

UNIVERZITA KONŠTANTÍNA FILOZOFA V NITRE

**Algoritmizácia
a úvod do programovania**

Skalka Ján - Cápaj Martin - Lovászová Gabriela -
Mesárošová Miroslava - Palmárová Viera

Nitra 2007

Obsah

Úvod	7
1 Algoritmizácia	9
Pojem problém	9
Algoritmus	10
Elementárnosť	11
Determinovanosť	12
Rezultatívnosť	12
Konečnosť	13
Hromadnosť	14
Efektívnosť	14
Ako algoritmizovať?	15
Algoritmický jazyk.....	15
2 Algoritmické štruktúry	18
Vývojové diagramy.....	18
Príkazy vstupu a výstupu	19
Premenná.....	19
Sekvencia	21
Vetvenie.....	23
Zložitejšie podmienky.....	29
Cyklus	30
Cyklus so známym počtom opakovaní	30
Cyklus s podmienkou na začiatku.....	33
Cyklus s podmienkou na konci	35
Neriešiteľné problémy	37
3 Programovacie jazyky	39
Prekladače	41
Typy programovacích jazykov.....	42
Jazyky aplikácií.....	44
Programovací jazyk pascal	46
Štruktúra programu	46
Pozdrav ma!	47
Prvý „program“	49
Parametre komponentu	50
Základná filozofia programu.....	52
Skladba aplikácie	54
Konečne pozdrav!	56
Príkaz výstupu.....	56
Prepis sekvenčných algoritmov	60
4 Prepis štruktúr do programovacieho jazyka	64
Vetvenie.....	64
Reálne čísla	68
Cykly.....	70

Názov: Algoritmizácia a úvod do programovania

Autori: Skalka Ján
Cápay Martin
Lovászová Gabriela
Mesárošová Miroslava
Palmárová Viera

Recenzenti:
PaedDr. Jozef Kapusta
Doc. Ing. Cyril Klimeš, CSc.

Edícia: Prírodovedec č. 276

Publikácia vyšla vďaka prostriedkom poskytnutým agentúrou KEGA v rámci projektu KEGA 3/3041/05

Schválené vedením FPV UKF v Nitre dňa 26. 10. 2007

Rukopis neprešiel jazykovou úpravou

© UKF v Nitre 2007

ISBN 978-80-8094-217-5
EAN 978-80-8094-217-5

Cyklus so známym počtom opakovaní	70
Cyklus s neznámym počtom opakovaní	71
5 Práca s textom	74
Textové reťazce	74
Číselné versus textové hodnoty	76
Podporné funkcie	79
Údajový typ Char	81
Ordinálne a neordinálne typy	83
Logický typ	85
Priorita operátorov	86
6 Chyby v programe	89
Testovanie a ladenie	90
Zložitosť a efektívnosť	91
7 Zoznamy	94
Viacnásobné vetvenie	94
Reprezentácia údajových typov	98
Štruktúrovaný typ pole	99
Listbox	101
Úprava prvku	104
Hľadanie v Listboxe	104
Listbox a pole	105
Konštanty	105
Využitie poľa	107
Náhodné čísla	109
8 Súbory	111
Uloženie údajov	111
Údaje v súbore typu integer	114
Textový súbor	115
Vylepšenie práce so súborami	117
9 Podprogramy	122
Dôvody používania podprogramov	122
Procedúry	123
Globálne a lokálne premenné	125
Funkcie	127
Parametre podprogramov	128
Mechanizmus volania podprogramu	134
10 Štruktúrované typy	135
Záznam (record)	135
Prostredie pre prácu so záznamami	138
Zložitejšie štruktúrované typy	143
Záznam v zázname	143
11 Tabuľka - matica	146
Pole polí	146
Vizuálna reprezentácia tabuľky	147
Záverový test	153

Úvod

Algoritmické (a analytické) myslenie je nástrojom, ktorý poskytuje svojim majiteľom silu schopnú vytvoriť z počítača poslušného otroka slepo vykonávajúceho zadané príkazy. Niežeby počítače na súčasnej úrovni boli schopné neposlušnosti, no najmä pri začínajúcich používateľoch môže nestranný pozorovateľ nadobudnúť dojem, že nie človek, ale počítač je riadiacou zložkou ich vzájomného vzťahu.

V praxi môžeme pozorovať niekoľko veľkých skupín používateľov informačno-komunikačných technológií:

- začiatocníci, ktorí s počítačom pracujú z donútenia a pred klávesnicou uprednostňujú papier napriek evidentným výhodám textového editora,
- používatelia aplikačných programov (textové editory, tabuľkové a databázové systémy), ktorí ovládajú funkcie svojich nástrojov a sú schopní využívať funkcie, ktoré poznajú,
- pokročilí používatelia aplikačného softvéru schopní využívať a prispôbovať riešenie problému existujúcim funkciami, prípadne vyhľadávať nové funkcie, analyzovať a transformovať problém do jazyka aplikačného softvéru,
- zdatní používatelia, často správcovia systémov so závideniahodnými vedomosťami, ktorí však nie sú ochotní riešiť problémy pomocou vlastných programov a vhodné riešenie radšej pohľadajú (a takmer vždy nájdu) na Internete,
- používatelia programátori, ktorí sú v prípade potreby ochotní riešiť problém vytvorením algoritmu (počítačového programu),
- programátori, ktorí čas venovaný hľadaniu funkcie v aplikačnom softvéri, strávia radšej vytvorením vlastnej aplikácie, ktorá je presne prispôbená ich požiadavkám a umožňuje v prípade potreby pridávanie ďalších funkcií.

Cieľom tejto publikácie je poskytnúť čitateľovi dostatočné množstvo informácií na to, aby sa dokázal zaradiť do jednej z dvoch posledne menovaných skupín.

Celý učebný text je rozdelený do lekcí, pričom na začiatku sú vždy vymenované požiadavky na vedomosti, ktorými by mal čitateľ disponovať na zvládnutie tej-ktorej lekcie. Vďaka tomu je možné postupovať po texte nie priamočiario, ale niektoré kapitoly vynechať, prípadne sa im venovať neskôr.

„Do programov (zjednodušene povedané predpisov na to, čo má vykonávať počítač) vkladá každý programátor kúsok seba a spolu s klasickým postupom riešenia dodáva k nemu aj svoj estetický a podľa neho efektívny pohľad na spracovávanú problematiku. O tom, či je to pre iného používateľa estetické a efektívne, by sa dalo meditovať, ale základom programovania je práve možnosť realizovať riešenie problému podľa svojich predstáv, dať riešeniu svoj vlastný imidž. Je to podobné umeniu i skutočnému životu, kde platí, že keď dvaja robia to isté, nikdy to nie je to isté.

Programovanie má oproti mnohým iným činnostiam, ku ktorým nás „dobrovoľne-povinne“ vedú v škole, obrovskú výhodu - často nás núti naplno roztočiť mozgové závitky, rozmyšľať o tom, čo je podstatné a čo nie, čo má byť viditeľné a čo skryté, realizovať svoju vlastnú predstavu použitím známych prostriedkov. Vždy, keď sa človek do niečoho pustí, musí mať správnu motiváciu. Tou môže byť aj skutočnosť, že s počítačmi a informačnými technológiami sa už dnes stretnete prakticky všade. Nechcete dokázať prinútiť stroj počítač, aby robil presne to, čo mu zadáte? Je to výzva, ktorej sa ťažko odoláva, ak chcete naozaj niečo dosiahnuť. Skúste to!“

(Drlík, P.: Turbo Pascal I, 1998)

V algoritmizácii a programovaní nie sú dôležité definície (i keď tým základným sa určite nevyhneme), ale schopnosť analyzovať a myslieť. Programovanie a algoritmizácia všeobecne sú odrazovým mostíkom pre riešenie nezvyčajných, ale aj každodenných problémov. Naučia nás rozdeliť zložitú operáciu na jednoduchšie a tak lepšie pochopiť podstatu problému.

1 Algoritmizácia

predpoklady na zvládnutie lekcie:

- nie sú vyžadované žiadne špeciálne vedomosti

obsah lekcie:

- problém - pojem, definícia a príklady problémov
- algoritmus a vlastnosti algoritmov
- algoritmický jazyk

cieľ:

- oboznámiť sa so základnými pojmami algoritmizácie

Pojem problém

Prvotným dôvodom vytvárania počítačového programu, alebo všeobecnejšie algoritmu, je existencia problému, ktorý potrebujeme riešiť.

Problémov je okolo nás obrovské množstvo a mnohé z nich si v každodennom kolobehu už ani neuvedomujeme. Keď sa však pozastavíme a zamyslíme, zistíme, že život je plný problémov a problémových situácií, ktoré sa človek už od malička snaží riešiť.

Spočiatku (prvých desať rokov života) mu na vyriešenie problému stačí začať plakať (jemnejší výraz pre slovné spojenie „vrešťať ako pavián“), pričom túto formu aplikuje dovtedy, kým dospelí neurobia to, čo chce. Tento spôsob je najjednoduchší, relatívne najmenej namáhavý, ale bohužiaľ nedá sa používať stále, pretože v určitom veku už rodičia nereagujú na náš plač poslušne, ale vytiahnu remeň (príp. varechu).

Vezmime si jednoduchý problém: chceme piť kakao. Predtým ako sa pustíme do riešenia, je vhodné naplánovať si postup. (Plánovanie robíme zvyčajne len zo začiatku, neskôr takéto problémy riešime automaticky – bez rozmyšľania).

Ako prvé zrejme musíme overiť, či máme mlieko, cukor a kakao – tieto suroviny pre nás predstavujú vstupné podmienky. Pokiaľ ich nemáme a nechce sa nám ísť na nákup (ďalší problém zložený z mnohých menších problémov), postup sa skončil.

Ak sú všetky suroviny k dispozícii, môžeme pristúpiť k samotnej operácii:

1. do hrnčeka nalejeme mlieko,
2. dáme ho zohriať,

3. v prázdnej šálke zmiešame cukor a kakao (môže byť aj Granko),
4. skontrolujeme mlieko,
5. ak nie je dost' teplé, vrátime sa do bodu 4.,
6. zalejeme zmes v šálke,
7. necháme vychladnúť,
8. vypijeme.

... a je po probléme.

Pozrime sa teraz na našu činnosť zo stránky informatickej.

Na počiatku bol problém. **Problém je stav, v ktorom jestvuje rozdiel medzi tým, čo v danom momente máme a tým, čo chceme dosiahnuť** (nemáme nič a chceme kakao). Problém je vždy viazaný na svojho vlastníka (pre iného to nemusí byť problém, ale nezmysel) a na problémové prostredie (okrem prostredia, v ktorom chceme piť kakao, môže ísť napr. o finančné, školské, citové).

Riešenie problému chápeme ako **odstraňovanie rozdielu medzi aktuálnym stavom a tým, čo chceme dosiahnuť**. Postup, ktorým sa pri tejto činnosti riadime, nazývame aj **algoritmus**. Keď sa nám podarí dosiahnuť pôvodný cieľ, hovoríme o vyriešení problému. Nie každý problém je však riešiteľný a nie vždy sa dopracujeme k požadovanému výsledku.

1. Popíšte postup pri prechode cez križovatku bez semaforov.
2. Popíšte postup pri nákupe topánok.
3. Vymyslite algoritmus pre nastupovanie päťčlennej rodiny do dvojdverového automobilu. Popíšte aj vystupovanie.

Riešiť pomocou algoritmu problémy reálneho života je dost' náročné, pretože správny algoritmus vždy berie do úvahy všetky možnosti, detaily, náhody alebo zriedkavé situácie. Napr. pri našom postupe s varením kakaa by sme mali vziať do úvahy, že vypnú prúd (zastavia plyn), príde návšteva a mlieko vykypí, susedov kocúr rozbije šálku a pod. Takéto algoritmy potom možno navrhnúť len približne a za obmedzených podmienok.

O algoritmoch má zmysel hovoriť vtedy, keď máme k dispozícii určitú obmedzenú množinu príkazov (môže byť aj veľmi veľká), pomocou ktorých dokážeme navrhnúť postup pri riešení.

Algoritmus

Či sa nám to páči alebo nie, informatickej definícii algoritmu sa nevyhneme. Vzhľadom na to, že ide o elementárny (základný) pojem, ako napr. v geometrii bod, dokážeme ho len opísať.

Algoritmus je návod na vykonanie činnosti, ktorý nás od (meniteľných) vstupných údajov privedie v konečnom čase k výsledku.

Algoritmus chápeme ako predpis, popis krokov, ktoré musíme realizovať, aby sme dosiahli výsledok. Vykonávanie činnosti na základe algoritmu označujeme ako **výpočet**.

Od algoritmu zvyčajne vyžadujeme splnenie nasledovných požiadaviek:

- **elementárnosť** – postup je zložený z jednoduchých krokov, ktoré sú pre vykonávateľa (počítač, nemysliace zariadenie, človeka) zrozumiteľné,
- **determinovanosť** – postup je zostavený tak, že v každom momente jeho vykonávania je jednoznačne určené, aká činnosť má nasledovať, alebo či sa už postup skončil,
- **rezultatívnosť** – výpočet dáva po konečnom počte krokov výsledok,
- **konečnosť** – výpočet (činnosť vykonávaná podľa algoritmu) vždy skončí po vykonaní konečného počtu krokov,
- **hromadnosť** – algoritmus je použiteľný na celú triedu prípustných vstupných údajov,
- **efektívnosť** – výpočet sa uskutočňuje v čo najkratšom čase a s využitím čo najmenšieho množstva prostriedkov (časových i pamäťových).

Splnenie týchto vlastností je dôležité, pretože algoritmy v informatike zvyčajne vykonáva nemysliace (jemnejší výraz pre hlúpe) zariadenie, ktoré si nedokáže uvedomiť, že výpočet sa vykonáva podozrivo dlho, nevie experimentovať, nemá žiadne skúsenosti a neučí sa z chýb.

Pokusme sa teraz popísať jednotlivé vlastnosti podrobnejšie.

Elementárnosť

Každý postup môže byť zapísaný viacerými spôsobmi. Pri jeho navrhovaní treba dbať na to, aby jednotlivé inštrukcie boli pre adresáta zrozumiteľné, jednoduché a jednoznačné.

Veľmi si napr. zohrievanie mlieka v mikrovlnnej rúre z predchádzajúceho príkladu. Ak sme majiteľmi tohto novodobého zariadenia dlhší čas, nerobí nám príkaz „zohreje mlieko v mikrovlnnej rúre“ problémy. Ak sme ju kúpili nedávno, alebo sme od prírody imúnni voči používaniu technických zariadení, nedokážeme túto činnosť vykonať, pretože pre nás nepredstavuje elementárnu operáciu (skôr ďalší problém).

Rovnako to môže dopadnúť v škole. Už deti na prvom stupni základnej školy vedia násobiť. Ak im však prikážeme: „Zistite šiestu mocninu čísla 2!“, zrejme nebudú vedieť reagovať. No keď im zadáme úlohu v tvare: „Zistite výsledok súčinu 2.2.2.2.2.2!“, bude to pre nich hračkou. Zistenie mocniny pre nich totiž nie je elementárnou činnosťou.

Pokiaľ je algoritmus určený pre človeka, máme veľkú výhodu – človek sa dokáže učiť a na základe predchádzajúcich skúseností je schopný riešiť stále zložitejšie a zložitejšie situácie ako elementárne (napr. pre vodiča začiatočníka je odbočenie na križovatke nekonečným peklom, zatiaľ čo skúsenejší ho považujú za jednoduchú záležitosť).

Pri formulácii si treba dávať pozor aj na jednoznačnosť.

Napr. príkaz: „Meľ dva dni staré rožky!“ alebo „Krájaj týždeň starú kapustu!“ sa dá vysvetliť všelijako. Príkaz „Pridaj cukor!“ alebo „Rozbi dve vajcia!“ môže pri nesprávnom vysvetlení spôsobiť v prvom prípade presladenie a v druhom flaky na koberci alebo na stene.

Determinovanosť

Determinovanosť kladie na postup požiadavku, aby bolo v každom kroku jasné, kam sa má riešenie uberať, čím pokračovať. Človek opäť dokáže pochopiť postup vďaka skúsenosti, no pre počítač musíme určiť postupnosť krokov jednoznačne.

Často rodičia svojim potomkom prikazujú: „*umyt, urobiť za sebou poriadok, vyzliecť a spať*“. Tieto príkazy sú rovnocenné a ak by si ich počítač vysvetlil po svojom mohlo by sa stať, že najprv pôjde spať (vykoná časovo najnáročnejšiu úlohu) a až keď sa vyspí, urobí poriadok, vyzlečie sa a umyje.

Logickým dôsledkom tejto vlastnosti je, že výsledok bude za rovnakých vstupných podmienok, vždy rovnaký. V bežnom živote sa to vždy podarí, nemusí, no ak ten istý postup vykonáva počítač, zvyčajne je táto požiadavka splnená.

Najčastejšie sa s rôznym výsledkom pri rovnakom recepte stretávame v kuchyni. Obed pripravovaný podľa toho istého receptu je raz slaný, inokedy štiplavý alebo nedovarený. Príčinou sú najčastejšie výrazy ako štipka soli, za hrst korenia, chvíľu variť a pod. Výsledok potom závisí od veľkosti štipky, hrsti alebo predstavy kuchára o chvíľke. Rovnako ani pojem dve veľké vajcia nemusí znamenať vždy rovnaké množstvo.

Rezultatívnosť

Rezultatívnosť od algoritmu vyžaduje, aby jeho realizácia dokázateľne viedla po konečnom počte krokov k správne výsledku pri riešení ľubovoľnej zo skupiny úloh, pre ktorú bol vytvorený.

Pri vykonávaní výpočtu na základe nesprávneho algoritmu zvyčajne tiež získame výsledok, ktorý je pre daný algoritmus správny. Problém je „len“ v tom, že tento algoritmus nerieši zadaný problém.

Niečo veľmi podobné sa nám stávalo aj na písomke z matematiky alebo fyziky. Pre rovnaké zadanie vyšlo päť (minimálne) rôznych výsledkov a každý si myslel, že práve ten jeho postup je správny.

Podobný stav môže navodiť i riešenie nasledovného matematického problému.

Troja chlapci si v športovom obchode kúpili loptu. Zaplatili za ňu spolu 30,- Sk. Keď odišli, predavač zistil, že lopta stála len 25,- Sk a so zvyšnými peniazmi poslal za nimi pomocníka. Ten im dal 3,- Sk a 2,- Sk si ponechal „od cesty“. Takže chlapci (keďže každý dostal 1,- Sk nazad) zaplatili za loptu po 9,- Sk a pomocníkovi zostali 2,- Sk.

Spolu: $9 \times 3 + 2 = 29$,- Sk. Kam sa podela zvyšná koruna?

Konečnosť

Splnenie tejto vlastnosti má zabezpečiť, aby sa výpočet vždy skončil. Človek, pracujúci s problémom na základe skúseností dokáže určiť, či jeho výpočet dá alebo nedá výsledok (resp. či skončí alebo nie). Počítač bez skúseností sa na takejto úrovni rozhodnúť nedokáže.

Nahliadnime zasa do kuchyne. Ak má počítač postupovať podľa nasledovných príkazov:

1. polož hrniec s jedlom na varič,
2. pusti plyn,
3. miešaj, kým nezačne vriet,

môže sa stať, že v prípade vypnutia plynu sa bude touto činnosťou zamestnávať najbližších 50 rokov.

Rovnako príkaz: „*Kop, kým nenarazíš na poklad!*“ môže v prípade nesprávneho miesta znamenať prekopávanie sa do Austrálie.

Podobne matematický postup: „*Kým je zadané číslo menšie ako jedna, vynásob ho dvoma!*“ nás v prípade zadania nuly alebo záporného čísla môže priviesť až do pokojnej staroby.

