Programování

Algoritmus jde o řešení nějakého problému pomocí předem dané posloupnosti kroků. Lze rozlišit tři základní druhy algoritmů: a) slovní – v praxi se jedná o nějaký psaný návod, např. na obsluhu nějaké elektroniky
b) grafický – v praxi jde o obrázkový návod, např. návod na slepení modelu, ale my se častěji setkáme s grafickým vyjádřením připravovaných programů s tzv. vývojovými grafy nebo strukturogrami
c) vyjádřený z instrukcí – jde o řešení nějakého problému v nějakém konkrétním programovacím jazyce

Algoritmus musí splňovat následující podmínky: a) jednoznačnost
b) obecnost
c) rezultativnost

Jednoznačnost

- každý krok algoritmu musí být přesně popsán a nesmí dovolovat více výkladů, jestliže se program má rozhodnout tak programátor musí přesně nastavit podmínky.

Obecnost

- algoritmus musí vyhovovat obecně řešení z nějaké skupiny úloh tzn. když budeme psát algoritmus na sčítání dvou čísel, čísla nejsou zadána v algoritmu napevno, ale uživatel je může libovolně zadat. Nejlepší řešení je napsat algoritmus na libovolnou matematickou operaci – kalkulačku. Správně obecně napsaný algoritmus zareaguje na jakékoliv zadání a to i na nesprávné.

Rezultativnost

- algoritmus musí vždy vysvětlit proces v nějakém časovém limitu nebo v nějakém počtu dílčích kroků, jestliže to nedokáže, tak se vykonávání musí se zastavit nebo přerušit.

Fáze při vytváření programu

1. Seznámení s problematikou
   - programátor musí danou problematiku nastudovat např. bude-li tvořit účetnický program musí nastudovat příslušnou legislativu a vytvoření základního návrhu algoritmu
2. Výběr vhodného programovacího jazyka
   - na základě nastudování problematiky programátor vybere vhodný programovací jazyk tzn. ne každý prog. jazyk je vhodný pro řešení nějakého problému
3. Vlastní programování
   - vytváření vlastního kódu programu
4. Testování
   - nejprve je při vlastním testování uvolněna alfa verze – tu testují jen vybraní testeři, kteří jsou za to zpravidla placeni, po té následuje uvolnění tzv. beta verze – tu testuje široká dobrovolné veřejnost. Nebo jde-li o program na zakázku je dán na otestování zákazníkovi. Po otestování a opravě zásadních chyb následuje jeho ostré používání.
5. Život aplikace
   - v průběhu používání aplikace jejího života následují upgrady tedy jeho vylepšování, opravování chyb či změny z důvodu legislativy.

Programovací jazyky

- existuje několik způsobů dělení, každý způsob dělení zobrazuje na ně jeden úhel pohledu tzn. programovací jazyk z jednoho úhlu pohledu je zařazen do nějaké skupiny jazyků a z jiného úhlu pohledu je ve skupině jazyků, které při předchozím úhlu pohledu spadaly to jiné skupiny.

1. Dělení dle úrovně abstrakce.

(a) vyšší – programování se provádí pomocí příkazů, program je psán v tzv zdrojovém kódu. Tyto příkazy jsou člověku bližší a programování je pro člověka jednodušší. Na druhou stranu počítač při vykonávání programu napsaném ve vyšším programovacím jazyku musí tento kód převést do strojového kódu, tedy do jazyka zpracovávaného počítačem. Představitelé C,C ++,C #, Visual Basic. Tuto skupinu lze rozdělit na dvě další podskupiny:
   i. Procedurální (imperativní) - tyto programovací jazyky popisují výpočet pomocí posloupnosti příkazů, které určují přesný postup (algoritmus), jak danou úlohu řešit. Program je soustavou proměnných, které v závislosti na vyhodnocení jednotlivých příkazů programu změní svůj stav.
      A. Strukturované programovací jazyky používají programovací techniku, kdy se implementovaný algoritmus rozděluje na dílčí úlohy (funkce, bloky příkazů), které se spojují v jeden celek. K implementaci v programu se používá vybraných řídicích struktur, ostatní struktury nejsou povoleny vůbec nebo jejich použití v programech je velice výjmečné (např. GOTO). Typickými představiteli jsou programovací jazyk C nebo Basic.
      B. Objektově orientované programovací jazyky respektive kód v nich napsaný je složen s tzv. objektů. Programování v nich spočívá v modelování nějaké části. Jednotlivé prvky modelované reality jsou jednotlivé objekty kódu, kde se seskupují do tzv. entit. Objekty si pamatují svůj stav a navenek poskytují operace. Každý objekt pracuje jako černá skříňka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje. Objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, jednotlivé objekty spolu komunikují pomocí rozhraní. Typickými představiteli jsou programovací jazyk C## nebo Java.
   ii. Neprocedurální (deklarativní) – tyto programovací jazyky jsou založeny na programování pomocí definic. Programátor píše co se má udělat, ale neřeší jak se to má udělat. U deklarativních programovacích jazyků je specifikován cíl a algoritmizace je ponechána programu daného jazyka. Opšt se dělí na dvě základní skupiny programovacích jazyků:
      A. Funkcionální programovací jazyky – výpočet je zpracován jako vyhodnocení matematických funkcí. Aplikace je složena z jednotlivých funkcí, tyto funkce jsou aplikované na argumenty a vypočítávající deterministicky jediný výsledek. Typickými představiteli jsou programovací jazyk Hascel, Lisp nebo Scheme.
      B. Logické programovací jazyky – využívají matematické logiky jako prostředku pro programování. Je zde velice důležité psát program, tedy jednotlivé výroky (věty) efektivně. Vyhodnocováním výroků zabezpečuje část programovacího jazyka zvaná dokazovač vět. Pro efektivní psání výroků by programátor, měl vědět jak dokazovač vět pracuje. Typickým představitelem je programovací jazyk Prolog.

(b) nižší – neboli jazyky symbolických adres (JSA). Jsou velice blízké strojovému kódu, tedy jejich zpracování - vykonání je velice rychlé, ale na druhou stranu je programování v nich pro člověka složitější. Prakticky jde skoro o 1 a 0. Program napsaný v nižším programovacím jazyku má nejčastěji příponu COM. Nejčastěji jsou v těchto jazycích napsány jádra operačních systému. Typickým představitelem je ASSEMBLER.

2. Dělení dle prostředí.
   (a) Kompilované – programátor program vytváří ve zdrojovém kódu a kompilátor nebo-li překladač ho převede do „strojového“ kódu a vytvoří spustitelný soubor s příponou EXE či COM. Při spouštění tohoto programu není nutné mít v počítači nainstalováno prostředí programovacího jazyka. Mezi hlavní představitele patří: C, C++, C#, Pascal.
   (b) Interpretované – programátor obvykle vytváří program ve zdrojovém kódu, ale neexistuje zde žádný překladač. Programy jsou spouštěny pomocí prostředí

programovacího jazyka, toto prostředí musí být bezpodmínečně nainstalováno v počítači.

Protože se program převádí do strojového kódu až při spuštění, tak programy v nich napsané bývají pomalejší něž u jazyků kompilovaných. V jistém ohledu je tato nevýhoda velikou výhodou, protože se program nepřevede do strojového (každá platforma má svůj jedinečný strojový kód), lze velice snadno ho přenášet mezi vzájemně nekompaktibilními platformami. Jediné co je nutné dodržet, aby jednotlivé platformy podporovaly daný jazyk. Například program napsaný por mobilní telefon v jazyku Java lze velice snadno spustit na PC.

Ve dřívější dobách bývalo téměř vždy pravidlem, že program napsaný v interpretovaném programovacím jazyku je pomalejší, ale dnes existuje spoustu interpretovaných jazyků, ve kterých jsou určité druhy programů rychlejší než, kdyby byly napsány v kompilovaném programovacím jazyce např. v C. Interpretovaný programovací jazyk MATLAB, je určen pro věděcké výpočty a provádění těchto výpočtů je v něm velice rychlé. Další představitelé jsou Basic, Perl, Java

Moderní zásady programování

1. Postup od shora dolů: Program (problém) je v podstatě pyramida v jejímž vrcholu je zadání řešení postupu je směrem dolů se problém postupně konkretizuje a nárůstá do šířky. Výsledkem je pak konkrétní řešení. Tento způsob umožňuje snadnou lokalizaci chyby a její odstranění

2. Modularizace: Program je rozčleněn do jednotlivých samostatných částí bloků (modulů) .Tyto moduly mají maximálně několik desítek řádků což nám umožnuje rozdělit program mezi programátory.

3. Stavebnicovost:Při tvorbě modulů musíme být co nejdříve univerzální,aby šel modul použít ve více projektech

Vývojové diagramy

- jde o grafické znázornění algoritmů
- vycházejí z normy ČSN ISO 5807 "Zpracování informací. Dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy programu a diagramy zdrojů systému"
- používají se při základním návrhu algoritmu, rovněž bývají přílohou součástí dokumentace, které popisuje zdrojový kód
- jde o soustavu symbolů, kde každému symbolu je přeřazena nějaká dílčí operace (vstup, výstup, podmínka...)
- symboly jsou propojeny spojnicemi
- prioritní způsob vyhodnocování vývojového diagramu je od shora dolů a zleva doprava

Spojnice

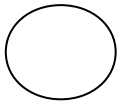
- jde o vertikální nebo horizontální čáry, které se mohou křížit nebo spojovat
- místo obyčejných čar lze použít i orientované šipky. Využívají se zpravidla, když směr vyhodnocování je jiný než standardní nebo je nutné směr zvýraznit.

Symboly:

Ovál – značí začátek a konec algoritmu. Na začátku v něm bývá obvykle uvedeno jméno algoritmu. Na konci pak informace o ukončení algoritmu (konec, end).

Obdélník – používá se pro nějakou operaci, která nemá žádnou speciální symboliku, například pro vyhodnocení aritmetického výrazu, přiřazení hodnoty do proměnné....

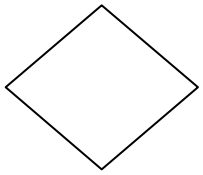
Kruh – používá se pro rozdělení vývojového diagramu na více stran, čili má funkci spojky. Na konci listu papíru ho uvedeme k poslední spojnici a na novém listu z něj vedeme první spojnici,. Zpravidla do něj zapisujeme jednoduchá označení jako písmena a čísla. Části, které na sebe navazují musí mít stejné označení.



Čtyřúhelník – je použit pro vstup dat, někdy se užívá pouze jako ruční vstup dat tzn. vstup z klávesnice, myši, světelného pera…

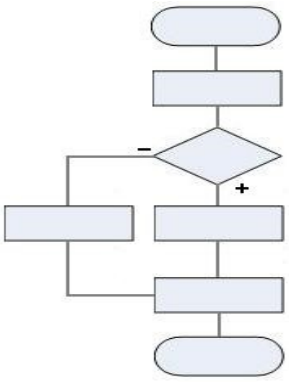
Lichoběžník – značí výstup dat, ať už na monitor, tiskárnu, či jiné výstupní zařízení…. Tento výstup je obvykle v nějakém formátu.

Kosočtverec – se používá pro podmínku, nebo-li rozhodování či větvení. Algoritmus se na základě vyhodnocení nějakého výrazu rozhodne jakou z předdefinovaných cestu bude pokračovat.

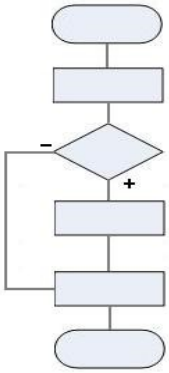


Větvení algoritmu má tři charakteristické konstrukce.

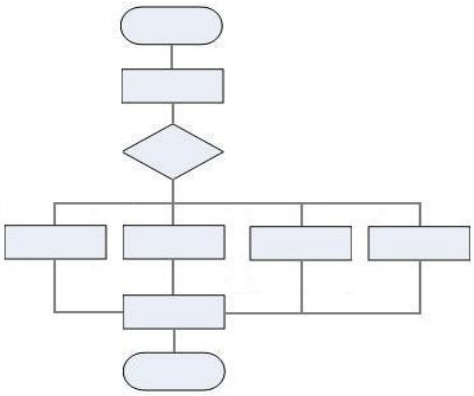
1. Úplná varianta - algoritmus se větví do dvou částí, pokud je podmínka splněna, provede se jedna část kódu, pokud ne, provede se druhá část kódu.




2. Neúplná varinta – Algoritmus se moc neliší příliš od předchozího, jen je jedna jeho větev prázdná. Znamená to, že pokud se podmínka splní, provede se zadaný kód a pak program pokračuje dalším příkazem. Pokud podmínka pravdivá není, část kód se nevykoná.



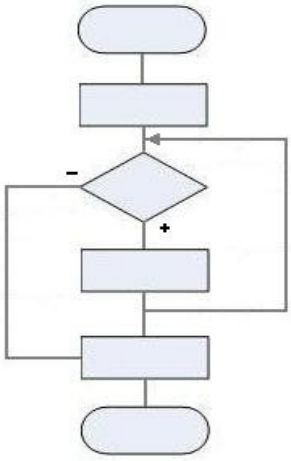

3. Několikanásobné větvení – dle vyhodnocení výrazu, tedy čemu se daný výraz bude rovnat, se provede určitý kód. Např.: Dejme tomu, že po uživateli vyžadujeme, ať zadá nějaké písmeno a podle toho, jaké písmeno zadal, provedeme to, co uživatel chce.



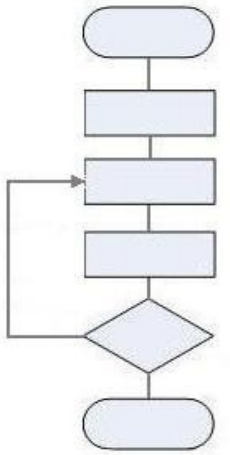
Jednotlivé druhy podmínek lze mezi sebou různě kombinovat.

