prostoru. Direktivy zatím nemusíme řešit, prostě na začátku programu budeme mít #include.

Dále máme „nějaký“ main a spoustu závorek. Main() je hlavní funkce programu a její kód píšeme do složených závorek { }, které k ní přiléhají. Mohou existovat i další funkce (později je budeme taky programovat), ale main je vždy nejdůležitější.

Každá funkce má svůj název (třeba main), za kterým jsou kulaté závorky a u některých funkcí máme v těchto závorkách parametry určující, s čím má funkce pracovat. Pokud třeba i jen v textu použijeme název funkce, je zvykem přidat kulaté závorky, i kdyby v nich nemělo nic být – pomáhá to odlišit funkce od jiných prvků v programu. Proto budeme také v těchto odstavcích psát main().

Složené závorky píšeme u každé funkce, do nich umístíme kód této funkce (příkazy).

Poznámka:

Pokud soubor se zdrojovým kódem uložíte s příponou .C, měl by být jako poslední příkaz funkce `main()`

```
return 0;
```

Pokud soubor uložíte s příponou .CPP, tak to obvykle není nutné.

4.2 Jednoduchý vstup a výstup

Pokud programujeme aplikaci s grafickým rozhraním, pak zřejmě budeme mít okna, tlačítka, menu atd. Jestliže však programujeme jednoduchou konzolovou aplikaci, tak máme k dispozici tyto možnosti:

- standardní vstup, výstup a chybový výstup,
- soubory, těmi se zatím nebudeme zabývat.

Standardní vstup (`stdin`) je obvykle z klávesnice – co nám tam uživatel naklepe, to bude náš vstup. Standardní výstup (`stdout`) je na obrazovku, standardní chybový výstup (`stderr`) taky. Nicméně uživatel může vstupy i výstupy přesměrovat někam jinam, ale tím se zatím nemusíme zabývat.

S tím jsme se vlastně už setkali v předchozí kapitole: to, jak budeme pracovat se vstupem a výstupem, záleží na I/O knihovně, kterou zvolíme.

- stdio.h se hodně používá v jazyce C, ale je použitelná i v C++, pro standardní vstup máme funkci `scanf(...)`, pro standardní výstup funkci `printf(...)`.
- iostream je pro C++, pro vstup používáme `cin`, pro výstup `cout`. Měli bychom určit namespace.

Úkol:
Použijte některý online překladač (viz str. 13), klidně několik, ať si otestujete, které prostředí vyhovuje nejlépe. Obvykle tam najdete něco jako „šablonu“ většinou vypisující „Hello World“ nebo něco podobného. Podívejte se také, jestli se dá kód uložit do souboru. Pokud ne, žádná tragédie, můžete zkopírovat a uložit si ručně. Vyzkoušejte kód z ukázek na straně 11. Pokud budete kopírovat, pozor na uvozovky. Srovnejte zápis funkcí printf/scanf a cin/cout.

4.3 Komentování

Jednoduchý kód není nutné vysvětlovat, ale u složitějšího či delšího kódu obvykle píšeme komentáře. V jazycích C a C++ používáme dva typy komentářů – krátký jednořádkový a delší, který může být přes víc řádků. Oba typy jsme mohli vidět v ukázkách.

```
// Toto je jednořádkový komentář: od dvojice lomítek do konce řádku.
/* Toto je víceřádkový komentář, který nicméně může být i na jeden řádek.
   je vždy od lomítka hvězdičky až k hvězdičce lomítku. */

/************************************************************************/
/*  Pokud chceme vytvořit informační záhlaví souboru se zdrojovým kódem, */
/*  můžeme to udělat třeba takto.                                        */
/************************************************************************/
```

4.4 S jakými daty můžeme při programování pracovat

Pokud chceme vynásobit čísla 1028 a 2392, nemusíme kvůli tomu psát program. Větší smysl má napsat program, který umí vynásobit jakákoliv dvě čísla, která mu budou zadána. Programy tedy píšeme tak, aby byly použitelné pro různé vstupy. Této vlastnosti říkáme *hromadnost*.

V kapitole o vývojových diagramech jsme se už setkali s proměnnými, obvykle číselnými. Proměnná (jak název napovídá) má proměnlivý obsah – může se měnit, ale jedna věc se nemění: *datový typ*. Datový typ určuje, jaký typ dat můžeme do proměnné uložit, většinou použijeme:

Tabulka 1 Základní datové typy

Datový typ	**Co s ním**	**Příklad:**
`int`	celé číslo, délka 4 B	int i = 25;
`short`	celé číslo, délka 2 B, zkratka pro `short int`	short s = 25;
`long`	celé číslo, délka 8 B, zkratka pro `long int`	long x = 25;
`float`	racionální číslo, délka 4 B	float m = 3.82;
`double`	racionální číslo, délka 8 B	double d = 82.55;
`char`	znak, délka 1 B, vnitřně je to taky číslo	char c = 'a';
`string`	řetězec (proměnné s knihovnou)	string s = "Ahoj";
`void`	tam, kde přímo nechceme žádný datový typ psát (= neurčitý typ)	

Dále existují odvozené varianty, které mají před „hlavním“ názvem ještě přídavek. Například `long` určuje variantu zabírající více paměti, tedy můžeme uložit i delší čísla:

```
long int velkeCeleCislo;         // můžeme napsat i zkráceně  long velkeCeleCislo
long long hodneVelke;            // ještě větší číslo, long long int
long double velkeRacionalni;     // zabere 12 B místo 8 B
```

Když napíšeme jenom `long`, pak to ve skutečnosti znamená `long int`, jde o zkratku.

Podobně funguje přídavek short, ale má opačný význam: takové číslo bude zabírat méně místa v paměti a je určeno pro malé hodnoty.

```
short int maleCislo;    // může být zkráceno na    short maleCislo
```

Přídavek unsigned určuje číslo bez znaménka. Standardně jsou typy int, float, double se znaménkem, tedy můžeme do nich ukládat i záporná čísla. Jenže v takovém případě je jeden bit z proměnné vyhrazen pro znaménko a nemůže být využit pro uložení číselné hodnoty proměnné. Takže unsigned znamená, že pro číselnou hodnotu proměnné máme o jeden bit víc místa, vejde se tam tedy větší číslo.

```
unsigned int cisloBezZnamenka;
unsigned short maleCisloBezZnamenka;
unsigned float realneBezZnamenka;
```

Kromě čísel a znaků máme k dispozici i další datové typy, ke kterým se dostaneme později.

Proměnnou je třeba deklarovat. Tedy předtím, než ji použijeme, vysvětlíme programu, že tato proměnná je určitého datového typu. Program to potřebuje vědět, protože pro proměnnou předem rezervuje potřebné místo v kódu. Při dekladaci je dobré proměnnou inicializovat, tedy přiřadit počáteční hodnotu (třeba nulu u čísla), abychom předešli omylu s náhodným obsahem.

```
int rok;                  // deklarace bez inicializace
int rok = 2022;           // deklarace s inicializací
int den, mesic, rok = 2022; // deklarace tří proměnných, jedna je inicializována
```

Kromě proměnných můžeme používat konstanty. Konstanta bývá taky určitého datového typu, ale na rozdíl od proměnné se její hodnota nemění. Může se to zdát jako nevýhoda, ale na druhou stranu nehrozí, že její hodnota bude změněna omylem. Některé postupy totiž mohou mít tzv. side effect (vedlejší efekt), který ne vždy čekáme.

```
const double PI = 3.14159;   // konstanta typu double
```

Přetypování znamená, že s určitou proměnnou či konstantou budeme zacházet jinak než jak odpovídá jejímu datovému typu.

Vyzkoušejte – jaký dostanete výsledek?

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

Přetypování může automaticky dělat samotný překladač, bohužel je někdy důsledkem to, že pokud programátor udělá chybu, nedozví se to (protože překladač myslel příliš samostatně).

Vyzkoušejte:

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

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

Další informace:

https://en.cppreference.com/w/cpp/language/types

4.5 Řetězce

Už jsme se setkali s tím, že můžeme třeba na výstup vypsat „Hello World!“ nebo nějaký jiný řetězec. Ať už používáme kterýkoliv způsob výstupu, do řetězce můžeme vpašovat tzv. *escape sekvence*:

Tabulka 2 Escape sekvence v řetězcích

Escape sekvence	**Význam**
\n	symbol konce řádku
\t	tabulátor
\r	návrat na začátek řádku
\v	vertikální tabulátor

Například:

```
printf("Hello World!\n");
cout << "Hello World!" << endl;      // bez escape sekvence
cout << "Hello World!\n";            // s escape sekvencí, nemusíme použít endl
```

Pokud bychom potřebovali pracovat s řetězci trochu víc než jen vypsat konstantní řetězec, máme k dispozici knihovnu `string`, kterou bychom načetli takto:

```
#include <string>
```

Práce s řetězci

Napíšeme program, který bude používat proměnnou datového typu string. Do této proměnné načte od uživatele jeho jméno (pozor, bez mezery), přidá ke jménu „z Opavy“ a pak vypíše ve formě pozdravu. Skloňování nebudeme brát v úvahu.

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

int main()
{
   string jmeno;
   cout << "Jak se jmenuješ? ";
   cin >> jmeno;

   jmeno += " z Opavy";
   cout << "Ahoj, " << jmeno  << "!\n";
}
```

Datový typ string umožňuje používat operátor „+“ pro zřetězení dvou řetězců, případně „+=“ pro přidání řetězce na konec jiného. Další operátory už vítány nejsou, a taky nemůžeme takto propojit řetězec a číslo.

Příklad

Máme tři řetězce, které spojíme v jeden výsledný:

```
string privlastek, podmet, prisudek, predmet;
podmet = "hrnek ";
privlastek = "modry ";
prisudek = "obsahuje ";
predmet = " kafe";
```

```
cout << privlastek + podmet + prisudek + predmet << endl;

privlastek = "zeleny ";
predmet = "caj";
cout << privlastek + podmet + prisudek + predmet << endl;

podmet = "cajovnik ";
prisudek = "poskytuje ";
cout << privlastek + podmet + prisudek + predmet << endl;
```

Všimněte si, že na konci hodnot většiny proměnných je přidaná mezera. Tento program by postupně vypsal tyto řádky:

```
modry hrnek obsahuje kafe
zeleny hrnek obsahuje caj
zeleny cajovnik poskytuje caj
```

Tentýž „vypisovací“ řádek by mohl být jinak:

```
cout << privlastek << podmet << prisudek << predmet << endl;
```

Popřípadě pokud bychom neměli na konci obsahu příslušných proměnných mezeru, museli bychom ji dodat navíc:

```
cout << privlastek << ' ' << podmet << ' ' << prisudek << ' ' << predmet << endl;
```

Nebo naopak můžeme přidat mezery pomocí operátoru „+“:

```
cout << privlastek + ' ' + podmet + ' ' + prisudek + ' ' + predmet << endl;
```

Úkol:

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 (místo mezery může uživatel použít podtržítko). Taktéž si pohlídejte, aby mezi dvěma prvky na tomtéž řádku byla mezera.

4.6 Jak vyrobit náhodné číslo

Když chceme náhodné číslo, použijeme generátor náhodných čísel. Jde sice jen o pseudonáhodná čísla (generátor použije například momentální čas, vlastnosti hardwaru, některé údaje od operačního systému apod.), ale obvykle to stačí.

V jazyce C/C++ potřebujeme tyto dvě funkce:

- `void srand(číslo)` – „nastartuje“ generátor, zvolené číslo ovlivňuje náhodnost čísel,
- `unsigned int rand()` – vrátí nám náhodné číslo.

První z těchto dvou funkcí použijeme jednou (jako iniciační číslo obvykle používáme momentální čas, tedy časové razítko), druhou používáme tolikrát, kolik potřebujeme náhodných čísel.

Příklad:

Vygenerujeme jedno náhodné číslo a vypíšeme. Pak počkáme, až uživatel stiskne jakoukoliv klávesu.

```
#include <iostream>
using namespace std;
int main()
{
   srand(time(0));
   cout << rand() << endl;
   cin.get();
}
```

Úkol:

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();
```

4.7 Zpět ke vstupům a výstupům

Téměř od začátku se setkáváme s tím, že je třeba komunikovat s uživatelem, tedy řešit vstupy a výstupy programu. Většinou jsme používali knihovnu `iostream.h`, tedy vstupy pomocí `cin` a výstupy pomocí `cout`. Ale už na začátku jsme si ukázali původní „céčkový“ přístup (knihovnu `stdio.h` s funkcemi `printf()` a `scanf()`).

Shrneme si obě možnosti. Je celkem jedno, kterou z nich použijeme. V C++ je sice více zvykem používat knihovnu `iostream.h`, protože je objektová, ale ani s knihovnou `stdio.h` není problém, záleží na zvyku a osobních preferencích.

4.7.1 Vstup a výstup tradičně: stdio.h

V knihovně `stdio.h` máme k dispozici především funkce `printf()` a `scanf()`, které už jsme měli v ukázkách na začátku semestru. U funkce `scanf()` obvykle načítáme obsah určité proměnné, její datový typ bychom měli určit pomocí modifikátoru. Například:

```
int x;
scanf("%d", &x);
```

U obou funkcí obvykle zadáváme nejdřív formátovací řetězec, a pak to, co se do daného řetězce má dosadit (třeba proměnné). vše navzájem oddělené čárkami. Konkrétní namapování proměnných do formátovacího řetězce musí být naprosto jasné a musí sedět i jejich počet. Například:

```
printf("Počet koček v domácnosti: %d", kocky);
printf("Doma máme %d koček a %d psů, k tomu %d myší. ", kocky, psi, mysi);
```

Modifikátor `%d` určuje, že od uživatele načítáme celé číslo. Nejobvyklejší modifikátory (řídicí řetězce) jsou v následující tabulce.

Tabulka 3 Modifikátory pro funkce z knihovny stdio.h

Datový typ	**V printf/scanf:**
`int`	%d (desítkově)
	%x, %X (hexadecimálně)
	%o (osmičková soustava)
`long`	%ld
`unsigned long`	%lu

float	%f (desetinná tečka)
	%e (exponenciální tvar)
double	%f, %lf
long double	%Lf (velé „L“!)
char	%c
unsigned int	%u
string	%s

Modifikátory se používají při vstupu i výstupu, a to jak u proměnných, tak i u konstant.

```
printf("Kolik je hodin: %d:%d", hod, min);  // hod a min jsou celočíselné proměnné
```

```
char c = 'a';
printf("Znak %c má ASCII hodnotu %d (%X hexadecimálně)");
```

K modifikátoru můžeme přidat číslo určující počet míst (třeba cifer celého čísla), případně počet desetinných míst. Následující příkaz zarovná hodiny i minuty vždy na dvě pozice:

```
printf("Kolik je hodin: %2d:%2d", hod, min);
```

Pro určení desetinných míst:

```
printf("Cena bonbónů: %3.2f Kč", cena);
```

Pokud chceme ve formátovacím řetězci použít něco se speciálním významem, například symbol procenta, zdvojíme ho. Totéž platí o zpětném lomítku.

```
printf("Úspěšnost řešení byla %d %%.", uspesnost);
printf("Zde používám zpětné lomítko \\, zde musí být dvojité, vypíše se jedno. ");
```

Poznámka:

Zatímco u `printf` píšeme za formátovacím řetězcem přímo proměnné, jejichž obsah se má dosadit, u `scanf` jste si určitě všimli ampérsándu (&):

```
scanf("%d", &hodnota);
printf("Zadáno: %d", hodnota);
```

Ampérsánd slouží jako reference (odkaz) na adresu dané proměnné, funkce scanf totiž potřebuje získat přímo tuto adresu, aby tam uložila načtenou hodnotu. Funkce printf tuto hodnotu nemění, pro získání hodnoty z proměnné přímo adresu nepotřebuje.

V knihovně `stdio.h` máme kromě funkcí printf() a scanf() také další funkce – pro práci s řetězci, soubory, streamy.

4.7.2 Vstup a výstup pomocí streamů: iostream.h

Pod pojmem stream rozumíme proud dat. Vstupní stream je jednoduše proud těch dat, která (postupně) přijímáme od uživatele, výstupní stream je proud dat, která (postupně) zapisujeme na výstup, aby si je uživatel mohl přečíst.

V knihovně `iostream.h` jsou definovány tři základní streamy:

- `cin` (objekt typu `istream`) – pro standardní vstup `stdin`, obvykle míří z klávesnice,
- `cout` (objekt typu `ostream`) – pro standardní výstup `stdout`, obvykle na obrazovku,
- `cerr` (objekt typu `ostream`) – pro standardní chybový výstup `stderr`, obvykle obrazovka.

Používáme většinou operátory << a >>, třebaže existují i funkce těchto objektů (vlastně s některými jsme se už taky setkali).

Co se týče chybového výstupu, umožňuje nám vypsat chybová hlášení, přičemž fakt, že je použit cerr (resp. stderr) místo běžného streamu pro výstup, je také informací pro okolí procesu. Takže například můžeme napsat:

```
cerr << "Chyba při aritmetické operaci – dělení nulou. ";
```

V předchozí sekci byla řeč o manipulátorech pro funkce printf, scanf, fprintf a fscanf. Manipulátory existují i pro objekty z iostream, jen se s nimi zachází trochu jinak.

Příklad:

Pokud chceme vypsat číslo tak, aby se zobrazilo v šestnáctkové soustavě, použijeme manipulátor hex:

```
cout << hex << 25;
```

Pro vstup přes cin by platilo totéž – uživatel zadá číslo v šestnáctkové soustavě:

```
cin >> hex >> x;
```

Některé manipulátory jsou dostupné přes knihovnu iostream.h (včetně hex nebo třeba endl, který už taky známe), pro jiné potřebujeme načíst knihovnu iomanip.h.