Predchádzajúce príklady boli určite nekonečné, no okrem nich existujú aj problémy, ktorých riešenie je síce konečné, ale nájdenie výsledku trvá veľmi dlho. Typickým príkladom sú šifrovacie algoritmy, keď síce teoreticky dokážeme rozšifrovať každú správu, no doba realizácie je taká dlhá, že obsah správy po rozšifrovaní (napr. po niekoľkých rokoch) stráca zmysel.

Podobné je to s počítaním buniek v ľudskom tele, molekúl v litri vzduchu alebo zrníka piesku na púšti.

Hromadnosť

Ak od postupu vyžadujeme, aby bol hromadný, zvyčajne doň vkladáme nejaké vstupné hodnoty – parametre. Nie každý algoritmus však vie byť hromadný. Niektoré algoritmy sú šité presne na konkrétny problém a nie je možné vstupné parametre meniť, lebo sú zložité alebo jednoducho iné neexistujú. Preto túto vlastnosť považujeme skôr za užitočnú ako nutnú.

Typickým praktickým príkladom využitia hromadnosti je výpočet brzdennej dráhy automobilu pri zadanej rýchlosti, povrchu vozovky a prevážanej hmotnosti. Výsledok dokážeme určiť bez zmien v algoritme, postačí zadať iné vstupné údaje.

Pri vytváraní všeobecných algoritmov nejde o vyriešenie konkrétneho problému, ale skôr o popísanie postupu, podľa ktorého sa výsledok získava.

Ak chceme algoritmicky zapísať výpočet objemu kvádra pre ľubovoľné rozmery, nemôžeme do algoritmu napísať jednoducho $3 \times 8 \times 11$, ale $a \times b \times c$, pričom ako vstupné hodnoty budeme zadávať práve hodnoty a , b a c .

Efektívnosť

Vytvoriť efektívny algoritmus znamená navrhnúť taký postup, ktorý s použitím minimálnych prostriedkov v čo najkratšom čase vyrieši náš problém. Aj algoritmus, ktorý nie je efektívny, je algoritmom, ale ak si zadávateľ, ktorý za vytvorenie algoritmu dokonca platí, môže vybrať, zvolí si určite ten najefektívnejší. Efektívnosť je veľmi dôležitá najmä pri spracúvaní veľkého množstva údajov, kde je rozdiel, či pri spracúvaní 2 000 údajov trvá jedna operácia sekundu alebo dve (rozdiel predstavuje viac ako polhodinu).

Pri zložitých problémoch je prvotným cieľom zvyčajne aspoň vytvorenie algoritmu a až po jeho otestovaní a v prípade potreby vylepšovanie a zrýchľovanie.

Pri prehliadke veliteľ potreboval zistiť počet nastúpených vojakov. Vojaci stáli v 32 radoch po 17. Úlohou poveril dvoch zástupcov. Prvý postupoval nasledovne: $17+17+17+17+\dots+17$, druhý to skúsil ako 17×32 . Čo myslíte, ktorý sa dopracoval k výsledku skôr?

Populárnym príkladom na prezentáciu efektívnosti je aj problém sčítavania čísel 1 až 100. Prvý spôsob, ktorému všetci rozumieme je postupovať $1+2+3+4+\dots+100$. K výsledku sa síce dostaneme, ale ak vezmeme dvojice čísel $1+100$, $2+99$, $3+98$, ..., $50+51$ (spolu ich je 50), vyriešime problém podstatne rýchlejšie: dvojíc je 50, ich súčet je 101, teda $101 \times 50 = 5\,050$.

Ako algoritmizovať?

Či chceme alebo nie, či si to uvedomujeme alebo nie, celý náš život pozostáva z algoritmov – postupov. Tým, že sa ich snažíme zapísať, sledujeme zvyčajne dva ciele:

- vďaka popisu dokážeme vykonávaním algoritmu poveriť iného človeka alebo počítač,
- vďaka vyjadreniu myšlienok sa nám problém stáva zrozumiteľnejším a sme schopní lepšie mu porozumieť a následne ho vylepšiť.

Proces, ktorý pri zápise algoritmu vykonávame, sa nazýva **algoritmizácia**. Na jej začiatku vždy potrebujeme určiť **vstupné podmienky** (napr. rozsah hodnôt, ktoré môžu do algoritmu vstupovať) a **výstupné podmienky** (vlastnosti výsledku). Zadanie algoritmu potom zapisujeme takto:

```
{VST: vstupné podmienky}
?
{VÝS: výstupné podmienky}
```

napr. pre výpočet objemu hranola bude zápis vyzerat' nasledovne:

```
{VST: a, b, c : kladné reálne čísla}
?
{VÝS: V - reálne číslo predstavujúce objem}
```

Algoritmický jazyk

Na to, aby sme mohli s niekým alebo niečím komunikovať, potrebujeme dorozumievací prostriedok – **jazyk**. Jazyk sám osebe však zvyčajne nestačí. Darmo ovládate deväť cudzích jazykov, keď osoba, s ktorou sa potrebujete dohovoriť je Eskimák a jazyk, ktorý používa, sa k tým vašim svojimi výrazovými prostriedkami nepribližuje ani zďaleka.

Na komunikáciu s iným subjektom (človekom, strojom) je potrebné používať jazyk, ktorému rozumie. Pôvodne bol jazyk iba prostriedkom komunikácie medzi ľuďmi, dnes je už aj prostriedkom komunikácie medzi človekom a strojom. Jazyk, pomocou ktorého sa komunikuje so strojom, má však oproti klasickému jazyku určité odlišnosti:

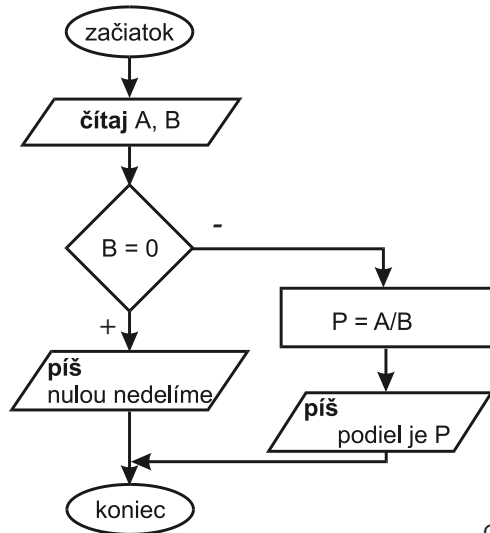
- ľudský jazyk obsahuje množstvo slov (napr. slovenčina pozná vyše 110.000 slov, angličtina takmer 800.000), je v neustálom vývoji, slová v jazyku pribúdajú a zanikajú. Jazyk na komunikáciu so strojom vyžaduje **stabilný a nemenný zoznam umožňujúci presnú špecifikáciu príkazov**, ktoré napr. výrobca do zariadenia implementuje pri jeho výrobe,

- zatiaľ čo ľudské jazyky obsahujú množstvo výnimiek, „umeleckých obrátov“ (frazologizmy, príslovia a porekadlá, zdrobneniny a pod.), synonym (slová s rovnakým významom), homonym (rovnaké slová s odlišným významom, napr. hlava, list, koreň) a tvarov (pády, časovanie slovík, časy a neurčitok), v strojových jazykoch je vyžadovaná **presnosť, konkrétnosť a adresnosť**,
- prirodzený ľudský jazyk obsahuje množstvo prvkov a konštrukcií, ktoré sú pri špecifikovaní postupov zbytočné.

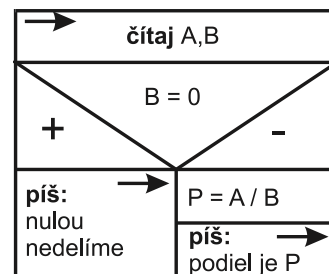
Nemožnosť využitia prirodzeného jazyka v komunikácii so strojom viedla k potrebe úpravy prirodzeného jazyka a redukcii jeho obsahu na úzku skupinu slov, pomocou ktorých je možné požadovanú činnosť popísať a špecifikovať jej parametre.

Podobne ako sa nepodarilo ľuďstvu dohovoriť na jednotnom jazyku (známym pokusom je napr. esperanto), ani pri algoritmických jazykoch nedošlo k dohode a vo všeobecnosti sa používa viacero z nich. Môžeme ich rozdeliť na:

- **graficky orientované**, kam patria:
 - **vývojové diagramy**, kde je postupnosť činností popisovaná prostredníctvom grafických značiek a textu v nich, pričom tok výpočtu je znázornený šípkami,



- **štruktúrogramy** sa skladajú z blokov základných algoritmických štruktúr, ktoré sa môžu do seba vnárať alebo radiť za sebou do postupnosti. Oproti vývojovým diagramom nie sú definované normou,



Obr. 1 Vývojový diagram a štruktúrogram

- **obrázkové jazyky** umožňujú programovať prostredníctvom spájania obrázkov, hlavným reprezentantom sú detské programovacie jazyky napr. *Baltik* alebo programovanie *RCX* kocky *Lega* v jazyku *LabView*,
- **textovo orientované** využívajú na popis činnosti text a prostredníctvom neho zapísané príkazy:
 - **programovacie jazyky** sú v oblasti algoritmizácie najpoužívanejším prostriedkom a predstavujú formalizované algoritmické jazyky s presne definovanou syntaxou,
 - **slovný zápis algoritmu v prirodzenom jazyku** býva často nejednoznačný, preto sa pre zápis algoritmov používajú len určité dohodnuté formulácie, od ktorých je už len krok k nasledujúcemu typu,
 - **rozhodovacie tabuľky** nepopisujú činnosti v poradí, v akom sa majú vykonať, ale definujú v tabuľke to, čo sa má robiť pre rôzne kombinácie hodnôt premenných.

1. Definujte pojem problém a pojem algoritmus.
2. Ktoré vlastnosti sú pre algoritmus povinné, a ktoré „vhodné“?
3. Popíšte spoločné a rozdielne črty rezultatívnosti a hromadnosti.
4. Prečo nie je štandardný „ľudský“ jazyk vhodným prostriedkom na komunikáciu s nemysliacim zariadením?
5. Navrhňte vlastný algoritmický jazyk pre zvolené nemysliace zariadenie.

Lekcia 2

2 Algoritmické štruktúry

predpoklady na zvládnutie lekcie:

- oboznámenie sa s pojmami algoritmus a algoritmický jazyk

obsah lekcie:

- algoritmické štruktúry – sekvencia, vetvenie, cyklus
- vývojové diagramy ako forma zápisu algoritmov
- pojem premenná
- príklady a popis hotových algoritmov

cieľ:

- naučiť sa transformovať úlohy do algoritmického jazyka

Na to, aby sme dokázali komunikovať prostredníctvom algoritmického jazyka, potrebujeme mať stanovené príkazy, ktorými dokážeme prikázať procesoru vykonať presne stanovené činnosti. Štandardne predstavuje príkaz elementárnu činnosť, ktorú je schopný vykonávať algoritmu realizovať. Vykonávaní príkazov v takom poradí, v akom sú zapísané, hovoríme **sekvencia**.

V niektorých prípadoch je potrebné zabezpečiť vykonanie príkazu len pri splnení definovaných podmienok. Možnosť rozhodnúť sa a vykonať príkazy na základe pravdivosti skúmaného znaku sa označuje ako **vetvenie**. Skladá sa z podmienky a z príkazov, ktoré sa vykonajú v prípade kladného a záporného výsledku.

Veľmi často potrebujeme časť algoritmu opakovať. Zápis umožňujúci opakovanie označujeme ako **cyklus**. Pri každom opakovaní je dôležité čo (telo cyklu) sa má opakovať a *dokedy* (podmienka cyklu) sa má opakovať.

Sekvenciu, vetvenie a cyklus označujeme ako **algoritmické konštrukcie** a platí, že každý algoritmus dokážeme zapísať ich vhodnou kombináciou.

Vývojové diagramy

Ako sme už spomenuli, zrejme najbližším jazykom je pre každého človeka jeho prirodzený jazyk. V algoritmizácii nie je však ani zďaleka najvhodnejším na zápis algoritmov, pretože je najmä pre začiatočníkov príliš neprehľadný. Pre túto kategóriu používateľov poskytujú omnoho realnejšiu predstavu o tom, ako algoritmus pracuje, skôr zápisy využívajúce grafické prvky, ktoré už svojím vzhľadom napovedajú o aký je ich účel.

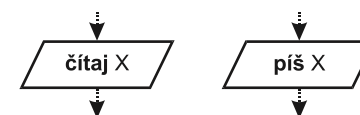
Najvhodnejším grafickým jazykom pre začiatočníkov sú vývojové diagramy umožňujúce intuitívne chápať postup a tok výpočtu znázornený šípkami aj bez vysvetľovania syntaxe.

Príkazy vstupu a výstupu

Jednou z vlastností algoritmu je hromadnosť, ktorá mu prisudzuje schopnosť byť použiteľný na riešenie problému so všetkými prípustnými hodnotami vstupných údajov. Prvým predpokladom, ktorý nám umožní využiť algoritmus skutočne hromadne, je možnosť vkladania **rôznych** vstupných údajov. Na získanie údajov od používateľa využíva algoritmus príkazy vstupu, prostredníctvom ktorých umiestni hodnoty zadané používateľom do premenných uvedených na vstupe.

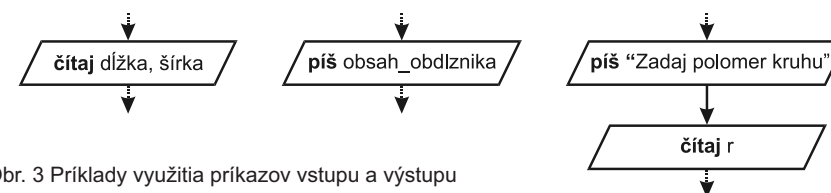
S prečítanými hodnotami potom realizuje predpísané operácie a výsledok vypíše (prípadne iným spôsobom zobrazí) prostredníctvom príkazov výstupu.

Pre operácie vstupu a výstupu využívajú vývojové diagramy bloky



Obr. 2 Príkazy pre vstup a výstup

Príkladom môže byť:



Obr. 3 Príklady využitia príkazov vstupu a výstupu

Premenná

Premenná je objekt (môžeme ju považovať za nejakú pamäť alebo miesto v pamäti) slúžiaci počas behu algoritmu na odkladanie údajov. Jej hodnota sa počas činnosti algoritmu môže meniť (a zvyčajne sa aj mení).

Môže obsahovať číslo, znak, textový reťazec a pod. Každá premenná má svoje meno (napr. obsah, dlzka, a2, priemer a pod.), ktoré zvyčajne začína písmenom a neobsahuje diakritiku.

Je žiaduce, najmä pri rozsiahlejších algoritmoch a neskôr programoch, voliť také mená premenných, z ktorých je okamžite zrejmé ich použitie, napr.

obsah, prepona, menoUcitela a nie xxx, pes, bryndza (i keď v niektorých prípadoch môže byť i takýto názov opodstatnený).

V nasledujúcich riadkoch budeme dodržiavať zásadu, že názvy jednoslovnej premennej budú zapísané malými písmenami a v prípade viacslovných názvov bude prvé slovo zapísané písmenami malými a každé prvé písmeno ďalšieho slova bude veľké (napr. obsahKruhu, menoPouzivatelaPocitaca). Podotýkame, že ide len o dohodu a jeden z používaných zápisov. Osvedčil sa nám najmä v programátorskej praxi, kde predstavuje časovú úsporu pri čítaní programového kódu a zvyšuje prehľadnosť napr. oproti zápisu používajúcemu oddeľovanie jednotlivých slov podčiarkovníkom (obsah_kruhu, meno_pouzivatela_pocitaca).

Premenná nadobúda hodnoty priradením alebo načítaním. Načítanie je realizované prostredníctvom operácií vstupu, priradenie umiestni (priradí) do premennej konkrétnu hodnotu priradovacím príkazom.

Priradovací príkaz budeme označovať symbolom „:=“. Platí, že do premennej uvedenej naľavo od symbolu priradenia vkladáme (priradíme) hodnotu alebo výsledok výpočtu uvedený na strane pravej - jej hodnota sa teda zmení. Premenné uvedené napravo od symbolu priradenia svoju hodnotu pre výpočet len poskytujú - ich obsah sa týmto použitím nemení.

Často sa príkaz priradenia pletie s rovnosťou známou z matematiky, ktorá sa v Pascale používa v podmienkach. Symbol „=“ v nasledujúcom texte predstavuje porovnanie, symbol „:=“ priradenie.

Priradenie môže byť realizované prostredníctvom:

- priradenia konkrétnej hodnoty (číselnej, textovej alebo inej), napr.:

```
polomer := 15
meno := 'Jozef'
```

- priradenia obsahu (hodnoty) inej premennej, napr. :

```
novyPolomer := polomer
meno2 := meno
```

Postup, ktorý algoritmus vykoná je veľmi jednoduchý: vezme hodnotu premennej umiestnenej na pravej strane a vloží ju do premennej uvedenej na strane ľavej.

- výpočtom, resp. zistením hodnoty výrazu zapísaného na pravej strane, do premennej sa vloží len získaná hodnota, nie spôsob výpočtu:

```
cislo := 15 + 12
obsah := vyska * sirka
obsahKruhu := 3,14 * polomer * polomer
vypocet := a * b + c - 3 * (d + f)
x := x + 1
```

Postup algoritmu: vyhodnocuje výraz na pravej strane rovnako ako matematický výraz t.j. uprednostňuje výpočty v zátvorkách i násobenie a delenie pred sčítaním a odčítaním. Namiesto mena premennej vždy dosadí hodnotu, ktorú premenná aktuálne obsahuje.

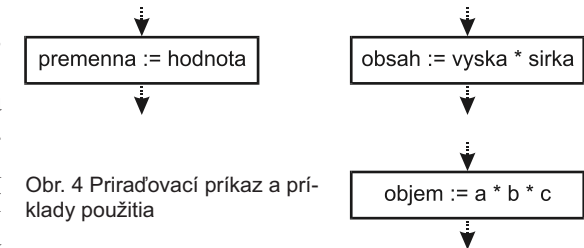
Nesprávne zápisy:

```
12345 := meno
a + b := c
```

Postupnosť znakov uvedených na ľavej strane ani v jednom prípade nepredstavuje platné meno premennej (v prvom prípade nezačína písmenom, v druhom obsahuje nepovolený znak „+“). Nezávisle na tom, čo chcel autor týmito zápsmi dosiahnuť, sú nesprávne.

Operácia priradenia využíva pre svoj zápis vo vývojových diagramoch tiež vlastnú značku.

Okrem premenných, ktoré svoje hodnoty počas behu algoritmu môžu meniť, sa niekedy používajú aj konštanty. Ich hodnota sa spravidla určí na začiatku algoritmu (programu) a zostáva nemenná - nie je možné vykonať do nich priradenie. Známe sú konštanty z matematiky a fyziky, napr. π , g , G atď.

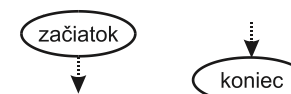


Obr. 4 Priradovací príkaz a príklady použitia

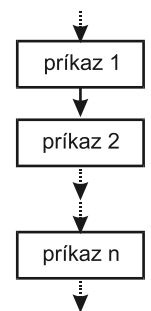
Sekvencia

Postupnosťou jednoduchých príkazov vstupu, niekoľkými priradeniami a operáciou výstupu dokážeme vyriešiť jednoduché problémy a zapísať pre ne algoritmy. Vo všeobecnosti možno sekvenciu zapísať ako postupnosť príkazov.

Ak vytvárame komplexné algoritmy, je potrebné uviesť ich vstupné a výstupné podmienky a označiť začiatok i koniec algoritmu. Na označenie začiatku a konca algoritmu vývojové diagramy používajú tiež samostatné značky.



Obr. 6 Označenie začiatku a konca algoritmu



Obr. 5 Postupnosť príkazov

V tomto okamihu sme sa dostali do štádia, keď naše vedomosti postačujú na vytvorenie jednoduchých sekvenčných algoritmov.

Napište algoritmus na výpočet obsahu a obvodu obdĺžnika.