Cykly – nebo-li opakování

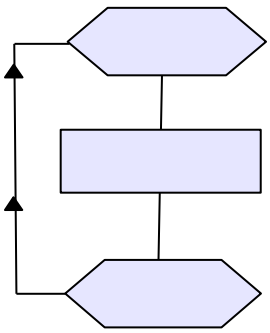
1. Cyklus se vstupní podmínkou na začátku - se bude provádět, dokud si nebude výraz v podmínce cyklu roven. Pokud si výraz bude roven ještě předtím, než se začne cyklus provádět, cyklus se neprovede vůbec.




2. Cyklus s podmínkou na konci – se provede, alespoň jednou.



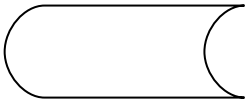
3. Cyklus se stanoveným počtem opakování – cyklus se vykoná přesně tolikrát kolikrát je stanoveno.



- počátek cyklu, nastavení poč. hodnoty proměnné
- tělo cyklu, zde se nachází zvyšování hodnoty proměnné a další příkazy
- ukončení cyklu zde se nachází maximální hodnota proměnné po jejím dosažení dojde k přerušení vykonávání cyklu

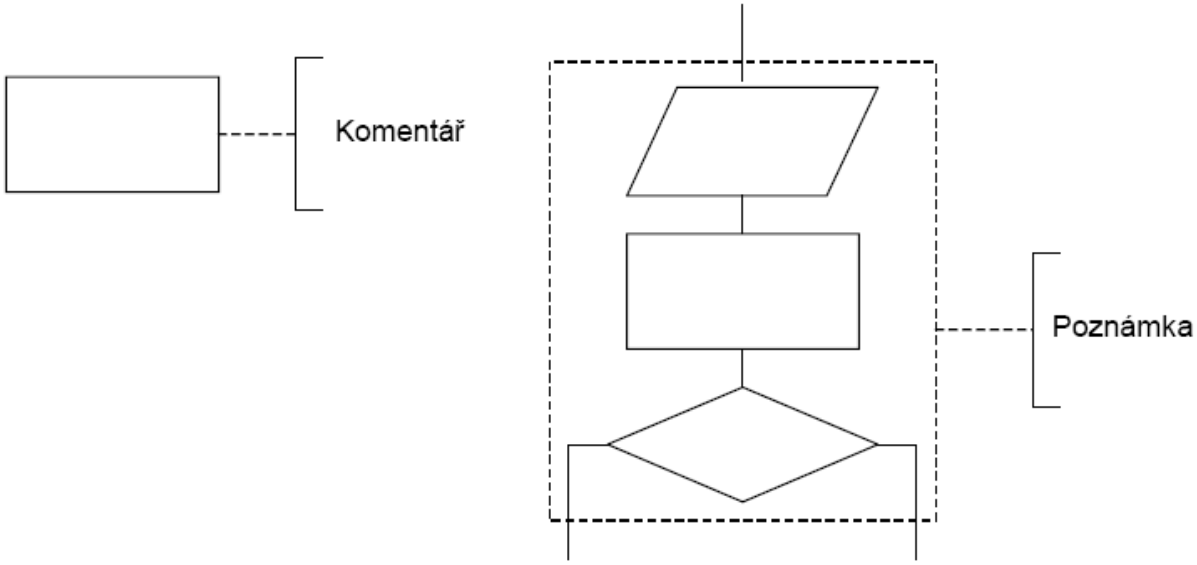
Jednotlivé druhy cyklů lze mezi sebou různě kombinovat.

„Neúplný ovál“ – tento symbol se používá pro zápis do souboru. Často se místo tohoto symbolu používá obyčejný blok příkazů (obdélník), protože se jedná o velice specifickou a důležitou funkci budeme jej značit tímto neúplným oválem.

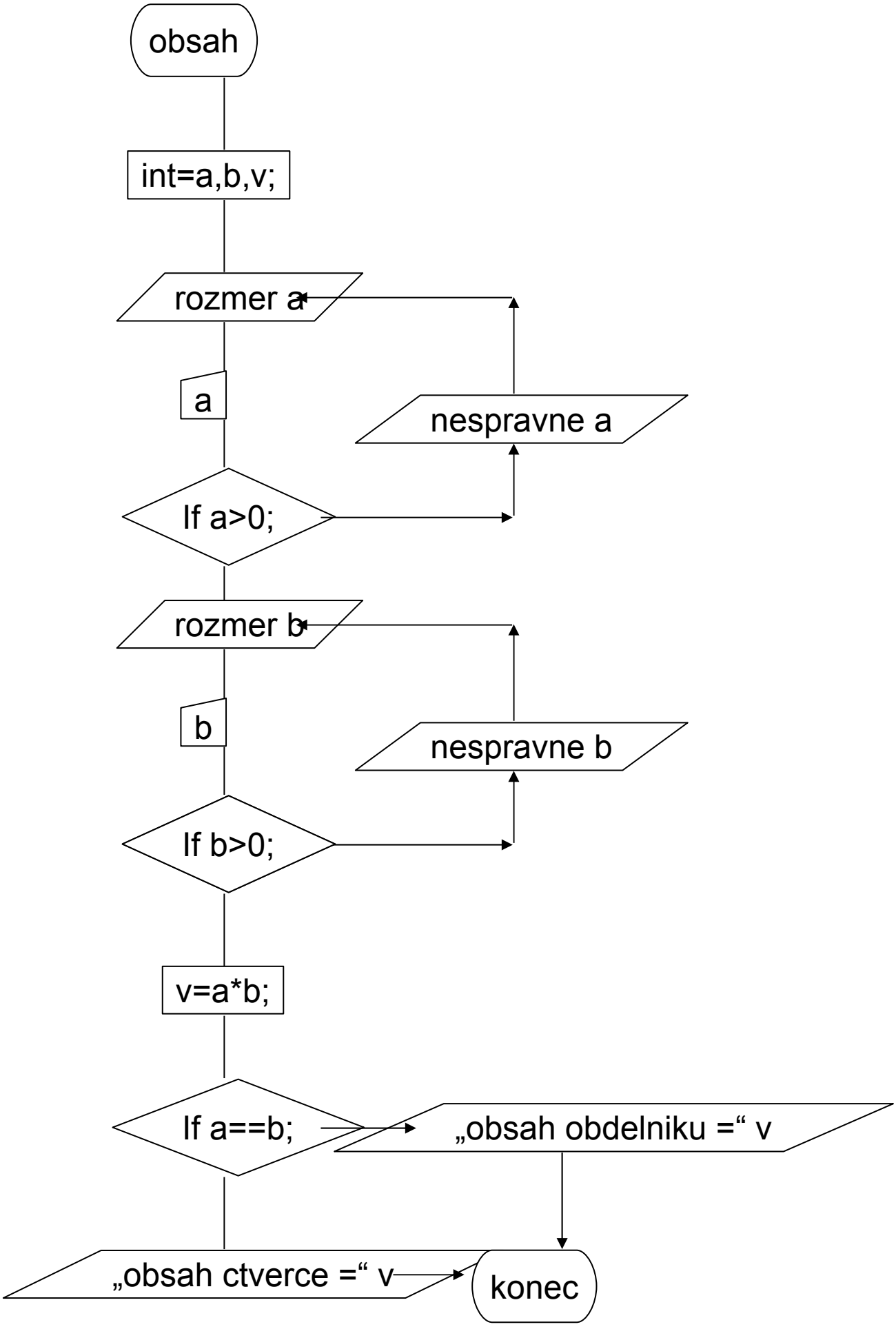


„Obdélník zakončený vlnovkou“ – tento symbol se používá pro zpracování souboru, tedy pro čtení a případnou jinou práci se souborem.

Anotace - symbol se používá k připojení popisných komentářů nebo vysvětlujících textů. Přerušovaná čára symbolu anotace je připojena k příslušnému výkonnému symbolu, nebo může být připojena k ohraničené skupině symbolů.




Na následující stránce je zobrazen vývojový diagram algoritmu, který zachycuje postup pří výpočtu obsahu čtverce nebo obdélníka.


obsah
int=a,b,v;
rozmer a
a
nespravne a
If a>0;
rozmer b
b
nespravne b
If b>0;
v=a*b;
If a==b;
„obsah obdelniku =“ v
„obsah ctverce =“ v
konec

Programovací jazyk C++

C++ je objektově orientovaný programovací jazyk, jde o rozšíření programovacího jazyka C. Podporuje procedurální, objektově orientovaný a generický styly programování. Autor jazyka C++ Bjarne Stroustrup, navrhl C++ díky svým předchozím špatným zkušenostem s programovacím jazykem Simul. K návrhu se dostal, protože chyběl programovací jazyk, který by byl objektově orientovaný, ale také zároveň dostatečně rychlý. Nejprve začal využívat jazyk C kam implementoval třídy jazyka Simul. Z tohoto vývojem vznikl programovací jazyk C++. v průběhu vývoje se odehrály následující úpravy: úprava objektově orientovaného programování, přidaní výjimek, šablon a jmenných prostorů.

První, ale však neoficiální normou se stala kniha The C++ Annotated Reference Manual z roku 1990. Organizace ANSI a ISO v průběhu následujících let vyvinuly. Standart, první schválení standartu proběhlo v roce 1998, v následujících letech byl standart několikrát aktualizován.

99% programů v jazyce C jsou programy v jazyce C++. Jinými slovy jazyk C je podskupinou jazyka C++. Jazyk C++ vznikl především převzetím jazyka C, který byl doplněn o objektovou stránku programování a některé další vlastnosti. Některé vlastnosti jazyka C byly v C++ vypuštěny. Z hlediska přenositelnosti kódu je možné rozumně napsaný program v jazyce C přeložit překladačem C++. Korektně napsaný program v C++ nepůjde přeložit překladačem C, ale jde napsat program v C++, který překladačem C přeložit půjde.

Existuje celá řada překladačů pro různé operační systémy. V systému Windows produkt od společnosti Microsoft nebo Borland, ale existuje také kvalitní volně šiřitelné překladače například Dev C++. V systému Linux se můžeme setkat s podporou v gcc.

Kompilace

V poslední kapitole před samotným programováním se seznámíme se způsobem převodu programu do spustitelné podoby v počítači. V C++ je nutné provést tzv. kompilaci, jde o proces kde se převádí náš zdrojový kód do podoby, které rozumí náš počítač tedy strojového kódu.

Kompilace je složena z třech základních kroků. Nejprve je kód předpřipraven na proces kompilace. Tuto činnost zajistí kompilátor zavoláním tzv. preprocesoru, jehož činnost konkrétně spočívá k vložení hlavičkových souborů, konstant... Následujícím krokem je samotná kompilace, při které dojde k přeložení kódu upraveného preprocesorem na kód, kterému rozumí náš procesor. Poslední fáze je fáze tzv. slinkování. Fyzicky jde připojení knihoven ke zkompilovanému kódu. V případě úspěšné kompilace je vytvořen spustitelný soubor. Pokud všechny tři fáze proběhnou v pořádku mělo by být možno program spustit.

Když kompilace neproběhne úspěšně kompilátor nám zahlásí nějaké chyby případně varování. V případě varování se nejedná o chybu, ale o jaké si upozornění na něco ne zcela standardního např. změna datového ty pu u proměnné tzv. přetypování. Varování jakožto takové není chybou a nebude-li vypsána žádná chyba, tak kód bude zkompilován. Chybou v klasickké=m slova smyslu je myšlen špatný zápis zdrojového kódu tzv. syntaktické chyba např. může jít o zapomenutí uzavření bloku, řádku, špatný identifikátor u proměnné.... Když se nám kompilace zdaří, tak nemáme vyhráno. Následuje mnohdy dlouhá fáze testování a ladění programu, kde hledáme tzv. sémantické nebo-li významové chyby např.: zapomněli jsme ošetřit dělení nulou…. Tyto chyby se mnohdy hledají velice obtížně.

Co vše budeme potřebovat ke psaní programu. Po teoretické stránce by nám stačil ASCII editor a samostatný kompilátor. Dále pak existují nadstavby, které sestavování programů značně zjednodušují např. zvýraznění syntaxe, nástroje pro ladění a krokování programu... Nejčastěji se však používají tzv. IDE (Integrated Development Enviroment - integrované vývojové prostředí). Tato prostředí v sobě spojují vlastní editor, kompilátor, debuger (nástroj pro odladění programu). Značná část IDE má podporu automatického doplňování syntaxe, generování části kódu, zvýraznění syntaxe..... všechny tyto doplňky usnadňují práci programátorům při vývoji programu. Mezi nejznámější IDE patří MS Visual Studio, Eclipse, NetBeans, Dev C++. Abychom vývojová

prostředí nepřechválili, tak mají také své nevýhody jako je někdy až značná hardwarová náročnost a někdy horší orientace ve vývojovém prostředí. Proto bych z hlediska operačního systému Windows doporučil poněkud jednodušší vývojové prostředí Dev C++. Mezi jeho výhody patří, lokalizace v českém jazyce, kterou ocení uživatelé nepříliš disponující znalostí anglického jazyka a hlavně je šířen zdarma.

Vlastní programování v jazyce C++

Úvodem musím říct, že nejprve se budeme zabývat programováním z hlediska strukturovaného stylu. Do objektového stylu programování budeme pronikat postupně, protože nejprve je nutné zvládnout strukturované programování.

První program v C++

Bývá zvykem na počátku výuky každého programovacího jazyka, napsat jednoduchý program „Hello world“. Program výpíše na monitor hlášení ahoj světe a ukončí se.