Tabulka 4 Přehled nejpoužívanějších manipulátorů pro streamy

Manipulátor	Význam
endl	zařádkuje
hex	od této chvíle bude vstup/výstup v šestnáctkové soustavě
oct	osmičkové soustavě
dec	desítkové soustavě
boolalpha	vypíše pravdivostní hodnotu „true“ nebo „false“, podle následující položky
setprecision(4)	nastaví počet míst (přesnost) pro výpis čísla, zde 4
showpos	bude se vypisovat znaménko čísla
setw(10)	nastavení šířky následující položky, zde 10 (například celkový počet číslic čísla nebo celková šířka řetězce), vytvořený prostor vlevo se vyplní výplňovým znakem (výchozí je mezera)
setfill('0')	nastaví výplňový znak, zde číslice nula (může být třeba tečka, mezera apod.)

Příklad:

U racionálních čísel (tedy s desetinnými místy) má smysl určovat, kolik desetinných míst se má vypsat. Často zneužívané Ludolfovo číslo π (3,14159…) využijeme i pro ukázku určení počtu desetinných míst. Pak si ukážeme, že počet desetinných míst můžeme zadávat nejen jako konstantní hodnotu, ale můžeme ji dodat v proměnné (kterou v našem případě načteme od uživatele).

```
float pi=3,14159265358;

cout << pi << endl;
cout << setprecision(2) << pi << endl;    // máte načtenou knihovnu iomanip?
cout << setprecision(3) << pi << endl;
```

```
cout << setprecision(4) << pi << endl;
cout << setprecision(5) << pi << endl;

int mista;
cout << "Kolik desetinných míst? ";
cin >> mista;
cout << setprecision(mista) << pi;
```

Příklad pro pokročilé:

Pokud chceme zobrazit momentální čas, můžeme použít manipulátor `put_time()`:

```
#include <iostream>
#include <iomanip>
#include <ctime>
#include <chrono>
using namespace std;
using namespace std::chrono;
using std::chrono::system_clock;

int main ()
{
   time_t myTime = system_clock::to_time_t(system_clock::now());
   struct tm * pDisplayedTime = localtime(&myTime);

   cout << "Tolik je právě hodin: " << put_time(pDisplayedTime,"%c") << endl;
}
```

Další informace:

O některých manipulátorech (přesněji: těch definovaných v knihovně `iomanip.h`) se dočteme například na https://cplusplus.com/reference/iomanip/

5 Výrazy a podmínky

5.1 Konstanty

Konstantu můžeme buď deklarovat a pojmenovat, nebo přímo psát v kódu. Obě možnosti už jsme použili, tedy pro „běžné“ datové typy. Pro zopakování:

```
const int N = 10;    // deklarovali jsme konstantu N, její hodnota je 10
x = 25 – y;          // použili jsme číselnou konstantu 25
cout << "Ahoj lidi"; // použili jsme řetězcovou konstantu "Ahoj lidi"
char zn = 'a';       // proměnnou jsme inicializovali znakovou konstantou 'a'
```

Řetězcové konstanty uzavíráme do uvozovek, znakové konstanty do apostrofů.

Co když chceme zadat něco jiného než “běžnou” hodnotu nebo použít v kódu konstantní hodnotu s určením datového typu?

Jiné číselné soustavy:

```
024     takto zadáme číslo v osmičkové soustavě (nula na začátku)
0x8af   takto zadáme číslo v šestnáctkové soustavě (nula a písmeno „x“ na začátku)
```

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).

Určení datového typu konstantní hodnoty není obvykle nutné, ale někdy se může hodit. Například písmeno „u“ nebo „U“ na konci čísla znamená „unsigned int“, písmeno „L“ znamená „long double“ (používejte raději velké „L“).

Příklad:

Vyzkoušejte a srovnejte výstup těchto příkazů:

```
cout << sizeof(25U) << endl;
cout << sizeof(25L);
```

Pokud v obou případech dostanete stejný počet Bytů, pak zrovna používáte platformu, která pro tyto datové typy vyhrazuje stejné množství paměti. Na některých platformách (zejména v 64bitových systémech) byste měli získat jiné hodnoty, pravděpodobně 4 B pro `unsigned` a 8 B pro `long`.

5.2 Aritmetické operace

S aritmetickými operacemi si již dokážeme celkem poradit. Víme, jak utvořit matematický výraz s běžnými operátory, vyzkoušeli jsme si také speciální unární operátory `++` a `--`.

Unární operátory:

```
-i      // unární "mínus", je to pouze operátor, není samostatný příkaz
i++     // inkrementace (přičítám jedničku), je operátor a zároveň příkaz
++i     // vpodstatě totéž, ale při použití ve výrazu pozor na pořadí operací
i--     // dekrementace (odečítám jedničku)
--i
```

Speciální formy přiřazovacího příkazu:

```
i += 5    // inkrementace o 5
i -= 5
i *= 5
i /= 5
i %= 5    // operace modulo (zbytek po celočíselném dělení)
k = --i+3 // jakýkoliv unární operátor lze využívat ve výrazu s jinými operátory
```

Protože operátor přiřazení vrací to, co je přiřazováno, můžeme ho takto zřetězit:

```
x = y = z = a+3
```

Platí běžné priority operátorů, například násobení a dělení má vyšší prioritu než sčítání a odčítání. Operátory se stejnou prioritou jsou prováděny v pořadí zleva (neplatí pro rovnítko).

Příklad:

Srovnejte tyto úseky kódu:

```
int i = 10;
int k = i++;
cout << k;
```

```
int i = 10;
int k = ++i;
cout << k;
```

V obou případech má druhý příkaz tzv. „vedlejší efekt“. Nejen že nastavuje hodnotu proměnné `k`, ale taky mění hodnotu proměnné `i`. Jde o to, co konkrétně skončí v proměnné `k`. V prvním případě 10,

protože inkrementace proběhne až po přiřazení, v druhém případě 11, protože nejdřív proběhne inkrementace a pak až přiřazení.

Příklad:

Podívejte se na následující kód:

```
char c = 65;
cout << c;
c += 1;
cout << c;
c += '1';
cout << c;
```

Znak jsme inicializovali na číslo, ale to určitě není problém, použije se ASCII kód (mimochodem, jde o písmeno velké „A“). Pak jsme přičetli jedničku, což znamená, že v ASCII tabulce jsme se posunuli na další písmeno („B“). Ale co dál? Přičetli jsme něco, co je sice taky jednička, ale ve smyslu znaku. Číslice ‚1‘ má ASCII kód 49, takže jsme vlastně přičetli 49 a pak proběhla automatická konverze na znak.

Příklad:

Vpravo vidíme ASCII tabulku pro kódování Windows 1250 (tedy ne celou, až od kódu 11, znaky s nižší hodnotou jsou netisknutelné). Existují různá kódování, mnohá mají první část tabulky stejnou (do kódu 127, což je právě polovina rozsahu pro 8bitové tabulky).

	0	1	2	3	4	5	6	7	8	9
10		♂	♀		♫	☼	►	◄	↕	‼
20	¶	§	▬	↨	↑	↓	→	←	∟	↔
30	▲	▼		!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	¦	}	~	⌂	Ç	ü
130	é	â	ä	ů	ć	ç	ł	ë	Ő	ő
140	î	Ź	Ä	Ć	É	Ĺ	ĺ	ô	ö	Ľ
150	ľ	Ś	ś	Ö	Ü	Ť	ť	Ł	×	č
160	á	í	ó	ú	Ą	ą	Ž	ž	Ę	ę
170		ź	Č	ş	«	»	░	▒	▓	│
180	┤	Á	Â	Ě	Ş	╣	║	╗	╝	Ż
190	ż	┐	└	┴	┬	├	─	┼	Ă	ă
200	╚	╔	╩	╦	╠	═	╬	¤	đ	Đ
210	Ď	Ë	ď	Ň	Í	Î	ě	┘	┌	█
220	▄	Ţ	Ů	▀	Ó	ß	Ô	Ń	ń	ň
230	Š	š	Ŕ	Ú	ŕ	Ű	ý	Ý	ţ	´
240	-	˝	˛	ˇ	˘	§	÷	¸	°	¨
250	↓	→	←	∟	↔	▲				

Všimněte si, jakou ASCII hodnotu mají číslice: začínají kódem 48 (to je číslice 0), končí kódem 57 (číslice 9). Malá písmena začínají na kódu 97 (písmeno ‘a’), kdežto velká písmena začínají už na kódu 65 (velké písmeno ‘A’). Takže nejdřív jsou velká písmena, pak až malá. Z velkého písmene bychom udělali odpovídající malé prostě tak, že bychom přičetli rozdíl mezi velkým a malým áčkem, tedy

```
znak += 'a' - 'A';
```

Naopak z malého písmene bychom udělali velké odečtením této hodnoty:

```
znak -= 'a' - 'A';
```

Ovšem museli bychom mít jistotu, že to předtím bylo velké, resp. malé písmeno.

Pokud potřebujeme něco víc než základní matematické operace, musíme načíst příslušnou knihovnu. Například hodně matematických funkcí je v knihovně `math.h`, tedy:

```
#include <math.h>
…
double f = sqrt(10);
```

Při práci s výrazy je třeba si dávat pozor také na hranice datového typu, C++ totiž při přetečení nevaruje. Vyzkoušejte si například toto:

Příklad:

```
unsigned short x = 70000;    // sedmdesát tisíc, jsou tam čtyři nuly
cout << x;
```

„Překvapivě“ se nevypíše 70000, ale 4464. Proč? Protože datový typ unsigned short zabírá 2 B, tedy 16 bitů, což nám dává 2^{16} = 65 536 různých hodnot – jinými slovy, do proměnné tohoto typu můžeme uložit čísla od 0 do $2^{16} - 1$, což je rozsah 0 – 65 535. Pokud jsme do proměnné uložili větší číslo než je horní ohraničení datového typu, došlo k přetečení (overlay). Pokud od 70 000 odečtete 65 536, dostaneme právě to číslo, které se vypsalo.

Vyzkoušejte následující:

```
unsigned short x = 65536;
cout << x;
```

Z toho plyne, že rozsah datového typu bychom si rozhodně měli hlídat.

5.3 Bitové operace

Každý byte se skládá z bitů, a někdy potřebujeme pracovat přímo s jednotlivými bity. Existují tyto bitové operátory:

```
x & y     bitový součin, AND
x | y     bitový součet, OR
x ^ y     bitový exkluzivní součet, XOR (stříška)
~x        bitová negace (vlnka)

x << n    bitový posun doleva o n bitů
x >> n    bitový posun doprava o n bitů
```

Binární operace (tj. ty se dvěma operandy) se dají používat i s přiřazovacím operátorem, například:

```
x &= 0xff;   // tímto jsme všechny bity kromě posledních osmi nastavili na nulu
x >>= 2;     // totéž jako x = x >> 2; posun o dva bity vpravo
```

Bitové operace typicky používáme na celá čísla, ideálně unsigned, protože znaménko může v bitových operacích udělat celkem chaos.

Vysvětlení bitových operací:

Bitovou operaci používáme na čísla tak, že postupně zpracováváme jednotlivé bity. Pokud je binární, vezmeme z každého operandu (čísla) bit na téže pozici a provedeme příslušnou operaci, toto postupně pro jednotlivé pozice.

Bitový součin AND vrací 1, pokud oba operandy jsou 1, jinak vrací 0:

AND	0	1
0	0	0
1	0	1

Pokud tuto operaci použijeme na dvě čísla, pak při ručním postupu je napíšeme pod sebe v binární soustavě a postupujeme po jednotlivých pozicích (po sloupcích). Například pokud provádíme výpočet 155 & 201:

155 je binárně 10011011, 201 je binárně 11001001.

```
     1 0 0 1 1 0 1 1
AND  1 1 0 0 1 0 0 1
-------------------
     1 0 0 0 1 0 0 1
```

Výsledek je binárně 10001001, což je v desítkové soustavě 137.

Pro bitový součet OR platí, že vrací 0, pokud oba operandy jsou 0, jinak vrací 1:

OR	0	1
0	0	1
1	1	1

Postupujeme podobně. Pro uvedená čísla vypočteme bitový součet 155 | 201:

```
      1 0 0 1 1 0 1 1
OR    1 1 0 0 1 0 0 1
-----------------------
      1 1 0 1 1 0 1 1
```

Výsledek je binárně 11011011, což je v desítkové soustavě 219.

Exkluzivní součet XOR má následující tabulku (vrací 1 pro dvojice 01 nebo 10, jinak vrací 0):

XOR	0	1
0	0	1
1	1	0

A u bitové negace je určitě zřejmé, že nulu zamění za jedničku a naopak: `~(11010) = (00101)`

Příklad:

Bitový součin se používá pro „vymaskování" určitých bitů, což vidíme o pár řádků výše. Například:

```
unsigned int x = 260;
x &= 0xff;           // hexadecimální 0xff je binárně 1111 1111
```

znamená:	*desítkově*	*binárně*	*hexadecimálně*
původní hodnota x	260	1 0000 0100	104
AND	256	0 1111 1111	ff
po změně	4	0 0000 0100	4

Úkol:

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.

Bitový součet (OR) se používá pro nastavení konkrétních bitů na jedničku (bez ohledu na jejich původní hodnotu).

Příklad:

Pokud je číslo sudé, zvýšíme ho o jedničku, ale pokud je liché, zůstane beze změny:

```
x |= 1;    // totéž jako  x = x | 1;
```

Bitový posun znamená, že posouváme bity čísla zadaným směrem o zadaný počet pozic. Příslušný operátor zapisujeme dvojicí znaků pro „menší" `<<` nebo „větší" `>>`, mezi nimi nemá být mezera. Směřování těchto dvojznaků (představme si dvojitou šipku) je stejné jako směr posunu binárních číslic.

Příklad:

Bitový posun `27 << 2` znamená, že číslo 27 (binárně 11011) v bitech posuneme o dvě pozice vlevo, tedy na pravé straně přidáme dvě nuly:

```
0001 1011    změním na:   0110 1100    desítkově = 108
```

Výše uvedený příklad předpokládá, že máme 8bitové číslo (char). Binární číslice na pravé straně se vlastně ztrácejí (zde jsme přišli o nuly, což není až takový problém, ale se ztracenými jedničkami by mohl problém nastat).

Bitový posun opačným směrem `27 >> 2` znamená, že se ztrácejí dvě číslice na levé straně.

```
0001 1011    změním na:   0000 0110    desítkově = 6
```

Bitové posuny lze s úspěchem použít pro násobení a dělení číslem 2.

Úkol:

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?

5.4 Logické operace, výrazy v podmínkách

S jednoduchými podmínkami jsme se už setkali – používáme různé *relační operátory*:

```
x == 10   vrací TRUE, pokud v proměnné x je právě číslo 10
x < 10    vrací TRUE, pokud v proměnné x je číslo menší než 10
x <= 10   vrací TRUE, pokud v proměnné x je číslo menší nebo rovno 10
x > 10    vrací TRUE, pokud v proměnné x je číslo větší než 10
x >= 10   vrací TRUE, pokud v proměnné x je číslo větší nebo rovno 10
x != 10   vrací TRUE, pokud v proměnné x je číslo jiné než 10 (negace rovnosti)
```

Pozor na rovnítka:

```
x == 10   zjišťujeme, zda v proměnné x je číslo 10 (vrací TRUE nebo FALSE)
x = 10    do proměnné x ukládáme číslo 10, tj. teď tam už bude (vrací číslo 10)
```

Relační operátor je tedy takový operátor, který jako operandy používá obvykle čísla a vrací hodnotu TRUE nebo FALSE, tedy pravdivostní hodnotu.

Pak máme ještě *logické operátory*. To jsou takové, které jako operandy mají pravdivostní hodnoty a vracejí taktéž pravdivostní hodnotu. V jazycích C a C++ používáme tyto logické operátory:

```
B1 && B2   logické AND: pokud oba operandy jsou TRUE, je výsledek TRUE, jinak FALSE
B1 || B2   logické OR: pokud oba operandy jsou FALSE, je výsledek FALSE, jinak TRUE
!B         logická negace: převrátí pravdivostní hodnotu
```

Všimněte si, že pro AND a OR používáme tytéž symboly jako pro bitové operace, jen jsou zdvojené. Nezapomínejte zdvojovat, jinak bude program reagovat poněkud jinak než bychom čekali!

Jeden relační výraz obsahuje právě jeden relační operátor, kdežto v logickém výrazu můžeme kombinovat jakýkoliv počet relačních výrazů spojených logickými operátory.

Příklad:

`((x > 0) && (x <= 25))` celý výraz vrací TRUE, pokud platí $x \in (0\,;25\rangle$

`((x-y > 0) && (x+y < 5))` dva relační výrazy jsme propojili logickým operátorem AND

`(((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z')))`

výraz vrací TRUE, pokud v `c` je malé nebo velké písmeno

Opět trochu předběhneme. Vytvoříme si vlastní výčtový datový typ BARVA (výčtový typ slouží k tomu, abychom měli prvky „slovně" pojmenované a zároveň překladač aby s nimi zacházel jako s čísly).

```
enum BARVA { bila, modra, zluta, zelena, cervena, hneda, fialova, cerna };
BARVA barva = bila;     // nová proměnná typu BARVA
…
if ((barva == cervena) || (barva == zelena) || (barva == modra)) { … }
```

Následující dva logické výrazy budou fungovat stejně, i když první možnost je „logičtější":

```
if (barva != modra)            if (!(barva == modra))
```

V jazyce C a odvozených se pravdivostní hodnoty TRUE a FALSE používají ve formě čísel 1 a 0. Takže to, že pravdivostní výraz má hodnotu FALSE, je určeno tím, že vyjde 0. Číslo 1 (nebo lépe řečeno nenulová hodnota) znamená TRUE.

Příklad:

Víme, že nulou se nemá dělit, tedy můžeme otestovat, zda je určitá proměnná různá od nuly. Následující dva příkazy jsou ekvivalentní, můžeme použít kterýkoliv z nich:

```
if (x != 0)                    if (x)
```

U prvního jsme použili relační operátor pro nerovnost, v druhém případě spoléháme na to, že nenulová hodnota znamená TRUE. Naopak pokud potřebujeme, aby v proměnné byla nula:

```
if (x == 0)                    if (!x)
```

V druhém případě jsme použili logický operátor, který vrací TRUE (tedy jedničku), pokud je v proměnné `x` nulová hodnota.