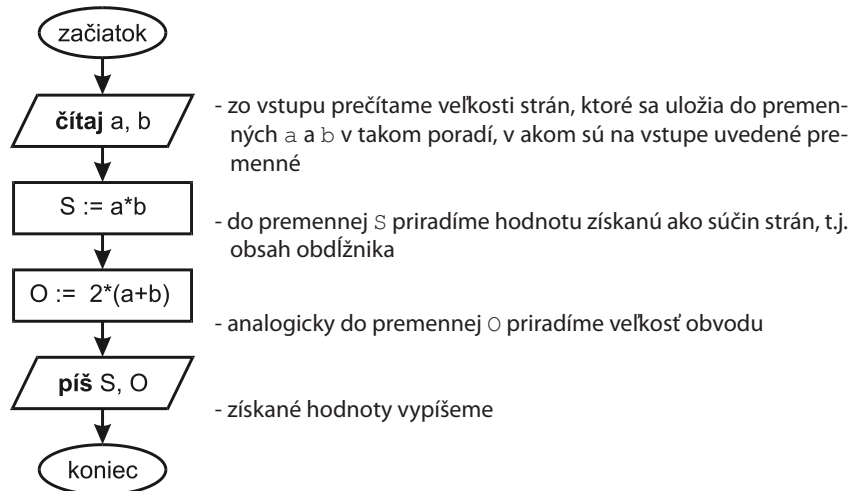
Vstupom do algoritmu budú rozmery strán obdĺžnika, výstupom požadované hodnoty obsahu a obvodu obdĺžnika:

{VST: $a, b > 0$ – strany obdĺžnika}

?

{VÝS: S – obsah obdĺžnika, O – obvod obdĺžnika}

Samotný algoritmus bude pozostávať zo štyroch krokov:



Obr. 7 Výpočet obsahu a obvodu obdĺžnika

V príklade sme použili také názvy premenných, na aké sme zvyknutí v matematike, no pokojne sme ich mohli nazvať *sírka*, *vyska*, *obsah* a *obvod* (v zložitejších príkladoch sú takéto názvy žiaduce kvôli lepšej prehľadnosti).

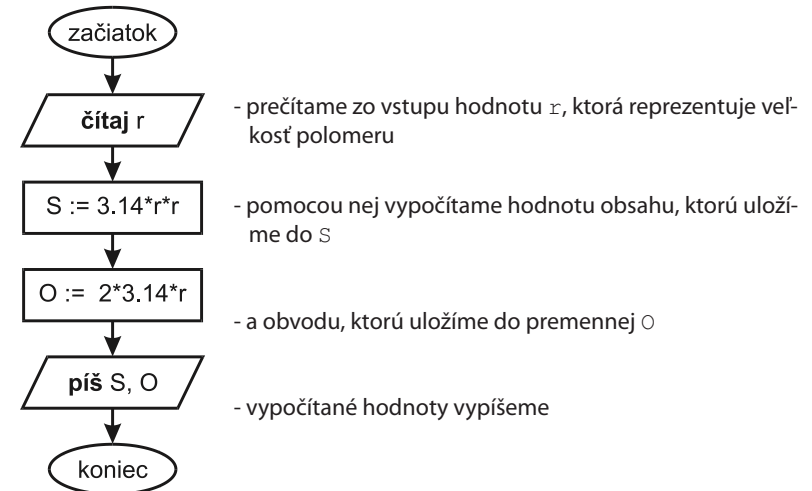
Napište algoritmus, ktorý pre zadaný polomer vypočíta obsah a obvod kruhu.

Riešime úplne analogicky:

{VST: $r > 0$ – polomer kruhu}

?

{VÝS: S – obsah kruhu, O – obvod kruhu}



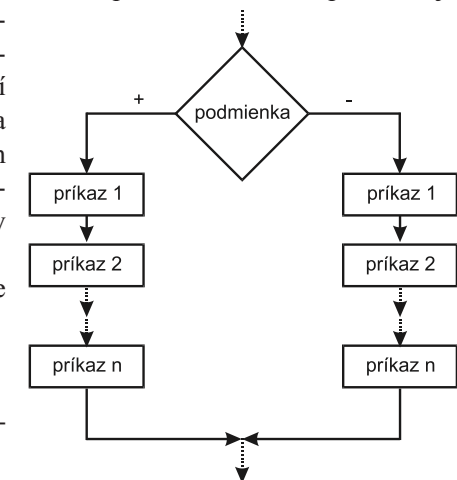
Obr. 8 Výpočet obsahu a obvodu kruhu

1. Napište algoritmus, ktorý pre zadané rozmery hrán zistí povrch a objem kvádra.
2. Napište algoritmus na zistenie vzdialenosti, ktorú prejde lietadlo pri zadanej rýchlosti a dobe letu.

Vetvenie

Vetvenie je v algoritmizácii reprezentované podmienkou, ktorá predstavuje možnosť rozhodnúť sa podľa pravdivosti skúmaného znaku. V závislosti od jej splnenia sa postup vetví na rôzne prípady. Ak je podmienka splnená, pokračuje sa vykonávaním vetvy označenej ako „+“, v opačnom prípade sa spracúvajú príkazy vo vetve „-“.

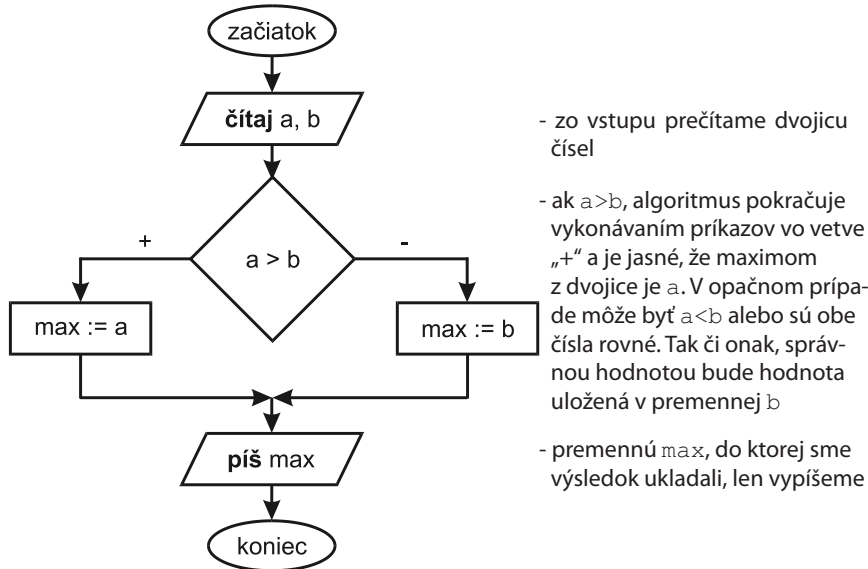
Vo všeobecnosti možno vetvenie zapísať ako na obrázku.



Obr. 9 Všeobecné vyjadrenie úplného vetvenia

Napište algoritmus na nájdenie maxima z dvoch čísel.

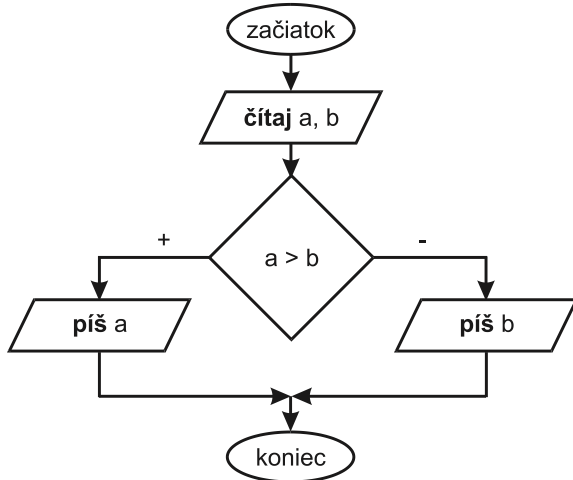
{VST: a, b – ľubovoľné čísla}
?
{VÝS: max – maximum z dvojice}



Obr. 10 Algoritmus pre nájdenie maxima

Algoritmus by sme mohli vytvoriť i priamym výpisom hodnoty bez použitia ďalšej premennej.

{VST: a, b – ľubovoľné čísla}
?
{VÝS: maximum z dvojice}

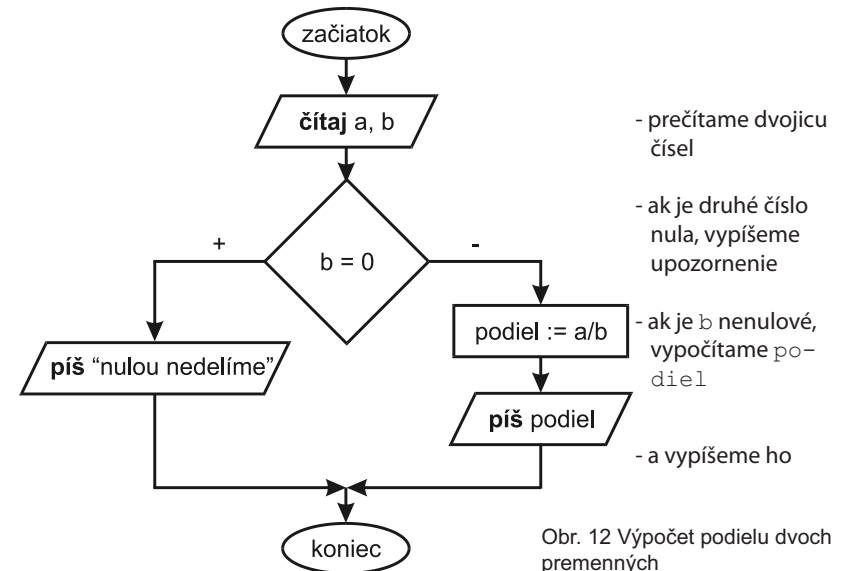


Obr. 11 Algoritmus pre nájdenie maxima bez použitia tretej premennej

Napište algoritmus na zistenie podielu dvoch čísel.

Zistiť podiel znamená vydeliť dve čísla. Mohli by sme prečítať čísla, priradiť ich podiel do premennej a vypísať ho, ale môže nastať prípad, že ako deliteľa zadáme nulu. A už v druhej triede na základnej škole nás učili, že nulou deliť nemožno. Preto potrebujeme takúto situáciu **ošetriť** – v prípade zadania nuly sa vypíše text „Nulou nedelíme!“, inak sa vypočíta podiel a vypíše sa.

{VST: a, b – ľubovoľné čísla}
?
{VÝS: podiel – podiel dvojice}



Obr. 12 Výpočet podielu dvoch premenných

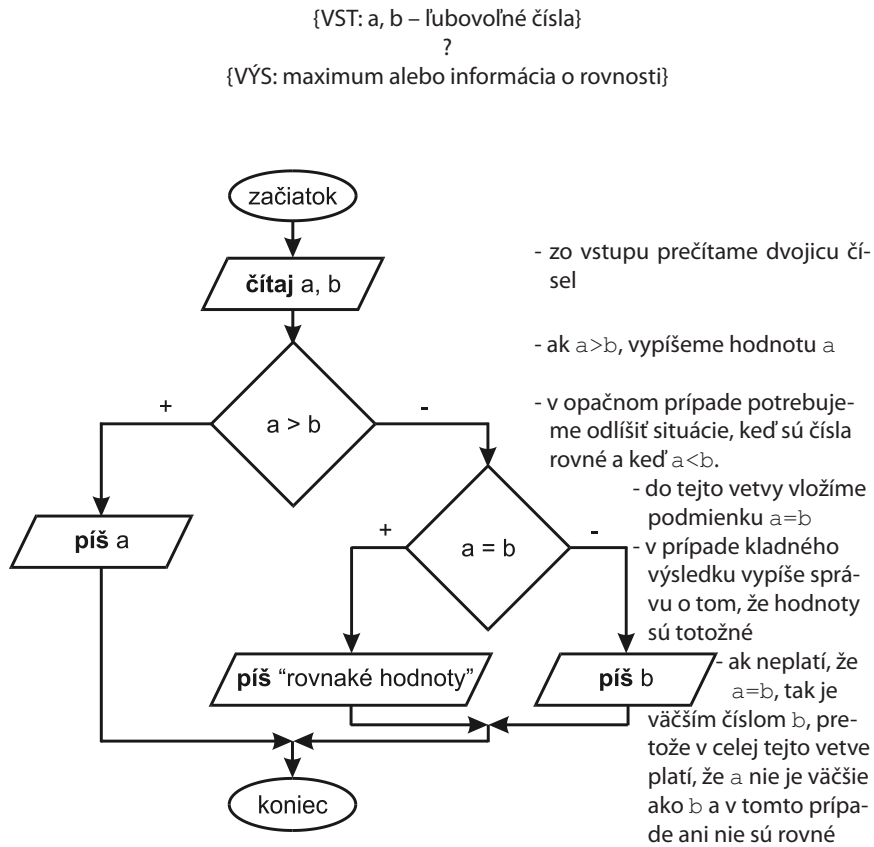
Pokiaľ je text za príkazom piš v úvodzovkách, vypíše sa presne to, čo je v úvodzovkách. Ak text za piš v úvodzovkách nie je, považuje sa za premennú.

Napište algoritmus na nájdenie maxima z dvoch čísel a v prípade, že sú rovnaké, podajte o tom informáciu.

Ak potrebujeme vetviť postup na viacero rôznych riešení v závislosti od podmienky, vkladáme viacero alternatív “do seba”.

Vetvenie, v ktorom sa príkazy vykonávajú ako v kladnej, tak i v zápornej vetve sa označuje ako úplné, no často sa môžeme stretnúť i so situáciou, že

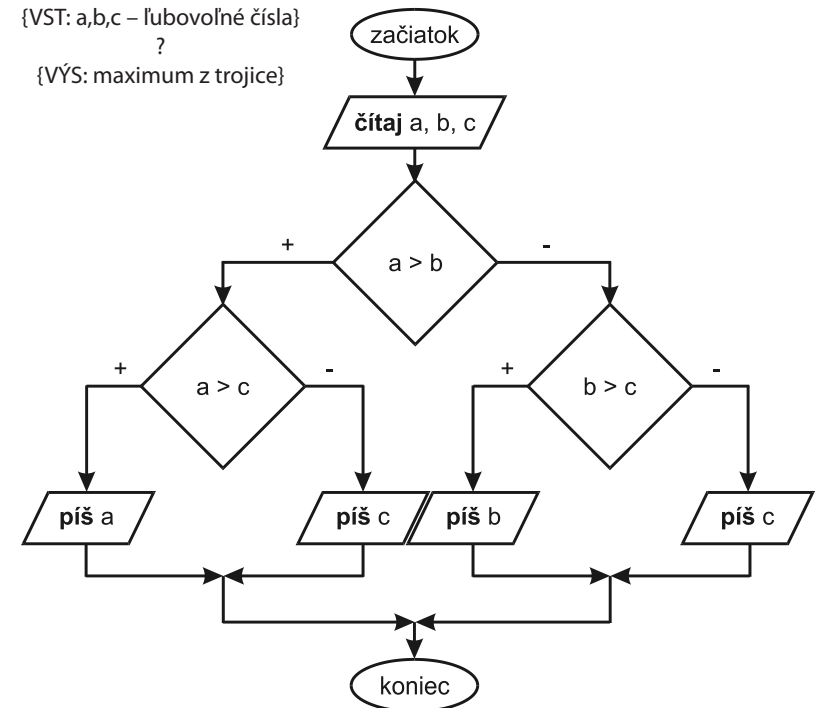
príkazy sú uvedené len v prípade splnenia či nespĺnenia podmienky. Takéto vetvenie sa označuje ako neúplné, no neznamená to, že je menejcenné – veľmi často totiž nie je potrebné niektoré príkazy pri splnení podmienky vykonávať.



Obr. 13 Ďalší algoritmus na nájdenie maxima

Napište algoritmus na nájdenie maxima z troch čísel.

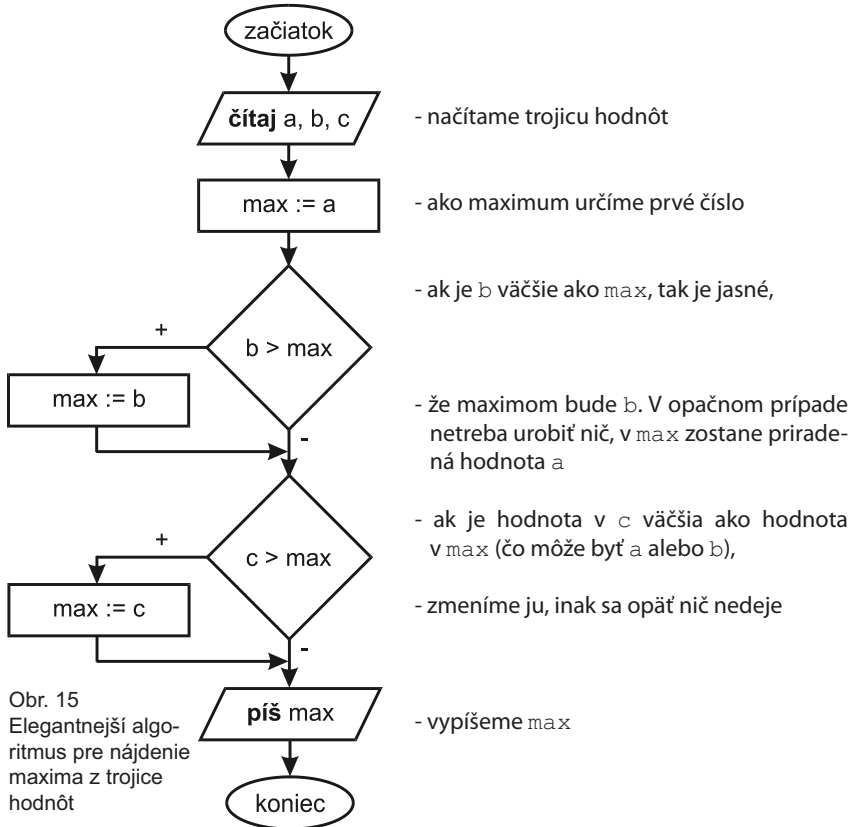
Prvé riešenie by mohlo vyzerat' nasledovne:



Obr. 14 Nájdenie maxima z trojice hodnôt

Ak sa však zamyslíme nad energiou, ktorú sme museli venovať uvedomeniu si situácie v jednotlivých vetvách, určite uprednostníme nasledujúce, na prvý pohľad možno dlhšie, no na rozmýšľanie i na možnosť vyvarovať sa chyby určite vhodnejšie riešenie.

{VST: a,b,c – ľubovoľné čísla}
?
{VÝS: maximum z trojice}

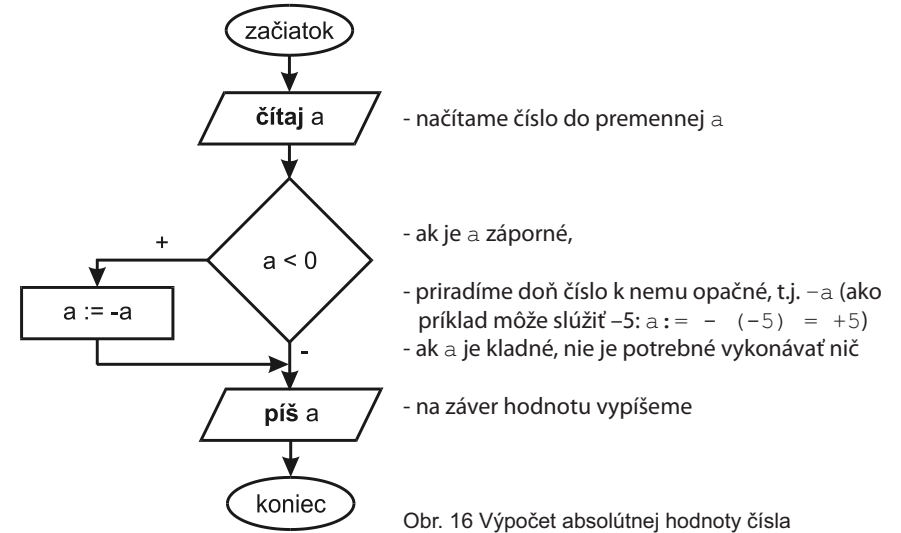


Záver: nie vždy platí, že najkratšie či na prvý pohľad zrejmé riešenie je najvhodnejšie.