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

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

Prvním řádkem tedy „#include <iostream>“ říkáme preprocesoru, že chceme do našeho kódu vložil hlavičkový soubor iostream. Tento soubor obsahuje základní funkce-objekty jazyka C/C++, které zabezpečují práci ze vstupem a výstupem. Lze tedy odvodit, že většina programů tento řádek vždy **bude obsahovat**.

Na druhém řádku je specifikování jmenného prostoru. Překladač je informován, že je používán jmenný prostor std. Jmenným prostorům budeme věnovat samostatnou kapitolu později. V současné chvíli je nutné tento řádek na začátek programu vždy uvést. V případě vynechání je nutné před každou třídu, kterou využíváme z hlavičkových souborů, připisovat předponu **std::** Například místo cout bychom museli napsat std::cout.

Kdy jsme zmínili pojem **třída** musíme se u něj trochu zastavit a vysvětlil si ho. Třída je základním základním pojmem klasifikace. Úplně nejzákladnějším pojmem je **objekt**, který si pamatuje svůj stav a poskytuje rozhraní operacím. Pomocí tohoto **rozhraní**, které se nazývá **metoda** se s ním pracuje. Při práci s objekty nás zajímají poskytované operace objektu a ne způsob provádění operací, tento přístup nazýváme pak **princip zapouzdření**. Jednotlivé objekty jsou organizovány stromovým způsobem. Jednotlivé objekty mohou **dědit** od jiných objektů. To v praxi znamená, že přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Právě problematika dědění a stromové struktury vytváří dělení objektů do jednotlivých tříd. Objekt je **instancí** nějaké třídy a třída může dědit od jiné třídy. V tomto odstavci jsme si vysvětlili základní pojmy objektově orientovaného programování.

Další důležitá věc v programu je použití příkazu středníku, v programovacím jazyce C/C++ má funkci oddělovače jednotlivých příkazů. Jeho zapomínání je velice častá syntaktická chyba.

Řádek int main() Tento řádek najdeme v každém programu nepsaném v programovacím jazyku C/C++. Jedná se o základní funkci, které musí být v každém programu. Pokud tuto funkci nenapíšeme program nepůjde zkompilovat, protože překladač hledá funkci main a z té se dostává do některé z eventuálních funkcí. Není nutné, aby všechny funkce byly volány jen z funkce main,

ale mohou se též volat mezi sebou. Rozeberem-li detailně funkci main, tak slovo main zde znamená název funkce, funkce main musí mít návratový typ celé přirozené číslo, což určuje právě slovo int. Do prázdných kulatých závorek mohou být uvedeny tzv. vstupní proměnné, jedná se o tzv. parametry funkce. Určují jaká data do funkce vstupují. Je také nutno zdůraznit, že středník za závorkou s parametry funkce se **nepíše**, bývá to jedna ze začátečnických chyb. Obecný tvar funkce lze vyjádřit takto:

```
navratova_hodnota nazev_funkce (typ vstupni_promena1,…)
{
     prikazy – telo funkce;
}
```

Za funkcí tady za jejími parametry následují složené závorky, které lze napsat na samostatný řádek. Závorky jsou vždy párové tzn. vždy musí existovat počáteční a koncová závorka. U funkce začíná hned za kulatými závorkami s parametry a končí za posledním příkazem, který do funkce patří. V těchto závorkách se nachází posloupnost příkazů, které tvoří tělo funkce. Prováděním těchto příkazů dojde k vykonávaní z programovaného algoritmu.

Řádek „cout <<"Hello world"<<;“ zajistí vypsání pozdravu na obrazovku. Objekt cout vypisuje obecně řečeno na obrazovku text v úvozovkách a obsah proměnných, které však k objektu cout zapisujeme bez úvozovek. Více si o tomto objektu povíme v některé z dalších kapitol.

Nyní jsme v našem programu na posledním řádku, tady na „return 0;“, tento příkaz má jediný úkol předat návratovou hodnotu funkce. V tomto konkrétním případě je předávána 0, která signalizuje úspěšný průběh funkce tzn. její úspěšné ukončení. Více si o návratových hodnotách si povíme v kapitole zabývající se funkcemi.

Poslední věcí, kterou si v naší úvodní kapitole povíme jsou komentáře. Slouží k vkládání poznámek do těla programu. Existují komentáře do jednoho řádku i přes více řádků. Lze do nich Nalezne-li kompilátor symbol komentáře, tak se za něj již nedívá. Díky této vlastnosti lze do nich schovat část kódu, kterou je nám líto smazat pro její nepotřebnost. V budoucnu by se mohla v programu hodit.

Začínající programátoři, je většinou nepíší a odůvodňují to slovy „V tomhle krátkém programu se vždy vyznám“, ale opak bývá mnohdy pravdou. I kdyby se programátor v jednoduchém programu vždy vyznal, tak časem bude psát složitější a programy. Hrozí, že když se k nějakému programu třeba po roce vrátí bude dlouho pátrat nad tím co jaká část kódu dělá. Dalším důvodem pro používání komentářů je fakt, že na programech pracuje mnohdy více programátorů a při nepsání komentářů každý programátoru musí si vše složitě zjišťovat procházením kódu. Psaní komentářů patří ke správné programátorské kultuře a proto si je zvykněme psát už od těch nejjednodušších programů!

Poslední věc kterou je nutné ke komentářům říci je, že píšeme je průběžně v průběhu psaní kódu! Nyní jejich syntaxe: jednořádkový komentář je značen: // a více řádkový /*text*/. Lze pro jejich text používat češtinu, ale doporučuji nepoužívat diakritická znaménka, protože otevřeme-li tento program pro úpravu na jiné platformě nebo počítači s jiným operačním systéme hrozí, že místo písmen s diakritickými znaménky budeme mít nesmyslné symboly.

```
/* Vse co je zde zapsano bude ignorovano
* a muze to byt rozepsano i na vice radkach
* jedna se o vice radkovy komentar */
*/
```

```
/*
Vice radkovy komentar nemusi na kazdem novem radku
obsahovat hvezdicku, i toto je spravny zapis
*/
//jednoradkovy komentar
```

Identifikátory

Jde o vlastní jména proměnných, funkcí a typů. Pro psaní identifikátorů existuje několik následujících omezení.

- maximální délka identifikátoru může být 1024 znaků
- první znak v identifikátoru musí být písmeno či podtržítko _(podtržítko využívají systémové proměnné, takže se jim budeme vyhýbat)
- pro další znaky v názvu lze použít různou kombinaci písmen, čísel a podtržítek tzn. ostatní znaky v identifikátor jsou nepřípustné
- název by měl být smyslu plný a měl by se snažit vystihovat to co proměnná obsahuje
- doporučuje např. proměnnou s názvem "pomocnapromenna", kvůli čitelnosti měla zapsat jako: mazana_promenna a nebo jako mazanaPromenna, ale pozor doporučuje se vybrat jen jeden styl a nemýchat to dohromady.
- někteří programátoři si názvy proměnných volí tak, aby věděli jakého typu je. např: int i_nazev, float f_nazev; v kódu je to užitečná pomůcka, kde je na první pohled vidět datový typ proměnné
- v identifikátorech jsou rozlišována velká a malá písmena např. cislo, CISLO, Cislo jsou tři různé identifikátory, bývá to častá začátečnická chyba
- jako název identifikátoru nelze použít klíčové slovo

Klíčová slova

- klíčové slovo je v podstatě zvláštní druh identifikátoru, který má přiřazen nějaký význam z hlediska jazyka C++
- asm, auto, bool, break, case, catch, char, class, const, const.cast, continue, default, delete, do, double, dynamic.cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret.cast, return, short, signed, sizeof, static, static.cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar.t, while
- významy a použitím jednotlivých klíčových slov se budeme zabývat postupně v následujících kapitolách

Proměnná

Jde o nějaké místo respektive adresu v paměti, kde je pod nějakým jménem - identifikátorem uložena nějaká hodnota, která se může měnit. Proměnné umožňují přístup k informacím. O přidělení konkrétní adresy se stará překladač.

Každá proměnná je nějakého datového typu. Tzn. je určeno jakých hodnot může nabývat a kolik místa zabere v paměti. Proměnné, které se používají pro uchovávání čísel lze rozdělit na dvě základní skupiny celočíselné a reálné.

Celočíselné

- mohou být znaménkové tedy od záporné hodnoty po kladnou
- nebo beznaménkové od nuly po kladnou hodnotu
- k určování tohoto parametru slouží tzv. modifikátory signed pro znaménkové a unsigned pro beznaménkové
- celočíselný datový typ bez zapsaného modifikátoru je standardně signed

1) char
   - využívá se při práci se znaky
   - označovaný jako znak
   - v paměti zabírá 1B
   - lze do něj uložit hodnotu o maximální velikosti 256, proto se používá při práci se znaky – ASCII tabulka má rovněž 256 znaků
   - signed char má rozsah od -128 do +127 (ne 128 do 128, protože je nutno si pamatovat také nulu)
   - unsigned char má rozsah od 0 do 255

2) int
   - využívá se při práci s celými čísly
   - označovaný jako celé číslo
   - v paměti zabírá 4B tedy 32b někdy 2B tedy 16b
   - lze do něj uložit celé číslo o velikosti 4294967296 (4B) a 65536 (2B)
   - signed int má rozsah od -2147483648 do +2147483647 (4B) a od -32768 do +32767 (2B)
   - unsigned int má rozsah od 0 do 4294967295 (4B) a od 0 do 32767 (2B)

3) short int
   - lze zapisovat též ve zkráceném formátu short
   - využívá se při práci s celými čísly
   - označované jako krátké celé číslo
   - v paměti zabírá 2B tedy 16b
   - lze do něj uložit celé číslo o velikosti 65536
   - signed int má rozsah od -32768 do +32767
   - unsigned int má rozsah od 0 do 32767

4) long int
   - lze zapisovat též ve zkráceném formátu long
   - využívá se při práci s celými čísly

- označované jako dlouhé celé číslo
- v paměti zabírá 4B tedy 32b
- lze do něj uložit celé číslo o velikosti 4294967296
- signed int má rozsah od -2147483648 do +2147483647 unsigned int má rozsah od 0 do 4294967295
- existuje i datový typ long long int skládá se ze dvou typů long, ale příliš se nevyžívá.

Reálné

Přetečení a podtečení

Musíme si z hlediska práce z reálnými datovými typy objasnit další dva pojmy přetečení a podtečení. Přetečení nastane, pokud číslo přesáhne horní mez pro datový. Dojde-li k podtečení datového typu znamená to, že je číslo příliš malé a výsledkem bude nula tzn. číslo méně platných číslic než je minimum u daného datového typu.

1) float

- číslo ve tvaru pohyblivé řádové čárky, jednoduchá přesnost
- v paměti zabírá 4B tedy 32b
- lze do něj uložit číslo se 7 desetinnými místy
- přesnost je nejméně 6 platných číslic

2) double

- číslo ve tvaru pohyblivé řádové čárky, dvojitá přesnost
- v paměti zabírá 8B tedy 64b
- lze do něj uložit číslo se 15 desetinnými místy
- přesnost je nejméně 10 platných číslic

3) long double

- číslo ve tvaru pohyblivé řádové čárky, zvýšená přesnost
- v paměti zabírá 10B tedy 8b
- lze do něj uložit číslo se 18 desetinnými místy
- přesnost je nejméně 10 platných číslic

Velikosti datový typů se mohou na jednotlivých počítačích lišit. C++ je jazyk přenositelný na více platforem a z tohoto důvodu jsou nepřesně definovány počty bytů pro jednotlivé datové typy. Záleží zda je kód, který píšeme určený pro 8, 16 či 32 bitový systém. Existuje soubor doporučení, které je z hlediska dobré kultury programování dodržovat, třeba velikost int nějakém systému má být přirozenou velikost celých čísel v daném systému tzn. 32 bitový systém má přirozenou velikost celých čísel 32 bitů.

V jednotlivých systémech musí být dodrženy tyto pravidla:

short <= int <= long

float <= double <= long double

char vyžaduje 8 bitů

Jak zjistit velikosti datových typů? Lze se podívat do hlavičkových souborů limits.h nebo float.h. V jazyce C++ lze použít operátoru sizeof, který nám vrátí velikost proměnné a tedy i datového typu. Předpokládejme, že existuje v paměti počítače proměnná a typu int.

```
sizeof(zjistovany_objekt)
```

```
sizeof(a)
```

Uvedený výraz vrací počet bytů nutných k uložení hodnoty proměnné a, tj. velikost datového typu int.

Deklarace a definice

Již jsme si řekli, že chceme-li pracovat s proměnou nebo s funkcí tak musím k ní stanovit nějaké jméno tedy její identifikátor. Teď si musíme objasnit pojem deklarace a definice, protože bývají dost často zaměňovány a přitom každý z nich představuje rozdílnou věc. Deklarace znamená existenci proměnné či funkce a určujeme typ objektu. Definice proměnné či funkce a přiděluje ji paměť, jde vlastně o přiřazení hodnoty do proměnné v případě funkce je to sled jednotlivých příkazů ve funkci.

Při práci se základními proměnnými je proměnná deklarována a definována zároveň a nemusí mýt hned při svém vzniku přiřazenu nějakou hodnotu. Podrobněji se rozdílem mezi deklarací a definicí budeme zabývat v kapitole o funkcích a o dynamicky alokované paměti.