Můžeme vyzkoušet také následující:

```
int x = 5;
cout << x << !x << !(!x);
```

Nedávejte dva vykřičníky přímo za sebou, ten vnitřní by měl být uzávorkován.

Pozor, v jazycích založených na C mají logické operátory nižší prioritu než relační, tedy je nutno závorkovat! Závorek raději více než méně (nicméně taky musejí být na správných místech).

Příklad pro pokročilé:

V C se dá použít tzv. podmíněné vyhodnocování (také zkrácené, líné, lazy). Například:

```
int x=10, y=5, vysledek;       // za "y" zkuste dosadit nulu místo pětky
vysledek = y && (x/y);
cout << y << " -> " << vysledek << endl;
```

Co to znamená? Pokud je `y==0`, pak se přímo načte do výsledku, jinak se použije jako dělitel (to by samozřejmě nešlo, kdyby v proměnné bylo číslo 0). Druhá část výrazu se provede pouze tehdy, pokud první část skončí s výsledkem FALSE (tedy pro nulové y).

Proč? Pro logické AND platí, že když má první část hodnotu FALSE, tak je jasné, že `FALSE && COKOLIV` bude vždy FALSE. Tak proč se namáhat s vyhodnocováním zbytku výrazu? Tedy se provede jenom první část. Ovšem pokud má první část hodnotu TRUE, pak jsme ve stavu nejistoty a druhá část musí být vyhodnocena (to je jedno, že to není jen pravdivostní výraz, ale příkaz).

Podobně pro logické OR.

```
int x=10, y=5, vysledek;       // za "y" zkuste dosadit nulu místo pětky
vysledek = (!y) || (x/y);
cout << y << " -> " << vysledek << endl;
```

Oproti předchozímu je jiná nejen struktura příkazu, ale i výsledek. Jestliže je `y==0`, pak jeho negace je jednička. Logické OR vyhodnotí druhou část výrazu jen tehdy, když je po první části nejistota o výsledku, což v tomto případě není (`TRUE || COKOLIV`) je vždy TRUE. Ale pokud by y mělo nenulovou hodnotu, pak by jeho negace byla nulová, čímž by vznikla „nejistota“ a druhá část výrazu by se vyhodnotila.

Rozdíl by však byl v tom, co se dostane do výsledné proměnné. Jestliže je v proměnné `y` nula, pak zde do výsledku dostaneme číslo 1, protože jsme nulu negovali.

V podmíněných výrazech jsme používali čísla, ale ve skutečnosti existuje i datový typ pro tento účel:

```
bool b = true;    // pozor, tu konstantu malými písmeny, TRUE by hodilo chybu
if (b) …
```

Ve skutečnosti je datový typ bool jen jedním z alternativních číselných datových typů, tedy pracujeme s čísly, podobně jako třeba u datového typu char. Ostatně do této proměnné můžeme klidně číslo načíst:

```
bool b = 2;     // číslo 0 znamená false, jakékoliv jiné znamená true
```

6 Podmíněné provedení kódu a větvení programu

6.1 Ternární operátor – pro pokročilé

Unární operátor má jeden operand, binární má dva operandy, ternární má tři operandy. V jazyce C a jazycích z něj odvozených (C++, Java, Python atd.) máme ternární operátor `?:` určený otazníkem a dvojtečkou:

```
podmínka ? příkaz-splněno : příkaz-nesplněno;
```

Nejdřív tedy napíšeme podmínku, pak za otazník ten příkaz, který se má provést, když podmínka platí, pak za dvojtečku příkaz pro případ, že podmínka neplatí.

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?

6.2 Příkaz IF

S větvením kódu do dvou částí podle podmínky jsme se setkali už ve vývojových diagramech a také se příkaz `if` mihl v předchozích příkladech. Takže teď se na něj podíváme důkladněji.

Základní syntaxe je následující – podle tohoto pseudokódu:

```
if (podmínka)
   příkaz, když je podmínka splněna
```

```
if (podmínka)
   příkaz, když je podmínka splněna
else
   příkaz, když podmínka není splněna
```

Podmínka musí být vžd uzavřena v závorkách, další závorky budou případně uvnitř výrazu, pokud je to třeba. Jestliže v těle příkazu potřebuji uvést více než jeden příkaz, uzavřu tyto příkazy do bloku (složené závorky).

Příklady:

```
if (delitel == 0) {
   cout << "Chyba dělení nulou, končím! ";
   return 0;
}
vysledek = x / delitel;
```

Konverze velkých písmen na malá se dá udělat i takto:

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

Chceme, aby v proměnné max byla hodnota větší z hodnot proměnných x a y:

```
if (x > y)      // ale klidně můžete pro větve použít složené závorky {…}
   max = x;
else
   max = y;
```

Potřebujeme maximum ze tří proměnných:

```
if (x > y) {
   if (x > z)
      max = x;
   else
      max = z;
}
else {
   if (y > z)
      max = y;
   else
      max = z;
}
```

V tomto konkrétním případě sice nejsou složené závorky nutné, protože v každé větvi máme jen jeden příkaz, nicméně pro přehlednost rozhodně nejsou na škodu.

Úkol:

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.

Jestliže potřebujeme otestovat, zda v proměnné je či není nula (nebo zda výraz má či nemá nulový výsledek), může se použít zkrácené vyhodnocování – to jsme ostatně zjistili už dřív v předchozí kapitole, protože víme, že pravdivostní hodnota není nic jiného než číslo.

Příklad:

`if (výraz != 0)`	je totéž jako	`if (výraz)`
`if (výraz == 0)`	je totéž jako	`if (!výraz)`

Příklad:

Různé příkazy také vracejí hodnotu. Může to být i hodnota, která říká, že vše proběhlo v pořádku. Vyzkoušejte například toto:

```
int i;
if (cin >> i)
   cout << "ok";
```

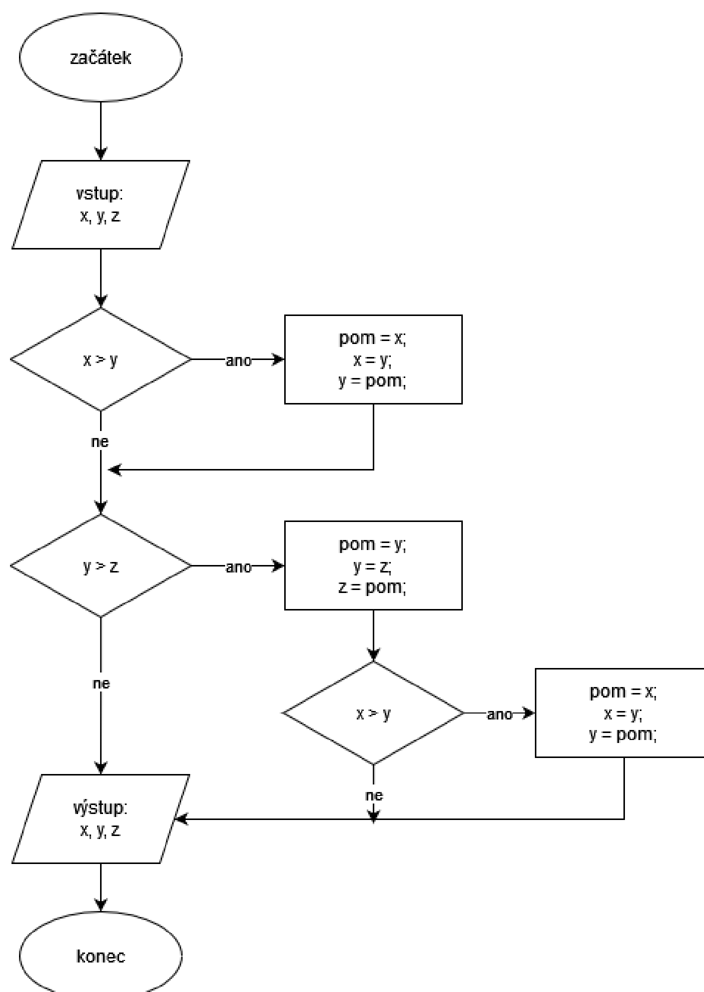
Co konkrétně se zde testuje? V tomto případě funkce cin nevrací tu hodnotu, kterou načetla, ale vrací buď 1 (vše v pořádku) nebo 0 (nastala chyba). Můžeme si to ověřit, když přidáme větev „else“:

```
int i;
if (cin >> i)
   cout << "ok";
else
   cerr << "chyba";
```

Když teď za běhu programu zadáme místo čísla třeba řetězec obsahující písmena, skočíme do druhé větve výpočtu.

Úkol:

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.



Úkol:

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.

6.3 Příkaz SWITCH

Když rozhodujeme mezi dvěma možnostmi, případně se nám každá možnost ještě dále větví, vystačíme si s příkazem `if`. Ale co když se potřebujeme rozhodnout mezi více možnostmi? K tomu nám slouží příkaz `switch`. Základní syntaxe je následující:

```
switch (výraz) {
   case hodnota1:
      příkazy;
      break;
   case hodnota2:
      příkazy;
      break;
   case hodnota3:
      příkazy;
      break;
   …
   default:
      příkazy;
}
```

Co platí:

- Výraz může být jakýkoliv, který vrací číslo typu int.
- Větev `default` je nepovinná, je to obdoba větve `else` u příkazu `if`.
- Pokud chceme, aby větve byly „disjunktní“, tj. abychom se po provedení jedné větve přesunuli na konec (a tedy neprováděli další větev), pak příkaz `break` musíme uvést, jinak ne.

Poslední podmínka se může zdát trochu zvláštní, ale tento princip nám umožňuje napsat kód platný pro více různých větví (za sebou následujících). Prostě když už se dostaneme do některé větve, opustíme celý `switch` buď až tehdy, když narazíme na `break`, nebo tehdy, když z místa, kde jsme, provedeme všechny příkazy až do konce switche. Podívejte se na následující:

```
switch (výraz) {
   case hodnota1:     // první větev, provedou se příkazy2
   case hodnota2:     // druhá větev, provedou se příkazy2
      příkazy2;
      break;
   case hodnota3:     // třetí větev, provedou se příkazy3 a příkazy4
      příkazy3;
   case hodnota4:     // čtvrtá větev, provedou se příkazy4
      příkazy4;
   …
}
```

Pokud se vyhodnocením výrazu dostaneme do čtvrté větve, provedou se pouze příkazy v bloku 4. Pokud se však dostaneme do třetí větve, provedou se příkazy bloku 3 a 4, protože nejsme zastaveni příkazem `break`. Pokud se dostaneme do druhé větve, provedou se pouze příkazy v bloku 2. Pokud se dostaneme do první větve, provedou se taktéž příkazy bloku 2, protože (třebaže v první větvi nemáme žádné příkazy) plynule přecházíme do bloku 2, neblokováni breakem.

Příklad:

Vypíšeme uživateli malé menu určující, co má zadat, aby se provedla určitá činnost.

```
#include <iostream>
using namespace std;
int main() {
   int menu = 0, x, y, vysl;
   cout << "Zadej:" << endl;
   cout << "1 ... scitani" << endl;
   cout << "2 ... odcitani" << endl;
   cout << "3 ... nasobeni" << endl;
   cout << "jinak ... nic" << endl;
   cin >> menu;
   if (menu == 1 || menu == 2 || menu == 3) { // (nebo: menu >= 1 || menu <= 3)
      cout << "x = ";   cin >> x;
      cout << " y = ";  cin >> y;
      switch (menu) {
         case 1: vysl = x + y; break;
         case 2: vysl = x - y; break;
         case 3: vysl = x * y; break;
         default:
            cout << "chyba";
            return 1;
      }
   cout << "Vysledek: " << vysl;
   }
}
```

Všimněte si, že zde máme větev `default`, třebaže to vlastně ani není nutné. Pro jistotu je však dobré zvyknout si mít ošetření chyby.

Příklad:

Protože datový typ `char` je vlastně číselný, můžeme ho zde taky využít:

```
char menu;
cout << "Smazat? [an] ";
cin >> menu;
switch (menu) {
   case 'a':
   case 'A':
      // tady bude kód, kde něco mažu
      break;
   case 'n':
   case 'N':
      // tady bude kód, který dělá něco jiného
      break;
   default:
      cout << "Takové písmenko neznám! ";
   }
}
```

Upozornění:

- Větve mohou být rozlišeny jen konkrétními hodnotami. Pokud bychom chtěli větve rozlišit pomocí intervalů (například „pokud jde o číslo, proveď xxx, pokud jde o malé písmeno, proveď

yyy, pokud jde o velké písmeno, proveď zzz, jinak proveď eee“), je praktičtější použít několik příkazů IF-ELSE.

- Pozor na závorky. Výraz (popř. proměnná), podle něhož se kód větví, musí být v kulatých závorkách (podobně jako podmínka u IF). Celé tělo switche musí být ve složených závorkách.
- Opravdu si dávejte pozor na to, kde má/nemá být `break`. Chyby tohoto typu jsou velmi časté a poměrně špatně se hledají.

7 Cykly a pole

7.1 Jednoduchý cyklus s podmínkou

Ve vývojových diagramech jsme se setkali s cykly, což je konstrukce umožňující opakování kroků. V jazycích C a C++ existují dva typy cyklů s podmínkou:

```
while (podmínka)
   příkaz;
```

```
do
   příkaz;
while (podmínka);
```

Obvykle míváme v cyklu více než jeden příkaz, pak musíme použít složené závorky:

```
while (podmínka)
{
   příkazy;
}
```

```
do
{
   příkazy;
}
while (podmínka);
```

Kondenzovanější forma:

```
while (podmínka) {
   příkazy;
}
```

```
do {
   příkazy;
} while (podmínka);
```

Uvnitř kulatých závorek tedy máme podmínku, části kódu uvnitř složených závorek říkáme tělo cyklu.

První typ cyklu má podmínku na začátku, příkazy v těle cyklu se provádějí tak dlouho, dokud podmínka platí (pokud neplatí ani při prvním testování, pak se neprovedou ani jednou). Druhý typ cyklu má podmínku na konci, příkazy v těle cyklu se taktéž provádějí tak dlouho, dokud podmínka platí, ale vždy nejméně jednou.

Úkol:

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.

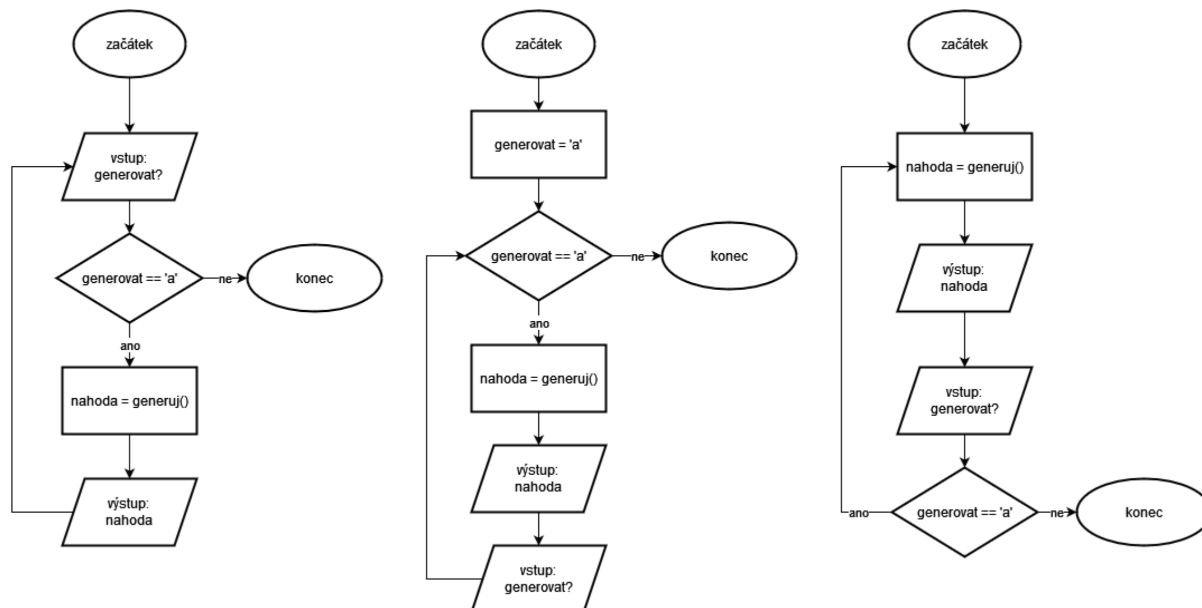
Konstrukci `i++` už známe, tento příkaz přičte k proměnné `i` jedničku. Ale co se stane, když plusy dáme před proměnnou? I to bychom už mohli znát, berte to jako připomenutí.

Úkol:

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.

Úkol:

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.

Úkol:

Napište program, který postupně od uživatele načte řadu čísel (přestane načítat až ve chvíli, kdy uživatel zadá něco, co není číslo – využijte postup testování výsledku `cin`, viz jeden z příkladů v sekci o větvení `if`). Průběžně v cyklu (zároveň s načítáním) zjišťujte maximum, minimum a průměr.

7.2 Složený datový typ pole

Zatím jsme pracovali s jednoduchými datovými typy – pro celá čísla, případně jsme na výstup vypsali konstantní řetězec. Datový typ může být i složený, tedy skládající se z více prvků, které mají vlastní datový typ.

Pole je složený datový typ prvků stejného základního typu zabírající souvislou oblast paměti (tj. jeho prvky jsou v paměti uloženy hned za sebou). Takže například můžeme mít pole celých čísel, pole znaků, pole racionálních čísel, ale také pole polí. Pole o `n` prvcích:

0	1	2	3	…	n-1

Každý prvek pole má svůj index, první prvek má index `0`. To znamená, že poslední prvek má index `n-1`. Pokud toto pole naplníme náhodnými čísly (což samozřejmě můžeme), bude vypadat třeba takto:

0	1	2	3	…	n-1
4286	702358	1137	30		91726

Příklad:

Pole celých čísel o délce 10 deklarujeme takto:

```
int moje_pole [10];     // před hranatou závorkou nemusí být mezera
```

K jednotlivým prvkům přistupujeme pomocí jejich indexů. Takže pokud chceme do prvku s indexem 3 načíst číslo 284, provedeme to takto:

```
moje_pole [3] = 284;
```

Kdybychom chtěli do prvku s indexem 8 načíst náhodné číslo, postup bude následující:

```
srand(time(0));
…
moje_pole [8] = rand();
```

Kolik místa v paměti zabírá položka pole a kolik celé pole? Tyto hodnoty si můžeme vypsat:

```
cout << sizeof(moje_pole[3]) << endl << sizeof(moje_pole);
```

Úkol:

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).