Napište algoritmus na zistenie absolútnej hodnoty zadaného čísla.

Opäť ide o jednoduchý príklad s neúplným vetvením – ako relatívne nový prvok si dovoľme použiť tú istú premennú na načítanie hodnoty i výpis výsledku.

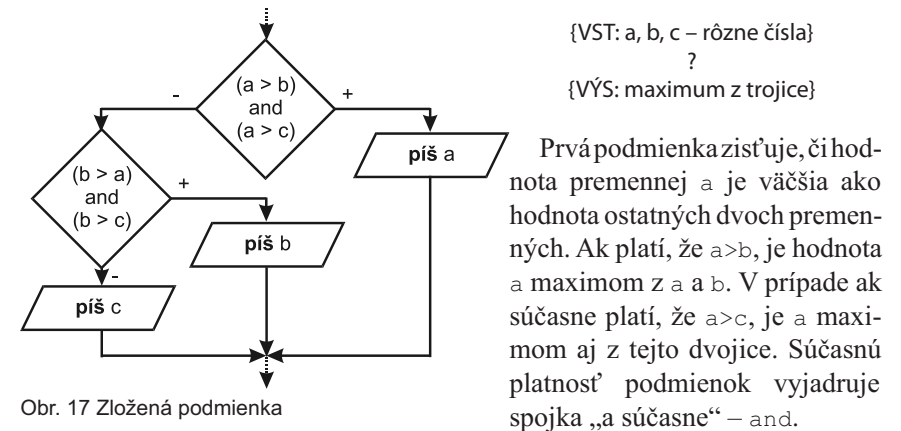
{VST: a – ľubovoľné číslo}
?
{VÝS: absolútna hodnota z a}



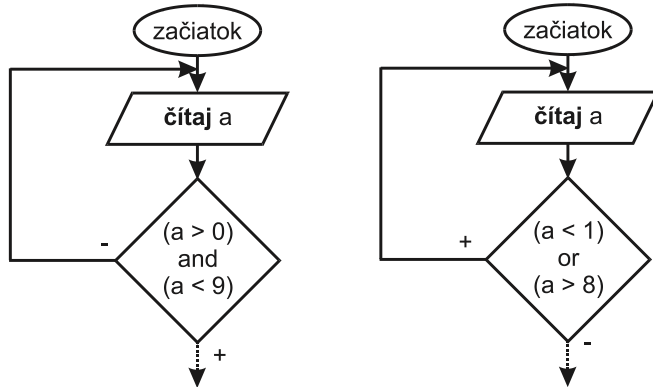
Zložitejšie podmienky

Zložená podmienka je v programovacích jazykoch chápaná ako logický výraz určujúci vzťahy medzi výrazmi, prípadne predstavujúci viac podmienok zviazaných logickými operátormi (and = “a súčasne”, or = “alebo” a not = “neplatí, že”). Spôsob použitia prezentujeme v nasledujúcich príkladoch.

Napište algoritmus, ktorý prostredníctvom zloženej podmienky nájde najväčšiu z trojice rôznych hodnôt.



Napište algoritmus, ktorý na vstupe ošetrí, či zadaná hodnota je z intervalu 1-8.



Obr. 18 Požiadavka na zadanie hodnoty z intervalu

Dvojica algoritmov rieši rovnaký problém. Zatiaľ čo v prvom prípade vyžadujeme, aby hodnota bola väčšia ako 0 a súčasne menšia ako 9 (použitá je spojka `and` – a súčasne), v druhom sme ju ochotní akceptovať, ak neplatí, že je menšia ako 1 alebo väčšia ako 8, na čo sme použili logickú spojku `or` (alebo).

Výsledný efekt je v oboch prípadoch rovnaký, rozdiel je, že v prvom prípade algoritmus pokračuje pri splnení podmienky, v druhom pri jej nespnení.

1. Napište algoritmus, ktorý pomocou jedinej zloženej podmienky zistí, či tri číselné hodnoty zadané na vstupe sú totožné.
2. Napište algoritmus, ktorý prostredníctvom jedinej zloženej podmienky zistí, či sa zadané číslo x nachádza v intervale zadanom hodnotami a a b .
3. Napište algoritmus, ktorý načíta strany trojuholníka a vypíše, či daný trojuholník existuje, ak áno, či je pravouhlý, rovnostranný, rovnoramenný alebo nemá ani jednu spomínanú vlastnosť.
4. Napište algoritmus, ktorý na základe zadania hodiny (1-12) a obdobia (a.m. = predpoludním, p.m. = popoludní) určí či je deň alebo noc, resp. svetlo alebo tma (predpokladajme, že slnko vychádza i zapadá o 6.00),

Cyklus

Cyklus nám poskytuje prostriedok umožňujúci opakovať činnosť alebo činnosti. Pri jeho použití je potrebné vedieť, čo sa má opakovať a **dokedy** sa to má opakovať. Činnosť, ktorá sa opakuje, označujeme ako **telo cyklu**, podmienku, ktorá určuje dokedy sa bude telo cyklu opakovať, nazývame **podmienka cyklu**.

V závislosti od vzťahu medzi telom a podmienkou cyklu môžeme cykly rozdeliť na:

- cyklus so známym počtom opakovaní,
- cyklus s podmienkou na začiatku,
- cyklus s podmienkou na konci.

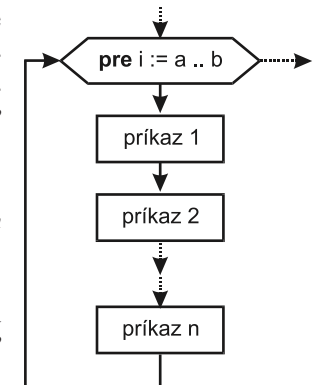
Cyklus so známym počtom opakovaní

Predpokladom využitia takéhoto cyklu je, že počet opakovaní sa dá vyjadriť pred jeho odštartovaním a operácie v tele naň nemajú žiaden vplyv. Vo všeobecnosti ho možno zapísať ako na obrázku.

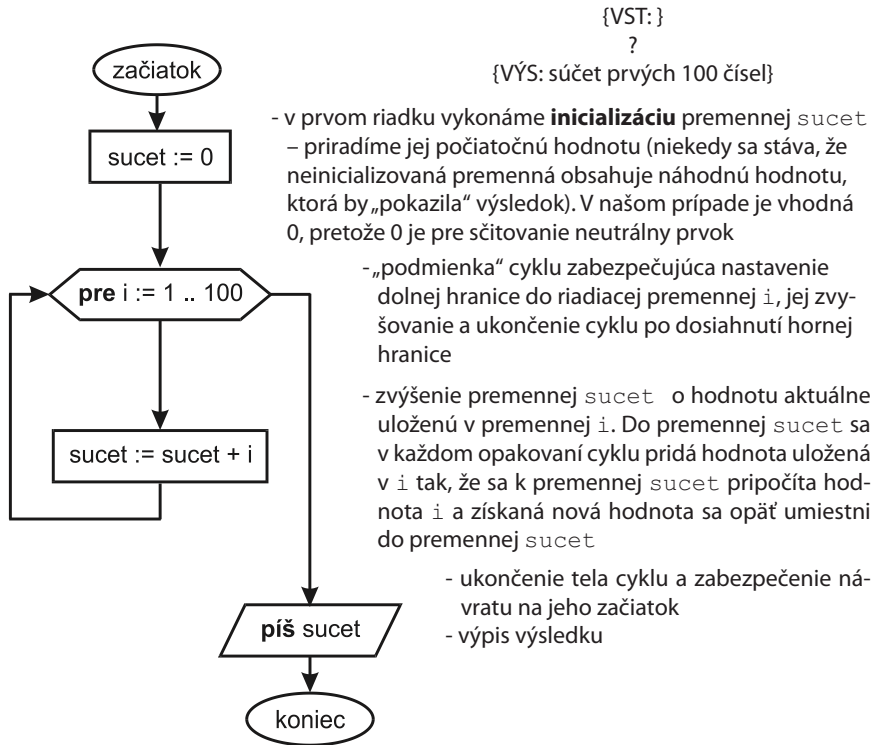
Napište algoritmus na zistenie súčtu prvých 100 čísel.

V tomto prípade nepoužijeme fintu, ktorú sme si ukázali vyššie, ale budeme pripočítavať čísla po jednom.

Vstup do algoritmu nepotrebujeme, napíšeme ho nehromadne tak, že vždy sčíta len prvých 100 čísel. Využijeme cyklus, ktorého počet opakovaní poznáme (100). Cyklus so známym počtom opakovaní používa premennú, hovorí sa jej **riadiaca premenná**, ktorá si „pamätá“ koľkokrát cyklus prebehol. V zápise cyklu určíme jej dolnú a hornú hranicu (pre i od 1 po 100). Po skončení tela cyklu sa automaticky pripočíta hodnota 1 a skočí sa na začiatok cyklu. Riadiacu premennú môžeme v tele cyklu používať ako hociktorú inú, neodporúča sa však priradovať do nej hodnoty (mohlo by sa stať, že cyklus skončí skôr ako predpokladáme alebo bude nekonečný, ak riadiaca premenná nedosiahne hornú hranicu cyklu).



Obr. 19 Všeobecné vyjadrenie cyklu so známym počtom opakovaní



Obr. 20 Algoritmus na zistenie súčtu prvých 100 čísel

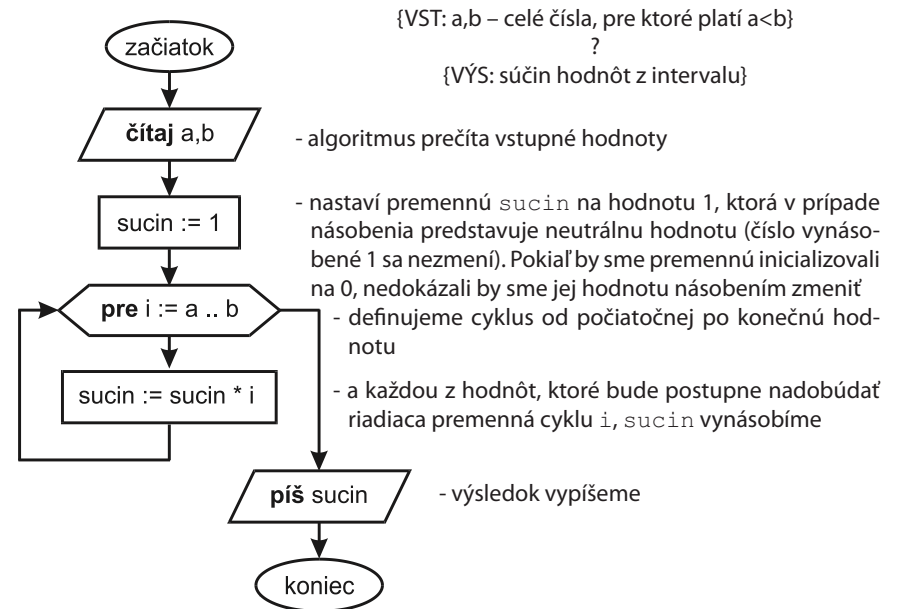
Na sledovanie hodnôt premenných a overovanie správnosti algoritmu sa často používajú sledovacie tabuľky, prostredníctvom ktorých sa nám často podarí pochopiť činnosť algoritmu oveľa jednoduchšie ako siahodlým slovným vysvetľovaním. Na ozrejmienie činnosti cyklu si jednu uvedieme i na tomto mieste.

i	poznámka	sucet
	pred začiatkom cyklu	0
1	sucet:=sucet+1, t.j. sucet:=0+1	1
2	sucet:=sucet+2, t.j. sucet:=1+2	3
3	sucet:=sucet+3, t.j. sucet:=3+3	6
4	sucet:=sucet+4, t.j. sucet:=6+4	10
...		
99	sucet:=sucet+99, t.j. sucet:=4851+99	4950
100	sucet:=sucet+100, t.j. sucet:=4950+100	5050

Tab. 1 Sledovacia tabuľka pre algoritmus na zistenie súčtu prvých 100 čísel

Upravte algoritmus tak, aby bolo možné zadať počet čísel, ktoré sa majú sčítať (na začiatku prečítajte napr. *n* a nahraďte ním číslo 100 v cykle).

Napište algoritmus na zistenie súčtinu celých čísel nachádzajúcich sa medzi dvoma zadanými hodnotami, napr. pre hodnoty 5 a 7 bude výsledkom 5*6*7.



Obr. 21 Algoritmus na zistenie súčtinu hodnôt v zadanom intervale

i	poznámka	sucin
	pred začiatkom cyklu, i ide od 5 do 7	1
5	sucin:=sucin*5, t.j. sucin:=1*5	5
6	sucin:=sucin*6, t.j. sucin:=5*6	30
7	sucin:=sucin*7, t.j. sucin:=30*7	210
8	bola dosiahnutá horná hranica, cyklus končí	210

Tab. 2 Sledovacia tabuľka pre hodnoty a=5, b=7

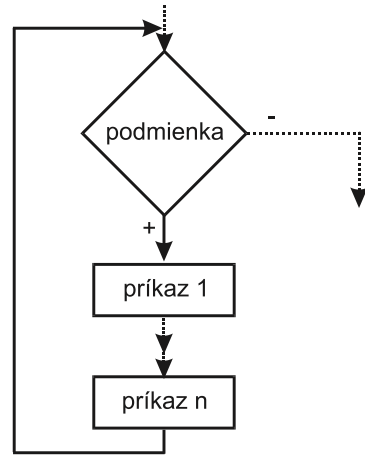
Napište algoritmus na zistenie súčtinu dvoch celých čísel pre zariadenie, ktoré nepozná operáciu násobenia (nahraďte ju sčítovaním v cykle).

Prostredníctvom cyklu so známym počtom opakovaní dokážeme vyriešiť všetky problémy, no niekedy je pohodlnejšie, prehľadnejšie a najmä v súlade so zásadami štruktúrovaného programovania, použitie ďalších dvoch typov cyklov. Tieto využívame vtedy, keď nám počet opakovaní nie je známy v momente vstupu do cyklu a/alebo ukončenie cyklu ovplyvňujú operácie v jeho tele. Kontrolu ukončenia cyklu môžeme realizovať:

- pred vykonaním tela cyklu – cyklus s podmienkou na začiatku,
- po vykonaní tela cyklu – cyklus s podmienkou na konci.

Cyklus s podmienkou na začiatku

Tento typ cyklu má podmienku, ktorá sa stará o ukončenie cyklu, umiestnenú pred telom. Ak je podmienka splnená, vykoná sa telo cyklu a opäť sa otestuje. Ak „vstupná“ podmienka nie je splnená už pri prvom vstupe do cyklu, nemusí sa tento vykonať vôbec. Graficky vyzerá zápis ako na obrázku.



Obr. 22 Všeobecné vyjadrenie cyklu s podmienkou na začiatku

Napište algoritmus, ktorý zistí zvyšok pri delení dvoch čísel.

Pre používateľov by bolo najjednoduchšie riešiť túto úlohu ručne. Počítač však natoľko inteligentný nie je a v prípade štandardného delenia môžeme získať len desiatinné číslo. Čo s tým?

Ak si uvedomíme, že delenie je vlastne zistenie počtu výskytov jedného čísla v druhom, môžeme zvyšok nájsť tak, že budeme od prvého čísla (delenca) odpočítavať druhé číslo (deliteľ) dovtedy, kým nebude zostatok menší ako deliteľ (prípadne nebude rovný 0).

Vezmime napr. 7 a 2.

$7-2=5$, 5 je väčšie ako 2, odčítavať môžeme ďalej,

$5-2=3$, od 3 stále dokážeme 2 odčítať,

$3-2=1$ a 1 je už menšie číslo ako 2, takže zostane – je zvyškom.

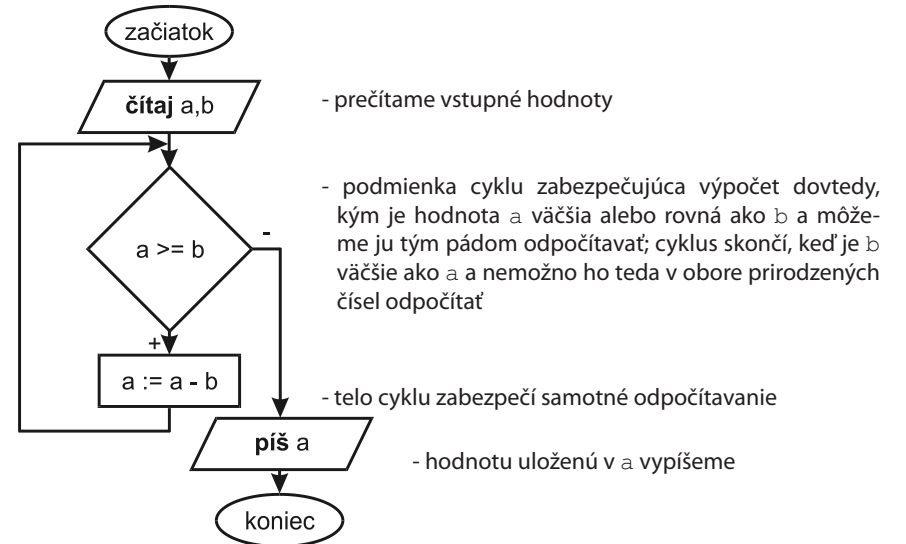
Algoritmus zovšeobecníme a realizujeme ho prostredníctvom cyklu (odčítavanie sa opakuje). Presný počet odčítaní nepoznáme, teda cyklus so známym počtom opakovaní môžeme zavrhnúť.

Podmienkou ukončenia bude test, či číslo, ktoré nám zostalo, je už menšie ako deliteľ. Kam dať podmienku? Môže sa stať, že delenec je menší ako deliteľ už na vstupe a vtedy by žiadne delenie prebehnúť nemalo – teda podmienka bude na začiatku.

{VST: a,b – celé kladné čísla}

?

{VÝS: zvyšok po vydelení a:b}



Obr. 23 Algoritmus na zistenie zvyšku pri delení

a	b	vyhodnotenie podmienky	nová hodnota a
15	4	$15 > 4$ - podmienka splnená	$15 - 4 = 11$
11	4	$11 > 4$ - podmienka splnená	$11 - 4 = 7$
7	4	$7 > 4$ - podmienka splnená	$7 - 4 = 3$
3	4	$3 > 4$ - podmienka nespĺnená - výpis a=3	3

Tab. 3 Sledovacia tabuľka pre hodnoty a=15, b=4

Pridajte do algoritmu výpočet celočíselného podielu (napr. pri každom odčítaní zväčšiť podiel o 1).

Algoritmické i programovacie jazyky už v sebe zvyčajne funkcie pre celočíselné delenie a zistenie zvyšku po celočíselnom delení obsahujú. Ak chce-

me dve čísla vydeliť, použijeme namiesto „/“ operáciu **div** ($7 \text{ div } 3$ je 2), ak chceme zistiť zvyšok, máme k dispozícii operáciu **mod** ($7 \text{ mod } 3$ je 1). Pri použití týchto operácií sa naše podmienkové algoritmy zmenia na štvorriadkovú sekvenciu.

1. Upravte predchádzajúci algoritmus podľa nových vedomostí.
2. Vypočítajte ciferný súčet čísl daného prirodzeného čísla N (využite fakt, že poslednú cifru z čísla viete „odtrhnúť“ prostredníctvom operácie mod).