Poslední pojem, který si vysvětlíme než ukážeme syntaxi deklarace a definici proměnných je inicializace. Inicializace znamená přiřazení hodnoty do proměnné. Dobrým programátorským zvykem bývá při stanovení proměnná do ní přiřadit nulu, protože nevíme jaká náhodná hodnota např. pozůstatek po jiném výpočtu nachází v paměti. Při nevynulování mohou ve výpočtech vznikat nesmyslné hodnoty.

```
datový_typ nazev_promene = inicializace;
```

```
int hodnota = 19; //deklarace a definice zároveň
int hodnota2; //deklarace a definice proměnné bez inicializace
hodnota2 = 19; //inicializace
```

Poznámka chceme-li více proměnných od jednoho datového typu nemusí být každá na vlastním řádku. Datový typ napíšeme pouze jednou a jednotlivé identifikátory s případnou inicializací oddělujeme čárkou.

```
int hodnotaA=0, hodnotaB=0;
```

Přetypování - konverze datových typů

Již jsme si vysvětlili základní datové typy, řekli jak vytvořit proměnou v paměti počítače a umíme ji přiřadit hodnotu. V průběhu psaní programu můžeme zjisti, že od nějakého místa v kódu potřebujeme změnit datový typ proměnné např. zjistíme že potřebuji reálné (double) číslo a mám proměnou celého čísla (int). Z těchto důvodů existuje přetypování.

Přetypování může být automatické nebo vynucené. O automatickém mluvíme, když do proměnné int uložíme hodnotu double.

```
int promenna = 0;
double realne_cislo = 3.78;
promenna = realne_cislo;
cout << promena; //vytiskne na monitor hodnotu 4
```

Jistě by mnozí očekávali, že na monitoru bude zobrazeno číslo 4, že prostě dojde k zaokrouhlení hodnoty. C++ provádí ve skutečnosti převod double na int oříznutím desetinných míst. Mylná informace o zaokrouhlování či neuvědomění si skutečnosti o oříznutí desetinných míst vede dost často k sémantickým chybám, tedy ke špatné činnosti programu. Převod opačným směrem ve většině případů nečiní žádné problémy, lze zcela běžně přiřadit do proměnné typu double či float celočíselnou hodnotu.

Jeden problém akorát nastane, když budeme dělit dvě celá čísla a ukládat výsledek do reálné proměnné. viz. následující příklad.

```
double vysledek = 0;
vysledek = 2/5;
cout << vysledek; //bude vypsana hodnota 0 a ne 0,4
```

Dojde opět k odseknutí desetinných míst. Po lepší pochopení co se děje při vyhodnocení si nyní vše podrobně popíšeme. V prvním řádku se deklaruje proměnná vysledek typu double, zde je vše v pořádku. Na druhém řádku se nejprve provede dělení dvou celých čísel, protože programovací jazyk C++ přiřazuje zprava doleva, tzn. dělí se dvě celočíselná čísla a ty nám zase vrátí celočíselný výsledek a ten se pak přiřadí do proměnné vysledek.

Jak se tomuto vyhnout? Existuje více variant, ale nejjednodušší je snad to, že jedno z čísel musí být reálné.

```
double vysledek = 0;
vysledek = 2.0/5;
cout << vysledek; //bude vypsana hodnota uz spravna 0,4
```

Nyní si ukážeme další řešení, které bude pracovat s čísly uloženými v proměnných.

```
double vysledek = 0;
int a = 2;
int b = 5;
vysledek = a/b;
cout << vysledek; //bude vypsana hodnota 0 a ne 0,4
```

Opět bychom mohli jedno číslo v proměnné zapsat jako reálné, či ho vynásobit číslem 1.0 a výsledek by byl pak reprezentován jako desetinné číslo. Lepší metodou je nespoléhat na automatické přetypování, ale vynutit si ho.

Vynucené přetypování se provádí uzavřením přetypovávané hodnoty do kulatých závorek a před ně napíšeme datový typ na který se má převádět.

```
datovy_typ(pretypovavana_promenna); //obecny zapi
```

```
double vysledek = 0;
int a = 2;
int b = 5;
vysledek = double(a)/b;
cout << vysledek; //bude vypsana hodnota uz spravna 0,4
```

Tento způsob přetypování je pro C++ typický, ale lze použít i přetypování z jazyka C která byl C++ s děděn.

```
(datovy_typ) pretypovavana_promena;
```

```
double vysledek = 0;
int a = 2;
int b = 5;
vysledek = (double)a/b;
cout << vysledek; //bude vypsana hodnota uz spravna 0,4
```

Tyto způsoby převodů, které jsme si vysvětlili lze použít i na ostatní datové typy. Poměnou typu char lze převést na int.

```
char znak;
1*znak;
int(znak);
(int)znak;
```

Nedivte se násobení čísla a písmen mělo by vám již být známo, že v počítači jsou veškerá písmena vyjádřena pomocí kombinace osmice bitů Osmice bitů nám dává 256 možností tzn. můžeme uložit 256 rozdílných symbolů. Symboly jsou jsou uspořádány do tzv. ASCII tabulky a každý z nich má své pořadové číslo - svůj kód. Na char se lze dívat jako na zvláštní druh celočíselné proměnné, která své číselné hodnoty reprezentuje jako čísla podle ascii tabulky. Např. písmenu a odpovídá hodnota 97.

Speciální datové typy

1) void
   - datový typ, který nemůže nebývat žádné hodnoty tzn. nic nereprezentuje
   - používá se jako návratová hodnota funkcí u funkcích, které nevrací žádnou hodnotu
   - použití si ukážeme v kapitole, která se bude zabývat funkcemi
2) bool
   - tento datový typ se označuje někdy jako „logický“. Z tohoto označení lze

již vyvodit jeho použití, slouží k uchování logické informace pravda(1) označovaná v terminologii jazyka C++ jako true nebo nepravda značená jako false.

- V paměti tento datový typ ačkoliv může představovat jen dvě hodnoty zabírá 1B tedy 8b. První dojem říká, že by měl zabírat jen 1b, ale to nezní možné jelikož nejmenší část paměti na, která má svojí adresu je 1B.

3) wchar_t

- unicode znak
- používá se pro kódování češtiny
- zabírá v paměti 2B tedy 16b
- Pro deklarovaní např. konstanty jako znaku či řetězce v Unicode je nutné ji zapsat s prefixem L – např. L'a' L"Svět".

```
wchar_t wstr[20] = L"text textovy";
```

4) enum

- výčtový datový typ
- využívá se pro vyjádření hodnot ze seznamu
- hodnoty se přiřazují pomocí celočíselných konstant
- zabírá v paměti 4B tedy 32b

```
enum [jmenovka]
   {
   položka1[=hodnota1],položka2[=hodnota2],...
   } [promenna1, promenna2,...];
```

- Jmenovka je v podstatě nepovinná. Je-li uvedena jde v podstatě se o deklaraci výčtového typu, jehož jméno bude jmenovka. Není-li jmenovka uvedena, definujeme touto deklarací pouze skupinu pojmenovaných konstant.
- Položka je to povinný údaj protože jde o identifikátor hodnoty výčtového typu.
- Hodnota je nepovinný údaj, který je představuje jemu odpovídající položka. Pokud ji uvedu tak se musí jednat o celočíselný konstantní výraz. Při jejím vynechání u některé položky, přidělí ji překladač hodnotu o 1 větší než u předchozí hodnoty. Není-li uvedena hodnota u první položky překladač jí přidělí hodnotu 0. Různé položky mohou mít stejné hodnoty.
- Proměnná je to opět nepovinná položka, jde o definici proměných případně ukazatelů....Můžeme ji definovat i později.
- Nyní si ukážeme fragment kód funkčního programu, kde si ve výpisu je nejprve deklarujeme výčtový datový typ Dny. Na konci deklarace bude

vytvořena instance (proměnná) typu Dny se jménem dnes. Ukážeme si správné a špatné definice proměnných viz. následující příklad.

```
enum Dny          //deklarace datového typu enum Dny
{
      pondeli,       //pondeli = 0
      utery = 1,     //utery = 1
      streda,        //streda = 2 (utery + 1)
      ctvrtek,       //ctvrtek = 3
      patek,         //patek = 4
      sobota,        //sobota = 5
      nedele = 5     //nedele = 5
} dnes;           //definice proměnné  dnes typu Dny

int  pondeli;    //chyba! redefinice
enum Dny vcera; //možné použití v C a C++
Dny zitra;       //možné použití pouze v C++

vcera = pondeli;

int i = utery;  //možné použití, i = 2
vcera = 0;       //chyba, není možná konverze
vcera = (Dny)0; //možné použití, ale výsledek nebude definován
```

5) pointer

- nebo-li ukazatel
- Zabírá v paměti 4B tedy 32b
- Umožňuje ovlivňovat dění v paměti počítače
- Pomocí identifikátoru pracujeme přímo vždy s konkrétní proměnou. Ukazatel nám s proměnou zabezpečuje práci tzv. nepřímo. Ukazatel ví od své deklarace na jaký datový typ bude ukazovat. Adresu proměnné kde je uložena hodnota na kterou ukazuje, získá až za běhu programu. V adrese identifikátoru je tedy uložena adresa proměnné na, kterou ukazuje.
- Nejčastější chybou bývá použití ukazatele, ještě před získáním platné adresy na kterou lze odkazovat. Jedná se pak o neinicializovaný ukazatel, protože ukazuje na nějaké neznámé místo v paměti počítače. Tato chyb může vést i k havárii nejen dané aplikace ale i operačního systému.
- Ukazatel vytváříme pomocí: *
- Je nepsaným pravidlem, že identifikátor ukazatele začíná na písmeno p (z

anglického pointer). Díky tomuto na první pohled odlišíme, že jde o ukazatel.

```
datovy_typ *pindetifikator;
```

- Adresu do ukazatele přiřadíme pomocí: &

&promenna = pidentifikator;

```
int a, *pa; //vyvtvoreni promene „a“ a idetifikatoru „pa“
pa = &a; //spravne ziskani adresy
*pa = 512; //prirazeni hodnoty do promenne a pomoci identifikatoru pa
```

Konstanty

Konstanta na rozdíl od proměnné má stálou ne měnou hodnotu. Její typické využití je např. při výpočtech s nějakým neměnným koeficientem. Definujeme si ho jako konstantu a v programu při práci s hodnotou používáme jen identifikátor konstanty. Proč? V průběhu realizace programu můžeme zjisti, že koeficient byl stanoven chybně a v případě nedefinování této hodnoty jako konstanty bychom pracně museli procházet celý kód a hodnotu jednu po druhé měnit a hrozilo by zapomenutí na nějakou hodnotu. Při definované konstantě jen pohodlně změníme jednu hodnotu.

Způsob zápisu konstanty spočívá v použití klíčového slova const, po něm následuje datový typ s identifikátorem a nějaké hodnoty. Končíme řádek opět středníkem. Toto označujeme jako konstantní výraz. Překladač při překladu vyhodnotí konstantu  a do kódu pak přímo vkládá její hodnotu. Překladač konstantu označí konastou pouze když nebude obsahovat: přiřazení, inkrementaci nebo dekrementaci (i++, i--), funkční volání či čárku.

Poznámka v případě nějakého pole hodnot následuje vektor s hranatými závorkami, kde je počet jeho dimenzí. První prvek pole je indexován od 0.

Po teoretické stránce vytvoříme konstantu takto:

```
const datovy_typ indefikator = hodnota;
```

V případě pole bude teoreticky syntaxe vypadat takto:

```
const datovy_typ indefikator[pocet_dimennzi] = {prvek1, prvek2,...};
```

Nyní si uveďme ukážeme program, kde si definujeme různé typy konstant.

```
# include <iostream>
# include <string>
using namespace std;
const bool pravda = true;
const int konstanta = 7;
const int celociselna = -512 ;
const double realna = 6.6256e-34;
```

```
const char male_a = 'a' ;
const string retezec = "Konstantni retezec." ;
const float meze[2] = {-20, 60} ;
const char rimska_znk[]={'I', 'V', 'X', 'L', 'C', 'D', 'M'} ;
const int arabska_hodn[]={1, 5, 10, 50, 100, 500, 1000} ;

void main () {
     cout << pravda << endl
     << konstanta << endl
     << celociselna << endl
     << realna << endl
     << male_a << endl
     << retezec << endl ;
     for (int i = 0;
     i < sizeof(arabska_hodn) / sizeof(int); i++)
     cout << "i = " << i
     << ", rimsky : " << rimska_znk[i]
     << ", arabsky " << arabska_hodn[i]
     << endl;
     }
```

Program vypíše:

1
7
-512
6.6256e-034
a
Konstantni retezec.
i=0, rimsky:I, arabsky 1
i=1, rimsky:V, arabsky 5
i=2, rimsky:X, arabsky 10
i=3, rimsky:L, arabsky 50
i=4, rimsky:C, arabsky 100

Celočíselné konstanty

Jde o zápis celého čísla ať už kladného nebo záporného, které lze zapsat v klasické desítkové soustavě. U záporného čísla před něj bez mezery zapíšeme mínus. V C++je možné pracovat osmičkovou soustavou, konstantu uvozujeme 0. Další podporovanou soustavou je šestnáctková kdy číslice uvozujeme symboly 0X nebo 0x, písmena pro označení hodnot 10-16 mohou být malá i velká. Samozřejmě je zde podporována desítková soustava.

```
const int soustava_10 = 455;
const int soustava_8 = 0707;
const int soustava_16 = 0x1C7;
```

Poznámka jiné číselné soustavy se dají použít rovněž u proměnných. Existují dva modifikátory, které se píší za hodnotu bez mezery. Prvním je U, které značí že jde o nezápornou hodnotu (unusigned). Druhé je L značí, že jde datový typ long.

Racionální – reálné konstanty

V paměti počítače se ukládají pomocí mantisy a exponentu. Např. 3.58e5 je zápisem racionálního čísla 358000. Výchozím datovým typem pro racionální konstantu je double. V případě, že chceme konstantu datového typu long napíšeme za konstantu modifikátor L. Počty platných desetinných míst jsou stejné jako u proměnných tedy pro float 7, double 15 a long double 19.