Úkol:

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.

Stejně jako číselné proměnné, také pole se dá inicializovat. Pokud při deklaraci zároveň pole inicializujeme, není nutné uvádět délku pole – vlastně je lepší ji neuvádět, abychom omezili chyby.

Příklad:

Definujeme dvě proměnné typu pole – pole celých čísel a pole racionálních čísel:

```
int pole1[] = { 1, 8, -54, 16 };             // pole čtyř prvků
double pole2[] = { 12.5, .46, -100, 3.02 }; // taktéž pole čtyř prvků

cout << pole2[1];    // vypíše se 0.46, protože jde o druhý prvek pole
```

Pozor, příkaz `cout << pole1[];` by byl špatně, celé pole bychom museli vypsat postupně po prvcích.

7.3 Cyklus s pevným počtem kroků

Pole, se kterým jsme pracovali v předchozí sekci, má neměnný počet kroků, a to je kandidát na zpracování jiným typem cyklu. Syntaxe příkazu `for` je následující:

```
for (inicializace ; podmínka ; krok)
```

První část závorky určuje, co se má provést předtím než samotný cyklus začne, může tam být i deklarace pomocné proměnné s její inicializací (a taky často bývá). Druhá část určuje, kdy má cyklus skončit, a tedy se testuje na konci každého cyklu (cyklus končí, když podmínka přestane platit). Třetí část bude provedena na konci každého cyklu (ještě před testováním podmínky ukončení).

Následuje tělo cyklu, pro které platí totéž jako u cyklů `while` a `do`: pokud jde o více než jeden příkaz, uzavřeme do složených závorek.

Příklad:

Nejdřív něco jednoduchého. Vypíšeme na výstup postupně čísla od 0 do 9 (mezi ně dáme mezeru).

```
for (int i = 0; i < 10; i++)
   cout << i << ' ';
```

Sečteme všechna čísla od 1 do 50, a pak vypíšeme výsledek, tedy provedeme výpočet $\sum_{i=1}^{50} i$

```
int soucet = 0;
for (int i = 1; i <= 50; i++)
   soucet += i;
cout << soucet;
```

Příklad – jiné řešení předchozího problému:

Do inicializační části můžeme umístit více než jeden prvek. Pak mezi ně umístíme čárku (říkáme tomu operátor čárky). Čárka v parametru znamená řetězení operací, zde řetězíme dva příkazy.

```
int i, soucet;
for (i = 1, soucet = 0; i <= 50; i++)    // provede se nejdřív i=1, pak soucet=0
   soucet += i;
cout << soucet;
```

Jak by fungovalo toto (bez řádku s deklarací)?

```
for (int i = 1, soucet = 0; i <= 50; i++)
   soucet += i;
cout << soucet;
```

Co konkrétně překladači vadí?

V kterékoliv části definice cyklu `for` (tím je myšlen obsah kulatých závorek) může být jakýkoliv počet příkazů či podmínek. V předchozím příkladu jsme si ukázali, že v první části (inicializační) mohou být dva příkazy oddělené *operátorem čárky*. Podobně může být více příkazů v třetí části určující krok, taktéž pro jejich oddělení použijeme operátor čárky. Ale co když je některá část prázdná?

Příklad:

Ukážeme si řešení s prázdnou inicializační částí cyklu `for`:

```
int i=1, soucet=0;
for ( ; i <= 50; i++)
   soucet += i;
cout << soucet;
```

Inicializační část cyklu je prázdná, ale nevadí to, protože veškeré proměnné, které se cyklu týkají, jsou vytvořeny a inicializovány předem. Ovšem pozor, je třeba si to ohlídat.

V předchozích příkladech často používáme proměnnou „i“ nebo podobné. Pokud tutéž proměnnou používáme vícekrát, můžeme ji klidně vytvořit někde nahoře a použít v různých cyklech („recyklovat“). Mnozí ale preferují deklarovat proměnnou, přes kterou běží cyklus, radši jen pro tento cyklus (ne předem pro víc cyklů), a to i z bezpečnostních důvodů – je menší riziko, že změna v jednom místě programu ovlivní jiné místo programu.

Úkoly:

1. Úkoly s poli z předchozí sekce přepište – místo cyklu s podmínkou použijte cyklus `for`.
2. Vytvořte pole s 5 prvky typu `int`. Od uživatele načtěte prvky tohoto pole a následně zjistěte (a vypište) součet a průměr těchto hodnot.

Úkol:

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?

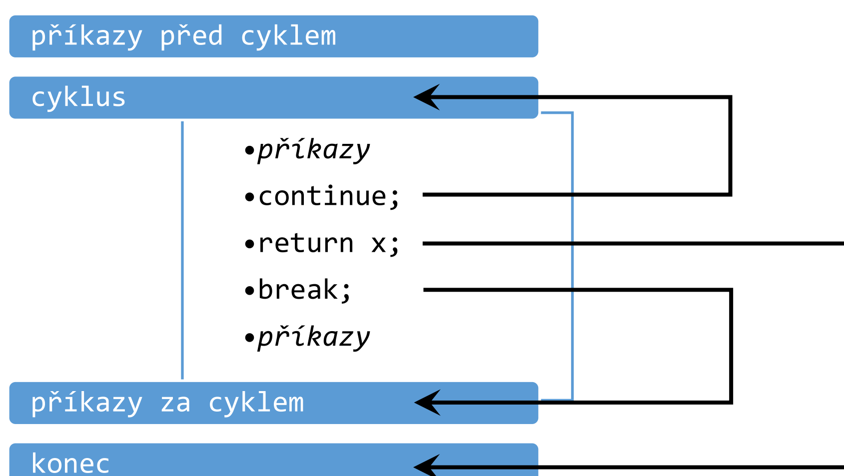
Úkol:

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

7.4 Přerušení cyklu

Co když potřebujeme z cyklu „vyskočit" někde uprostřed, případně potřebujeme zprostředka skočit znovu na začátek? K tomu máme tyto příkazy:

- `break` – okamžitě vyskočí z cyklu, princip je stejný jako u příkazu `switch`
- `continue` – ukončí momentální průběh cyklu a přesune se opět na jeho začátek, začne nový běh cyklu včetně testování podmínky, pokud jde o `while`
- `return` – ukončí funkci (pokud jsme ve funkci `main()`, tak ukončí celý program), což použijeme, pokud došlo k takové chybě, která vylučuje další běh programu (tomuto příkazu obvykle zadáváme jako parametr číslo určující kód chyby)



Takže pokud se v kódu dojde k příkazům `break` nebo `return`, cyklus rozhodně skončí (v případě `return` skončí i funkce/program). Pokud se provede příkaz `continue`, cyklus nekončí, ale přeruší se právě probíhající krok a začne nový krok cyklu.

Příklad:

Vytvoříme jednoduchou kalkulačku, která umí sčítat a odčítat, přičemž tyto operace bude provádět opakovaně, dokud uživatel neurčí konec. Použijeme cyklus s podmínkou na konci, ale místo samotné podmínky ukončíme cyklus příkazem `break`.

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

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

   do {
      cout << "1: sčítání\n";
      cout << "2: odčítání\n";
      cout << "jinak: konec\n";
      cin >> menu;

      if (menu < 1 || menu > 2)  {      // něco jiného než sčítání nebo odčítání
         cout << "konec";
         break;                         // vyskočíme z cyklu, tj. o dva řádky níže
      }
      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;
         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 podmínky – číslo 1 znamená true, tedy cyklus je opravdu možné ukončit jen tak, že průběh programu bude zahrnovat příkaz `break` nebo `return`. Je to nebezpečné (můžeme omylem vytvořit cyklus, který je v určité situaci nekonečný), ale celkem hodně používané. Ovšem musíme si dobře pohlídat, aby všechny možné průchody cyklem bylo možno ukončit.

Příkaz `break` můžeme použít například pro účely přerušení cyklu v případě, že zjistíme chybu (třeba uživatel zadal nesprávně něco na vstupu – například požádáme ho, aby zadal číslo z určitého intervalu, ale on zadá číslo mimo tento interval).

Poznámka:

Příkazy `break` a `continue` platí vždy pro nejvnitřnější cyklus, takže je dobré si dávat pozor, co vlastně přerušujeme.

Příkaz `break` známe také jako ukončovací příkaz větví ve `switch`i. Pokud máme switch uvnitř cyklu, pak ovšem příkaz `break` ve větvích `switch`e nebude ukončovat cyklus, ale pouze danou větev `switch`e.

Jestliže potřebujeme „vyskočit“ z cyklu, ale přitom jsme uvnitř některé větve `switch`e, musíme to řešit jinak. Například můžeme použít `continue` a na začátku cyklu vyřešit vyskočení z cyklu třeba pomocí `if`, nebo stojí za úvahu, zda místo příkazu `switch` nepoužít `if`, který na `break` nereaguje. Následující kód ukazuje cyklus `while`, ve kterém je vnořen `switch`. Potřebujeme ukončit celý cyklus (protože nastal problém, kvůli kterému nelze v cyklu pokračovat), ale `break` uvnitř `switch`e pouze ukončil ten `switch`. Proto použijeme například proměnnou, kterou v případě problému nastavíme na 1.

```
int chyba = 0;
while ((!chyba) && dalsi_podminky) {
   ...
   switch(zn) {
      case 'a': ...
         break;
      case 'b':
         ...
         if (nastal_problem) {
            chyba = 1;
            continue;
         }
         ...
         break;
      default: ...
   }
}
```

Úkol:

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`?

7.5 Vícedimenzionální pole

Pole prvků typu `int` nebo podobného je vlastně vektor – prostě máme řádek nebo sloupec prvků. Ale z matematiky známe i matice, což je vlatně dvoudimenzionální pole.

Příklad:

Vytvoříme matici 2×3 s prvky typu `int`, necháme uživatele zadat její obsah, následně matici vynásobíme číslem 2 a vypíšeme na výstup. Nejdřív nejméně optimální způsob:

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

int main() {
   int matice[2][3];
   int i,j;

   cout << "Zadej 2x3 cisla:" << endl;
   for (i=0; i<2; i++)                // fáze načítání matice
      for (j=0; j<3; j++)
         cin >> matice[i][j];

   for (i=0; i<2; i++)                // fáze násobení prvků matice dvěma
      for (j=0; j<3; j++)
         matice[i][j] *= 2;

   for (i=0; i<2; i++) {              // fáze vypisování prvků matice
      for (j=0; j<3; j++)
         cout << matice[i][j] << '\t';
      cout << endl;
   }
}
```

Proč je tento kód neoptimální? Máme tu tři „dvojcykly“ `for`, které postupně zpracovávají prvky téhož pole bez vzájemného ovlivňování těchto prvků (v druhém a třetím případě). Buď první nebo třetí případ bude muset být samostatný, protože načtení a výpis pole není možno míchat, ale druhý můžeme sloučit v našem případě s třetím.

Například:

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

int main() {
   int matice[2][3];
   int i,j;
```

```
   cout << "Zadej 2x3 cisla:" << endl;

   for (i=0; i<2; i++)                // fáze načítání matice
     for (j=0; j<3; j++)
       cin >> matice[i][j];

   for (i=0; i<2; i++) {              // fáze násobení a zároveň vypisování
     for (j=0; j<3; j++)
       cout << (matice[i][j] *= 2) << '\t';
     cout << endl;
   }
}
```

Všimněte si:

- Řádek matice je vypsán na jednom řádku, čísla jsou oddělena tabulátorem.
- Vynásobení dvěma je „voperováno" do příkazu pro výstup (nemusí, mohli jsme nejdřív vynásobit dvěma a pak vypsat).
- Přiřazovací příkaz je ve skutečnosti operátor, který vrací výsledek operace, a ten je tedy vypsán.
- V druhém „dvojcyklu" `for` máme složené závorky (tj. ohraničení bloku) pouze u vnějšího cyklu, u vnitřního nejsou potřeba (tam je uvnitř jen jeden příkaz). Pokud bychom ale nejdřív násobili dvěma a pak teprve vypisovali, museli bychom použít složené závorky i pro vnitřní cyklus:

```
   for (i=0; i<2; i++) {
     for (j=0; j<3; j++) {
       matice[i][j] *= 2;
       cout << matice[i][j] << '\t';
     }
     cout << endl;
   }
```

Úkol:

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 dělení „modulo", která se zapisuje symbolem procenta: `x = y % 100`

Taktéž vícedimenzionální pole se dá inicializovat přímo s deklarací, ale v takovém případě necháváme nevyplněnou jen první dimenzi.

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?
```

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.

```
const int R = 5, S = 6;        // pro počet řádků a sloupců použijeme konstantu
const int hodnota_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() % hodnota_modulo;   // zbytek po dělení číslem 1000

cout << "Zadej číslo menší než " << hodnota_modulo << ": ";
cin >> porovnani;
if ((porovnani < 0) || (porovnani >= hodnota_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.

Úkol pro ty, kdo potřebují procvičit matematiku:

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.

Úkol pro pokročilé:

Zkuste vymyslet, jak využít 2D pole pro zjištění všech prvočísel menších než 100 algoritmem Eratosthenovo síto. Pokud nevíte, co to je, „papírová“ varianta je následující:

Napíšeme si všechna čísla, třeba do matice, kde každý řádek má 10 čísel. Například v prvním řádku matice by byla čísla od 1 do 10 (nulu nebereme v úvahu, protože není přirozené číslo a nemůže být prvočíslo), v druhém řádku od 11 do 20, atd.

Dále procházíme postupně všechna čísla od 2 do konce. Pro každé toto číslo vyškrtáme všechna následující čísla, která jsou násobkem tohoto čísla (tj. ro číslo 2 vyškrtneme 4, 6, 8, atd., pro číslo 3 vyškrtneme 6, 9, 12, atd., stejně pro další čísla).

Po průchodu celé matice (kde uvnitř cyklu procházíme určité následující prvky, taktéž v cyklu) zůstanou jen prvočísla.

Nápověda: je zbytečné mít v matici přímo čísla (ostatně – jak byste je škrtali?), protože samotné číslo je odvoditelné ze souřadnic. Je lepší mít v matici hodnoty TRUE/FALSE (resp. 1 či 0), které určují, jestli je číslo ještě „použitelné“ nebo bylo škrtnuto (pozor, je třeba matici nejdřív celou inicializovat dvěma vnořenými cykly FOR). Dávejte si velký pozor na hranice, ať už při inicializaci, nebo následně při hledání násobků.

8 Složené a uživatelské datové typy

8.1 Výčtový typ

S výčtovým typem jsme se už ve skutečnosti setkali – na straně 31 máme výčtový typ BARVA definovaný takto:

```
enum BARVA { bila, modra, zluta, zelena, cervena, hneda, fialova, cerna };
BARVA barva = zelena;
...
if (barva == fialova) ...
if (barva < cervena) ...
```

Příklad:

Při deklaraci nového výčtového typu nemusíme nutně souhlasit s tím, že „vnitřní“ hodnota prvního prvku je 0, druhého 1, atd. Pokud u některého určíme ručně hodnotu, pak se změna projeví i u dalších. Například zde chceme začít od jedničky:

```
enum TRIDA { prima = 1, sekunda, tercie, kvarta, kvinta, sexta, septima, oktava };
TRIDA trida = sekunda;
cout << trida;
```

Tímto jsme vytvořili uživatelský datový typ, který můžeme používat stejně jako typy předdefinované.

Častou chybou je umístění rovnítka mezi název a složenou závorku:

```
enum BARVA = { bila, ...     // vyhodilo by chybové hlášení
```

8.2 Struktura

Jeden složený datový typ už známe – pole. Pole je *homogenní* datový typ, což znamená, že všechny jeho prvky jsou stejného „vnitřního“ datového typu. Například můžeme mít pole prvků typu `int` či `float`, nebo u více dimenzí pole prvků typu pole prvků typu `int`,…

Struktura je naproti tomu *heterogenní* datový typ, tedy může obsahovat prvky různých „vnitřních“ datových typů.

Příklad:

Nadefinujeme strukturu pro adresu. Existuje několik různých způsobů, jak to udělat. První způsob by byl použitelný, pokud jednoduše potřebujeme jednu či více proměnných s danou strukturou a dále se k tomuto typu už nebudeme vracet:

```
#include <iostream>
#include <string.h>
using namespace std;
...
struct {
   string ulice;
   int cislo;
   string mesto;
   string psc;
} adresa1, adresa2;        // proměnné, jejich datový typ ale není pojmenován

adresa1.cislo = 158;       // takto přistupujeme k prvkům struktury
adresa2.mesto = "Opava";
```

Druhý způsob přidává pojmenování datového typu:

```
struct ADRESA {
   string ulice;
   int cislo;
   string mesto;
   string psc;
} adresa1, adresa2;       // tady nemusejí být žádné proměnné, stačí středník
```

Proměnné adresa1 a adresa2 jsou pak přístupné stejně jako v předchozím případě, navíc můžeme kdykoliv dále v kódu vytvořit další proměnnou stejného datového typu:

```
ADRESA adresa3;
```

Třetí způsob spočívá v oddělení definice datového typu a deklarace proměnných, přičemž používáme klíčové slovo typedef:

```
typedef struct ADRESA {
   string ulice;
   int cislo;
   string mesto;
   string psc;
} ;                    // nezapomeňte tady středník

ADRESA adresa1, adresa2;
```

Nejběžnější je v současné době druhý způsob, tedy hned za informací, že jde o strukturu, napíšeme název datového typu. Třetí způsob není moc používaný, komu by se chtělo navíc psát „typedef“…

K prvkům struktury pak přistupujeme přes tečku, jak už bylo naznačeno, ať už použijeme kterýkoliv způsob vytvoření. Pokud tedy máme proměnnou adresa výše uvedeného datového typu:

```
adresa.ulice = "Programátorská";
adresa.cislo = 152;
adresa.mesto = "Opava";
adresa.psc = "76502";

cout << adresa.ulice << " " << adresa.cislo << endl;
cout << adresa.psc   << " " << adresa.mesto << endl;
```

Vnitřní prvky struktury mohou být různé, včetně výčtového typu či polí (výčtový typ bychom ovšem měli definovat předem). A naopak, můžeme vytvořit pole prvků typu (určitá konkrétní) struktura. Samozřejmě lze vytvořit i strukturu, jejíž některé prvky jsou taky typu struktura.