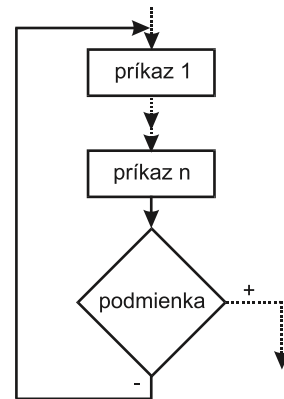
Cyklus s podmienkou na konci

Tento cyklus na prvý pohľad vyzerá oproti cyklu s podmienkou na začiatku ako opačný – najprv sa vykoná telo cyklu a až potom sa zisťuje splnenie podmienky. Ak je podmienka cyklu splnená, vykonávanie cyklu sa ukončí, v opačnom prípade sa pokračuje opätovným vykonávaním tela cyklu. Dôsledkom takéhoto riadenia je, že cyklus vždy prebehne minimálne raz.

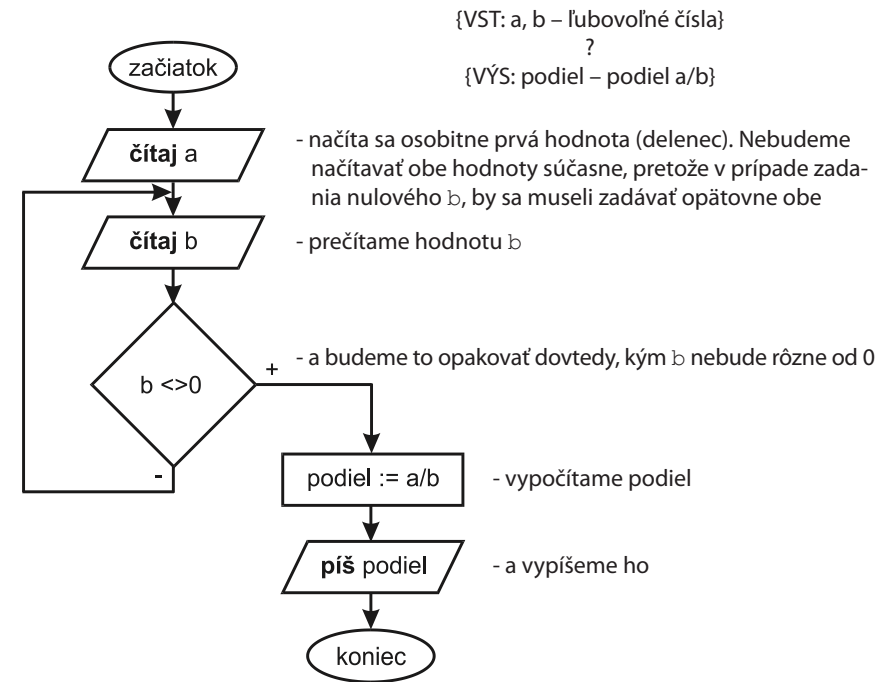
Tento cyklus môžeme označiť ako „neopatrný“: najskôr sa niečo vykoná, potom sa rozhoduje, či to bolo dobre; cyklus s podmienkou na začiatku sa naproti tomu niekedy nazýva ako opatrný – najprv otestuje platnosť podmienky a až potom sa vykonáva.

Tento typ cyklu sa často využíva napr. pri vkladaní vstupných hodnôt a testovaní ich správnosti – v prípade algoritmov síce vieme určiť vstupné podmienky, no pri programovej realizácii ťažko dokážeme nariadiť nedisciplinovanému používateľovi, aby ich aj skutočne dodržiaval – ošetrorenie musíme zabezpečiť na úrovni samotného algoritmu.

Vráťme sa k algoritmu, ktorý delí dve čísla. Zabezpečte, aby algoritmus požadoval zadanie druhej hodnoty dovtedy, kým používateľ nezadá nenulovú hodnotu.



Obr. 24 Všeobecné vyjadrenie cyklu s podmienkou na konci



Obr. 25 Testovanie správnosti vstupnej hodnoty

1. Napíšte algoritmus, ktorý bude sčítavať dve čísla. Zabezpečte, aby sa po skončení výpočtu opýtal, či chce používateľ pokračovať a aby skončil až vtedy, keď odpoveď na otázku bude „nie“.
2. Pre interval zadaný prostredníctvom hraníc a a b zistite, koľkokrát sa v ňom nachádzajú čísla deliteľné hodnotou d.
3. Napíšte algoritmus, ktorý zistí počet deliteľov zadaného čísla.
4. Napíšte algoritmus, ktorý nájde najväčšieho spoločného deliteľa dvoch čísel.
5. Napíšte algoritmus, ktorý vykráti zlomok zadaný prostredníctvom čitateľa a menovateľa.
6. Napíšte algoritmus, ktorý pre zadané číslo vráti jeho zrkadlový obraz (za akých podmienok je riešenie správne?)
7. Pre postupnosť čísel ukončenú nulou vypíšte maximum.

8. *Pre postupnosť rôznych čísel ukončenú nulou nájdite najmenšie a druhé najmenšie číslo.*
9. *Pre zadanú postupnosť čísel ukončenú nulou zistite, koľko sa v nej nachádza párnych a koľko nepárnych čísel.*

Neriešiteľné problémy

Všetky nami doteraz riešené problémy boli algoritmizovateľné – existoval algoritmus, ktorým sme dokázali popísať postup vedúci k vyriešeniu. V praktickom živote možno však veľmi často nájsť prípady, keď nájsť algoritmus na vyriešenie je vzhľadom na obrovské množstvo vstupných údajov veľmi náročné (získovanie viny alebo nevinu v právnickej praxi, správanie sa Zeme v najbližších 10 000 rokoch, atď.). Na základe tejto náročnosti sa potom mnohí bádatelia snažia vyhlásiť problémy za nealgoritmizovateľné. Lenže na základe čoho, ak nie vstupných informácií a postupov v ľudskom mozgu sa určí vina alebo nevina, alebo sa predpokladá život na Zemi o 10 000 rokov? Otázka algoritmizovateľnosti v týchto prípadoch nepredstavuje problém, problémom je len množstvo vstupných údajov.

Napriek tomu existuje skupina problémov, ktoré možno označiť pojmom **algoritmicky neriešiteľné**.

Ako príklad si môžeme vziať hru na život:

Majme k dispozícii populáciu buniek (štvorčekový papier) s niekoľkými bunkami (krížikmi v políčkach). Bunky sa na základe určitých pravidiel rozmnožujú, na základe iných pravidiel umierajú. Žijú v určitých cykloch - pri prechode do nového cyklu sa každá existujúca bunka skontroluje a buď zahynie, alebo sa rozmnoží alebo sa nezmení.

Pravidlá môžu byť určené nasledovne:

- a) *ak bunka nemá suseda (bunku v ôsmich okolitých políčkach) – zahynie na samotu,*
- b) *ak bunka susedí s dvoma inými bunkami, vznikne štvrtá,*
- c) *ak má bunka viac ako piatich susedov – zahynie od hladu.*

Samovražednou sa nazýva taká počiatočná populácia, ktorá po určitom počte generácií vymizne. Neriešiteľným problémom je napísať taký algoritmus, ktorý pre **ľubovoľnú** zadanú populáciu zistí, či je alebo nie je samovražedná.

Túto úlohu možno riešiť tak, že budeme simulovať život zadanej populácie. V prípade kladnej odpovede, keď populácia je samovražedná, môžeme simuláciu ukončiť po vymretí všetkých bu-

niek. Ak zadaná populácia nie je samovražedná, simuláciu jej života môžeme ukončiť v prípade zacyklenia alebo vytvorenia pravidelného vzoru, pri ktorých je zrejmé, že k vyhynutiu nedôjde. Avšak nie každá začiatočná populácia vedie k vytvoreniu pravidelného alebo cyklicky sa opakujúceho vzoru.

V takom prípade nevieme v konečnom čase rozhodnúť, či populácia niekedy vymrie alebo sa bude ďalej nepravidelne vyvíjať. Simulácia vývoja populácie preto nie je algoritmom, lebo nespĺňa vlastnosť konečnosti.

1. *Vymenujte a popíšte základné algoritmické štruktúry.*
2. *Vysvetlite pojem premenná.*
3. *Akú úlohu zohráva definovanie vstupných podmienok pri vytváraní algoritmu?*
4. *Vymenujte typy vetvenia a popíšte ich na príkladoch.*
5. *Na príkladoch vysvetlite používanie logických spojok and, or a not.*
6. *Na príkladoch popíšte vhodnosť použitia cyklu s podmienkou na začiatku a s podmienkou na konci.*

Lekcia 3

3 Programovacie jazyky

predpoklady na zvládnutie lekcie:

- schopnosť tvorby algoritmov prostredníctvom vývojových diagramov

obsah lekcie:

- programovací jazyk ako nástroj na prepis algoritmu
- úvod do vývojového prostredia Borland Delphi
- tvorba jednoduchých programov
- prepis sekvenčných algoritmov do programovacieho jazyka

cieľ:

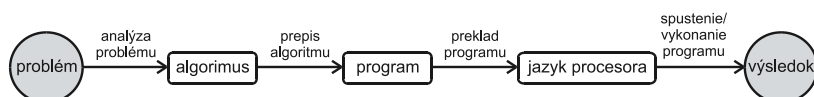
- oboznámiť sa s prepisom algoritmu do programovacieho jazyka
- zvládnuť pohyb v prostredí Borland Delphi
- pochopiť filozofiu udalost'ami riadeného programovania

Algoritmus a algoritmizácia sú určitým medzikrokom medzi zadaním problému a jeho vyriešením na počítači. Pomocou algoritmu dokážeme vyriešiť problém my, no takmer vždy ho potrebujeme preložiť do jazyka počítača (alebo iného stroja či človeka), ktorý poveríme jeho riešením.

Na komunikáciu s akýmkoľvek zariadením schopným vykonávať algoritmy potrebujeme jazyk. Aby bolo dorozumievanie pre človeka čo najpriateľnejšie, vytvárajú sa umelé jazyky, pomocou ktorých môžeme jednoducho a pritom jednoznačne vyjadriť algoritmus, a ktoré dokáže naše zariadenie interpretovať. Takéto jazyky nazývame **programovacie**.

Jazyk zrozumiteľný procesoru počítača sa nazýva **strojový kód**. Pre programátora je však veľmi ťažké komunikovať s počítačom v tomto jazyku. Preto boli vyvinuté programovacie jazyky, ktoré sú pre človeka zrozumiteľnejšie a prekladače, ktoré programy zapísané v týchto jazykoch prekladajú do strojového kódu.

Činnosť, ktorú vykonávame pri zápise algoritmu do programovacieho jazyka, označujeme ako **programovanie**. V súčasnosti ju uľahčuje množstvo podporných prostriedkov a techník a tak možno povedať, že programovanie už nie je umením, ale často len remeslom.



Obr. 26 Riešenie problému od jeho vzniku až po spustenie programu

Program sme doposiaľ chápali len ako súbor (uložený na disku počítača) obsahujúci inštrukcie, podľa ktorých dokáže procesor pracovať – iným slovom aplikáciu, ktorá je uložená v tvare strojového kódu.

Programom však budeme nazývať aj **algoritmus zapísaný v programovacom jazyku**, ktorý môže byť do strojového kódu preložený. Súboru obsahujúcemu takýto program hovoríme **zdrojový kód**. Pre prekladač je zdrojovým (vstupným) súborom, z ktorého sa vygeneruje cieľový (výstupný) súbor v strojovom kóde – spustiteľná aplikácia.

Ak chceme v nejakej aplikácii robiť zmeny (napr. sa nám nepáči farba pozadia), potrebujeme ich vykonať v zdrojovom kóde. Teoreticky je možné urobiť zmeny aj v preloženej aplikácii, avšak takéto krkolomné kúsky dokáže len málokto.

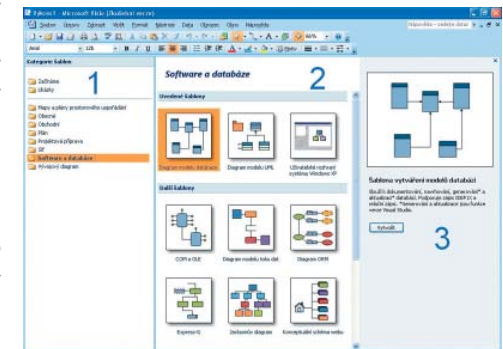
Programovanie v širšom slova zmysle pozostáva z troch základných fáz:

- **algoritmizácia** predstavuje premyslenie algoritmu, prípadne jeho textový alebo grafický zápis,
- prepis algoritmu do počítača spočíva v **prepísaní algoritmu do inštrukcií programovacieho jazyka**, pričom treba algoritmus prispôbiť špecifickosti toho-ktorého jazyka. Program okrem inštrukcií obsahuje rôzne nastavenia a prispôbenia, popis premenných a ošetrenie chýb, ktoré môžu nastať,
- **ladenie a testovanie programu** je poslednou fázou a zahŕňa v sebe hľadanie chýb spôsobených nedokonalosťou programátora, testovanie a kontrolu správnosti výsledku.

Niektorí „tiežprogramátori“ sa snažia prvú (niekedy aj poslednú) fázu vynechať. Programovať začínajú priamo do počítača, často nielen bez predchádzajúcej prípravy, ale aj bez premyslenia postupu. Čas, ktorý ušetria vynechaním algoritmizácie, potom mnohonásobne prekročia prepracovaním zdrojového kódu.

Tento spôsob programovania možno aplikovať len pri jednoduchých problémoch. Riešeniu náročných problémov musí predchádzať dôkladná analýza často v kombinácii s modelovaním postupov prostredníctvom špecializovaných jazykov a nástrojov.

Spustenie programu je podmienené jeho prekladom do strojového kódu, ktorý jediný je schopný procesor vykonávať. Program doň prekladáme pomocou prekladača, ktorý je v sú-



Obr. 27 Modelovanie v prostredí MS Visio

tá, v súčasnosti sú najrozšírenejšími jazyky postavené na základoch *Basicu*, *Pascalu* a *C*. Tieto jazyky existujú v mnohých dialektoch, no jadro a spôsob zápisu zostáva stále rovnaký.

Prostredníctvom prekladača sa programy napísané v jazyku vyššej úrovne prekládajú do nízkoúrovňových jazykov. Tieto sú už šité presne na mieru procesora a pokiaľ napíšete program v niektorom z nich, je takmer isté, že iný typ procesora mu rozumieť nebude. Do tejto kategórie patrí strojový jazyk a jazyk symbolických adries (*assembler*). Oba jazyky pracujú priamo s hardvérom počítača (strojový kód pomocou číselných inštrukcií, *assembler* má čísla nahradené symbolickými názvami inštrukcií a operandov).

Napr. pre sčítanie dvoch čísel v assembleri sa najprv musia z pamäti (treba zadať presné miesto, kde sú čísla zapísané) premiestniť do „sčítačky“, sčítať a výsledok zasa premiestniť na konkrétne miesto v pamäti.

obdobie	jazyk	popis
40. roky 20. stor.	strojový jazyk počítača	časovo i intelektuálne vysoko náročný, pretože príkazy vďaka potrebe kódovania a jeho nezrozumiteľnosti často obsahovali chyby
1951	assembler	jednoduché príkazy vo forme anglických slov
1956	FORTRAN	FORmula TRANslation – určený primárne na vedecké a inžinierske výpočty
1958	ALGOL	ALGOritmic Language – rozšírenie a zuniverzálnevanie fortranu
1959	COBOL	COmmon Business Oriented Language – spracovanie údajov pre ekonómov, možnosť tvorby makier a hierarchických údajových štruktúr
1960	Algol 60	základ takmer všetkých ďalších programovacích jazykov - štruktúrovaný jazyk s podporou podprogramov a rekurzie
1961	BASIC	Beginners All-Purpose Symbolic Instruction Code – prvý masovo rozšírený jazyk, primárne určený pre výučbu programovania
1968	Pascal	pomenovaný po Blaise Pascalovi navrhnutý Niklausom Wirthom pre výučbu programátorov na univerzite v Zürichu
1980	C	obsahuje prvky štruktúrovaného a súčasne nízkoúrovňového programovacieho jazyka vďaka čomu poslužil ako základ pre tvorbu mnohých operačných systémov, svojou verziou C++ (1985) sa postaral o zavedenie a propagáciu objektovo orientovaného programovania
1991	Visual Basic	udalosťami riadené programovanie
1992	Comenius Logo	„detský“ programovací jazyk
1995	Delphi	obrovský prelom v programovaní vďaka kombinácii vizuálneho a udalosťami riadeného programovania

Tab. 4 Vývoj programovacích jazykov

Našťastie, prepis z vyššieho programovacieho jazyka do jazyka strojového zabezpečuje prekladač toho-ktorého vyššieho programovacieho jazyka, a samotný *assembler* alebo strojový jazyk sa používajú pri programovaní len zriedkavo.

Výhodou vyšších programovacích jazykov je prehľadnosť a vďaka nej možnosť rýchlejšieho nájdania chýb a jednoduchšie úpravy. Nevýhodou oproti jazykom nižšej úrovne je menšia efektivita kódu po preložení. Je to daná za to, že môžeme programovať na úrovni bližšej ľudskému mysleniu a za to, že do strojového kódu sa program prekladá automaticky pomocou prekladača. Pri súčasných frekvenciách procesorov to však ani nepostrehneme.

Medzi jazykmi vyššej úrovne hrajú v súčasnosti významnú úlohu **objektovo orientované programovacie jazyky**. Vychádzajú z faktu, že objektový pohľad na programovanie je veľmi podobný pohľadu človeka na svet – človek vidí objekty, rozlišuje ich vlastnosti a pozná ich funkcie – za objekt považuje prakticky všetko, čo dokáže popísať.

V objektovo orientovanom jazyku je základnou stavebnou jednotkou objekt, ktorý má svoje vlastnosti a dokáže vykonávať definované operácie (označujú sa ako metódy). Vyššiu formu objektov predstavujú komponenty, ktoré sú v moderných vývojových prostrediach základným stavebným kameňom aplikácie. Okrem vlastností a metód disponujú mechanizmami, prostredníctvom ktorých sú schopné spracovať udalosti (napr. kliknutie na komponent, stlačenie tlačidla, pohyb myši a pod.). Veľmi často sa programovanie, pri ktorom sa využívajú komponenty schopné reagovať na udalosti, označuje i ako **udalosťami riadené programovanie**.

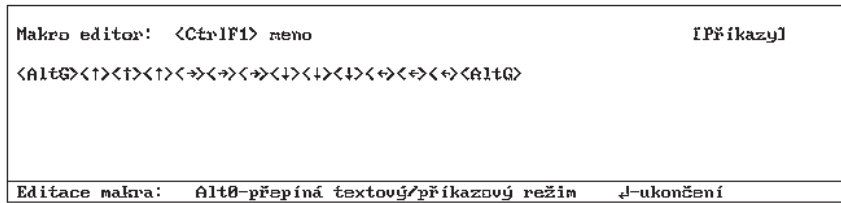
Každé, t.j. i objektovo orientované, programovanie nesie v sebe na úrovni programovania základných operácií črty štruktúrovaného programovania a ani to, že pri tvorbe aplikácií využívame komponenty, neznamená, že programujeme objektovo. Analógiu medzi dvoma prostrediami na vývojovo rôznych stupňoch budete môcť posúdiť pri programovaní aplikácií v prostredí Turbo Pascal a Borland Delphi v ďalších kapitolách.

Jazyky aplikácií

S algoritmizáciou sa okrem programovacích jazykov môžeme stretnúť v mnohých vyspelejších aplikáciách, ktoré nám umožňujú „naprogramovať“ často sa opakujúce postupy. Zvyčajne ide o sekvenciu niekoľkých činností, no možno sa stretnúť aj s inými algoritmickými konštrukciami. Takýmto „programom“ hovoríme makrá (skripty) a často má aplikácia na ich vytváranie vlastný programovací jazyk.

Makrá možno vytvárať **interaktívne** – aplikácia si uchováva činnosti, ktoré vykonávame a neskôr ich dokáže zopakovať alebo **manuálne**, keď príkazy makra môžeme zapisovať ručne. Často sa oba prístupy kombinujú – kostru vytvoríme interaktívne a potom ju upravujeme manuálne.