U reálných čísel respektive konstant (float, double) můžeme číslo zadávat buď jako desetinné číslo (3.45, 2.34, ...). nebo v exponenciálním tvaru (3e5, 2e-4)

```
const float obsah = 3.56;
const float objem = 6.2e2;
```

Poznámka zápis v exponenciálním tvaru lze také rovněž užít i při práci s proměnnými. Jazyk C++ užívá desetinou tečku!!!

Znakové konstanty

Jde o zápis jednotlivých znaků, který provádíme zápisem znaku mezi apostrofy. Sami jistě tušíte, že budeme-li jako znakovou konstantu chtít apostrof nebo jiný speciální znak třeba odstránkování, přechod na novou řádku, tak nepůjde symbol vyjádřit jen zapsáním do apostrofů. Pro některé znaky ani není tisknutelný znak, kterým bychom je mohli reprezentovat. Tyto znaky vyjadřujeme pomocí tzv. escape sekvencí. Escape sekvence je zpětné lomítko a nějaký znak či symbol napsaný opět v apostrofech. Jednotlivé escape sekvence a jejich význam jsou uvedeny v následující tabulce. Chceme-li použít české symboly musíme užít datový typ wchar_t a před znakovou konstantou užít modifikátoru L.

```
'a' \\znaková konstanta a
'\a' \\pípnutí
```

| **posloupnost** | **jméno** | **význam** |
|---|---|---|
| \a | Alert (Bell) | pípnutí |
| \b | Backspace | návrat o jeden znak |
| \f | Formfeed | odstránkování |
| \n | Newline | na začátek nového řádku |
| \r | Carriage return | na začátek aktuálního řádku |
| \t | Horizontal tab | na další tabelační pozici |
| \v | Vertical tab | stanovený přesun dolů |
| \\ | Backslash | zpětné lomítko |
| \' | Single quote | apostrof |
| \“ | Double quote | uvozovky |
| \? | Question mark | otazník |

| | | |
|---|---|---|
| \000 | | znak zadaný osmičkově |
| \xHH | | znak zadaný šestnáctkově |

Konstantní řetezce

Řetězec jde posloupnost (pole) jednoho či více znaků. Kterou zapisujeme do úvozovek. Pozor nezaměňovat 'a' znakovou konstantu a konstantní řetězec s jedním symbolem "a", jde o dvě rozdílné věci! Chceme-li v konstantních řetězcích použít speciální symboly zapisujeme je obdobně jako o znakových konstant pomocí escape sekvencí. Chceme-li použít české symboly v konstantním řetězci musíme užít datový typ wchar_t a před znakovou konstantou užít modifikátoru L.

```
"dve slova"
"a"
"Ahoj!"
"Cela tato veta tvori jeden retezec."
"Cela tato veta tvori jeden retezec.\n"
L"Můžeme psát s diakritikou"
```

Jelikož, C++ je objektově orientovaný programovací jazyk tak si představme řetězec jako objekt. Trochu si nyní o tomhle pohledu povíme. Je nutné říct, že řetězce budeme povídat ve velké většině všech programů, které budeme vytvářet. Z předchozího odstavce a příkladu víme, že lze používat jednoduché řetězce, které byly přvzaty z programovacího jazyka C. V C++ lze pracovat s řetězci efektivněji v knihovnách najdeme třídu string. Tato třída nabízí mnohem lepší práci s řetězci. Viz. následující příklad.

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

void main () {
     string pocatek = "Bylo odstartovano";
     string datum = "15.12.2010", vypis;

     vypis = pocatek + ' ' + datum + '.'

     cout << vypis << endl

     return 0;

     }
```

Jak je vidět z příkladu s řetězci v jazyce C++ můžeme pracovat jako s klasickými výrazy. V starém jazyce C se používali složitější funkce a příkazy. Těmi se však s časových důvodů nebudeme zabývat.

Preprocesor pod lupou z hlediska uživatele

Již jsme si řekli, že samotné kompilaci kód předchází zpracování kód preprocesorem. Je nutné si však uvědomit, že výstupem z preprocesoru je opět text a do samotného strojového kódu ho převede až samotné kompilace! Preprocesor neprovádí kontrolu syntaxe, tedy správnost zápisu z hlediska pravidel jazyka C++. Činnost preprocesoru spočívá tedy konkrétně ve vložení volaných hlavičkových souborů do kódu, odstranění komentářů, rozvinutí maker a hlavně provádění

podmíněného překladu. Tzn. přeložení jen nějaké části kódu na základě podmínek.

Preprocesor pracuje s tzv. direktivami. Je nutné zdůraznit, že se nejedná o příkazy jazyka C++ a neukončujeme je tedy středníkem. Direktivu musí vždy uvozovat „křížek“ #. Existují následující direktivy: #define, #error, #include, #elif, #if, #line, #else, #ifdef, #pragma, #endif, #ifndef, #undef.

Makra

Jelikož C++ je podmnožinou jazyka C, tak v něm lze pracovat i makry z jazyka C. Doporučuje se však používat konstanty, funkce, případně šablony funkcí. Konstanty jsme si již vysvětlili a funkce s jejich šablonami si vysvětlíme v některé z následujících kapitol. Proč používat konstanty? Překladač u nich na rozdíl od maker může provádět typovou kontrolu. Ukážeme si však definování a oddefinování makra.

```
#define makro_jm[posloupnost příkazů]
#undef makro_jm
```

Podmíněný překlad

Během činnosti preprocesoru může dojít k vyhodnocení, je-li nějaké podmínka splněna či nikoliv a dle výsledků splnění či nesplnění této podmínky lze přeložit jen vybranou část kódu nebo nějaká část kódu muže být odfiltrována. Podmínky podmíněného překladu jsou si velice podobné s podmínkami v jazyce C++, ale jejich zpracování provádí preporcesor.

Chceme-li použít rozsáhlejších logických výrazů musíme použít argumentu preprocesoru defined, které nemusíme uzavírat do závorek, ale lze ho užít jen za #if nebo #elif. Proto si teď uveďme malý příklad.

```
#if defined LIMIT && defined MALA && VELKA==1
```

Teď jsme si ukázali jak by vypadalo složitějšího výrazu u podmíněného příkladu. Jak vše v praxi užit? Podívejme se do následujícího příkladu.

```
#if test [
     kod]
[#else
     kod2]
#endif
```

Nyní si příklad vysvětlíme. Části kódu označené jako test představují porovnání hodnot resp. Vyhodnocení logického výrazu na úrovni preprocesoru. Části označené jako kód označují vlastní program v jazyce C++, respektive jeho části. Kód jazyka C++ musí začínat na samostatném řádku než příkazy preprocesoru.

Místo #if lze používat i #ifdef nebo #ifndef. Nesmíme opomenout určit, kde jaká podmíněná část kódu končí tzn. je nutné nezapomenout na příslušné místo zapsat direktivu #endif nebo #elif.

Direktiva #include

Tato direktiva bude existovat v každém programu napsaném v jazyce C nebo C++. Zabezpečuje nám vložení nějakého zdrojového kódu z jiného souboru. Nejčastěji se používá pro již předdefinovené objekty z hlavičkových souborů. Existují 4 druhy zápisu této direktivy.

```
#include <hlavička>
```

Používá se pro standardní hlavičkové soubory jazyka C++, chceme-li načíst hlavičkový soubor jazyka C jméno hlavičkového souboru zapíšeme s příponou .h. Umístění těchto hlavičkových souborů je definováno v překladači.

```
#include <hlavičkový_soubor>
```

Tento druhý způsob je velice podobný předchozímu, vlastně jde o stejný způsob zápisu jen je prohledávám adresář include, kde jsou umístěny standardní hlavičkové soubory. Pokud není soubor nalezen je ohlášena chyba.

```
#include "hlavičkový_soubor"
```

Jméno hlavičkového souboru zapisujeme do uvozovek jedná-li se zpravidla o námi vytvořený hlavičkový soubor, který je uložen v pracovním adresáři. Není-li nalezen je opět ohlášena chyba.

```
#include jméno_makro
```

Je nahrazen expandovaným makrem. Pokud není-li makro nalezeno je opět hlášena chyba.

Direktiva #pragma

Jde o zvláštní typ direktivy, jejíž „úkol“ spočívá v ignorování varování překladače, při nepodporování nějakého druhu direktivy.

S ostatními direktivami se zmíníme v průběhu této knihy, protože jejich potenciál bychom nevyužili.

Vstup a výstup z programu jazyka C++

Ještě než se v další kapitole budeme zabývat prací s výrazy, tak bychom si měly trochu víc popsat vstup a výstup z programu v jazyce C++. Základní vstup do programu nám zajistí objekt cin. Tento objekt najdeme v základním hlavičkovém souboru respektive třídě iostream. Syntaxe objektu cin je následující:

```
cin >> proměnná >> proměnná_2;
```

Pro shrnutí hodnoty, které chceme zadat přiřazujeme do již deklarovaných proměnných. Jednotlivé zadávané proměnné udělujeme „špičatými“ závorkami. V jakém pořadí je zapíšeme v takovém je při běhu programu zadáváme. Za poslední zadávanou proměnnou nejsou závorky a příkaz je ukončen oddělovačem tedy středníkem. Při běžícím programu zadání každé hodnoty potvrdíme entrem.

Nyní si popíšeme jednoduchý výstup z jazyka C++ objekt cout. Tento objekt opět patří do základní třídy iostream. Můžeme jím na monitor vytisknout obsah proměnné, konstanty tzn. můžeme zobrazit číslo ať už reálné nebo kladné, znakovou konstantu nebo řetězec. Jeho syntaxe bude následující:

```
cout << "tisknu řetězec" << ' ' << proměnná << endl;
```

Teď si opět výstup opět shrneme. Jednotlivé zobrazované údaje jsou v příkazu oddělené „špičatými“ závorkami tentokrát obrácenými, což je logické. Jednotlivé zobrazované údaje mohou

být napsány v libovolné kombinaci. V našem příkladu se nám nejprve vypíše řetezec: tisknu řetězec, poté se vloží mezera pomocí znakové konstanty a pak se vypíše obsah proměnné proměnná. Nakonec je odřádkováno pomocí příkazu endl. Endl je dobré používat téměř za každým výpisem, protože jinak nám program bude vše vypisovat do jednoho řádku a odřádkuje až zaplní řádek, výstup by pak byl nepřehledný. Řádek lépe řečeno příkaz je zakončen opět středníkem.

Do objektu cout, lze zapsat výraz bez přiřazení do proměnné. Ten se vyhodnotí a vypíše.

```
cout << proměnná*2;
```

Poslední věc, kterou si zatím v rámci vstupu a výstupu z programu ukážeme bude objekt setw. Tento objekt patří do třídy respektive hlavičkového souboru iomanip. V této třídě nalezneme objekty, které zajišťují formátování výstupu.

Objekt setw určí šířku vypisované hodnoty, tedy kolik znaků je na monitoru vyhrazeno pro vypsání obsahu proměnné. Můžeme tím hodnoty vypisovat do několika sloupců. Teď k samotné syntaxi, setw zapisujeme semostatně do špičatých závorek k němu do kulatých závorek píšeme číslo kolik znaků je pro vypsání vyhrazeno. Tímto formátovacím objektem bude ovlivněna vypisovaná hodnota v následujících špičatých závorkách.

```
cout <<proměnná<<setw(počet_míst)<< proměnná_2<< endl;
```

Nyní si ukážeme praktický příklad, kde zadáme hodnotu do proměnné i tu vypíšeme třikrát. Jednou takovou jakou jsme jí zadali, po druhé její dvojnásobek a po třetí její druhou mocninu. Celý program se vypíše pomocí setw do dvou sloupců první bude mít k dispozici 16 znaků a druhý 4.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
     int i;
     cout << "Zadejte hodnotu: ";
     cin >> i;
     cout << setw(16) <<"hodnota:"<< setw(4) << i << endl;
     cout << setw(16) <<"dvojnasob:"<< setw(4) << i*2 << endl;
     cout << setw(16) <<"na druhou:"<< setw(4) << i*i << endl;
     return 0;
}
```

Takže výstup z programu bude vypadat takto (zadali jsme jako hodnotu číslo 3):

```
Zadejte hodnotu: 3
        hodnota:   3
      dvojnasob:   6
      na druhou:   9
```

Jistě jste si při spuštění programu všimli, že program provedl svojí činnost a okno kam vypsal výsledek se hned zavřelo. Co s tím? Máme několik možností a proto začneme tou méně elegantní, kterou bych Vám nedoporučoval. Kód programu nebudeme vůbec nijak upravovat a otevřeme jsi nabídku start, vybereme volbu spustit a zadáme cmd. Spustí se nám interpret příkazového řádku. V tomto okně zapíšeme cestu k zkompilovanému programu a potvrdíme enterem. Vidíte, že tato volba je moc zdlouhavá a neustále přepínání při testování programu mezi prostředím jazyka C++ a příkazovým řádkem je přinejmenším neefektivní. Pokud program bude určen pro laického uživatele, těžko toto po něm můžeme chtít.

Druhou možností je použití objektu - funkce system() a do závorek vepíšeme v uvozovkách

PAUSE. Je to poměrně násilná možnost ukončení programu, protože pro zastavení běžícího programu je založen celý nový proces, který zavolá systémovou funkci pause. Dále je potřeba říct, že toto bude fungovat pouze pod operačním systémem Windows.

A poslední třetí možností, kterou si ukážeme je využití objektu cin, kde se bude čekat na zadání enteru. Jediná menší nevýhoda je, že překladače někdy tento řádek ignorují a proto ho napíšeme dvakrát pod sebe. Konkrétně příkaz bude vypadat takto: cin.get();.

Existují také další způsoby, ale ty si uvádět nebudeme. Doporučuji užívat pro zastavení běhu programu objekt cin. Poznámka jak system("PAUSE"); tak cin.get(); se píší na konec funkce main před příkaz return. Nyní si ukážeme nějaké praktické příklady.