Příklad:

```
enum OBOR { informatika, matematika, fyzika, historie, filologie, filozofie };
struct STUDENT {
   string jmeno, prijmeni;
   OBOR obor;
   int rocnik;
} ;

STUDENT studenti[20];

for (int i = 0; i <= 14; i++)
   studenti[i].obor = informatika;

studenti[15].obor = filologie;
```

```
for (int i = 16; i < 20; i++)
   studenti[i] = matematika;

studenti[0].jmeno = "Jane";       // první student se jmenuje Jane
studenti[0].prijmeni = "Doe";     // příjmení prvního studenta je Doe

/* Pokud inicializujete při deklaraci, dejte si pozor na pořadí jednotlivých
   součástí struktury, má být stejné jako u definice datového typu: */
STUDENT novy = { "Albert", "Einstein", fyzika, 2 };   // deklarace s inicializací
```

Úkol:

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).

8.3 Union pro určení variant

Někdy potřebujeme mít (nebo prostě chceme mít) místo v paměti, které je sdíleno více proměnnými, tedy tyto proměnné se na daném místě v paměti překrývají. Typické využití je v případech, kdy ve struktuře ukládáme různé typy záznamů, které jsou navzájem disjunktní (tj. případy, které nemohou nastat zároveň). Pak je zbytečné plýtvat pamětí v určení datového typu.

Příklad s úkolem:

Vytvoříme zjednodušenou strukturu pro zachycení údajů o události v programu.

- Pokud půjde o událost klávesnice, potřebujeme uložit číslo, ve kterém je zakódována stisknutá klávesa (příp. zda zároveň byla stisknuta některá speciální klávesa jako Ctrl, Alt apod.).
- Pokud půjde o událost myši, pak chceme evidovat souřadnice, na kterých se zrovna nacházel kurzor.
- Pokud půjde o událost procesoru, pak budeme ukládat číslo chyby, které procesor nahlásil.

```
enum TYP_UDALOSTI { klavesnice, mys, procesor };

struct SOURADNICE {
   int x, y;
};

struct UDALOST {
   int identifikator;
   TYP_UDALOSTI typ;
   union {
      int klavesa;              // pro událost klávesnice
      SOURADNICE souradnice;    // pro událost myši, máme strukturu ve struktuře
      int chyba;                // pro událost procesoru
   };
};

UDALOST nova_udalost;
nova_udalost.identifikator = 3581;
nova_udalost.typ = mys;
```

```
// všimněte si, jak přistupujeme k prvkům vnořené struktury:
nova_udalost.souradnice.x = 84;
nova_udalost.souradnice.y = 150;

cout << nova_udalost.souradnice.x;
```

Co když v našem případě provedeme následující:

```
cout << nova_udalost.klavesa;
```

Vyzkoušejte, jestli se vypíše chybové hlášení nebo to „projde“.

S uniony se ve skutečnosti dá zacházet stejně jako se strukturami (co se týče deklarace, inicializace apod.). Ale obvykle má union smysl opravdu jen jako součást struktury, tedy není nutno ho pojmenovávat.

Úkol:

Napište stukturu KNIHA pro evidenci knih. Každá kniha bude mít název, autora (pro zjednodušení jen jednoho), rok vydání, ISBN, dále pomocí výčtového typu určeno, zda je elektronická nebo tisknutá, a pomocí unionu v případě tisknuté bude počet výtisků, v případě elektronické řetězec s webovou adresou, na které je kniha zveřejněna.

Definujte pole KNIHOVNA o 10 prvcích, prvky jsou typu KNIHA. Vyplňte v kódu první dva prvky (jeden typu tisknutá kniha, druhý typu elektronická), případně to nechte udělat uživatele (veďte ho, aby věděl, co má zadávat). Pak oba prvky pro kontrolu vypište.

Při vypisování prvků unionu používejte příkaz IF-ELSE podle typu knihy.

8.4 Řetězce

Zatím jsme se seznámili s řetězci podle knihovny `string.h`, ale to nejsou jediné řetězce, které můžeme použít. Další možnost je tato:

```
char s1[] = "Ahojte";
char s2[] = { 'A', 'h', 'o', 'j', 't', 'e', '\0' };
```

Vypíšeme si velikost obou řetězců:

```
cout << sizeof(s1) << endl << sizeof(s2) << endl;
```

Jak to dopadlo? V obou případech se vám pravděpodobně vypíše číslo 5, třebaže první řetězec má zdánlivě délku 4. V obou případech totiž vytváříme řetězec ukončený nulovým znakem (v angličtině se mu říku null-terminated string) – tedy znakem s ASCII kódem 0.

Tyto řetězce jako celek nemůžeme přepsat, můžeme však pracovat s jednotlivými znaky:

```
s1[0] = 'H';
s1[1] = 'e';
```

Tento řetězec nemůžeme prodlužovat. Ale můžeme takový řetězec celý vypsat na výstup:

```
cout << s1;
```

Rozhodně bychom neměli přepisovat nulový znak na konci. Ovšem není nutné, aby tento znak byl na konci, můžeme ho umístit kdekoliv v řetězci. Tím zajistíme, že zbytek řetězce se stane irelevantní, neviditelný (až do chvíle, kdy nulový znak posuneme na některou vzdálenější pozici).

```
s1[2] = '\0';     // tímto „zakryjeme“ část řetězce od pozice s1[3] dále
cout << s1 << endl;
```

```
s1[2] = 'l';      // teď ji zase „odkryjeme“, nulový znak jsme přepsali písmenem
cout << s1 << endl;
```

Řetězec zakončený nulovým znakem nemusíme inicializovat hned při deklaraci, v tom případě ale musíme stanovit délku (což bychom však mohli i v případě, že inicializujeme).

Příklad:

Vyzkoušejte tento kód (nepotřebujeme knihovnu `string.h`):

```
char s3[27];

for (int i = 0 ; i < 26 ; i++)
   s3[i] = i+(int)'a';

s3[26] = '\0';
cout << s3;
```

Všimněte si, co ve skutečnosti načítáme do prvků pole, a také jaké limity jsou použity pro indexy prvků pole. Proč jsme se v cyklu zastavili už před indexem 26?

Pokud v kódu použijeme konstantní řetězec (tzv. řetězcový literál), tedy „ručně“ napsaný v uvozovkách, ve skutečnosti jsme napsali řetězec ukončený nulovým znakem.

9 Pointer a dynamická alokace paměti

9.1 Pracujeme s pointery

Zatímco například `int` je datový typ, do kterého ukládáme celé číslo, pointer je typ určený pro uložení adresy. Nejdřív se podíváme, jak taková adresa vypadá:

Příklad (jak vypadá adresa):

```
int x = 25;
cout << "x ma hodnotu " << x << " a adresu " << &x << endl;
```

Vypsalo se nám něco podobného (adresa bude zřejmě jiná):

```
x ma hodnotu 25 a adresu 0x7ffc53181a54
```

Adresa označuje místo v paměti, na kterém začíná v našem případě proměnná `x`. O jedničku vyšší číslo má následující byte v pořadí, a protože je naše proměnná 4- nebo 8bytová, tak jí taky bude patřit, stejně jako několik dalších.

Příklad:

Vytvoříme nejdřív celočíselnou proměnnou a pak pointer na tuto proměnnou.

```
int x = 25;
int *px = &x;   // px obsahuje adresu proměnné x, *px obsahuje hodnotu proměnné x

(*px)++;
cout << "x ma hodnotu " << x << ", pointer ukazuje na " << *px;
*px = 100;
cout << "x ma hodnotu " << x << ", pointer ukazuje na " << *px;
```

Všimněte si, jak byl použit unární operátor `++`. Hvězdička je totiž taky unární operátor a v našem případě by bez závorek měla inkrementace přednost před hvězdičkou (oba operátory mají stejnou prioritu, ale zde se vyhodnocuje zprava), takže by se nejdřív zvětšila adresa uložená v pointeru a pak by byla vrácena hodnota na této posunuté adrese, což v našem případě zrovna nechceme.

V předchozím příkladu vidíme použití dvou nových unárních operátorů:

- `&` (ampérsánd, referenční operátor, reference) vrací adresu svého parametru, v příkladu jsme tak zjistili adresu proměnné `x`,
- `*` (hvězdička, dereferenční operátor) naopak považuje parametr za adresu a zjistí její obsah.

Pointer sice nese adresu, ale kromě toho je úzce spjat s určitým datovým typem. Proměnná `px` v příkladu je ukazatel na typ `int`, což je důležitá informace.

Co když tedy přece jen inkrementujeme pointer? Může to být k něčemu dobré? Ano, a to ve spojitosti s datovým typem, na který daný pointer ukazuje. Podívejme se na následující příklad.

Příklad:

```
const int hranice = 5;
int pole[hranice] = { 45, 87, 4, 92, -10 };
int *p = &pole[0];                // ukazuje na první prvek pole

p++;                    // takto se posuneme na druhý prvek pole
p--;                    // couvneme zpět na první prvek
```

```
for (int i = 0; i < hranice; i++, p++)
   cout << *p << '\t';
```

Proměnná p je typu pointer na celé číslo, přičemž byla „namířena“ na první prvek pole prvků typu int. To je samozřejmě v pořádku. Následují dva „demonstrační“ řádky ukazující, jak se dá pohybovat mezi jednotlivými prvky pole, na které máme namířeno. V tomto případě se reálně nepřičítá/neodečítá jednička, ale číslo odpovídající délce základního datového typu (v našem případě int). Pokud by délka datového typu int byla 8 B, pak se přičte/odečte osmička.

Upravíme kód tak, ať vidíme, na kterých reálných adresách se nacházejí jednotlivé prvky pole:

```
for (int i = 0; i < hranice; i++, p++)
   cout << *p << '\t' << p << endl;
```

Tedy pro každý prvek pole vypíšeme nejdřív jeho hodnotu a pak po tabulátoru jeho adresu. Všimněte si rozdílu těchto adres – měl by odpovídat délce datového typu int na vaší platformě.

Upozornění:

Procházení pole pomocí pointeru je rychlejší než přístup pomocí indexů. U několikaprvkového pole to je celkem jedno, ale u hodně dlouhých polí to může mít význam.

Ukážeme si poněkud netradiční použití cyklu for – bez číselné řídicí proměnné.

Příklad:

```
const int hranice = 5;
int pole[hranice] = { 45, 87, 4, 92, -10 };
int *zacatek = &pole[0], *konec = &pole[hranice-1], *p;

for (p = zacatek; p <= konec; p++)
   cout << *p << '\t';
```

Proměnnou p jsme použili jako „běžce“, který postupně projde všechny prvky pole a vypíše je. Všimněte si, že ukončující podmínka je „větší nebo rovno“, protože pointer konec ukazuje na poslední prvek pole s indexem hranice-1, nikoliv na prvek pole[hranice].

```
*(zacatek+2) += 8;   // Právě pracujeme s proměnnou pole[2], přičítáme číslo 8
```

Shrneme si, jak zacházet s pointery:

- když chceme zjistit adresu proměnné, použijeme referenční operátor (ampérsánd),
- k obsahu pointeru (tedy k tomu, co se nachází na adrese v pointeru uložené) se dostaneme přes hvězdičku,
- pokud chceme, aby pointer ukazoval tamtéž co jiný, použijeme přiřazovací operátor,
- pokud chceme ve výrazu použít hvězdičku s jinými operátory, musíme dát pozor na priority, radši závorkujeme,
- pokud chceme pracovat s místem, na které pointer ukazuje, nesmíme zapomenout hvězdičku,
- když k pointeru (bez hvězdičky) přičteme číslo, ve skutečnosti přičítáme toto číslo násobené velikostí datového typu, na který pointer ukazuje, podobně s dalšími operacemi.

Úkol:

Vytvořte pole řetězců o délce 4 (číslo 4 použijte jako konstantu typu int). Dále deklarujte tři pointery typu ukazatel na řetězec zacatek, konec a p: první bude ukazovat na první prvek pole, druhý na

poslední prvek a třetí použijete později jako jezdce po poli. Dejte pozor, aby všechny pointery byly typu ukazatel na řetězec, jinak by jejich použití na pole řetězců bylo poněkud problematické.

Obsah pole načtěte ze vstupu (nejdřív sdělte uživateli, že má zadat čtyři řetězce, pak v cyklu for podle výše naznačeného příkladu načtěte tyto čtyři řetězce – pozor, cin považuje mezery za ukončení řetězců, navíc datový typ string je jen pro „krátké" řetězce). Dále pro každý prvek pole vypište nejdřív jeho adresu a následně jeho obsah, taktéž pomocí pointerů.

Všimněte si rozdílu adres sousedních prvků. Pro jejich vypočtení můžete použít kalkulačku…

9.2 Pointer na strukturu

Pointer může ukazovat také na strukturu. Možnosti přístupu k prvkům struktury jsou následující:

Příklad:

```
struct ADRESA {
   string ulice;
   unsigned int cislo;
   string mesto;
} ;
ADRESA adresa, *padresa;

adresa.ulice = "Novatorska";
adresa.cislo = 1024;
adresa.mesto = "Zabovresky";

padresa = &adresa;
(*padresa).cislo += 8;         // zde bydlící osoba se přestěhovala o pár domů dál

// Jiný (obvyklejší) přístup k prvkům struktury z pointeru:

padresa -> ulice = "Sipkova";
padresa -> cislo += 10;          // další stěhování
```

9.3 Dynamická alokace paměti v C++

Zatím jsme se zabývali *statickou alokací paměti*, kdy každá proměnná má přesně vymezenou oblast paměti od chvíle, kdy je deklarována, až do chvíle, kdy přestane existovat, přičemž tato oblast zůstává proměnné k dispozici po celou dobu v dané velikosti.

Ale co když v okamžiku deklarace ještě nevíme, kolik paměti bude zapotřebí, případně tento parametr budeme chtít za běhu programu měnit? Pak použijeme *dynamickou alokaci paměti*. V kterémkoliv okamžiku určíme pro proměnnou, kolik paměti má k dispozici, a v kterémkoliv (následujícím) okamžiku jí tu paměť můžeme vzít.

Pro dynamickou alokaci (tj. vytvoření prostoru v paměti) používáme operátor new, pro uvolnění operátor delete. Záleží, jestli alokujeme místo pro jednu proměnnou nebo celé pole. Postup si ukážeme na následujícím příkladu.

Příklad:

```
int delka, *prvek, *pole;

cout << Zadej delku pole: ";   cin >> delka;
prvek = new int;        // vytvořili jsme dynamický prostor pro celé číslo
```

```
pole = new int [delka];// vytvořili jsme dynamické pole o délce určené uživatelem

*prvek = 25;
*pole = 25;
pole[1] = 26;
pole[2] = 27;
cout << "pointer prvek ukazuje na hodnotu " << *prvek << endl;
cout << "prvni tri prvky pole jsou\t" << *pole << '\t' << *(pole+1) << '\t' <<
  *(pole+2) << endl;

int *pom = pole;
*(++pom) = 30;       // Co se provelo dřív – přiřazení nebo inkrementace?
cout << "druhy prvek pole jsme zmenili na " << *pom;

delete prvek;     // Takto uvolníme dynamicky alokovaný prostor pointeru
delete[] pole;    // Takto uvolníme dynamické pole
prvek = NULL;
pole = NULL;
```

Všimněte si rozdílů mezi dynamickým a statickým polem: nejen že si u dynamického můžeme určit kdykoliv jeho délku, ale také jeho velikost může být stanovena proměnnou, nemusí to být konstanta.

K čemu jsou poslední dva řádky příkladu? Vpodstatě tam nejsou nutné, ale je to dobrý zvyk – pokud uvolníme paměť, měli bychom sdělit, že daný pointer momentálně nikam neukazuje, a to pro případ, že bychom například omylem znovu tentýž pointer uvolnili. Při použití operátoru delete na uvolněný prvek bez tohoto přiřazení by totiž program mohl spadnout, kdežto po takovém ošetření nespadne (a nic se nestane, ani katastrofa typu pádu programu).

NULL je ve skutečnosti nula přetypovaná na datový typ pointer. To si můžeme vyzkoušet:

```
cout << (int)NULL;
```

Pokud bychom chtěli programovat ještě bezpečněji, můžeme tuto hodnotu přiřazovat pointerům hned při deklaraci (tedy inicializovat na NULL), především tehdy, když paměť tohoto pointeru budeme řešit až někde dál v kódu. Kdykoliv pak můžeme provést test:

```
int *pole = NULL;
...
if (pole == NULL) ...
```

Není jiný způsob, jak zjistit, zda je dynamické pole již vytvořeno nebo ještě ne. Takto také můžeme po příkazu `new` otestovat, jestli alokace paměti proběhla v pořádku.

Časté chyby při práci s dynamickou pamětí:

- Dynamická paměť se alokuje v části paměti zvané heap (halda), heap není nekonenčný. Pokud zapomínáme uvolnit paměť, kterou jsme předtím alokovali, může se kapacita heapu vyčerpat.
- Pokud naopak paměť uvolníme, ale dále s ní pracujeme, taky to není v pořádku. V tom nám pomůže přiřazení NULL do uvolněného pointeru.
- S pointery často pracujeme tak, že jimi procházíme jiný (klidně statický) prostor. Může se stát, že při tom překročíme hranici vymezeného prostoru a pracujeme s úplně jinou pamětí než předpokládáme. Pak můžeme omylem něco přepsat nebo číst jiná data než myslíme.