Jedným z prvých programov, ktoré podporovali makrá, bol v našich zemepisných šírkach textový editor T602. Tu však neexistoval žiaden programovací jazyk, makrá si len pamätali postupnosť stlačených klávesov. „Program“ potom mohol vyzerat napr. ako na obrázku.



Obr. 29 Makro vytvorené zo stlačenia klávesov v T602

S nástupom kancelárskeho balíka *Microsoft Office* sa dostali k slovu makrá vytvárané v programovacom jazyku *Visual Basic for Application*. Opäť ich možno vytvárať interaktívne alebo manuálne, no nie sú obmedzené veľkosťou. Je možné spúšťať ich nielen klávesovou skratkou, ale aj tlačidlom na paneli nástrojov alebo položkou v menu či dokonca automaticky pri otvorení súboru. Výhodou takéhoto spustenia makra je, že dokáže pred používateľom skryť nepotrebné položky a používateľ sa potom nestráca v množstve funkcií programu, ale postačia mu tie, ktoré pre neho zobrazí tvorca makra.

Vlastnosť automatického spustenia makra veľmi radi využívali tvorcovia tzv. makrovírusov, ktorí prostredníctvom textových a tabuľkových dokumentov šíрили škodlivý obsah medzi neznalými používateľmi. Súčasnú kancelárske aplikácie už obsahujú niekoľkoúrovňovú ochranu pred škodlivým kódom tohto typu.

Databázové systémy okrem jazyka makier obsahujú zvyčajne aj nástroje pre vytváranie dotazov – požiadaviek na zobrazenie údajov. Tieto sa obmedzujú na niekoľko kľúčových slov a dovoľujú manipulovať (triediť, mazať, zobrazovať, počítať a pod.) s údajmi v tabuľke. Štandardom pre túto kategóriu je jazyk SQL (*Structured Query Language*), ktorý je univerzálny a využívajú ho nielen osobné počítače, ale možno sa s ním stretnúť aj v iných systémoch a komunikovať s databázami prostredníctvom počítačovej siete.

Ako typický príklad využitia makier nám môžu slúžiť rozliční sprievodcovia v kancelárskych aplikáciách a grafických editoroch, ktorí po vložení niekoľkých údajov dokážu vygenerovať webovú stránku, životopis, kalendár, fax, vizitky a pod.

Ďalšími kategóriami sú skriptovacie jazyky využívané v počítačových hrách, v dávkových súboroch operačných systémov i v prostredí Internetu (PHP, ASP).

Programovací jazyk pascal

Programovací jazyk pascal vytvoril profesor informatiky v Zürichu *Niklaus Wirth* (nar. 1934) s cieľom zabezpečiť vyučovanie systematického programovania. Jeho cieľom bolo vytvoriť programovací jazyk, ktorý by bol kompromisom medzi abstraktnými štruktúrami algoritmov a konkrétnou reprezentáciou spracovávaných údajov v počítači. Skúsenosti, ktoré získal ako vedúci návrhár pri tvorbe iných jazykov (navrhol *Algol W*, *Modula*, *Modula-2*, *Oberon*) zúročil v roku 1971 prvou verziou pascalu, ktorú následne v roku 1974 upravil prakticky do súčasnej podoby. V roku 1981 bola pre pascal vytvorená ISO norma a začal sa uplatňovať najmä na poli vzdelávania.

Najväčší rozmach dosiahli verzie *Turbo Pascalu* od firmy *Borland*, ktoré sa ešte i dnes pomerne často používajú na stredných školách. Niektoré verzie boli *Borlandom* uvoľnené na nekomerčné používanie a možno ich získať na ich domovskej stránke **zdarma**.

Nasledovníkom *Turbo Pascalu* a obrovským prelomom v programátorskem myslení sa v roku 1995 stalo vývojové prostredie *Delphi*, ktoré je založené na jazyku *Object Pascal*. Jeho prínosom je charakteristika označovaná ako **RAD** (*Rapid Application Development* – rýchly vývoj aplikácií), ktorá umožňuje aplikáciu poskladať z malých hotových častí – komponentov. Oklieštené verzie sú na nekomerčné používanie opäť k dispozícii **zdarma** na webovej stránke tvorca.

Štruktúra programu

Program má zvyčajne svoje meno, no tento údaj nie je povinný. Zapisuje sa do prvého riadku za kľúčové slovo `program`. Môže obsahovať ľubovoľné znaky a nesmie byť rovnaké ako niektoré kľúčové slovo jazyka. V *Delphi* sa názov programu vytvára automaticky podľa názvu súboru, v ktorom je program uložený.

Telo programu pozostáva z príkazov a je ohraničené dvojicou kľúčových slov `begin` (začiatok) a `end` (koniec). Túto dvojicu zvykneme nazývať programovými zátvorkami a okrem ohraničenia začiatku a konca programu sa používa i na ohraničenie viacpríkazových sekvencií napr. vo vetve podmienky alebo v tele cyklu (čítaj ďalej). Príkazy v tele programu je potrebné oddeľovať bodkočiarkou.

Pri písaní programu je vhodné dodržiavať určitú štandardizovanú úpravu a zabezpečiť tak lepšiu čitateľnosť programu:

- príkazy odsadiť o čosi viac doprava ako sú `begin` a `end` slúžiace na ich uzavretie,

- písať jeden príkaz do jedného riadku,
- vhodne striedať veľké a malé písmená vo viacslovných názvoch atď.

Zátvorky „{“ a „}“ je možné používať na umiestnenie komentára pre lepšie pochopenie. Text, ktorý je v nich, počítač ignoruje. Rovnako ignoruje i text za príkazom „end.“, v ktorom znak bodky hovorí o tom, že program skončil a ďalej netreba pokračovať.

Pozdrav ma!

Borland Delphi je vývojové prostredie primárne určené na tvorbu aplikácií pod *Windows*. Na programovanie používa jazyk *Object Pascal*.

Tvorba aplikácie v *Delphi* je oproti vývojovým prostriedkom používaným v prostredí *Turbo Pascal* pod *MS DOS* na prvý pohľad veľmi odlišná, pretože vizuálna stránka aplikácie je štandardne tvorená pomocou formulára, ktorý predstavuje okno – základný stavebný prvok operačného systému *Windows*.

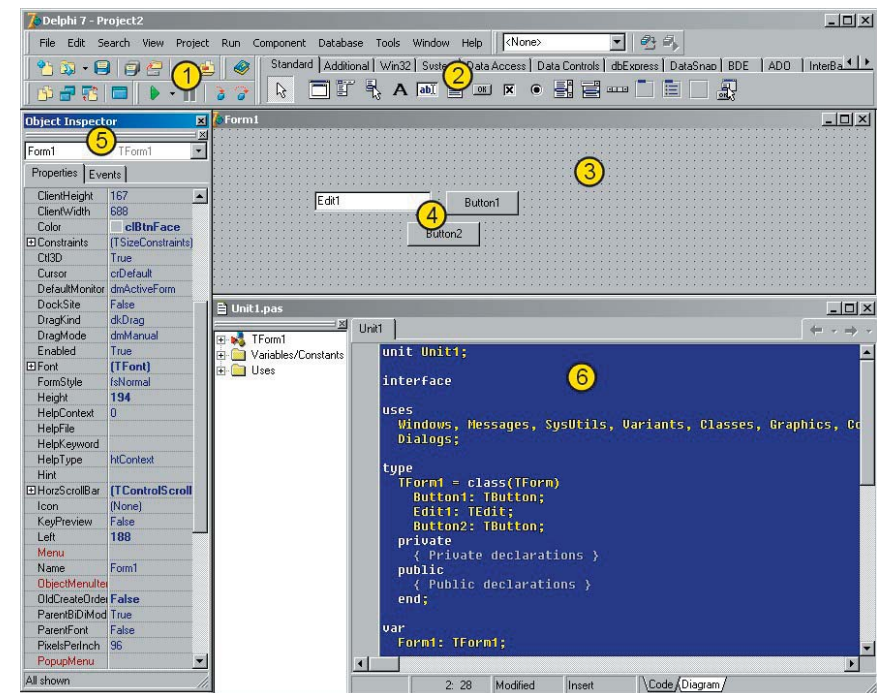
Na formulár sa umiestňujú komponenty, ktoré sú stavebnými kameňmi aplikácie (napr. tlačidlá, texty, editovacie riadky, menu, posúvače, grafické plochy atď.). Komponent predstavuje logicky uzavretý celok, ktorý má svoje vlastnosti a je schopný reagovať na rozličné udalosti – kliknutie, prechod myšou, stlačenie tlačidla a pod. Zvyčajne zabezpečuje niektorú činnosť alebo funkciu aplikácie. Vďaka komponentom nie je potrebné vytvárať každú aplikáciu úplne od začiatku, ale môžeme v nej využiť už hotové (existujúce) časti. Tieto môžu byť súčasťou vývojového prostredia (autori *Delphi* nám dali k dispozícii niekoľko stoviek komponentov zabezpečujúcich najčastejšie sa opakujúce funkcie), môžeme ich nájsť v prostredí Internetu alebo si vytvoriť vlastné.

Kombináciou, vzájomným prepojením a skladaním komponentov a samozrejme doplnením vhodného zdrojového kódu potom vytvárame vlastnú aplikáciu. Filozofia programovania je založená na spracúvaní udalostí, ktoré sa v aplikácii v rámci jednotlivých komponentov odohrávajú – z toho pramení aj označenie *udalostami riadené programovanie*.

Po spustení aplikácie sa dostaneme do prostredia pozostávajúceho z hlavných častí:

- **riadiaci panel** slúži na manipuláciu so súbormi *Delphi* (otvorenie, uloženie, spustenie, ladenie atď.),
- **formulár aplikácie** predstavuje to, čo vo *Windows* poznáme pod pojmom okno. Samotný formulár je vlastne okno (niekedy celá aplikácia), ktorá môže (ale nemusí) obsahovať ďalšie prvky,

- **zoznam komponentov** obsahuje skupiny komponentov, ktoré možno používať pri tvorbe aplikácie. Na formulár sa prenášajú kliknutím na komponent a kliknutím do formulára alebo dvojklikom na komponent. Pre prehľadnosť sú jednotlivé komponenty rozdelené do skupín, pričom v začiatkoch určite vystačíme s tými, ktoré sú umiestnené na karte *Standard*.
- **Object Inspector** je miestom, na ktorom možno nastaviť a meniť vlastnosti aktuálnych komponentov (to sú tie, na ktorých sme nastavení vo formulári). Vyvolať ho možno v ľubovoľnom momente cez *View – Object Inspector* (stlačením klávesu *F11*),
- **okno pre zdrojový kód** umožňuje písanie kódu a obsluhovanie udalostí jednotlivých komponentov.



1 – riadiaci panel

2 – paleta komponentov

3 – formulár aplikácie

4 – vložené komponenty

5 – Object Inspector

6 – zdrojový kód


Obr. 30 Prostredie Borland Delphi

Na obrazovke zvyčajne najviac miesta zaberajú formulár a kód. Často sa prekrývajú alebo niekedy jeden z nich zmizne. Prepínanie, resp. zobrazenie „toho druhého“ dosiahneme cez *View – Toggle Form/Unit* (alebo efektívnejšie klávesom *F12*).

Prvý „program“

Komponenty *Delphi* sú prispôbené tak, aby boli schopné meniť svoj vzhľad a funkciu podľa požiadaviek programátora a v rámci svojich možností.

Ak si vezmeme prázdny formulár, ktorý máme k dispozícii okamžite po spustení *Delphi* (alebo vytvorení novej aplikácie), predstavuje už v tomto momente samostatnú aplikáciu. Bez napísania jediného riadku kódu dokáže všetko, na čo sme zvyknutí pri oknách *Windows*.

Spustiť túto „aplikáciu“ môžeme položkou menu *Run – Run*, stlačením klávesu *F9* alebo najjednoduchšie ikonou  na riadiacom paneli.

Pre „aplikáciu“, ktorú sme „vytvorili“, sa vytvorí okno s názvom `Form1`, ktoré je schopné presunu, minimalizovania, maximalizovania, zmeny veľkosti uchopením za okraj a ťahaním (a to všetko bez jediného riadku kódu). Kliknutím na „x“ aplikáciu vieme ukončiť.



Obr. 31 Aplikácia bez programovania

Na tomto mieste by sme radi upozornili na veľmi často opakovanú začiatočnú chybičku. Je rozdiel, či máte pred sebou okno spustenej aplikácie alebo okno formulára, ktoré možno meniť. Často sa totiž stáva, že študent sa snaží meniť formuláru parametre a diví sa,

že tento nereaguje. Ak máte pred sebou formulár s hladkou plochou, ide zrejme o bežiacu aplikáciu. Ak okno formulára obsahuje bodky uložené do mriežky, ide o návrh formulára v prostredí *Delphi*.

Ďalšia vec, ktorá vás informuje o tom, s čím pracujete, je panel úloh systému *Windows*. Ak

je aktívne tlačidlo *Delphi*,  Obr. 32 Panel úloh a tlačidlá aplikácie

pracujete zrejme s návrhom, ak tlačidlo aplikácie, zrejme ide o spustený formulár.

Niekedy sa stáva, že programátor zmení parametre okna v *Delphi*, chce si spustením skontrolovať výsledok, ale aplikácia nejde spustiť, tlačidlo spustenia je deaktivované a *Delphi* na *F9* nereaguje. V takomto prípade je prav-

depodobné, že aplikácia nebola pred zmenou parametrov ukončená a zostala spustená v predchádzajúcom stave. Situáciu dokážeme identifikovať prostredníctvom panela úloh.

Každú aplikáciu možno v ľubovoľnom momente ukončiť funkciou *Run – Program Reset* (alebo kliknutím do okna zdrojového kódu a stlačením kombinácie *Ctrl+F2*).

Prišli sme teda k záveru, že na vytvorenie prázdneho okna ako bežiacей aplikácie pod *Windows* nepotrebujeme prakticky nič. Čo však, ak chceme meniť vlastnosti okna a vkladať doň vlastný obsah?

Parametre komponentu

Ako sme už vyššie spomenuli, meniť vlastnosti komponentov nám dovoľuje *Object Inspector*. Kliknutím na formulár sa tu zobrazia jeho vlastnosti.

Pokiaľ sa *Object Inspector* nezobrazí, neukončili ste spustenú aplikáciu aktuálneho formulára (pozri vyššie).

Skúsme teraz zmeniť záhlavie formulára – text, ktorý sa zobrazuje v titulkovom pruhu aplikácie. Doposiaľ sme v ňom mali zobrazený text „Form1“.

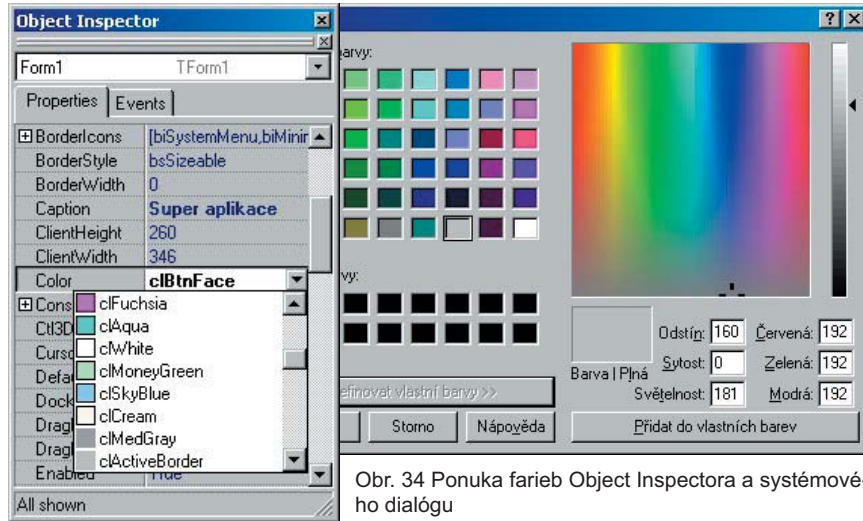
Vlastnosť, ktorá tento text určuje, sa označuje `Caption` a jej prepísaním sa okamžite zmení i záhlavie formulára.

Podobným spôsobom môžeme zmeniť formuláru i farbu: v položke `Color` nájdeme po rozbalení zoznamu preddefinované farby. Tieto môžu byť buď konkrétne (`clWhite`, `clRed`,...) a po našom nastavení nemenné alebo môžu byť prepojené na vlastnosti systému *Windows* a po zmene jeho farieb cez nastavenia obrazovky sa prispôbia (`clBtnFace`, `clHighlight`,...).

Dvojklikom do praveho stĺpca položky vyvoláme systémový dialóg s farbami, v ktorom dokážeme požadovanú farbu „namiešať“.



Obr. 33 Object Inspector a vlastnosti formulára



Obr. 34 Ponuka farieb Object Inspectoru a systémového dialógu

Zmena farby a záhlavia formulára prostredníctvom *Object Inspectoru* zrejme nenaplní naše vývojárske očakávania a pravdepodobne budeme požadovať viac (pre začiatok aspoň vypísanie textu, nejaké tlačidlá, po stlačení ktorých sa spustí nejaká činnosť atď.).

Základné komponenty, ktoré nám poskytnú prostriedky na vstup, výstup údajov a základné ovládanie, sú nasledovné:

Label – komponent slúžiaci na zobrazenie textu. Text, ktorý doň vložíme, zostane zobrazený po spustení aplikácie, no možno ho využiť i na zobrazovanie výsledkov. Vlastnosť určujúca text, ktorý chceme zobraziť, je označená ako *Caption* (rovnako ako pri formulári).

Edit – komponent určený na vkladanie textu používateľom, na zadávanie vstupov. Po spustení aplikácie môže obsahovať nejaký reťazec. Okamžite po vložení z panela komponentov obsahuje meno komponentu, napr. *Edit1*. Prednastavený text môžeme vymazať (alebo prepísať) tak, že vyprázdňime (alebo prepíšeme) vlastnosť *Text*.

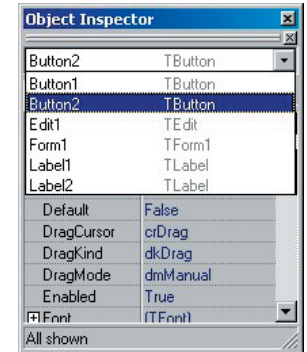
Button – tlačidlo, ktorého úlohou je zvyčajne po stlačení vykonať nejakú činnosť. Dôležitou je pre nás opäť vlastnosť *Caption*, ktorá určuje text zobrazený na tlačidle.

Ak chceme manipulovať s vlastnosťami konkrétneho komponentu, je ho potrebné najprv označiť (napr. kliknutím myši). O tom, že s ním skutočne manipulujeme, nás informuje aj prvý riadok *Object Inspectoru*. Pomocou neho sa môžeme me-

dzi jednotlivými komponentami umiestnenými na formulári aj prepínať (ak nevieme nájsť komponent na formulári, alebo je ich množstvo už značne neprehľadné).

Pokiaľ sa parametre nemenia podľa vášho očakávania, skontrolujte si, či naozaj ide o vybraný komponent.

S touto trojicou (štvoricou, ak rátame aj formulár) komponentov si pre začiatok vystačíme.



Obr. 35 Zoznam komponentov v Object Inspectoru

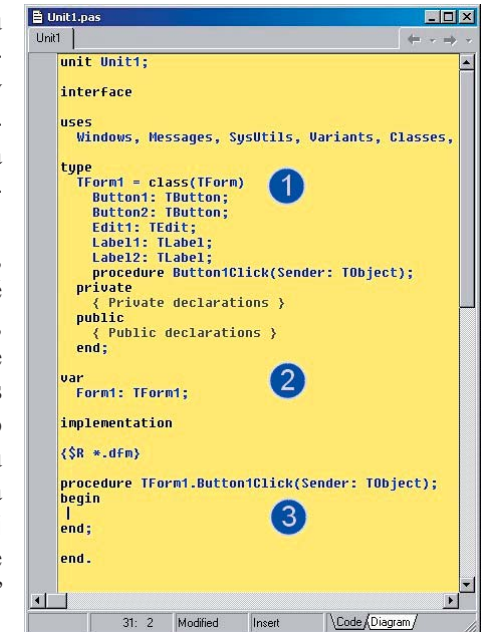
Základná filozofia programu

Vytvorte aplikáciu, ktorá po stlačení tlačidla zmení farbu formulára – postačia tri farby: červená, zelená a modrá.