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

      int main(){
            cout <<"Hello world"<<;
            system ("PAUSE")
            return 0;
      }
```

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

      int main(){
            cout <<"Hello world"<<;
            cin.get();
            cin.get();
            return 0;
      }
```

Operátory

Jde o symbol, který může být tvořen jedním či více znaky, tyto znaky představují nějakou operaci. Jsou velice významným prvkem v jazyce C++, protože bez nich bychom nemohli ani přiřadit hodnotu do proměnné.

Jak už bylo řečeno operátory jsou nepostradatelnou součástí programovacího jazyka C++ a nejen jeho, ale používají se v programování všeobecně. B jednotlivých programovacích jazycích se mohou lehce lišit. Pomocí operátorů se vytvářejí tzv. výrazy. Výraz se skládá z operátoru a z operandů. Operátory jsme si již vysvětlili a nyní si vysvětlíme operandy. Operand může představovat nějakou konstantu nebo identifikátor tzn. objekt, řetězec nebo volání funkce. Např.:

```
a + b + 2
```

Z hlediska normy jazyka C++ má každý operátor stanovenou svojí prioritu (významnost) z hlediska vyhodnocování, podobně jako je tomu v matematice. Dále u operátoru rozlišujeme směr vyhodnocování tedy zprava doleva nebo zleva doprava tzn. jaký směrem se bude probíhat „výpočet". Poslední pro nás důležitým faktorem je tzv. l-hodnota. L-hodnota říká zda-li je možné výsledek nějakého výrazu použít nalevo od přiřazení. Nyní si jednotlivé operátory seřadíme do tabulky od největší priority po nejnižší.

| priorita | operátor | popis | směr | l-hodnota |
|---|---|---|---|---|

| | | | vyhodnoc. | |
|---|---|---|---|---|
| 1 | ( ) | volání funkce | → | ano |
| 1 | [ ] | indexování | → | ano |
| 1 | . | přístup k položkám | → | ano |
| 1 | -> | přístup k položkám přes ukazatele | → | ano |
| 1 | :: | rozlišení platnosti | → | ano |
| 2 | ! ~ | logická a bitová negace | ← | ne |
| 2 | ++ -- | inkrementace a dekrementace | ← | ne |
| 2 | + - | unární plus a mínus | ← | ne |
| 2 | (typ) | přetypování | ← | ne |
| 2 | * | dereference ukazatele | ← | ne |
| 2 | & | získání adresy | ← | ne |
| 2 | sizeof | velikost v bytech | ← | ne |
| 2 | new | alokace paměti | ← | ne |
| 2 | delete | uvolnění paměti | ← | ne |
| 2 | const_cast | přetypování | ← | ano |
| 2 | static_cast | přetypování | ← | ano |
| 2 | dynamic_cast | přetypování | ← | ano |
| 2 | reinterpret_cast | přetypování | ← | ano |
| 2 | typeid | identifikace typu | ← | ne |
| 3 | .* ->* | deref. třídních ukazatelů | → | ano |
| 4 | * / % | násobení, dělení, modulo | → | ne |
| 5 | + - | sčítání, odečítání | → | ne |
| 6 | << >> | bitový posun vlevo, vpravo | → | ne |
| 7 | < > <= >= | menší než, větší než, menší nebo rovno, větší nebo rovno | → | ne |
| 8 | = != | rovno, nerovno | → | ne |
| 9 | & | bitová konjunkce | → | ne |
| 10 | ^ | bitová nonekvivalence | → | ne |
| 11 | \| | bitová disjunkce | → | ne |
| 12 | && | logická konjunkce | → | ne |
| 13 | \|\| | logická disjunkce | → | ne |
| 14 | ?: | podmíněný výraz | ← | ano |
| 15 | = | přiřazení | ← | ano |
| 15 | += −= *= /= %= | složená přiřazení | ← | ano |
| 15 | &= ^= \|= | složená přiřazení | ← | ano |
| 15 | >>= <<= | složená přiřazení | ← | ano |

| | | | | |
|---|---|---|---|---|
| 16 | , | postupné vyhodnocení | → | ano |

Nyní si jednotlivé operátory popíšeme a vysvětlíme si je. Zatím se nebudeme zabývat všemi operátory, ale zaměříme se zatím na ty nejdůležitější. Zbylými operátory se budeme zabývat postupně.

Přiřazení

Jedná se snad o nejdůležitější operátor. Jako symbol se pro něj používá znaménko rovná se. Pozor nezaměňovat s matematickým rovná se, které znamená rovnost, pro něj existuje jiný operátor, který si ukážeme v relační operátorech (= =). Co to přiřazení znamená? Nejlépe bude, když si vše ukážeme na příkladu.

```
a = b * (c - 1);
```

Nejprve se provede vyhodnocení nebo-li vyčíslení výrazu, který je napravo od operátoru a po té výsledek se přiřadí-uloží do proměnné a. Shrne-li se to tak nalevo od operátoru přiřazení musí být výraz, který se odkazuje do paměti a nazýváme ho adresový výraz. Nalevo je označeno pomocí identifikátoru místo v paměti kam se výsledná hodnota uloží, která vznikla při vyhodnocení pravé části, která se jmenuje l-hodnota. Pokud má nějaký výraz l-hodnotu tak tzn., že výsledek nějakého vyčíslení je přiřazen nalevo do adresového výrazu.

Datový typ výrazu tedy jeho výsledku je závislý na datovém typu jednotlivých operandů, je-li u všech operandů stejný pak je výsledek opět stejného datového typu v případě , že operandy budou rozdílných datový typů výraz bude takový datový typ jako byl největší datový typ u jednoho z operandů. Viz. jedna z předcházejících kapitol datové typy a přetypování.

Pomocí operátoru přiřazení můžeme provést i inicializaci více proměnných stejnou hodnotou zároveň. Např.:

```
int a, b, c;
a = b = c = -1;
```

Podle normy C++ se nedoporučuje používat výrazy, ve kterých se vlevo i vpravo od operátoru pčřazení mění stejná proměnná. Např.:

```
cc = cc++ * 2;
a[i] = i++;
```

Aritmetické výrazy

Vytváříme je opět pomocí operandů a aritmetických operátorů + pro sčítání, - pro odčítání, * pro násobení, / pro dělení a % pro zbytek po celočíselném dělení. S aritmetickými výrazy jsem se již nevědomky setkali a zde bude nejlepší si rovnou po názornost ukázat nějaký ten příklad.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){

     int op = 2 ,op2 = 45 ,op3 = 30, v1, v2, v3;
     int c1 = 20000, c2 = 20001, vc;
     v1 = op * op2;
     v2 = op3 / 2;
```

```
v3 = op2 % 2;
cout << op << " * " << op2 << " = " << setw(5) << v1 << endl;
cout << op3 << " / 2 = " << setw(5) << v2 << endl;
cout << op2 << " % 2 = " << setw(5) << v3 << endl;
vc = c1 * c2 ;
cout << endl << " nyni pozor :" << endl;
cout << setw(10) << c1 << " * ";
cout << setw(10) << c2 << " = " << vc << endl;
cout << setw(10) << vc << " * " << setw(10) << 10;
cout << " = " << vc*10 << endl;
return 0 ;
}
```

Výstup programu z programu by pak vypadal takto:

```
2 * 45 =    90
30 / 2 =    15
45 % 2 =     1

 nyni pozor :
     20000 *      20001 = 400020000
 400020000 *         10 = -294767296
```

Všimněte si posledního vypsaného řádku, došlo k tzv. celočíselnému přetečení tzn. výsledek se do datového typu již nevejde a je vrácena nesmyslná hodnota. Vznikne když, násobíme velká čísla. 32bitový překladač má omezený počet platných míst pro celá čísla!!!

Nyní si dáme ještě jeden příklad kde argumenty jsou opět celočíselné, ale levá strana je racionální. V příkladu je jasně vidět, že podíl 13/3 v tomto případě vrátí výsledek 4 místo správných 4,333.

```
#include <iostream>

using namespace std;
int main(){
     int i=13;
     double r;
     r = i / 3;
     cout <<"r= " << r << endl;
     return 0;
     }
```

Poznámka: pro změnu priorit operátorů užíváme stejně jako v matematice závorek.

Použití zkráceného tvaru přiřazovacího operátoru

Jde o velice jednoduchou variantu standardního přiřazovacího výrazu. V programovacím jazyku C/C++ se tento způsob zápisu velice často používá, protože zjednodušuje kód a tím jej dělá přehlednější. Obecně ho zapisujeme takto:

```
l-hodnotota operátor= výraz
```

Tento druh zápisu lze využít por následující operátory: +, -,*, /, %, <<, >>, &, ^, !. Zkácené verze budou vypadat tedy takto: +=, -=,*=, /=, %=, <<=, >>=, &=, ^=, !=. Nyní si uvedeme příklad.

```
x /= y - 3;     // znamena x = x / (y - 3)
```

Pozor v hodně případech začínající programátoři si význam uvedeného výrazu vyloží jako: x = x / y – 3. Což je chybné, sami víte z matematiky, že použití respektive jejich nepoužití nám dá dva rozdílné výsledky.

Operátory inkrementace a dekrementace

Tyto dva operátory jsou jedno z specifik jazyka C/C++. V podstatě jde o zvýšení hodnoty proměnné o jedničku (inkrementace) nebo snížení hodnoty o jedničku (dekrementace). Nyní si uvedeme malý příklad.

```
i++;//znamena i=i+1
i--;//znamena i=i-1
```

Existují dvě varianty každého z těchto dvou operátorů tzv. prefix a prefix. Prefixový zapisujeme před operandem. Čili nejprve provede inkrementace nebo dekrementace a teprve pak je vyhodnocena hodnota operandu tzn. hodnota operandu je nejprve zmenšena či zvětšena o jedničku a teprve pak je výsledná hodnota vrácena. Postfixový zapisujeme za operandem, jeho funkce je opačná než u prefixového, tzn. nejprve dojde vyhodnocení hodnoty – vrácení operandu a ta je inkrementována či dekrementována až pak. Vše nejlépe uvidíme na následujících příkladech:

```
int a = 5, b = 0;
b =  a++;
```

V tomto prvním příkladě se do proměnné b přiřadí hodnota 5 a proměnná a je zvětšena o jedničku až po přiřazení.

```
int a = 5, b = 0;
b =  ++a;
```

Ve druhém příkladě se do proměnné b přiřadí hodnota 6, protože proměnná a je nejprve zvětšena o jedničku.

Logické operátory

V C++ existuje logický datový typ bool o kterém jsme se již zmínili. Řekli jsme si, že může nabývat dvou hodnot true a false. True tedy pravda a je reprezentována logickou jedničkou. False tedy nepravda a je reprezentována logickou nulou.

Existují zde tři logické operátory konjunkce (and), disjunkce (or) a negace (not). Značí se v programovacím jazyce C++ takto: && pro and, || pro or a ! pro not. Slouží k provádění výpočtů u logických výrazů. Výpočty se provádějí pomocí pravidel počítání v Boolově algebře viz. následující tabulka.

| A | B | !A | A && B | A \|\| B |
|---|---|---|---|---|
| false | false | true | false | false |
| false | true | true | false | true |
| true | false | false | false | true |
| true | true | false | true | true |

Relační operátory

Relační operátory jsou: < menší než, > větší než, <= menší nebo rovno, >= větší nebo rovno, = rovno a != nerovno. Výsledek při relačních operacích bývá logická hodnota tedy true nebo false. Dají se použít u všech základních datových typů. Operátory rovnosti a nerovnosti se používají při práci s ukazateli. Ostatní relační operátory se používají při porovnávání polí, struktur.....

Bitové operátory

Zajišťují provádění operací s jednotlivými bity. Ne všechny vyšší programovací jazyky mají k dispozici práci bitové operátory. Protože je nutná znalost umístění jednotlivých bitů v paměti počítače, tak se touto oblastí budeme zabývat jen stručně.

K dispozici jsou << bitový posun vlevo, >> bitový posun vpravo, & bitová konjunkce, | bitová disjunkce, ~ bitová negace a ^ bitová nonekvivalence. Pravidla pro počítání s nimi jsou uvedena v následujíc tabulce.

| A | B | ~A | A & B | A \| B | A ^ B |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |

Čísla v paměti počítače jsou řadou 1 a 0 a s těmi to 1 a 0 se provádějí jednotlivé operace. Pozor bitové operace lze provádět pouze s celočíselnými hodnotami. Jelikož, je tato problematika pro začátečníky obtížná a zatím nepotřebná vrátíme se k tomu někdy hodně později respektive spíš se k této problematice vrátíte při svém dalším studiu na vysoké škole.

Adresový operátor

Pomocí adresového operátoru & získáme adresu objektu, na který ho použijeme. Používá se v souvislosti s ukazateli. K čemu nám je dobré znát adresu nějakého objektu? Není nutné kontrolovat umístění proměnný v paměti, to překladač udělá lépe než bychom to udělali my.
V některé z následujících kapitol tuto znalost využijeme při předávání výsledků z funkcí.

Díky adresovému operátoru získáme adresu proměnné a přiřadíme ji ukazateli. Poté lze na výstup vypsat jak hodnotu proměnné i její adresu

```
#include <iostream>
using namespace std;
int main() {
     int i = 12, *pi;
     pi = &i;
     cout << "promenna i=" << i;
     cout << " je umistena na adresu: " << pi << endl;
     return 0;
     }
```

Reference - Odkaz

Jde o konstrukci, která je vytvářena pomocí adresového operátoru. Jedná se o umístění

adresového operátoru jakou součást deklarace proměnná za datový typ proměnné.