Úkol:

Vytvořte jako nový datový typ strukturu `KNIHA`, který bude mít tyto vnitřní prvky: název, jméno autora, příjmení autora a isbn typu `string`, dále počet stran typu `unsigned int`. Vytvořte dynamické pole

knihovna prvků typu KNIHA, jeho délku zjistěte předem od uživatele (protože knihovna může mít různý počet knih).

Obsah prvního prvku tohoto pole načtěte od uživatele. Upozornění: s prvky dynamického pole se zachází stejně jako s prvky statického pole, tj. například k názvu první knihy v poli se dostaneme takto:

```
knihovna[0].nazev
```

Dále vytvořte pointer na typ KNIHA, nasměrujte ho na první prvek pole knihovna a vypište ho na obrazovku – tedy postupně jednotlivé části dané struktury (použijte „šipkovou" notaci, která je ukázána v příkladu v sekci 9.2).

Úkol:

Jako uživatelské datové typy jsou vytvořeny dvě struktury – jedna pro adresy a druhé pro osoby. U každé osoby je jako bydliště použit pointer na adresu:

```
struct ADRESA {
   string ulice;
   unsigned int cislo;
   string mesto;
} ;

struct OSOBA {
   string jmeno, prijmeni;
   ADRESA *bydliste;
} ;
```

Vytvořte dvě dynamická pole: jedno pro adresy a jedno pro osoby. Jejich délku určete na 3 adresy a 5 osob (na téže adrese může bydlet více osob). Abyste kdykoliv věděli, kolik je právě načteno adres a osob, mějte tuto informaci uloženou v proměnných typu unsigned int:

```
unsigned int adres = 0, osob = 0;
```

Po vytvoření načtěte od uživatele adresy (použijte cyklus for). Po každé načtené adrese nezapomeňte inkrementovat příslušnou proměnnou.

Dále podobným způsobem načtěte osoby, přičemž u bydliště se zeptejte, na které adrese daná osoba bydlí (po uživateli chtějte číslo z rozsahu 0 až adres-1, ošetřete případné chyby, kdy uživatel zadá něco jiného). Vnitřní proměnnou bydliste pak nasměrujte na daný prvek pole s adresami.

Následně vypište všechny prvky pole osob (opět použijte cyklus for), a to včetně bydliště.

9.4 Jiné možnosti dynamické správy paměti

Také v C++ jsou k dispozici funkce malloc, calloc, realloc a pro uvolnění free, to vše z knihovny stdlib.h, které pocházejí z původního jazyka C. Jejich použití je mírně složitější než použití new a delete, ale žádná hrůza (navíc umožňují pole kdykoliv prodlužovat).

Když půjdeme opačným směrem, tedy k tomu, co v jazyce C vůbec není a souvisí to s objektovým přístupem v C++, dá se použít konstrukce vector.

9.5 Dynamika na n-tou

Můžeme mít pole pointerů, což si můžeme představit jako matici, jejíž každý řádek má jinou délku podle potřeby (například se tak dá snadno vytvořit diagonální matice). Nebo můžeme mít dynamické pole pointerů…

10 Funkce

10.1 Jak na funkce

Obvykle se dá v kódu najít část, která je prováděna na různých místech, jen třeba s trochu jinými daty. Pokud takovou sekvenci příkazů označíme jako funkci a pojmenujeme, můžeme se na ni odkázat odkudkoliv v kódu a tento kód provést.

Příklad:

Vytvoříme funkci vracející větší ze dvou zadaných čísel.

```
int max (int x, int y)          // záhlaví funkce max()
{                       // tady začíná tělo funkce max()
   if (x > y)
      return x;
   else
      return y;
}                       // tady končí tělo funkce max()
```

Nebo stručněji pomocí ternárního operátoru:

```
int max (int x, int y) {
   return (x > y ? x : y);
}
```

V každém případě bychom pak funkci `max()` volali takto:

```
int prvni = 10, druhy = -8;
int srovnavam = max(prvni, druhy);     // funkce mi vrátila zřejmě číslo 10
cout << max(53, srovnavam);            // funkce vrátila číslo 53 a to se vypíše
```

Každá funkce může mít návratovou hodnotu, v našem případě je typu `int` stejně jako oba parametry.

Ve skutečnosti už jednu funkci běžně používáme – funkci `main()`, což je hlavní funkce programu. Také tato funkce může mít parametry (argumenty) a návratovou hodnotu, zatím jsme je nepoužívali.

Upozornění:

Na předchozích stranách jsme se dozvěděli, že příkaz `return` ukončí program. Ono to není tak úplně pravda – ukončí tu funkci, ve které je použit. Pokud použijeme příkaz return ve funkci `main()`, pak samozřejmě ukončí program. Jestliže chceme opravdu ukončit program a přitom nejsme ve funkci `main()`, existuje příkaz `exit(číslo)`.

Příklad:

Vytvoříme funkci, která vypíše zadanou strukturu (adresu) ve formátu, v jakém bývá na štítku obálky. Datový typ parametru deklarujeme ještě před funkcí, protože ve funkci už ho budeme potřebovat:

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

struct ADRESA {
   string ulice;
   int cislo;
   string mesto;
   string psc;
} ;
```

```
void vypisAdresu(ADRESA a) {                    // funkce pro výpis adresy
   cout << a.ulice << " " << a.cislo << endl;
   cout << a.psc << " " << a.mesto << endl;
}
```

Tuto funkci bychom volali následovně:

```
int main()
{
   ADRESA adresa = { "U kola", 12, "Loukotovice" , "14785" } ;
   vypisAdresu(adresa);
}
```

Všimněte si, že funkce pro výpis adresy je mimo funkci `main()`. Definice funkcí totiž nekřížíme, funkce v jazyce C a jeho potomcích se navzájem nevnořují.

V předchozím příkladu se objevil jeden nový datový typ – `void` (je to návratová hodnota naší funkce). Je to „prázdný" datový typ, který používáme tehdy, když na daném místě nezáleží na datovém typu (což je náš případ, zrovna tato funkce nemusí nic vracet, nebo tehdy, když se reálně může jednat o různé datové typy a při použití pllánujeme přetypování. V některých programovacích jazycích se funkce bez potřeby návratové hodnoty nazývá *procedura*.

Úkol:

Vytvořte funkci, která podle požadavku vypočte a vrátí buď obvod nebo obsah obdélníka: první parametr bude číslo 0 nebo 1 (0 znamená obvod, 1 znamená obsah), druhý a třetí parametr budou délky stran v cm.

Použití funkce: V programu vytvořte nekonečný cyklus, kde se vždy zeptáte uživatele, zda chce vypočíst obvod nebo obsah nebo skončit, pokud chce skončit, tak ukončete program (nebo cyklus). Pokud se má pokračovat, načtěte dvě čísla (to budou délky stran) a použijte funkci pro výpočet. Vrácenou hodnotu přímo vypište.

Rozlišujeme mezi deklarací a definicí funkce. U *deklarace* (neboli prototypu) jen sdělíme, jak se funkce jmenuje, jaké má parametry a jakou návratovou hodnotu, ovšem *definice* (tj. včetně kódu) musí někde následovat. Pokud je definice funkce až za funkcí `main()`, pak před funkci `main()` dáme deklaraci.

Příklad:

Funkci `soucet()` nejdřív deklarujeme, definice této funkce bude až za funkcí `main()`:

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

int soucet (int, int);    // deklarace: u parametrů jsou důležité typy
                          // mohou být i jejich názvy:  int soucet (int x, int y);
int main() {
   int cislo1 = 54, cislo2 = -10;
   int vysledek = soucet(cislo1, cislo2);
}

int soucet (int x, int y) {
   return x+y;
}
```

10.2 Volání funkce z funkce a rekurze

Funkci můžeme samozřejmě volat i z jiné funkce, jenom si musíme pohlídat, aby vždy bylo deklarováno to, co chceme použít. Tj. před volající funkcí musí být buď deklarace nebo definice funkce.

Rekurze je případ, kdy funkce volá samu sebe.

Příklad:

Jak víme, vzorec pro výpočet faktoriálu nezáporného čísla je tento:

```
0! = 1                        tj. F(0) = 1
1! = 1                        tj. F(1) = 1
pro n>1 platí n! = n*(n-1)!   tj. F(n) = n*F(n-1)
```

Výpočet faktoriálu pomocí rekurze bude následující:

```
unsigned long faktorial (long c) {
   if (c == 0 || c == 1)            // pozor, mezi svislicemi není mezera!
      return 1;
   return c * faktorial(c-1);
}
```

Tuto funkci pak můžeme použít třeba takto:

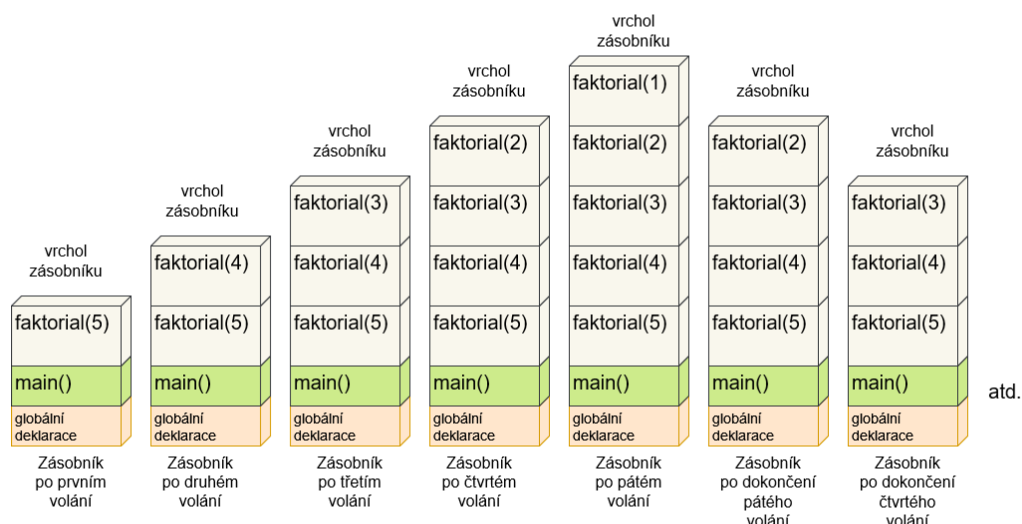
```
unsigned long x = faktorial(12);
cout << "Faktorial cisla 12 je " << x << endl;
cout << "Faktorial cisla 5 je " << faktorial(5) << endl;
```

Co se vlastně děje, pokud funkci zavolám například na číslo 5? První volání funkce zůstane rozpracováno, přičemž zavoláme tutéž funkci znovu, s jiným parametrem, v našem případě o jedničku menším číslem. Ovšem ta taky zůstane rozpracována a uvnitř provedeme třetí volání téže funkce, ovšem zase s jiným parametrem. Takto pokračujeme dále, dokud hodnota parametru neklesne až k jedničce. Funkce s parametrem 1 už nepůjde do rekurze, ale vrátí číslo 1 a ukončí se. Výsledek je předán předchozí funkci (která byla volána pro číslo 2), ta vynásobí 2*1, vrátí číslo 2 a ukončí se. Výsledek je předán předchozí funkci (pro číslo 3), ta vynásobí 3*2, atd.

```
faktorial(5)           ...rozpracováno, rekurze jde dolů
5 *  faktorial(4)           ...rozpracováno, rekurze jde dolů
5 *  4 *  faktorial(3)         ...rozpracováno, rekurze jde dolů
5 *  4 *  3 *  faktorial(2)      ...rozpracováno, rekurze jde dolů
5 *  4 *  3 *  2 *  faktorial(1)    ...otočka, konec rekurze
5 *  4 *  3 *  2 *  1              ...návrat z rekurze pro c=2
5 *  4 *  3 *  2                ...návrat z rekurze pro c=3
5 *  4 *  6             ...návrat z rekurze pro c=4
5 *  24          ...návrat z rekurze pro c=5
120         ...návrat z rekurze, výsledek vrácený „prvním“ voláním funkce
```

Jak vidíme, v jednom okamžiku máme rozpracovaných až 5 funkcí. Veškeré struktury, které s těmito funkcemi souvisejí (včetně parametrů funkce), jsou po celou dobu existence funkce v paměti, tedy je třeba si dávat pozor na hloubku rekurze = počet funkcí, které jsou v jednom okamžiku „v linii“ rozpracované. V našem případě je hloubka rekurze 5.

Jak bylo výše napsáno, ke každé zavolané funkci je třeba mít uloženy určité informace. Tyto informace se ukládají do části paměti zvané zásobník (stack). Pro každou „aktivní“ funkci je zde vytvořen tzv. aktivační záznam, který obsahuje „běhové“ informace o dané funkci, především: skutečné parametry (u nás například číslo 5, dále 4, atd. pro konkrétní funkci), lokální proměnné (takové, které platí jen v této funkci) a místo pro návratovou hodnotu. Ukázka takové struktury je na následujícím obrázku.



Obrázek 1 Zásobník aktivačních záznamů právě prováděných funkcí

Zásobník si můžeme představit jako vázu, do které dáváme míčky o šířce přesně padnoucí na šířku vázy. Vytáhnout lze ten míček, který byl do vázy hozen jako poslední. Zásobník je tedy typ struktury, kde jako první vytahujeme to, co jsme jako poslední vložili.

Úkol:

Pomocí rekurze napište funkci, která vrací n-tý prvek Fibbonacciho posloupnosti. Pro připomenutí:

```
Fib(0) = 0
Fib(1) = 1
Pro každé n>1 je Fib(n) = Fib(n-1) + Fib(n-2)
```

Takže máme globální a lokální proměnné (a taky konstanty, datové typy apod.). Globální proměnné se deklarují mimo jakoukoliv funkci a jsou dostupné kdekoliv, kdežto lokální proměnné se deklarují a jsou dostupné vždy jen v dané funkci. Je tu ještě další rozdíl: umístění v paměti. Globální proměnné, konstanty, uživatelské datové typy jsou umístěny v segmentu paměti určeném speciálně pro tento účel. Lokální proměnné, konstanty a uživatelské datové typy jsou v zásobníku, který má svůj vlastní segment v paměti. Globální segment a zásobníkový segment jsou v operační paměti na jiných místech, což poznáme podle adres.

Příklad pro zamyšlení (pointery a paměťové segmenty):

Podívejme se na následující kód:

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

int main() {
   int pole[]={0,1,2,3,4};
   int *p=pole;

   cout << setw(10) << "hodnota" << setw(12) << "*p" << setw(12) << "&p" << endl;
   for (; p<=&pole[4]; p++)
      cout << setw(10) << *p << setw(12) << p << setw(12) << &p << endl;
}
```

Tento kód zkopírujte a spusťte, prohlédněte si výstup.

Někoho možná zaujme, že u cyklu `for` je v závorce hned na začátku středník, tedy chybí první parametr. Nicméně to je zcela v pořádku, pokud nepotřebujeme, aby se na začátku cyklu něco inicializovalo. Používáme manipulátor `setw()` z knihovny `iomanip.h` pro nastavení šířky sloupce.

Proměnnou p jsme inicializovali – nasměrovali na začátek pole. V tomto případě nepoužíváme ampérsánd, ale mohli bychom, pokud bychom chtěli p nasměrovat na první prvek pole:

```
int *p=&pole[0];
```

Takže ampérsánd použijeme pro nasměrování na běžnou proměnnou, ale pole je jiný případ, samotné pole je ve skutečnosti odkaz na svůj první prvek, vnitřně jeho reprezentace nemá daleko k pointeru.

Pro každý prvek pole vypisujeme hodnotu v prvku pole, pak adresu daného prvku pole (všimněte si, o kolik se liší hodnoty na různých řádcích tohoto sloupce – souvisí to s délkou datového typu `int`?), a pak se v každém řádku vypíše totéž číslo. Co myslíte, že to číslo znamená? Všimněte si, že toto číslo je menší než čísla v předchozím sloupci.

Program trochu pozměníme:

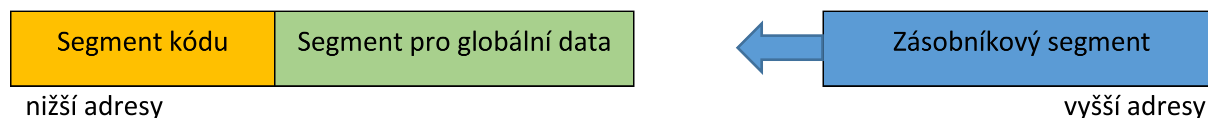
```
#include <iostream>
#include <iomanip>
using namespace std;

int pole[]={0,1,2,3,4};
int *p=pole;

int main() {
   cout << setw(10) << "hodnota" << setw(12) << "*p" << setw(12) << "&p" << endl;
   for (; p<=&pole[4]; p++)
      cout << setw(10) << *p << setw(12) << p << setw(12) << &p << endl;
}
```

Znovu spusťe. Všimněte si rozdílů: adresy vypadají jinak, a také je rozdíl v tom, co je větší nebo menší. V čem se kód liší? V prvním případě jsou proměnné deklarovány uvnitř funkce `main()`, tedy jsou *lokální*. V druhém případě jde o *globální proměnné*.

Délka zásobníkového segmentu se neustále mění. Z bezpečnostních důvodů je tento segment „otočený“: zatímco v jiných segmentech (včetně toho pro globální data) se paměť přiděluje od nižších adres k vyšším, u zásobníku je to naopak. Jak se postupně deklarují proměnné, přiděluje se jim paměť od vyšších adres k nižším, tedy zásobník roste směrem k nižším adresám. Je to proto, aby v případě narušení bezpečnosti (přetečení zásobníku – stack overflow) při nadměrném ukládání dat do zásobníku byly narušeny segmenty téhož procesu, a nikoliv segmenty jiných procesů.



10.3 Parametry volané hodnotou a parametry volané odkazem

V předchozích sekcích jsme psali funkce s *parametry volanými hodnotou*, tedy se nepočítalo s tím, že funkce do svých parametrů bude nějak zasahovat. Kdyby to udělala, stejně by se změna mimo prostor funkce neprojevila. *Parametry volané odkazem* jsou něco jiného – pokud do nich funkce zasáhne, pak se změna projeví i mimo tuto funkci (ve volajícím kódu).

Příklad:

```
// použijeme parametr volaný hodnotou, výsledek je v návratové hodnotě:
int abs1(int x) {
   if (x >= 0) return x;
   else return -x;
}                       // použití: acislo = abs1(cislo);

// použijeme parametr volaný odkazem s využitím dereference, výsledek je
void abs2(int *x) {                                         // v parametru
   if (*x < 0)
      *x = -(*x);
}                       // použití: abs2(&cislo);

// použijeme parametr volaný odkazem s využitím reference, výsledek je v parametru
void abs3(int &x) {
   if (x < 0)
      x = -x;
}                       // použití: abs3(cislo);
```

Pokud potřebujeme jako parametr funkce použít pole, pak používáme druhý z uvedených způsobů. Pokud bychom potřebovali funkci pro výpočet maximálního prvku pole, může deklarace vypadat takto:

```
int maximum (int pocet, int *pole);
int maximum (int pocet, int pole[]);
```

Vzhledem k blízkému vztahu mezi pointery a poli jsou tyto zápisy ekvivalentní a dá se s nimi běžně pracovat. Ovšem jak vidíme, potřebujeme také parametr určující délku pole. Takže volání pro pole o délce 20 čísel by bylo následující:

```
vysl = maximum (10, moje_pole);
```

Úkol:

Napište funkci, která má jako parametr pole (a parametr s délkou pole), vypočte a vrátí nejvyšší prvek tohoto pole.

V programu deklarujte pole 20 čísel, v cyklu do něj načtěte náhodná čísla menší než 1000 a pak pomocí sestavené funkce zjistěte maximum.

Vypište celé pole a pak vypište zjištěné maximum.

10.4 Globální a lokální proměnné

Každá (nejen) funkce může mít své vlastní proměnné. To už víme o funkci `main()` – vlastně skoro všechny proměnné, které jsme až dosud vytvářeli, byly *lokální* ve funkci `main()`, tedy mohly být používány jen ve funkci `main()`, ale týká se to i dalších funkcí a jiných programových konstrukcí.

Nejdřív si ujasníme jeden pojem: *blok* je sekvence příkazů uzavřená do složených závorek `{…}`. Kde jsme se už s bloky setkali:

- tělo cyklu (`while`, `do`, `for`), případně větev při větvení kódu (`if`, `switch-case`),
- tělo funkce.

Proměnná (nebo třeba uživatelský datový typ) je lokální v daném bloku, pokud je deklarace proměnné v tomto bloku. Pouze v daném bloku je „viditelná“, tedy platná, tam ji můžeme používat. Přesněji: i v daném bloku platí až od místa své deklarace.

Příklad:

V následujícím kódu je cyklus `for` s blokem ohraničeným složenými závorkami. Pouze uvnitř cyklu je platná proměnná `i`, uvnitř i vně těla cyklu (tj. v nadřízeném bloku) jsou navíc platné i další proměnné.

```
pocet = 0;
soucet = 0;
for (int i = 0; i < limit; i++) {
   if (pole[i] == 0)    // nulové hodnoty ignorujeme, přejdeme na další prvek
         continue;
   if (pole[i] == -1)   // při výskytu "-1" konec cyklu, zbytek pole ignorujeme
         break;
      soucet += pole[i];
      pocet++;
}
if (pocet > 0)
   prumer = soucet / pocet;
else
   prumer = 0;
```

Globální proměnná bývá deklarovaná ještě před začátkem funkce `main()`, případně před začátkem jakékoliv funkce, která v programu existuje, a je tedy platná všude (ve všem, co následuje).

Příklad:

Podívejme se na následující program, všimněte si, kde konkrétně je která proměnná deklarována.

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

int globalni = 10;
int soucet(int x, int y);          // deklarace funkce soucet()
                                   // mohlo by být taky:    int soucet(int, int);

void prehod(int &x, int &y)        // definice funkce prehod()
{         // v tomto bloku vidím proměnné globalni, x, y, pom
   int pom = x;
   x = y;
   y = pom;
}

int dalsi = -2;

int main()
{     // v tomto bloku vidím proměnné: globalni, dalsi, lokalni1
   int lokalni1 = 30;   // lokalní ve funkci main()

   cout << globalni << endl;
   cout << "soucet: " << soucet(3,-2) << endl;
   prehod(globalni, lokalni1);

   for (int i = 0; i < 10; i++)
   {    // v tomto bloku vidím proměnné: globalni, dalsi, lokalni1, i
      for (int j = 0; j < 12; j++)
      {    // v tomto bloku vidím proměnné: globalni, dalsi, lokalni1, i, j
         pole[i][j] += 10;
         cout << pole[i][j] << '\t';
      }
   }
```

```
}

int soucet (int x, int y)
{       // v tomto bloku vidím proměnné: globalni, dalsi, x, y
   return x+y;
}
```

Pozor, funkce `soucet()` je nejdřív pouze deklarovaná, definici najdeme až za funkcí `main()`. Naproti tomu funkce `prehod()` má nad `main()` celou definici.

Z toho vyplývá i důsledek ohledně viditelnosti: protože funkce `prehod()` má svůj kód nad deklarací proměnné `dalsi`, tato proměnná není ve funkci `prehod()` viditelná. Kód funkce `soucet()` je až po deklaraci proměnné `dalsi`, tedy s viditelností nebude problém.

V předchozí sekci byla řeč o zásobníku. Lokální proměnné platné jen v dané funkci jsou fyzicky uloženy právě v aktivačním záznamu této funkce, tedy v „obdélníku“ funkce. Co když ve stejném okamžiku existuje stejně pojmenovaná proměnná ve více „obdélnících“? To není problém. Do zásobníku koukáme shora (třebaže se můžeme prohrabovat i v níže položených záznamech), proto při průchodu shora první výskyt takto pojmenované proměnné bude ten správný.

Příklad:

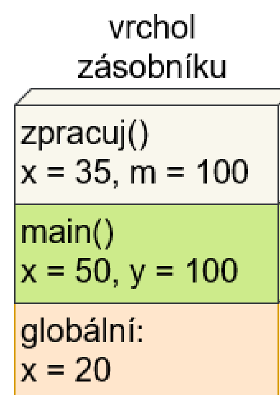
Podívejme se na následující kód (pozor, takhle radši neprogramujte – názvy proměnných by měly být výstižné):

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

int x = 4;

int zpracuj(int m) {
   int x = 35;
    return m+x;
}

int main() {
    int x = 50, y = 100;
    cout << zpracuj(y);
}
```



Na obrázku vedle vidíme situaci v zásobníku ve chvíli, kdy se zpracovává funkce `zpracuj()`. V zásobníku jsou celkem tři proměnné s názvem `x`, ale nejvýše (tedy první „na ráně“) je ta lokální uvnitř funkce.

Příklad:

Vytvoříme funkci načítající od uživatele prvky pole. Prvky sice budou typu int, ale chceme, aby byly nezáporné. To uděláme tak, že z každého načteného čísla uložíme jeho absolutní hodnotu. K tomu si taky vytvoříme funkci.

```
int absolutni(int cislo) {
   if (cislo < 0)
      return -cislo;
   else
      return cislo;
}
```

```
void nactiPoleNezapornych(int delka, int *pole)    // nebo (int delka, int pole[])
{
   int nacteno;
   cout << "Zadej pole (nezapornych) cisel: " << endl;
   for (int i = 0; i < delka; i++) {
      cout << i << ":\t";
      cin >> nacteno;
      pole[i] = absolutni(nacteno);
   }
}
```

Tuto funkci bychom pak volali následovně:

```
int main()
{
   const int delkaPole = 10;
   int mojePole[delkaPole];
   ...
   nactiPoleNezapornych(delkaPole, mojePole);
```

Funkce má návratový typ `void`, což (jak už víme) znamená, že nás „nezajímá“. Nicméně určitě by bylo lepší, kdyby vracela třeba celé číslo, podle kterého by se dalo poznat, jestli vše dopadlo dobře (ve funkci můžeme použít příkaz `return` pro předání návratové hodnoty znamenající určitou chybu).

Úkol:

Napište funkci nactiPole() s vhodnými parametry, která od uživatele postupně načte pole prvků typu int. Ve funkci použijte metodu pointerů, tedy namiřte jeden pointer na začátek pole, druhý na konec, třetí použijte jako „běžce“.

Na straně 35 je ukázka, jak při načtení pomocí objektu `cin` zjistit, zda vše proběhlo v pořádku. Tento postup použijte, abyste měli jistotu, že vstup od uživatele proběhl v pořádku. Tedy budete mít „vnější“ cyklus `for` běžící přes prvky pole, pak vnitřní cyklus (nejlépe `while` formovaný jako `if` na uvedené straně), kde pro právě načítaný prvek pole budete uživatele tak dlouho „otravovat“, dokud `cin` proběhne bez chyby.

Dále napište funkci, která vypíše prvky pole (mezi nimi tabulátor). Pokud se na to cítíte, pak v případě pole delšího než 10 vypisujte prvky vždy po 10 číslech na řádek.

Ve funkci `main()` vytvořte pole 15 prvků a použijte výše popsané funkce pro načtení a vypsání obsahu tohoto pole. Pro účely ladění můžete nejdřív vytvořit kratší pole a až po odladění funkcí jeho délku prodloužit. Pro délku pole použijte konstantu deklarovanou pomocí `const`, protože dané číslo budete potřebovat i jako jeden z parametrů obou funkcí.

Upozornění:

Funkce by neměly být moc dlouhé (a to platí i o funkci `main()`), doporučuje se, by se vešly na obrazovku (plus mínus). Měly by být přehledné a s naprosto jasným účelem. Název funkce by měl být dostatečně popisný (což nutně neznamená dlouhý – pojmenováváme stručně, jasně, výstižně).

Pokud není nutné použít parametry volané odkazem, tak je nepoužíváme, protože se jedná o „vedlejší efekt“. Vedlejší efekty mohou působit problémy, pokud si někdo dostatečně pozorně nepročte kód, protože jejich změnu mimo funkci nemusíme čekat.

11 Soubory a streamy

11.1 Soubory projektu

Kód jazyka C ukládáme do *zdrojového souboru* s příponou .c, kód jazyka C++ ukládáme do souboru s příponou .cpp. To už víme. V rámci projektu, ze kterého má vzniknout aplikace, můžeme mít i několik zdrojových souborů, v každém kód pro různé části aplikace, případně programovaný různými lidmi.

Dále můžeme používat *hlavičkové soubory*, které mají příponu .h. K čemu jsou nám dobré?

V kapitole o funkcích jsme si vysvětlili rozdíl mezi definicí a deklarací funkce. Pokud chceme, aby se určitá funkce používala i mimo ten konkrétní soubor s kódem, kde je její definice, potřebujeme deklaraci této funkce „zveřejnit“. To jde právě v hlavičkovém souboru.

Podobně můžeme v hlavičkovém souboru deklarovat globální proměnné, uživatelské datové typy, konstanty, prostě cokoliv, co není přímo kód a má být přehledně dostupné. Hlavičkové soubory totiž bývají typicky kratší než zdrojové soubory, tedy snadněji se v něm hledá „jak se co jmenuje“. Komentáře k jednotlivým funkcím a globálním proměnným či konstantám je taky dobré dávat sem (a taky do kódu).

Příklad:

Předpokládejme, že v souboru `hlavni_modul.cpp` máme definici (tj. včetně kódu) funkce `nactiPole()` a chceme, aby tato funkce byla použitelná i v jiných modulech projektu. Dále tam budou deklarace dalších funkcí a další informace. Vše se bude překládat do jediného spustitelného souboru, všechny soubory projektu si navzájem „důvěřují“ co se týče deklarací.

V hlavičkovém souboru `hlavni_modul.h`:

```
// nacte pole od uzivatele, první parametr je delka pole, druhy samotne pole
int nactiPole(int, int*);

// vypise pole, jednotlive prvky oddelene tabulatorem, vzdy po 10 na radek,
// prvni parametr je delka pole, druhy samotne pole
int vypisPole(int, int*);
```

Ve zdrojovém souboru `hlavni_modul.cpp`:

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

#include "hlavni_modul.h"

int main() {
... kod funkce main()
}

int nactiPole (int delka, int *pole) {
... kod funkce nactiPole()
}

int vypisPole (int delka, int *pole) {
... kod funkce vypisPole()
}
```

Všimněte si, že (téměř) na začátku zdrojového souboru je pomocí direktivy #include načten hlavičkový soubor. Tato direktiva způsobí, že daný soubor je opravdu znak po znaku vložen na dané místo.

Úkol:

Dokončete kód, který je naznačen v příkladu na předchozí straně. Přeložte soubor s příponou .cpp. Pak vyzkoušejte, zda vše funguje jak má.

Pokud chcete mít projekt skládající se z více než jednoho souboru (tedy včetně hlavičkových), pak si možná nevystačíte s online editory/překladači a je třeba použít některý „lokální" instalovaný překladač. Pokud máte po ruce Linux, není problém, obvykle tam bývá nainstalovaný překladač gcc, případně g++. Jinak použijte návod k programovacím nástrojům na straně 12.

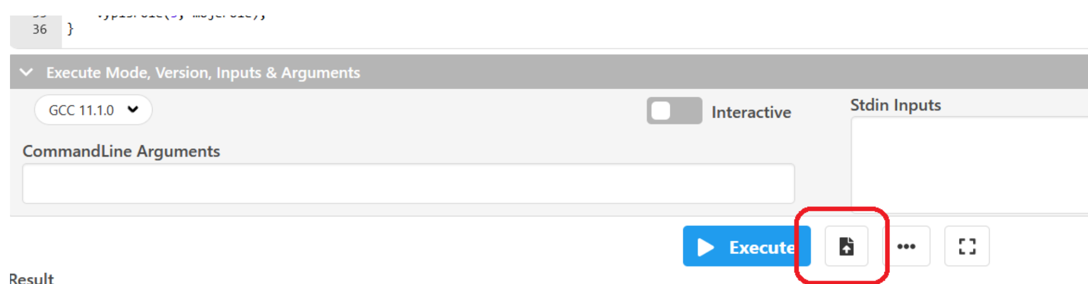
11.2 Soubory jako vstup a výstup

Při programování se často dostáváme do situace, kdy nám nestačí vstup zadaný na klávesnici uživatelem, ale potřebujeme vstup načíst ze souboru, případně uložit (zapsat) výstup do souboru.

Nejdřív upozornění:

Toto jde i v některých online nástrojích, třebaže je tam práce soubory někdy poněkud kostrbatá (musíme vědět, kam předem načíst vstupní soubory a kde následně hledat výstupní soubory).

Například u jDoodle to je možné. Mohu předem načíst soubory, které budou sloužit jako vstup, pak v kódu bude takový soubor přístupný ve složce „`/uploads`". Na obrázku níže je vyznačeno tlačítko (vedle tlačítka pro spuštění programu), přes které načítáme soubory, na něž se budeme v kódu odkazovat.



Jestliže takto předem načteme například soubor `test.txt`, pak se na něj v kódu budeme odkazovat jako na `/uploads/test.txt`.

Pokud chci v kódu vytvořit a používat soubor pro výstup, bude přístupný ve složce „`/myfiles`". Takže pokud například v kódu vytvořím nový soubor `vystup.txt`, budu se na něj odkazovat plným názvem `/myfiles/vystup.txt`.

V C++ existují dva možné způsoby (přístupy) práce se soubory: původní přístup z jazyka C implementovaný v knihovně stdio.h, a pak novější přístup přidaný při přechodu do C++, pomocí streamů (proudů).

11.2.1 Práce se soubory objektově podle C++: streamy

Otevřený soubor je při použití tohoto přístupu chápán jako proud (stream) dat, který je vytvořen v operační paměti během otevírání souboru. Stream tedy otevřeme pro určitý typ operace (čtení či zápis) a pak s ním můžeme pracovat. Ve skutečnosti tento přístup už dávno používáme, protože objekty `cin` a `cout` jsou právě takto implementovány.

Princip je následující: pro práci se soubory potřebujeme mít načtenou knihovnu `fstream.h`. Při otevírání souboru určíme, zda má být otevřen pro čtení, zápis nebo obojí. Po příkazu pro otevření

bychom si pro jistotu měli ověřit, jestli vše proběhlo v pořádku, a pak už můžeme se souborem pracovat.

Otevřený soubor je vlastně proměnná určitého datového typu. Máme na výběr:

- `ifstream` je datový typ pro vstupní stream, tedy soubor určený pro čtení,
- `ofstream` je datový typ pro výstupní stream, tedy soubor určený pro zápis, případně tak můžeme vytvořit nový soubor,
- `fstream` je datový typ pro obojí – umí vše, co umí `ifstream` i `ofstream`.

U všech tří datových typů je definována funkce `open()`, které zadáme název souboru (případně včetně cesty k němu, pokud se nachází jinde než kde bude spustitelný soubor) a dále mód otevření. Módy otevření ukazuje Tabulka 5, můžeme zkombinovat více módů (které se navzájem nevylučují).

Tabulka 5 Módy otevření souboru typu fstream v prostoru ios

Název módu otevření souboru	Význam
ios::in	(input) otevře soubor pro čtení
ios::out	(output) otevře soubor pro zápis, pozice v souboru je na začátku
ios:: app	(append) otevře soubor pro zápis, veškerý výstup bude zařazen na konec, původní data nebudou přepisována
ios::ate	(at end) soubor se otevře pro zápis, pozice je na konci souboru, ale zapisovat se může kamkoliv
ios::trunc	(truncate) soubor je při otevření oříznut na nulovou délku, tedy původní obsah se ztratí, pozice v souboru je na začátku
ios::binary	otevře soubor v binárním módu; pokud se nepoužije, soubor bude otevřen jako textový (v UNIXových systémech není nutný, ale jinde ano)

Příklad:

Otevřeme soubor `text.txt` pro čtení. Nejdřív si ukážeme základ bez textování chyb:

```
#include <iostream>
#include <fstream>   // u obou můžeme, ale nemusíme uvést příponu .h
using namespace std;

int main() {
   ifstream vstupniSoubor;
   char data[20];

   vstupniSoubor.open ("text.txt", ios::in);   // pozor, dvojice dvojteček
   vstupniSoubor >> data;                      // zapisujeme do souboru
   ... // zde pracujeme s načtenými daty
   vstupniSoubor.close();
}
```

Ovšem lepší by bylo mít ošetřeny případné problémy s otevřením souboru:

```
   vstupniSoubor.open("text.txt", ios::in);
   if (!vstupniSoubor.is_open()) {      // soubor se nepodařilo otevřít
      cout << "Chyba, soubor text.txt se neotevrel" << endl;
      return 1;
   }
```

K chybě může dojít samozřejmě i při uzavírání souboru, což zjistíme takto:

```
vstupniSoubor.close();
if (vstupniSoubor.is_open())
   cout << "Chyba, soubor text.txt se nepodarilo uzavrit, zmeny se neprovedly"
   << endl;
```

Ve skutečnosti nemusíme u otevírání souboru deklarovaného jako `ifstream` uvádět mód `ios::in`, protože je implicitní (tj. přednastavený). Nicméně můžeme.