Na formulár vložíme trojicu tlačidiel a napíšeme do nich (zmenou vlastnosti *Caption*) názvy príslušných farieb. Ako však vysvetliť systému, že po kliknutí na tlačidlo by bolo dobré niečo vykonať? Cesty sú dve:

- Prvá rýchlejšia a jednoduchšia, avšak nie univerzálna, dvojité kliknutie na tlačidlo (pozor, klikajte naozaj na tlačidlo, nie na formulár). Dvojklik nás preniesie do okna zdrojového kódu, kde vytvorí procedúru (časť algoritmu), ktorá sa vykoná vtedy, keď v spustenej aplikácii používateľ klikne na tlačidlo. Mala by vyzerat' podobne ako na obrázku.

Názov procedúry hovorí, že patrí triede formulára *TForm1* (t.j. formuláru *Form1*), ide o tlačidlo *Button1* a udalosť



1 – definícia triedy (pozri ďalej)
2 – deklarácia formulára
3 – automaticky vytvorená procedúra

Obr. 36 Automatické vytvorenie procedúry

Click, t.j. kód, ktorý sem napíšeme sa vykoná po kliknutí na toto tlačidlo.

Trieda formulára predstavuje jeho abstraktnú definíciu, je uvedená na začiatku unitu zvyčajne s rovnakým názvom ako má formulár, pred ktorým je umiestnené T (z anglického Type). O čosi ďalej je formulár deklarovaný.

Ak sa vytvorila procedúra s iným názvom (nie button a nie click), vráťte sa klávesom F12 do okna formulára a skúste kliknúť znova. Nesprávne vytvorenú procedúru si nevšímajte a nemažte ju – pokiaľ zostane prázdna, pri uložení aplikácie sa vymaže sama.

Medzi `begin` a `end` môžeme teraz vložiť príkazy, ktoré zabezpečia zmenu farby formulára na červenú:

```
Form1.Color:=clRed;
```

Formulár sa volá `Form1`. Tento jeho názov môžeme overiť vo vlastnostiach v položke `Name` (pozor `Name` nie je to isté, čo `Caption`).

Chceme mu zmeniť vlastnosť `Color` – vlastnosť sa od mena komponentu oddeľuje bodkou.

Samotné priradenie realizujeme priradovacím príkazom `:=`.

Priradovanú farbu môžeme zadať číslom (pokiaľ si pamätáme) alebo konštantou, ktorú systém pozná: `cl` znamená, že ide o farbu (*color*), `Red`, že farba bude červená.

Takýmto spôsobom sa nazývajú konštanty v mnohých prípadoch, pre nás postačí zatiaľ vedieť, že druhé tlačidlo využije `clBlue` a tretie `clGreen`.

- Druhá možnosť na ošetrenie kliknutia tlačidla je možno trochu zložitejšia, avšak poodhalí nám princíp toho, čomu sa hovorí udalosťami riadené programovanie.

Windows pracuje tak, že každé stlačenie klávesu, kliknutie alebo pohyb myši vyvolá udalosť. Túto udalosť potom môžeme ošetriť. Pokiaľ sa k nej nevyjadrujeme (nenapíšeme žiaden kód), nedeje sa nič, ak nejaký kód ako reakciu na udalosť naprogramujeme, postupuje sa pri jej vzniku presne podľa neho.

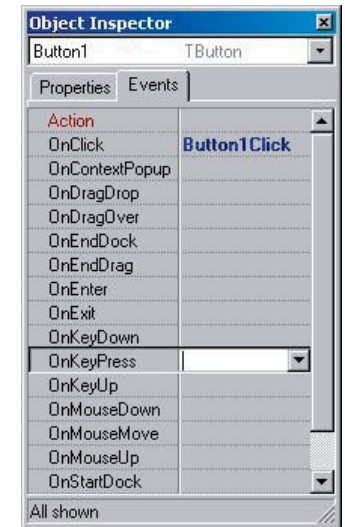
Klikneme si na ľubovoľné tlačidlo a vyvolajme *Object Inspector*. Možno ste si všimli, možno nie, *Object Inspector* pozostáva z dvoch záložiek – *Properties* (vlastnosti) a *Events* (udalosti).

O *Properties* už vieme, že nastavujú vlastnosti (vzhľad komponentu), *Events* obsahuje udalosti, na ktoré je schopný komponent reagovať. Pre `Button` sú k dispozícii tie, ktoré vidíte na obrázku.

Nás zrejme zaujme jedna z prvých udalostí – `onClick`. Popisuje to, čo sa má stať pri kliknutí na objekt (tlačidlo). Ak do prázdneho poľa za `onClick` dvojklikneme, dostaneme sa na to isté miesto ako v predchádzajúcom prípade po dvojkliku na tlačidlo.

Mohli by ste namietnuť, že prvý prípad je oveľa jednoduchší a rýchlejší, s čím možno len súhlasiť. Jeho nevýhodou je však to, že vždy vyvoláva len jednu udalosť, zvyčajne tú, o ktorej tvorcovia systému predpokladali, že sa najčastejšie v súvislosti s daným komponentom využíva (každý dvojklik na tlačidlo vždy vyvolá udalosť `onClick`). My však niekedy potrebujeme ošetriť aj iné udalosti. Napr. by sme chceli, aby sa niečo udialo, keď budeme pohybovať myšou nad tlačidlom (udalosť `onMouseMove`), alebo ak stlačíme kláves na klávesnici (udalosť `onKeyPress`) atď.


K týmto udalostiam máme prístup len cez záložku *Events*.



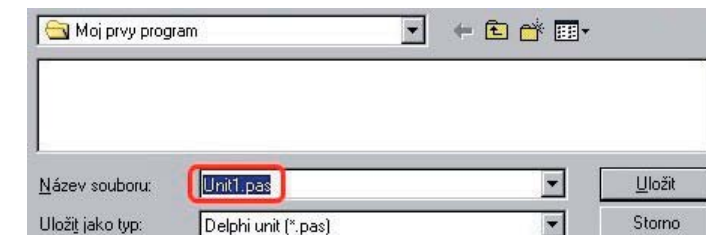
Obr. 37 Events (udalosti) pre tlačidlo `Button1`

Skladba aplikácie

Uložme teraz našu aplikáciu a pokúsme sa ju preniesť na iný počítač tak, aby sme s ňou mohli ďalej pracovať.

Na uloženie všetkých súčastí naraz slúži ikona , ktorá uloží všetko neuložené. Pokiaľ budete ukladať aplikáciu, postupnosť bude zrejme nasledovná:

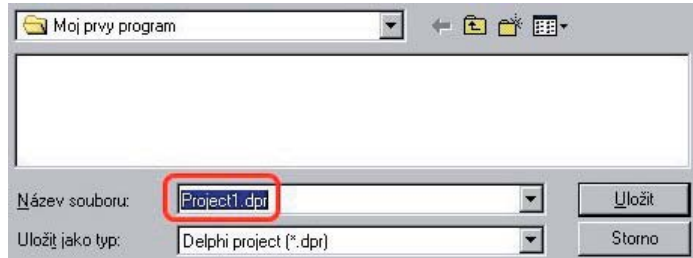
- ako prvá časť sa ukladá **unit**. Unit predstavuje základnú stavebnú jednotku aplikácie v *Delphi*. Pozostáva zvyčajne z dvoch častí. Prvá popisuje vizuálny model formulára – zoznam, umiestnenia a nastavenia



Obr. 38 Ukladanie unitu

komponentov na formulári, druhá je vlastný zdrojový kód, ktorý vidíme, a ktorý upravujeme prostredníctvom už známeho editora. Pre každú aplikáciu odporúčame vytvoriť osobitný priečinok a unitom dávať také mená, aby ste vedeli čo obsahujú,

- za unitmi nasleduje uloženie projektového súboru – tento obsahuje zoznam všetkých použitých unitov, inicializáciu, štart a prípadné ukončenie aplikácie.



Obr. 39 Ukladanie projektového súboru

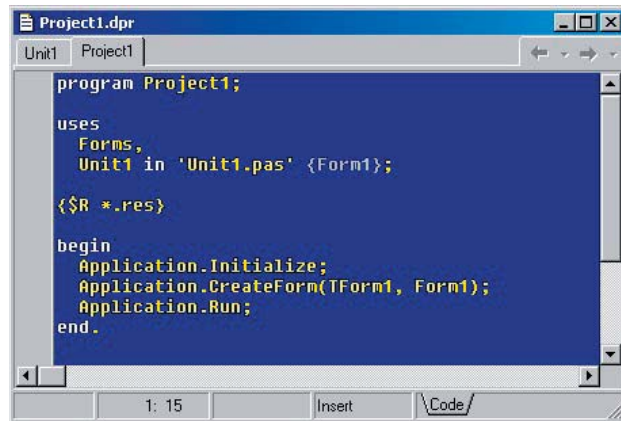
Jeho názov sa bude zobrazovať ako názov aplikácie jednak pri vytvorenej ikone, a jednak na paneli úloh po spustení aplikácie.

Bez projektového súboru unit nespustíte!!!

Po úspešnom uložení sa môžeme pozrieť na obsah priečinka, kam sme výsledok uložili.

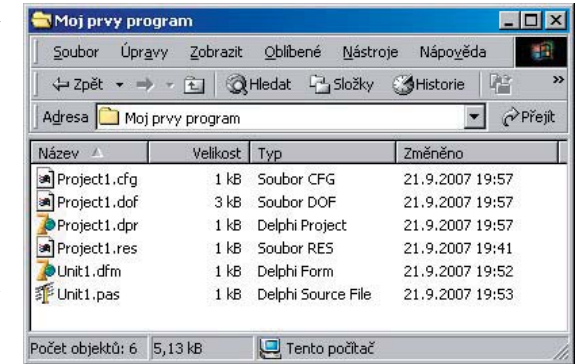
Z tohto zoznamu sú dôležité nasledovné typy súborov:

- *dpr* – súbor projektu obsahujúci zoznam unitov a formulárov, ktoré projekt (aplikácia) používa, inicializačný a štartovací kód. Jeho obsah možno v prípade potreby zobrazíť cez *Project – View Source*.



Obr. 40 Obsah projektového (dpr) súboru

- *pas* – zdrojový kód unitu napísaný používateľom,
- *dfm* – vizuálny model – obsah formulára (vzhľad a zoznam komponentov) priradeného k zdrojovému kódu s rovnakým názvom (vytvára sa automaticky pri ukladaní zdrojového kódu).



Obr. 41 Obsah priečinka

Bez týchto súborov zdrojový kód aplikácie na inom počítači neotvoríte a neskompilujete. Ostatné súbory obsahujú nastavenia projektu (majú názov totožný s projektom, odlišujú sa len koncovkou) a skompilované unity (*dcu*). Súbor označený ako aplikácia (prípadne s koncovkou *exe*) predstavuje vytvorenú aplikáciu, ktorú môžeme spustiť ako na aktuálnom, tak i na ľubovoľnom inom počítači.

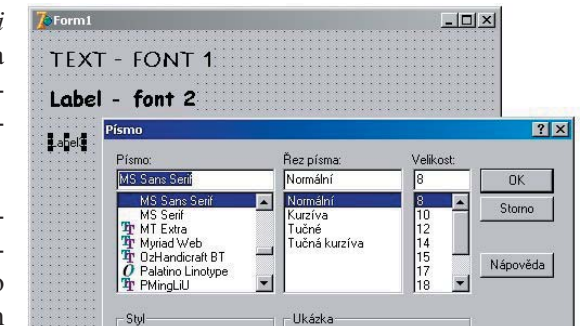
Konečne pozdrav!

Vráťme sa teraz k prvotnému zadaniu – požadujeme od aplikácie, aby nás pozdravila. Vzhľadom na to, že v *Delphi* zvyčajne nepracujeme v režime konzoly (čierna obrazovka s bielym výpisom podobne ako v *Turbo Pascale* alebo v *MS DOSe*), ale využívame prostriedky grafického prostredia, je potrebné prispôbiť naše uvažovanie novým štandardom.

Príkaz výstupu

V prostredí *Delphi* máme k dispozícii dva spôsoby, ktorými môžeme používateľovi poskytnúť údaje.

V prvom prípade môžeme zobrazíť výstupný text do niektorého zo štandardných vizuálnych komponentov.



Obr. 42 Vložený Label a jeho nastavenia pre font

Pre nás bude na tento účel najvhodnejším zrejme `Label`.

Na formulár umiestnime `Label`, ktorému môžeme prostredníctvom vlastnosti `Font` zmeniť nastavenia písma (napr. zmeniť font, zväčšiť, prípadne ho zafarbiť) a prostredníctvom vlastnosti `Caption` doň vložiť text (napr. „*Sem príde pozdrav*“).

Samotné vypísanie pozdravu naprogramujeme ako udalosť po kliknutí na tlačidlo, ktoré po stlačení do `Label1` vypíše samotný pozdrav. Text, ktorý sa má v komponente zobraziť umiestnime do apostrofov (klávesová skratka `Alt+39`):

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  Label1.Caption:='Nazdar, som Tvoj pocitac.';
end;
```

V tomto prípade sa môžeme na `Label1` odvolať priamo, nepotrebujeme uvádzať jeho umiestnenie na `Form1` (`Form1.Label1.Caption`) – pokiaľ pri odvolaní sa na komponent nevedieme formulár, automaticky sa o ňom uvažuje, že je umiestnený na formulári, ktorému patrí procedúra (`TForm1`).

Názov formulára sme nemuseli uvádzať ani v príklade s farbami, zjednodušil nám však vysvetľovanie.

Doplňte na formulár ešte jedno tlačidlo, ktoré do príslušného `Labelu` vypíše pozdrav „Ahoj, som PC.“. Pozorujte zmeny v texte pri striedavom stlačení tlačidiel.

Druhým spôsobom, ktorý nám umožňuje informovať používateľa dôraznejšie ako výpisom textu do `Labelu` (ktorý si niektorí používatelia ani nemusia všimnúť), je zobrazenie okna so správou.

Na (nový) formulár nám v tomto prípade postačí umiestniť tlačidlo, ktoré po stlačení realizuje nasledovný kód:

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  ShowMessage('Nazdar, som Tvoj pocitac.');
```



Príkaz `ShowMessage` zobrazí okno, v ktorom bude text umiestnený v apostrofoch.

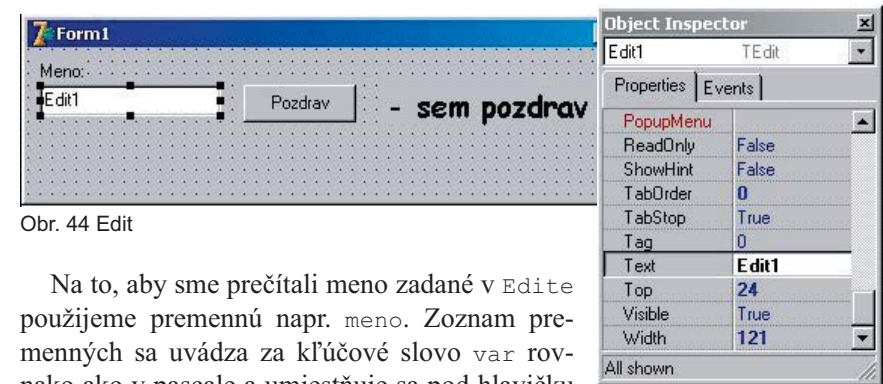
Obr. 43 Okno so správou a pozdravom

Napište program, ktorý na základe zadaného mena osloví a pozdraví používateľa (napr. „Ahoj Jozef“).

Na to, aby program dokázal od používateľa získať nejaké vstupné hodnoty, potrebujeme navyše komponent, ktorý nám umožní vkladanie textu napr. prostredníctvom klávesnice.

Na vkladanie jednoriadkového textu používame komponent `Edit`. Obsah (napísaný text) je k dispozícii vo vlastnosti `Text`.

Po vložení komponentu obsahuje `Edit` v sebe text totožný s jeho názvom – môžeme ho vymazať vymazaním obsahu vo vlastnosti `Text`. Prostredie programu môžeme vylepšiť pridaním `Labelu` popisujúceho, čo sa má do `Editu` umiestniť.



Obr. 44 Edit

Na to, aby sme prečítali meno zadané v `Editu` použijeme premennú napr. `meno`. Zoznam premenných sa uvádza za kľúčové slovo `var` rovnako ako v pascale a umiestňuje sa pod hlavičku procedúry. Kód riešiaci naše zadanie skryjeme do udalosti pre tlačidlo a samotná realizácia môže potom vyzeráť nasledovne:

```
procedure TForm1.Button1Click(Sender:TObject);
var meno:string;
begin
  meno:=Edit1.Text;
  ShowMessage('Ahoj '+meno);
end;
```

Proces sa spustí po stlačení tlačidla, premenná `meno` typu `string` nám poslúži na uloženie textu.

V prvom riadku programu do premennej `meno` vložíme text napísaný používateľom do `Edit1`. Zapamätané meno potom spojíme (operáciou `+`) s textom obsahujúcim pozdrav a pošleme prostredníctvom `ShowMessage` do samostatného okna.

Daný problém by sme mohli riešiť i nasledovne:

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  ShowMessage ('Ahoj ' +Edit1.Text);
end;
```

pričom by sme ušetrili minimálne jeden riadok kódu a jednu premennú – do okna so správou by sa priamo prečítal text vložený používateľom do komponentu `Edit1`.

Vzhľadom na potrebu používania premenných v náročnejších príkladoch a na zlepšenie čitateľnosti kódu však takéto možnosti zľahčenia budeme obchádzať a ponecháme ich na šikovnejších čitateľov.

Napište program, ktorý po zadaní mena, priezviska a oslovenia vypíše vetu v tvare: „Vítam ťa, pán Jožko Mrkvička“.

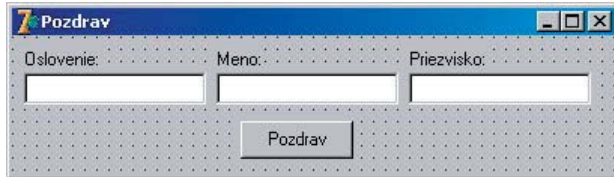
Na tomto príklade si predstavíme najjednoduchšie operácie, ktoré možno vykonať s textom: po umiestnení hodnôt do premenných vytvoríme pomocou operácií sčítania textových reťazcov celú vetu, ktorú následne vypíšeme.

Vzhľadom na to, že vstupných údajov je už o čosi viac

ako v predchádzajúcich príkladoch, je nevyhnutné pridať ku každému `Editu` popis prostredníctvom `Labelu`.

Spúšťajúcou udalosťou bude opäť kliknutie na tlačidlo.