```
int i = 0;
int & o_i = i;
o_i = 2; // umístí 2 do proměnné i přes odkaz o_i
```

Proměnná i je normální proměnná datového typu int. Odkaz je aliasem na proměnou i.

Podmíněný operátor

Je jediný operátor v jazyce C++, kde jsou nutné tři hodnoty. Skládá se z otazníku a dvojtečky. Nyní si ukážeme syntaxi:

```
výsledek = (podmínka) ? splňeno : nesplňeno;
```

Pokud bude podmínka splněna do výsledku bude přiřazeno splněno pokud ne bude do výsledku přiřazeno nesplněno. Nyní si ukážeme praktický příklad na kterém se vše pochopí nejlépe. Budeme počítat absolutní hodnotu zadaného čísla.

```
#include <iostream>
using namespace std;
     int main() {
     int a, abs_a;
     cout << endl << "Zadej cele cislo: ";
     cin >> i;
     abs_i = (i < 0) ? -i : i;
     cout << "abs(" << i << ") = " << abs_i << endl;
     return 0;
     }
```

Zadáme-li záporné číslo třeba -850 tak program vypíše: abs(-850) = 850.

Operátor čárky

Čárka jako operátor slouží k postupnému vyhodnocení několika výrazů, které jsou odděleny čárkou. Jak je vidět z naší tabulky, kde jsme si vypsali všechny operátory čárka má nejnižší prioritu. Vyhodnocování zde probíhá zleva doprava. Výsledek je hodnota výrazu, který je nejvíce napravo. Tento operátor je používán třeba v cyklu for.

Příkazy ovlivňující (řídící) vykonávaní programu

Kdyby neexistovali příkazy, které dokáží řídit chod programu mohl by program běžet pouze lineárně odshora dolů. Tyto příkazy nám umožňují vykonat jen nějakou část programu, případně jinou vynechat apod.

V jazyce C++ máme k dispozici tyto příkazy, které řídí chod programu:

1. výrazový příkaz
2. blok
3. podmíněný příkaz
4. přepínač
5. cyklus
6. skok
7. výjimky

Výrazový příkaz

Jde vlastně o výraz, které známe již z předchozích kapitol. Pod pojmem výraz rozumíme aritmetický výraz, konstantní hodnotu, konstantu, proměnou, ale rozhodně jsem nepatří volání funkce nebo přiřazení. Jak se stane z výrazu výrazový příkaz? Úplně jednoduše ukončíme ho středníkem. Nebo též lze ho označit jen pojmem příkaz.

Existuje také prázdný příkaz to je vlastně je výrazový příkaz bez výrazu čili jde o samotný středník.

Blok příkazů

Jak jsme si na začátku naší knihy řekli tak funkce tělo funkce je ohraničeno složenými závorkami, není to však jediné využití složených závorek. Složené závorky vymezují nějaký **blok příkazů**.

Blok příkazů je vykonán v programu místo jediného příkazu a jednotlivé příkazy v bloku jsou prováděny v pořadí v jakém jsou zapsány. Do bloku mohou být zapsány tedy příkazy a také v něm lze deklarovat a definovat proměnné. Proměnné vytvořené v bloku se nazývají lokální a existují pouze po dobu zpracování bloku.

Tyto bloky příkazů se v jednotlivých funkcích mohou do sebe libovolně zanořovat, ale nesmějí se křížit. Jejich využití především spočívá při  podmínečném zpracování části kódu, opakovaném vykonávání.....

Poznámka blok příkazů se nezakončuje středníkem, přesněji řečeno nemusí zakončovat.

```
#include <iostream>
using namespace std;
int main ()
{
	{
		BLOK_PRIKAZU;

	}
	{
		BLOK_PRIKAZU_2;
		{
			BLOK_PRIKAZU_3;
		}
	}

	}
```

Všimněte si jak se jednotlivé bloky do sebe zanořují tedy jsou postupně odsazeny  zleva. Není to sice povinností, ale patří to k dobrému programátorskému stylu. Zvyšuje to přehlednost programu. Proto jsi zvykněme toto zanořování používat!!!

Oblast platnosti identifikátorů

V minulé kapitolce jsme se zmínili o lokální proměnné, které existuje pouze v bloku. Nyní

se pobavíme o této problematice trochu detailněji. Jak už bylo v předchozích kapitolách řečeno identifikátor je nějakým označením, který lze používat pouze tehdy pokud je spojen s nějakou proměnou, konstantou, funkcí.... Deklarace z hlediska jednoho souboru začíná na jeho začátku a a končí na jeho konci. Deklarace z hlediska argumentů funkce má platnost od deklarace argumentů do ukončení funkce. V bloku deklarace platí do jeho konce.

Pokud budeme mít dvě proměnné se stejným identifikátorem přičemž jedna z nich bude lokální v nějakém bloku a druhá bude mít platnost v rámci celého programu tedy tzv. globální. Lokální proměnná zastíní platnost globální tzn. hodnota bude přiřazena do proměnné, která platí v rámci jednoho bloku. Používat stejné identifikátory se přesto nedoporučuje!!! Snižuje to přehlednost kódu a díky tomuto překrývání může dojít k přiřazení nevědomky do nesprávné proměnné.

Prostory jmen

Nebo-li též jmenný prostor je to velice důležitá část v kódu programu, která úzce souvisí s jednotlivými objekty a s identifikátory. Jde o velice silný nástroj, který umožňuje vyřešit konflikty mezi identifikátory.

V krátkém programu není problém pracovat s identifikátory. Více programátorů na rozsáhlejším projektu může trochu činit problém s domluvou kdo jaké identifikátory bude používat. Z tohoto důvodu každý programátor používá svůj jemný prostor, u proměnných je to pak něco jako příjmení. U identifikátorů je to vlastně druhé jméno, které je spojuje do jedné skupiny.

My jsme se v naši programech setkali s používáním předdefinovaného jmenného prostoru using namespace std; Vlastní jemný prostor vytvoříme takto:

```
namespace idetifikátor {seznam deklarací}
```

Teoreticky je identifikátor nepovinný, ale pokud ho nenapíšeme nebude jak se k jmennému prostoru dostat. Chceme-li použít něco z jmenného prostoru máme dvě možnosti. První možnost je, že napíšeme nám již známe:

```
using prostor_jmen
```

V tomto případě objekty v prostoru jmen budou viditelné od bloku, funkce před, kterou ho uvedeme. Chceme-li, aby prostor jmen byl viditelný z celého programu uvedeme ho před funkcí main za direktivou.

Druhou možností je před identifikátorem zapíšeme pomocí čtyř tečky :: název jmeného prostoru ve kterém se nalézá. Uvedeme-li pouze čtyřtečku a identifikátor jedná se o přístup do globální úrovně.

```
prostor_jmen :: identifikátor
```

Nyní si pro lepší pochopení ukážeme příklad.

```
#include <iostream>

const char *id = "globalni";// globalni konstanta
namespace nas_prostor {
     const char * id = "programovani";
     // konstanta ve jmenném prostoru nas prostor
}
int main() {
     const char *id = "telo funkce main";
```

```
    // lokální konstanta ve funkci main
    std::cout << id << std::endl; // lokální
    std::cout << ::id << std::endl; // globální
    std::cout << nas_prostor::id << std::endl;
    return 0;
    }
```

V programu vidíme zachycené všechny tři případy které jsme si teoreticky popsali. Ukázkový výstup programu:

```
telo funkce main
globalni
programovani
```

Podmínka nebo-li podmíněný příkaz

Jde o větvení běhu programu do dvou různých alternativ. Program se dělí dle splnění nebo nesplnění nějaké podmínky. Vždy je dojde k vykonání pouze jedné alternativy v závislosti na vyhodnocení podmínky. Příkaz pro podmíněný příkaz je if-else. Tento příkaz existuje ve více variantách a my si je teď postupně ukážeme. Je důležité říct, že jde o základní příkaz, který umožňuje dělené kódu do více alternativ. Syntaxe tohoto podmíněného příkazu vypadá následovně:

```
    if (podmínka) příkaz1 else příkaz2
```

Jak to celé pracuje? Je-li podmínka splněna je vrácena hodnotu true, vykoná se příkaz1. Při nesplnění podmínky je vrácena hodnota false a vykoná se příkaz2. Po vykonání jednoho z příkazů program pokračuje za podmíněným příkazem.

Nepřehlédněte podmínka se uzavírá do kulatých závorek, jednoduché příkazy1 či 2 musí být ukončeny středníkem, místo jednoduchých příkazů1 či 2 lze užít celé bloky příkazů ve složených závorkách a za závorkami středník není. Nyní si již ukážeme praktický příklad, který seřadí dvě čísla od nejmenšího.

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

int main() {
    int a, b;
    cout << "Zadej dve cisla: ";
    cin >> a >> b;
    if (a < b)
    cout << a <<", "<< b << endl;
    else {
         cout << "cisla bylo nutne prohodit" << endl;
         cout << b <<", "<< a << endl;
        }
   return 0;
```

Druhou variantou příkazu if je vypuštěním varianty else. Syntaxe vypadá pak takto:

```
    if (podmínka) příkaz
```

V případě splnění podmínky je vrácena hodnota true a vykoná se příkaz či blok příkazů v podmínce. V případě nesplnění podmínky se nastane nic a pokračuje vykonávaní programu. Nyní si opět ukážeme příklad na vypsaní absolutní hodnoty zadaného čísla.

```
#include <iostream>

using namespace std;

      int main() {
      int i;
      cout << endl << "Zadej cele cislo: ";
      cin >> i;
      if (i < 0)
           i=i*-1;
      cout << i << endl;
      return 0;
      }
```

Nyní si ukážem poslední možnost využití příkazu if-else. Klasický příkaz podmínky má nevýhodu, že se pomocí něho lze rozhodnout jen mezi dvěma alternativami. Jak tu uděláme když potřebujeme vybrat z více alternativ vykonávaného kódu? Jde tu úplně jednoduše příkaz if-else zanoříme několikrát do sebe, někdy je tato konstrukce označována if-else-if. Syntaxe by vypadala takto:

```
if (podmínka1) příkaz1
else if (podmínka2) příkaz2
else if (podmínka3) příkaz3
...
else if (podmínkaN) příkazN
else příkazN+1
```

Zde bude proveden vždy ten příkaz nebo blok příkazů, u kterého bude splněna podmínka, v poslední části nemusí být opět jako v předchozím případě uvedena větev else a pak se tedy nic nevykonná. Nyní si již ukážeme praktický příklad, který vypíše hlášení jaký typ znaky byl zadán.

```
#include <iostream>

using namespace std;

int main() {
      char znak;
      cout << "Zadej alfanumericky znak:";
      cin >> znak;
      cout << endl << "Zadal jsi ";
      if ((znak >= 'a') '' (znak <= 'z'))
      cout << "male pismeno";
      else if ((znak >= 'A') '' (znak <= 'Z'))
      cout << "velke pismeno";
      else if ((znak >= '0') '' (znak <= '9'))
      cout << "cislici";
      else
      cout << "Nezadal jsi alfanumericky znak!";
      cout << endl;
      return 0;
      }
```

Přepínač

Jak jsme si již koncem minulé podkapitoly řekli, tak nám kolikrát nestačí vybírat jen ze dvou variant běhu programu a používat příkaz if-else-if je značně zdlouhavé a neefektivní pro běh programu, protože program testuje jednu a tu samou podmínku několikrát, přesto se bez příkazu if-else-if někdy neobejdeme.

V jazyce C++ existuje příkaz zvaný přepínač, který nám umožní větvení programu na více než dvě variamty. Pro přepínač v jazyce C++ je defonováno klíčové slovo switch. Syntaxe tohoto příkazu je podobné příkazu if, tedy po příkazu switch je v kulaté závorce podmínka, pak z obecného hlediska následuje příkaz.

```
switch (podmínka) příkaz
```

Pod pojmem příkaz je většinou myšleno tělo přepínače. V těle přepínače jsou jednotlivá návěští. Každé z návěští pak představuje jednu variantu běhu po vyhodnocení podmínky. Jednotlivá návěští jsou složena z příkazu case, celočíselnou konstantou a dvojtečkou. Používá se také speciální návěští default po kterém následuje rovnou dvojtečka. Dále je podstatné říci, že celočíselná konstanta v jednotlivých návěštích musí být jedinečná. Nyní si ukážeme syntaxi těla přepínače:

```
{
case konst1: příkaz1
case konst2: příkaz2
...
case konstN: příkazN
default: příkazD
}
```

Jak vypadá vyhodnocení? Podmínka je vyhodnocena a na základě hodnoty, kterou podmínka vrací je skočeno do příslušného návěští, jehož konstanta je schodná s vyhodnocením podmínky. Po té se zpracují všechny příkazy, které jsou obsaženy v návěští. Na konci návěští by program automaticky pokračoval zpracováváním dalšího navěští tedy vykonáváním jeho příkazů bez ohledu na to jaká konstanta je v přepínači. Jak tedy zabránit vykonávaní dalšího návěští? Velice jednoduše na konec návěští zapíšeme příkaz break, který zajistí pokračování programu za tělem přepínače.

Pokud není nalezeno návěští s konstanoum které se rovná vyhodnocení podmínky je otevřeno do návěští default. Jsou pak provedeny příkazy nalézající se v tomto návěští. Není-li definováno program přeskočí tělo přepínače a pokračuje prvním příkazem za tělem přepínače.