Příklad:

Při čtení ze streamu můžeme data načítat do řetězce typu `char*`, ale taky lze použít datový typ `string`. Zde přepíšeme obsah souboru na standardní výstup, s funkcí `getline()` nebudeme zacházet „objektově". „Objektová" `getline()` má trochu problém s mezerami v řetězci (považuje je za oddělovač řetězců, nikoliv jejich součást), neobjektová bere i mezery jako součást řetzce:

```
ifstream vstupniSoubor;
string retezec;            // nezapomente na knihovnu string.h
const int delka = 81;
char radek[delka];      // 80 znaku plus ukoncujici nulovy znak \0

vstupniSoubor.open("text.txt");
if (vstupniSoubor.open()) {
   while (getline(vstupniSoubor, retezec))
      cout << retezec << endl;
}
vstupniSoubor.close();
```

Funkci `getline()` můžeme dodat i další parametry, což je užitečné hlavně tehdy, když načítáme do nulou ukončeného řetězce `char[]`. Například pokud chceme načíst celý (kratší) řádek, nebo v případě, že by řádek byl příliš dlouhý, tak jen zadaný počet řádků (tj. co vyjde kratší):

```
while (!vstupniSoubor.eof()) {
   vstupniSoubor.getline(radek, delka, '\n');
   cout << radek << endl;
}
```

Zde s funkcí `getline()` zacházíme „objektově", tj. voláme ji jako funkci objektu `vstupniSoubor`.

Příklad:

Ukážeme si možnosti zápisu do souboru:

```
ofstream vystupniSoubor;
char data[] =   "prvni radek\n"
                "druhy radek\n"
                "treti radek\n";

vystupniSoubor.open("text.txt");  // ios::out je implicitní mód, nemusí se uvádět

vystupniSoubor << "Zapisuji do souboru tento konstantni retezec\n";
vystupniSoubor << data;

// nebo jiný způsob:
vystupniSoubor.write(data, strlen(data));

vystupniSoubor.close();
```

Příklad:

Otevřeme soubor pro zápis v režimu append:

```
fstream soubor;      // mohlo by byt ofstream
char retezec[] = "Tento text pridame na konec souboru.\n";

soubor.open("text.txt", ios::of | ios::app);   // stačilo by ios::app
soubor << retezec;
soubor.close();
```

Příklad:

Abychom neměli pocit, že do souboru nutně musíme zapisovat pouze řetězece – můžeme tam zapisovat cokoliv, co třeba do objektu cout, ovšem v souboru to bude uloženo jako textový řetězec.

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

struct OSOBA {                  // musí být před deklaracemi funkcí, protože je
   string jmeno;                //  používána v jejich parametrech
   int identifikator;
};
// deklarace funkcí, jejich definice jsou za main()
void nactiOsobu (OSOBA &o);     // všimněte si: parametr volaný odkazem, reference
void zapisOsobuDoSouboru (fstream &f, OSOBA o);   // první odkazem, druhý hodnotou

int main() {
   fstream soubor;
   const int delka = 4;
   OSOBA osoby[delka];

   soubor.open("soubor.txt", ios::out | ios::trunc);
   if (!soubor.is_open()) {
      cout << "Soubor se neotevrel\n";
      return 1;
   }
   for (int i = 0; i < delka; i++) {
      nactiOsobu (osoby[i]);
      zapisOsobuDoSouboru(soubor, osoby[i]);
   }
   soubor.close();
}

void nactiOsobu (OSOBA &o) {
   cout << "Zadej jmeno: ";
   cin >> o.jmeno;
   cout << "Zadej identifikator: ";
   cin >> o.identifikator;
}

void zapisOsobuDoSouboru (fstream &f, OSOBA o) {
   if (f.is_open())
      f << o.jmeno << '\t' << o.identifikator << endl;
   else
      cout << "Soubor neni otevren, zapis nebyl proveden\n";
}
```

Příklad:

Pro objekty typu `ifstream` jsou kromě `getline()` k dispozici také funkce `get()` a `read()`. Podobně pro objekty typu `ofstream` máme kromě `write()` také funkci `put()`. Ukážeme si použití `get()` a `put()` na příkladu kopírování obsahu jednoho souboru do druhého po jednotlivých znacích:

```
ifstream vstupni;
ofstream vystupni;

vstupni.open("soubor1.txt");
if (!vstupni.is_open()) {
   cout << "Neotevrel se soubor soubor1.txt!\n";
   return 1;
}

vystupni.open("soubor2.txt");
if (!vystupni.is_open()) {
   cout << "Neotevrel se soubor soubor2.txt!\n";
   return 2;
}

char znak;
while (vstupni.get(znak))
   vystupni.put(znak);

vstupni.close();
vystupni.close();
```

V reálné situaci bychom zřejmě nekopírovali soubory po znacích, je to zde jen jako příklad.

Funkce `get()` ve skutečnosti umí pracovat i s řetězci, jen jí musíme dodat trochu jiné parametry:

```
int delka = 81;
char retezec[delka];
vstupni.get (retezec, delka, EOF);  // do proměnné retezec se načetlo buď 80
          // znaků, nebo méně, pokud je konec souboru (EOF) blíž než 80 znaků
```

11.2.2 Pohyb v streamu souboru

Při otevření souboru jsme vždy na určité pozici v souboru, v závislosti na módu. Může to být na začátku, může to být na konci. Vždy, když čteme nebo zapisujeme, se mění naše pozice v souboru, ale my ji třeba chceme měnit i bez čtení či zápisu.

Do otevřeného streamu ukazují dva „pointery“ – jeden pro čtení, druhý pro zápis. Nicméně v případě streamů pro soubory se obvykle kryjí, ukazují na totéž místo. Přesto je rozlišujeme takto:

- „get“ pointer – ukazuje na pozici pro čtení, například pomocí `get()`,
- „put“ pointer – ukazuje na pozici pro zápis, například pomocí `put()`.

Jak pohnout „get“ pointerem:

```
fstream soubor;
... otevřeme soubor, atd.

// ve streamu se přesuneme na zadanou pozici, počítáno od začátku souboru:
soubor.seekg(25);

// ve streamu se přesuneme na danou pozici, počítáno od momentální pozice:
soubor.seekg(25, ios::cur); // směrem dále
soubor.seekg(-8, ios::cur);  // směrem zpět
```

```
// ve streamu se přesuneme na danou pozici, počítáno od konce souboru:
soubor.seekg(-25, ios::end);
```

Podobně se používá funkce `seekp()`, jen se týká „put“ pointeru.

Hodilo by se mít taky funkci, která nám řekne momentální pozici v souboru. To je `tellg()` a `tellp()`.

```
int pozice = soubor.tellg();
... tady se souborem pracujeme, pohybujeme se v něm
soubor.seekg(pozice);     // návrat na uloženou pozici
```

11.2.3 Práce se soubory „céčkovým“ stylem

V knihovně `stdio.h` používané už od dob „neobjektového“ jazyka C, máme k dispozici trochu jiný přístup k souborům.

Příklad:

Ukážeme si, jak otevřít soubor pro čtení či zápis, dále po znacích zkopírujeme obsah jednoho souboru do druhého.

```
#include <stdio.h>

int main() {
   FILE *souborVstup, *souborVystup;
   char znak;

   if ((souborVstup = fopen("soubor1.txt", "r")) == NULL) {
      printf("Vstupni soubor se nepodarilo otevrit.\n");
      return 1;
   }
   if ((souborVystup = fopen("soubor2.txt", "w")) == NULL) {
      printf("Vystupni soubor se nepodarilo otevrit.\n");
      return 2;
   }

   while ((znak = getc(souborVstup)) != EOF)
      putc(znak, souborVystup);

   fclose(souborVstup);
   fclose(souborVystup);
}
```

Kromě přístupu po znacích máme k dipozici funkce `fprintf()` a `fscanf()`, které dokážou zapisovat či číst různé typy dat. Například:

```
char data1[] = "Pokusny retezec", data2[30];
int cislo1 = 25, cislo2;
fprintf(souborVystup, "%d %s\n", cislo1, data1);
fscanf(souborVstup, "%d %s\n", &cislo2, data2);
```

Modifikátory datových typů (zde `%d` pro `int` a `%s` pro `char*`) jsou stejné jako u funkcí `printf()` a `scanf()`, vlastně i to ostatní, s tím rozdílem, že u funkcí pracujících se soubory je na začátku přidáno písmeno `f`. Nejdůležitější modifikátory najdeme v tabulce na straně 17.

11.2.4 Jak zpracovat celý soubor

Při načítání textového souboru bychom si měli dát pozor na konec – omylem můžeme načíst o jeden (prázdný) řádek víc, protože symbol konce souboru EOF nebude včas detekován. Mělo by to vypadat takto (třebaže to znamená pro každý řádek jedno testování navíc):

Příklad:

Obsah vstupního souboru po řádcích zkopírujeme do výstupního souboru.

```
fstream vstupniSoubor, vystupniSoubor;
string radek;

... // otevřeme soubory atd.
while (!vstupniSoubor.eof()) {
   getline(vstupniSoubor, radek);      // nebo soubor.getline(radek);
   vystupniSoubor << radek;            // obsahuje i symbol konce řádku
   if (!vstupniSoubor.eof())
      vystupniSoubor << endl;   // aby byl správně identifikován poslední „Enter“
}
```

11.3 Řešení problémů se streamy

Občas se stane, že několik vstupů (`cin`) za sebou „splyne“, tedy že uživateli není umožněno část vstupů zadat. Je to způsobeno tím, že předchozí použití objektu `cin` ještě nebylo odstraněno z bufferu a má se za to, že následující vstup už proběhl. Pokud se něco takového stává (což je možné, pokud v krátké době po sobě následuje více vstupů), je řešením použít funkci objektu `cin` pojmenovanou `ignore()`.

Příklad:

Funkci `ignore()` voláme buď bez parametrů, nebo s parametrem určujícím velikost obsazeného bufferu (případně prostě uvolnit buffer celý):

```
#include <limits>
...
cout << "Zadejte nazev: ";
cin >> kniha.nazev;
cin.ignore();

cout << "Zadejte jmeno autora: ";
cin >> kniha.autorJmeno;
cin.ignore(numeric_limits<streamsize>::max(), '\n');   // je v knihovně limits

cout << "Zadejte prijmeni autora: ";
cin >> kniha.autorPrijmeni;
```

Příklad:

Prohlédněte si následující kód pro vygenerování obsahu knihy. Najdeme v něm jak nastavení šířky položek pro `cout`, tak i nastavení výplňového znaku (tečky), pole struktur, ignorování „nechtěného“ obsahu bufferu pro `stdin` a samozřejmě pointery, definice a deklarace funkcí.

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

const int maxKapitol = 10, sirkaStranky = 70;
```

```
struct KAPITOLA {
   string nazev;
   unsigned int strana;
};

int nactiKapitoly (int, KAPITOLA*);    // načte názvy kapitol do pole stringů
void vypisObsah (int, KAPITOLA*);

int main() {
   unsigned int pocetKapitol;
   KAPITOLA kapitoly[maxKapitol];

   pocetKapitol = nactiKapitoly (maxKapitol, kapitoly);
   vypisObsah (pocetKapitol, kapitoly);
}

int nactiKapitoly (int delka, KAPITOLA *pole) {
   int pocet = 0;
   cout << "Pokud misto nazvu klepnete na Enter, ukonci se nacitani.\n\n";

   while(pocet < delka) {
      cout << "Nazev: ";
      getline(cin, pole[pocet].nazev); // aby to bralo i mezery

      if (pole[pocet].nazev.length() == 0)
         break;     // uzivatel uz nechce dalsi knihu
      cout << "Strana: ";
      cin >> pole[pocet].strana;
      cin.ignore();
      pocet++;
   }
   return pocet;
}

void vypisObsah (int pocet, KAPITOLA *pole) {
   KAPITOLA *prvni, *posledni, *p;
   int sirkaTecek;
   if (pocet > 0) {
      prvni = &pole[0];
      posledni = &pole[pocet-1];
      cout << endl;
      for (p = prvni; p <= posledni; p++) {
         sirkaTecek = sirkaStranky - p->nazev.length();
         cout << p->nazev << setw(sirkaTecek) << setfill('.') << p->strana << endl;
      }
   }
}
```

Užitečnou funkcí pro tyto objekty je `fail()`, pomocí které zjistíme, zda poslední vstup/výstup provedený s daným objektem náhodou neselhal nebo zda nedošlo k chybě ve formátu. Funkce `bad()` má podobný význam, jen si nevšímá chyb ve formátu, pouze sděluje, zda výstup selhal.

Příklad:

Ošetření vstupu může být například následující:

```
int x;
cout << "Zadej cislo: ";
```

```
do {
   cin >> x;          // teď bychom schválně zadali místo čísla řetězec s písmeny:
   if (cin.fail())
      cerr << "chyba pri zadavani cisla, prosim znovu: ";
   else break;
} while (1);          // „nekonečná“ smyčka, která by skončila příkazem break
```

Také existuje funkce `good()`, která funguje opačně: pokud vše ve streamu dopadlo dobře, vrací true, jinak false.

Úkol:

Vymyslete si vlastní program podobný tomu na předchozí straně (tam šlo o vygenerování obsahu). Mělo by to být něco, kde použijete pole struktur (je na vás, čeho konkrétně se bude týkat a jaké budou vnitřní proměnné dané struktury), měla by tam být funkce pro načítání a funkce pro výpis zformátovaného výstupu. Výstupem by mělo být něco na způsob tabulky, kdy alespoň některý ze sloupců bude mít pevnou šířku.

Například: půjde o soutěž či soutěžní výstavu, máte vygenerovat výsledky soutěže. Pro každého soutěžícího bude například počet bodů či skóre (tam, kde jsme u obsahu měli číslo strany).

Vstup může být od uživatele, nebo můžete použít soubor.

12 Řazení

Předpokládejme, že máme (potenciálně velmi dlouhé) pole či jinou strukturu prvků (čísla, řetězce, struktury, objekty – cokoliv) a potřebujeme toto pole seřadit podle určitého kritéria. To nemusí být nutně „od nejmenšího po největší“ nebo „podle abecedy“, kritérium může být i jiné. Zvažujeme:

- je třeba určit porovnávací kritérium, tedy pro dvě hodnoty – podle čeho poznat, v jakém mají být pořadí,
- je třeba prohodit tyto dvě hodnoty v poli, pokud jsou ve špatném pořadí,
- potřebujeme celkový algoritmus, který určí, kdy se které dvě hodnoty mají porovnávat a případně prohazovat.

Třetí problém řešíme tak, že vybereme vhodný řadicí algoritmus. Existuje jich celá řada: nejznámější jsou Bubble Sort (bublinkové řazení, používá se spíše pro menší sady dat) a Quick Sort (celkem rychlé řazení i pro velká čísla), ve skriptech pro přednášky jich najdete spoustu.

První a druhý problém jsou taky snadno řešitelné: pokud jen porovnáváme čísla či řetězce, prostě použijeme relační operátor a přehodíme třeba s pomocnou proměnnou. Pokud porovnáváme něco složitějšího, pak stojí zato napsat si funkci, které předáme souřadnice porovnávaných prvků (případně pointery) a tato funkce jak porovná, tak i v případě potřeby přehodí.

Příklad:

Ukážeme si jednoduchý Bubble Sort na poli čísel:

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

void bubbleSort(int delka, int *pole) {
   int zastavit = delka-1, pom;
   for (int i = 0; i < zastavit; i++)
      for (int j = 0; j < zastavit-i; j++)
         if (pole[j] > pole[j+1]) {
            pom = pole[j];
            pole[j] = pole[j+1];
            pole[j+1] = pom;
         }
}

int main() {
   const int delkaPole = 20;
   int mojePole[delkaPole];
   srand(time(0));
   for (int i = 0; i < delkaPole; i++) {
      mojePole[i] = rand() % 1000;
      cout << setw(5) << mojePole[i];
   }
   cout << endl;

   bubbleSort(delkaPole, mojePole);
   for (int i = 0; i < delkaPole; i++)
      cout << setw(5) << mojePole[i];
   }
```

V IS máte ukázkový soubor s kódem několika různých řadicích algoritmů, vyzkoušejte.

Doporučená literatura

[1] MAREŠ, Martin a Tomáš VALLA. *Průvodce labyrintem algoritmů*. Druhé. Praha: CZ.NIC, z. s. p. o., 2022. ISBN 978-80-88168-66-9. Dostupné také z: https://knihy.nic.cz/files/edice/pruvodce_labyrintem_algoritmu_v2.pdf

[2] WRÓBLEWSKI, Piotr. *Algoritmy*. Dotisk 1. vydání. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.

[3] https://alvinalexander.com/programming/printf-format-cheat-sheet/

[4] https://en.cppreference.com/

[5] https://cplusplus.com/doc/tutorial/

[6] Myslíme v jazyku C++: https://books.google.cz/books?id=fSk99oy_mKcC&printsec=frontcover

[7] Codecast: editor, který umí i vizualizovat použitou paměť: https://codecast.france-ioi.org/v6/