Nyní si ukážeme program, který simuluje házení kostkou.

```
#include <iostream>
#include <string>
using namespace std;
int main()
	{
	string s;
	cout << "Hazim kostkou..." << endl;
	srand(time(NULL));
	switch (rand() % 6 + 1)
		{
		case 1: s = "jednicka";
		break;
		case 2:
		case 3: s = "dvojka nebo trojka";
```

```
        break;
        case 4: s = "ctyrka";
        break;
        case 5: s = "petka";
        break;
        default: s = "sestka";
        break;
        }
    cout << "Padla " << s << endl;
    return 0;
    }
```

Jak to celé funguje? Zcela jednoduše pomocí funkce srand je nastavena posloupnost náhodných čísel. Jako počáteční hodnotu funkce srand je zvolena návratová hodnota funkce time, tato hodnota je díky hodinám počítače (tedy generátoru hodinových impulsů) pokaždé jiná a zajistí nám náhodnost. Protože nic s funkcí nebudeme dělat použijeme argument NULL. Samotné házení bude náhodně generovat číslo o maximální velkosti 32 767. Jak dostaneme tedy z tohoto číslo 1 až 6? Opět existuje jednoduchý způsob, na funkci srnad použiji operaci modulo tedy zbytek po celočíselném dělení, konkrétně pak 6 a jsou nám vráceny hodnoty 0 až 5. Pří čteme-li jedničku budou nám vráceny požadované hodnoty. Na základě těchto vrácených hodnot je skočeno do jednotlivých návěští. Všimněte si v návěští 2 není break a padne-li na kostce dvojka pokračuje vykonávaní do návěští 3. V jednotlivých návěštích je přiřazena konkrétní hodnota do řetezce s. Ta je pak za přepínačem vytištěna.

Nyní si ukážeme ještě jeden příklad, kde uživatel jen zadává čísla a program mu vypisuje co zadal:

```
#include <iostream>
using namespace std;
int main()
    {
    int i;
    cout << "Zadej hodnotu od 1 do 4" << endl;
    cin >> i;
    switch (i)
        {
        case 1:
            {
            cout << "Zadel jsi jednicku" << endl;
              break;
            }
        case 2:
        case 3:
            {
            cout << "Zadel jsi dvojku nebo trojku" << endl;
              break;
            }
        case 4:
            {
            cout << "Zadel jsi ctyrku" << endl;
              break;
            }
        default:
            {
```

```
            cout << "Zadel jsi spatnou hodnotu" << endl;
              break;
            }
        }
return 0;
}
```

Poznámka:

- Přepínače lze do sebe libovolně zanořovat.
- Nelze použít dvě návěští se stejnou hodnotou.
- Příkaz break může být v návěští vložen v libovolných příkazech, ale vložíme-li ho do cyklu je přiřazen k němu a ne k přepínači.
- Z hlediska syntaxe je velice častá chyba zapomenutí klíčového slova case a zapsání konstanty pouze s dvojtečkou. Tento zápis je sice přípustný, ale znamená něco úplně jiného a k tomu se dostaneme hned po následující kapitole. Při zapomenutí slova case se program bude chovat nestadartně.
- if na rozdíl může testovat jakékoliv hodnoty, ale switch jen celočíselné
- U if-else-if se provede jeden z příkazů či blok z příkazů. Naproti tomu u přepínače se numusí provést, žádný z příkazů nebo více z příkazů či bloků.
- V přepínači se návěští default může vyskytovat kdekoliv, ale v konstrukci if-else-if je tato hodnota vždy na konci a jde o poslední else.

Cykly

Mnohdy je nutné část programu provádět opakovaně, proto existují konstrukce jménem cykly. Činnost cyklu je tedy opakované provádění části kódu, které je závislé na nějaké podmínce. Cyklus se skládá z podmínky cyklu a těla cyklu. Podmínka rozhoduje o znovu provedení či neprovedení těla cyklu. V těle cyklu může být jediný příkaz, ale nejčastěji jde o blok příkazů.

V jazyce C++ existují tře druhy cyklů. Dělíme je dle toho kolikrát se provede tělo cyklu. Prvním základním cyklem je cyklus while u něhož nevíme kolikrát se vykoná, ale nemusí proběhnou ani jednou. Druhý cyklu je jeho opakem víme přesně počet opakování těla cyklu a ten se jmenuje for. Posledním cyklem je cyklus do, ten má společného s cyklem while to, že nevíme kolikrát proběhne, ale víme, že musí proběhnout alespoň jednou. Ve většině případů to co je zapsáno pomocí jednoho cyklu jde přepsat do druhého, ale cykly bychom měli volit vzhledem k počtu jejich průběhů.

Cyklus while

Má podmínku na začátku, tzn. že podmínka cyklu while se testuje před průchodem těla cyklu. Tento cyklus tedy vůbec nemusí proběhnout a když proběhne tak nevíme kolikrát proběhne. Syntaxe je velice jednoduchá:

```
while (výraz) příkaz
```

Klíčové slovo while po, kterém následuje testovací výraz v kulatých závorkách. Je-li výraz vyhodnocen hodnotou true bude prováděno tělo cyklu. Po vykonání těla cyklu se opět provede vyhodnocení testovacího výrazu. Takto běří cyklus neustále do kola pokud vyhodnocením testovacího výrazu není hodnota false. To se pak program pokračuje s vykonáváním příkazu, který následuje za cyklem.

V těle cyklu se mohou objevit i příkazy break a continue. Příkaz ukončí provádění příkazů těla cyklu a přenese chod programu za probíhající cyklus. Příkaz continue rovněž ukončí provádění příkazů těla cyklu, ale řízení je předáno na řídicí podmínku cyklu.

Tělo cyklu vždy tvoří blok příkazů uzavřený ve složených závorkách. Protože může dojít k této situaci.

```
while (výraz)
   přikaz1;
   přikaz2;
```

```
while (výraz)
   {
   přikaz1;
   přikaz2;
   }
```

V prvním případě do těla cyklu patří jen příkaz1 ve druhém, pak do těla cyklu patří oba dva příkazy. Na to bychom měli dát pozor bývá to dost častá chyba.

Tak a teď už se podíváme ke konkrétnímu příkladu. Budeme opět házet kostkou, kde v těle cyklu budou počítány počty hodů dokud nepadne šestka 10x.

```
#include <iostream>
#include <string>

using namespace std;

int main()
     {
     static int celkem, pocet;
     const int POCET = 10;
     cout << "Hazim kostkou dokud mi nepadne " << POCET ;
     cout << "krat sestka..." << endl;
     srand((unsigned) time(NULL));
     while (pocet < POCET)
          {
          celkem++;
          if ((rand() % 6 + 1) == 6)
               pocet++;
          }
     cout << "A je to! Hodu bylo celkem " << celkem << endl;
     return 0;
     }
```

Cyklus for

Tento cyklus jev v kódu využíván, když je známý počet opakování těla cyklu. Tělo opět může být tvořeno jen jedním příkazem nebo celým blokem příkazů. Cyklus for je vykonán vícekrát nebo ani jednou. Pokud je prováděn vícekrát provádí se dokud je vyhodnocení podadmínky testovacího výrazu true. Jak vypadá syntaxe? Za klíčovým slovem for jsou v kulaté závorce oddělené středníky uvedeny: inicializační příkaz, podmínka a výraz.

```
for (inic_příkaz; podmínka; výraz)
     {
     tělo cyklu;
     }
```

Inicializační příkaz je vlastně nepovinný, jeho vykonání je provedeno před první průchod cyklu, použití je pro inicializaci proměnných před cyklem. Jde vlastně o počáteční hodnotu proměnných. Poté následuje podmínka, která je rovněž nepovinná, podmínka je vyhodnocována

před každým průchodem cyklu. Když doběhne tělo cyklu je vykonána třetí část cyklu tedy příkaz., který je opět nepovinný. Tento příkaz je vlastně příprava pro další testování, nječastěji jde o vyhodnocení nějakého výrazu a s tím spojené změny hodnoty nějaké proměnné. Opět se zde mohou objevit příkazy break a continue. Teoreticky minimální povinná syntaxe příkazu for vypadá takto:

```
for (;;)
     {
     tělo cyklu;
     }
```

Pojďme rovnou k příkladu, ne kterém je vě vidět nejlépe. Program vytiskne část ASCII tabulky znaků s kódy 32 až 127 včetně.

```
#include <iostream>

using namespace std;

int main()
     {
     cout << endl;
     for (int znak = 32; znak < 128; znak++)
          {
          if (znak % 16 == 0)
          cout << endl;
          cout << (char) znak;
          }
     cout << endl;
     return 0;
     }
```

Výstup bude vypadat pak takto:

```
!"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~Ś
```

Cyklus do

Jde vlastně o cyklus while, který je napsaný obráceně tzn. podmínka je na konci. Přítomnost podmínky na konci nám zajistí, že se vykoná tělo cyklu alespoň jednou tzn. k testování podmínky dojde až po průchodu těla cyklu. Úplně ideálně se toto hodí pro testování správnosti zadané hodnoty, pokud uživatel zadá správnou tak se pokračuje dalším vykonáváním kódu a pokud ne je ji nucen zadat znova. Syntaxe vypadá následovně:

```
do příkaz while (výraz);
```

První je klíčové slovo do po, kterém následuje příkaz, může se jednat o jeden příkaz nebo blok. Pak je klíčové slovo while a v kulatých závorkách je podmínka. Cyklus opět akceptuje příkazy break a continue.

Poznámka v minulých dvou cyklech jsme si říkali logické hodnot, které byli vyhodnocením podmínky tedy true nebo false. U příkazu do toto neplatí do-while může vyhodnocovat jakékoliv

výraz a ne jen logický jako u předchozích dvou cyklů. Je dobrým programátorským zvykem podmínku dávat na stejný řádek s koncovou závorkou těla cyklu, protože tím předejdeme vizuální záměně za cyklus while. Teď si ukážem stejný příklad jako u cyklus for a rovnou s tím prokážeme zaměnitelnost cyklů.

```
#include <iostream>
using namespace std;
int main()
      {
      int znak = 32;
      cout << endl;
      do {
            cout << (char) znak++;
            if (znak % 16 == 0)
            cout << endl;
      } while (znak < 128);
return 0;
}
```

Nyní pro doplnění si ukážeme ten samí program pomocí cyklu while:

```
#include <iostream>
using namespace std;
int main()
      {
      int znak = 32;
      cout << endl;
      while (znak < 128)
            {
            cout << (char) znak++;
            if (znak % 16 == 0)
            cout << endl;
            }
return 0;
}
```

Příkaz skoku

Jazyk C/C++ podporuje i příkaz nepodmíněný skok v kódu programu, který se nazývaá goto. Tento příkaz se v kódu používá velice ve vyjímečných případech, protože jeho použitím je narušuna struktura programu a tím pádem k´do je nepřehledný. Příkaz skoku také velice zpomaluje běh programu. Syntaxe příkazu goto je následující:

```
navěští: příkaz;
goto návěští;
```

Výjimky

Jedná se o velice moderní programovací techniku, která v jazyce C++ slouží jako mechanismus obsluhy chyb. Syntaxe výjimek je založena na třech klíčových slovech:

- try – zapisují se jsem veškeré výjimky, které chceme monitorovat
- catch - popisuje jak reagovat na výjimky
- throw - vyvolává výjimky

Využívání výjimek souvisí s řízením chodu programu, můžeme si uvést několik příkladů kdy se nám mohou velice hodit. Nejčastější výjimky souvisí s uživatelovou obslouhou námi vytvořeného programu. Uživatel může odpojit externí disk zrovna když na něj probíhá zápis dat, může zadat čas nebo datum jiném formátu.... Mechanismus výjimek je adekvátní řešení situace, požádá o připojení disku a provede zápis znova, nebo požádá u času či data o uvedení ve správném formátu....

Syntaxe výjimek je velice jednoduchá:

```
try {
      hlídaný blok
      throw výjimka;
}
catch (typ_výjimky){
      zpracování výjimky
}
```

V bloku za klíčovým slovem try je zapsána ta část programu kde by se mohli vyskytnout problémy. Nastane-li v hlídaném bloku problém, je vyvolána musí být vyvolána výjimka. Výjimku můžeme vyvolat dvěmi možnostmi. Jsou dvě možnosti, jak se dá výjimka vyvolat. První možností je vyvolání výjimky pomocí některé z knihoven jazyka C++, tuto možnost lze použít pouze jedná-li se o výjimku, která je podporována normou jazyka C++. Druhou možností je využití klíčového slova throw s typem výjimky, kterým může být celé číslo, řetězec nebo jiný objektový. Zachycení výjimky má na starosti ta část programu která je za klíčovým slovem catch. Catch obsahuje pár kulatých závorek, obsahující buď výpustku ... - pak jsou zachyceny výjimky všech typů, nebo typ zachycované výjimky a případně i identifikátor id.

Pozn. Abychom mohli výjimku zachytit tak musí nejprve nastat.

Nyní si pro praktickou ukázku uvedeme příklad, který má pomocí výjimek ošetřeno dělení nulou.

```
#include <iostream>
float delenec=0, delitel=0, podil=0;

using namespace std;

int main() {
      cout <<"Deleni dvou cisel"<<endl;
      cout <<"Zadejte delenec: ";
      cin >> delenec;
      cout <<"Zadejte delitel: ";
      try {
            cin >> delitel;
            if (delitel==0)
            throw "Deleni nulou.\n";
            podil=delenec/delitel;
            cout << delenec <<" / "<<delitel<<" = "<<podil<<endl;
      }

catch (const char* exception){
      cout << "Byla zachycena vyjimka - " << exception;
      }

return 0;
      }
```

V programu může dojít většinou k více výjimkám, je možné napsat více bloků catch. Každý blok catch bude mít na starost konkrétní typ výjimky. Existuje však blok catch, který dokáže zachytit všechny výjimky.

```
try{
     NebezpecnaFunkce();
     }

catch (...){
     // Timto blokem je mozne chytit kazdou vyjimku
     }
```

Tento způsob však není moc vhodný, protože je vhodné každou výjimku ošetřit rozdílným způsobem, v tomto případě to není možné. Lepší řešení je napsat více bloků catch, a blok catch (...) dát až jako poslední, jako takový "záložní", který zajistí výjimku co programátor nepředpokládal.