```
procedure TForm1.Button1Click(Sender:TObject);
var meno,priezvisko,oslovenie,celaVeta:string;
begin
  oslovenie:=Edit1.Text;
  meno:=Edit2.Text;
  priezvisko:=Edit3.Text;
  celaVeta:='Vitam Ta, '+oslovenie+' '+meno+' '+priezvisko;
  ShowMessage (celaVeta);
end.
```



Obr. 45 Vizualný návrh aplikácie

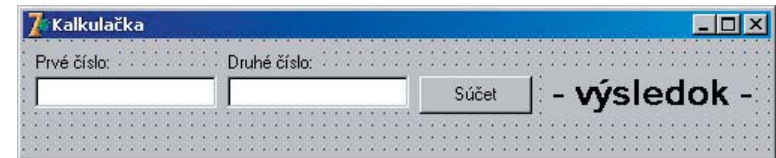
Text v apostrofoch bude do premennej priradený v rovnakej podobe, ako je napísaný v zdrojovom kóde programu, namiesto premenných bude dosadený text, ktorý obsahujú. Medzery slúžia na oddelenie textov v premenných. Pokiaľ by sme ich nepoužili, výsledkom by bol text bez medzier v podobe „panJozkoMrkvička“.

Pokiaľ sa pozdrav nezobrazil v správnom poradí (sú prehodené napr. oslovenie s priezviskom), pravdepodobne načítavate údaje z iných `Editov` ako náš program (napr. oslovenie z `Edit3` a pod.).

Prepis sekvenčných algoritmov

Vytvorte aplikáciu, ktorá dokáže sčítať, odčítať a vynásobiť dvojicu čísel.

Vytvoríme najprv aplikáciu, ktorá bude sčítavať dve celé čísla.



Obr. 46 Prostredie kalkulačky

Ako prvé potrebujeme zrejme vytvoriť prostredie. Mohlo by byť podobné tomu na obrázku. Zobrazené texty sú komponenty `Label`, ktorým sme zmenili vlastnosť `Caption` a napísali sme do nej zobrazený text.

Textové polia na zadávanie čísel sú komponenty `Edit`, ktoré síce po vložení obsahujú vpísaný text (`Edit1`, `Edit2...`), ale ten sme zrušili jeho vymazaním vo vlastnosti `Text`. Tlačidlo je `Button` so zmenenou vlastnosťou `Caption`, no a na zobrazenie výsledku použijeme opäť `Label`. Jeho obsah môžeme cez `Caption` prepísať na `0`, alebo pokojne nechať taký ako je.

Po vytvorení prostredia nám zostáva už len napísať obslužný kód pre vykonanie operácie.

Sčítanie sa má zrejme spustiť vtedy, keď stlačíme tlačidlo – kód programu teda zapíšeme do udalosti pri kliknutí (`onClick`).

Na vyriešenie problému použijeme premenné `a`, `b` a `vysledok`. Doposiaľ použitý typ `string` by nesplnil naše očakávanie a sčítanie takýchto premenných by realizoval ako ich spojenie. Pokiaľ chceme s údajmi pracovať ako s celými číslami, potrebujeme pre ne ako vhodný typ zvoliť `integer`. Prvým krokom by teda mohla byť ich deklarácia v tele automaticky vytvorenej procedúry:

```

procedure TForm1.Button1Click(Sender:TObject)
var a,b,vysledok:integer

begin

end;

```

Premenná *a* bude slúžiť na uloženie prvého čísla, do premennej *b* prečítame druhé. *Vysledok* bude obsahovať ich súčet.

Pred ďalším písaním kódu je potrebné zdôrazniť, že akýkoľvek reťazec (je jedno či text alebo číslo), ktorý sa číta z *Editu* (prípadne zapisuje do väčšiny komponentov) je nevyhnutne textový (teda typu *string*). Z tohto dôvodu potrebujeme pred uložením zadanej hodnoty vykonať prevod zo *stringu* na *integer* (z textového reťazca na číslo). Používa sa na to funkcia *StrToInt* (*string* na *integer*).

Výsledný program bude na základe týchto informácií začínať konverziu a uložením hodnôt z *Editov* do premenných, zistením ich súčtu na úrovni celých čísel (teda sčítanie, na aké sme zvyknutí z matematiky) a napokon zobrazením výsledku v *Labeli*.

```

procedure TForm1.Button1Click(Sender:TObject)
var a,b,vysledok:integer

begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);
  vysledok:=a+b;
  Label1.Caption:=IntToStr(vysledok);
end;

```

Vieme, že zobrazovaný text *Labelu* je uložený vo vlastnosti *Caption*. Táto je opäť typu *string*, a preto potrebujeme číselnú hodnotu previesť na textovú prostredníctvom „opačnej“ funkcie – *IntToStr*. A výsledok je na svete...

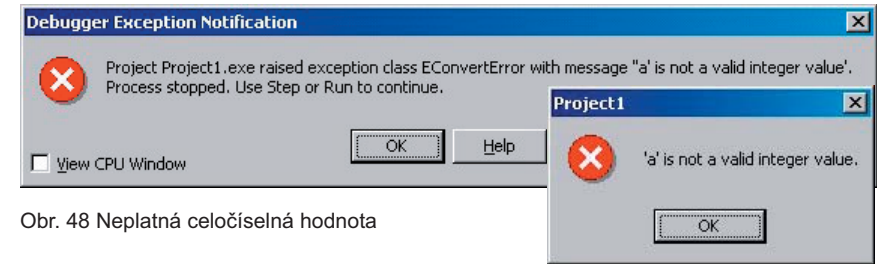
V prípade, že váš *Label* sa nevolá *Label1*, treba jeho názov v zdrojovom kóde upraviť podľa svojho. Správny názov nájdete vo vlastnosti *Name* alebo rýchlejšie v hornej časti *Object Inspector*a po kliknutí na príslušný objekt.



Obr. 47 Názov komponentu

Na záver je nevyhnutné upozornenie, že funkcia *StrToInt* je pomerne hlúpa – ak sa pokúsíte previesť na číslo prázdny text alebo neplatné číslo (pís-

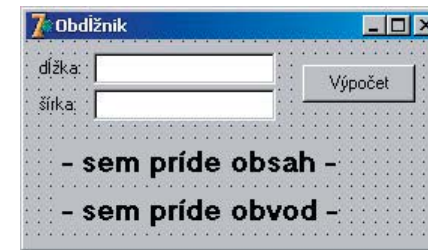
mená, medzery a pod.), program vyhodí chybu a preruší vykonávanie programu. Nedivte sa preto, ak v prípade nesprávneho čísla program zaprotestuje.



Obr. 48 Neplatná celočíselná hodnota

Správa informuje o tom, že zadaný reťazec nie je celé číslo. Vyriešiť tento problém sa naučíme neskôr prostredníctvom špeciálnej procedúry.

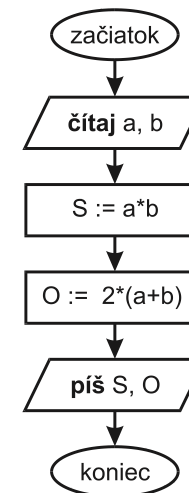
1. Doplňte do existujúceho programu tlačidlá pre súčin a rozdiel. (Postup je analogický ako v predchádzajúcom prípade, mení sa len znamienko pri vý-



počte. Premenné je potrebné deklarovať v každej procedúre vždy znova a znova.)

2. Prerobte do Delphi algoritmus na výpočet obsahu a obvodu obdĺžnika.

Obr. 49 Návrh prostredia pre výpočty



```

procedure TForm1.Button1Click(Sender:TObject);
var a,b,s,o:integer;

begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);

  s:=a*b;

  o:=2*(a+b);

  Label1.Caption:='Obsah je '+IntToStr(s);
  Label2.Caption:='Obvod je '+IntToStr(o);

end;

```

Algoritmus predstavuje sekvencia príkazov, ktoré prepíšeme do procedúry odohrávajúcej sa po kliknutí na tlačidlo v prostredí podobnom ako na obrázku.

1. Popíšte základné prostredie Borland Delphi.
2. Vysvetlite pojem komponent.
3. Vysvetlite pojem udalosťami riadené programovanie.
4. Aké komponenty využíva aplikácia v Delphi na komunikáciu s používateľom?
5. Pre zadané rozmery a, b, c vypočítajte objem a povrch hranola, kde hodnoty predstavujú rozmery jednotlivých hrán.
6. Napíšte program, ktorý pre zadaný vek vypočíta počet rokov do dôchodku (predpokladajte, že odchod na zaslúžený odpočinok je striktné stanovený na vek 65 rokov).
7. Napíšte program, ktorý pre zadané hodnoty a, b na jedno kliknutie vypočíta ich súčet, súčin a rozdiel.

4 Prepis štruktúr do programovacieho jazyka

predpoklady na zvládnutie lekcie:

- schopnosť tvorby algoritmov prostredníctvom vývojových diagramov
- základy práce v prostredí Borland Delphi

obsah lekcie:

- prepis vetvenia a všetkých druhov cyklov do programovacieho jazyka
- tvorba programov obsahujúcich spomenuté štruktúry
- údajový typ pre reálne čísla
- prehĺbenie vedomostí prostredníctvom riešenia úloh

cieľ:

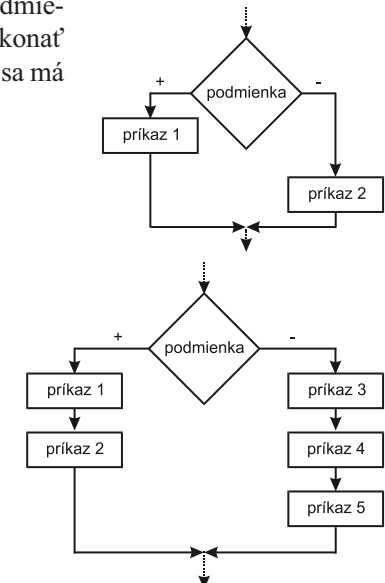
- zvládnuť prepis všeobecného algoritmu do programovacieho jazyka
- získať zručnosť pri tvorbe vizuálneho prostredia programu

Vetvenie

Prostredníctvom vetvenia vieme podmieniť vykonanie príkazov len v prípade splnenia zadanej podmienky. Rovnako ako vo vývojových diagramoch aj v programovacom jazyku potrebujeme v podmienom príkaze rozlíšiť časť, ktorá sa má vykonať v prípade splnenia podmienky a časť, ktorá sa má vykonať, ak podmienka splnená nie je.

```
if podmienka then príkaz1
    else príkaz2;
```

```
if podmienka then
begin
    príkaz1;
    príkaz2;
end
else
begin
    príkaz3;
    príkaz4;
    príkaz5;
end
```



Obr. 50 Prepis podmieneného príkazu

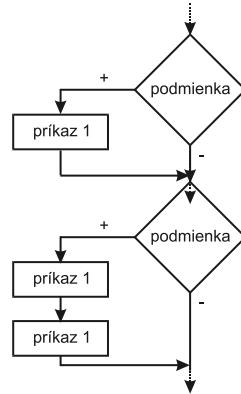
V prípade splnenia podmienky sa vykoná príkaz uvedený za kľúčovým slovíčkom `then`, v prípade nesplnenia ten, ktorý je umiestnený za kľúčovým slovom `else`. Pokiaľ chceme v jednotlivých vetvách vykonávať viac príkazov, treba ich uzavrieť medzi `begin` a `end`.

Všimnite si, že pred `else` sa bodkočiarka nedáva, pretože nejde o nový príkaz, ale len o pokračovanie príkazu `if`. Vo vetvách príkazu vetvenia môžeme použiť ľubovoľné príkazy, teda aj ďalší príkaz vetvenia, cyklus alebo inú algoritmickú konštrukciu.

V prípade neúplného vetvenia vetvu `else` jednoducho vynechávame.

```
if podmienka then príkaz1;
```

```
if podmienka then begin
    príkaz1;
    príkaz2;
end;
```



Obr. 51 Prepis podmienených príkazov

Povolená je i konštrukcia

```
if podmienka then
    else príkaz1;
```

kde sa vynechá kladná vetva a za `else` sa vložia príkazy vykonávané v prípade nesplnenia podmienky.

Napište program, ktorý nájde maximum z dvoch čísel.

Problém sme riešili už prostredníctvom vývojových diagramov, na tomto mieste ho len prepíšeme do programovacieho jazyka. Ako prvé však vyrobíme pre aplikáciu prostredie.

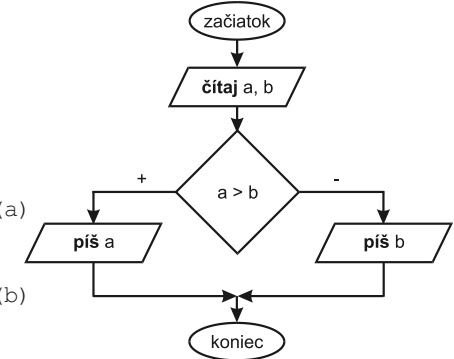


Obr. 52 Prostredie aplikácie v Delphi

Nasledovný kód sa môže odohrať po kliknutí na tlačidlo *Maximum*.

```
procedure TForm1.Button1Click(Sender:TObject);
var a,b:integer;

begin
    a:=StrToInt(Edit1.Text);
    b:=StrToInt(Edit2.Text);
    if a>b then
        Label1.Caption:=
            'Maximum je '+IntToStr(a)
    else
        Label1.Caption:=
            'Maximum je '+IntToStr(b)
end;
```



Zabezpečte, aby v prípade rovnosti podal program o tom informáciu.

```
procedure TForm1.Button1Click(Sender:TObject);
var a,b:integer;

begin
    a:=StrToInt(Edit1.Text);
    b:=StrToInt(Edit2.Text);
    if a>b then Label1.Caption:='Maximum je '+IntToStr(a)
    else begin
        if a=b then begin
            Label1.Caption:='Hodnoty sú rovnaké'
        end else begin
            Label1.Caption:='Maximum je '+IntToStr(b);
        end;
    end;
end;
```

V programe sme do podmienky `a=b` schválne vložili dvojicu `begin - end`, čím sme chceli zdôrazniť, že ich použitie je možné a niekedy kvôli prehľadnosti i žiaduce aj v prípade, ak je vo vetve len jeden príkaz.

Napište program, ktorý nájde najväčšie z troch čísel.

Pri riešení problému pomocou vývojových diagramov sme riešili úlohu dvoma spôsobmi, tu uvádzame ďalší algoritmus na riešenie tejto úlohy.


```

procedure TForm1.Button1Click(Sender:TObject);
var a,b,c,max:integer;

begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);
  c:=StrToInt(Edit3.Text);
  max:=a;
  if b>max then max:=b;
  if c>max then max:=c;
  Label1.Caption:='Maximalna hodnota je '+IntToStr(max);
end.

```

Na začiatku sme si za maximum zvolili *a* a túto hodnotu sme potom porovnávali s ostatnými dvoma – ak bola porovnávaná hodnota väčšia, našli sme nové najväčšie číslo, ktoré sme v kladnej vetve príkazu vetvenia do premennej *max* priradili.

Napište program, ktorý vydolí dve celé čísla a určí podiel a zvyšok (využite operácie div a mod pre celočíselné delenie). Ošetrte delenie nulou.

Riešenie zrejme nepotrebuje dlhý komentár. Testovanie podmienky na nulové *b* rozdelí program na dve časti. V prípade splnenia podmienky výpočet nemôže ďalej pokračovať a vypíše sa o tom informácia. Ak *b* je nenulové prebehne jednoduchá sekvencia s výpočtom a výpisom výsledkov.

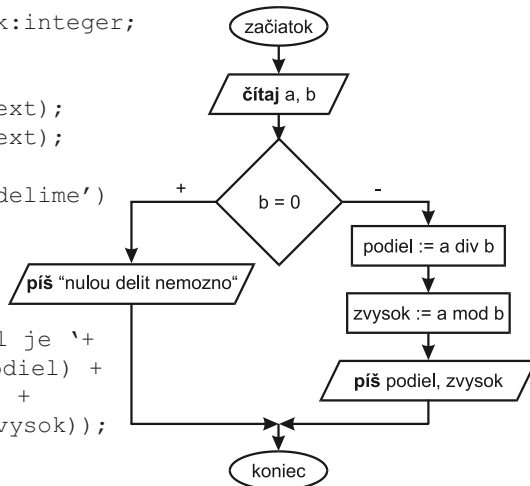
```
var a,b,podiel,zvysok:integer;
```

```

begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);
  if b=0 then
    ShowMessage('0 nedelime')
  else begin
    podiel:=a div b;
    zvysok:=a mod b;

    ShowMessage('podiel je '+
      IntToStr(podiel) +
      ' zvyšok: ' +
      IntToStr(zvysok));
  end;
end;
end;

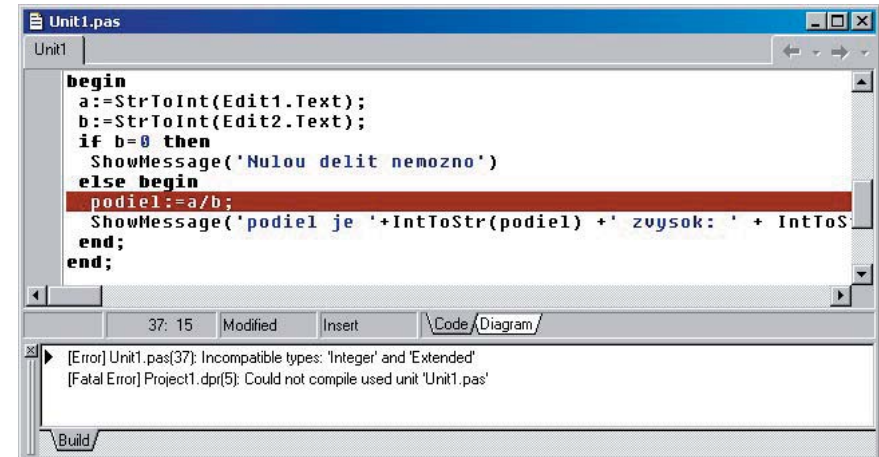
```



Reálne čísla

Napište program, ktorý vypočíta podiel dvoch celých čísel a ošetrí delenie nulou.

Program, ktorý sme vytvorili v predchádzajúcej kapitole, realizuje celočíselné delenie. Ak chceme získať ako podiel desatinné číslo, použijeme operáciu delenia „/“. Ak ňou však nahradíme operáciu `div`, kompilátor nám odmietne program preložiť.



Obr. 53 Vyhlásenie kompilátora

Dôvodom je nesúlad typov – pre celočíselný typ (`integer`) nie je operácia reálneho delenia („/“) definovaná. Na prácu s desatinnými číslami je potrebné zaviesť typ `real` podporujúci všetky (nám známe) matematické operácie.

Na prevod použijeme funkciu `FloatToStr`. Pojem `Float` je vyjadrením pre číslo s pohyblivou rádovou (desatinnou) čiarkou. Táto funkcia prevedie reálne číslo na text v „ľudskej podobe“, nie je potrebné špecifikovať počty desatinných miest.

Napísané programy (ako v *Delphi*, tak i v *pascal*) dokážu deliť len celé čísla, typ vstupných premenných sme ponechali `integer`. Pokiaľ by sme chceli používať program nielen na delenie celých čísel, zmeníme ho na `real`.

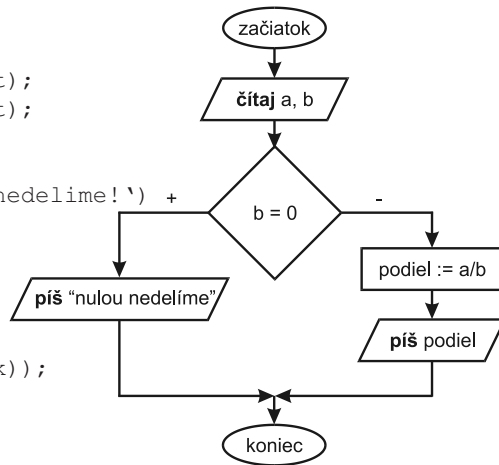
Program by pri použití celočíselných vstupných hodnôt mohol mať tvar:

```

var a,b:integer;
    vysledok:real;

begin
a:=StrToInt(Edit1.Text);
b:=StrToInt(Edit2.Text);
if b=0 then
begin
ShowMessage('Nulou nedelíme!');
end
else
begin
vysledok:=a/b;
ShowMessage(
FloatToStr(vysledok));
end;
end;
end;

```



V prípade prevodu reálneho čísla z Editu používame funkciu StrToFloat.:

```

a:=StrToFloat(Edit1.text);
b:=StrToFloat(Edit2.text);

```

Funkcia opäť nie je odolná voči chybnému vstupom rovnako ako StrToInt. Ako desatinný oddeľovač je nutné používať znak definovaný v *Miestnych nastaveniach Windows* (zvyčajne desatinná čiarka).

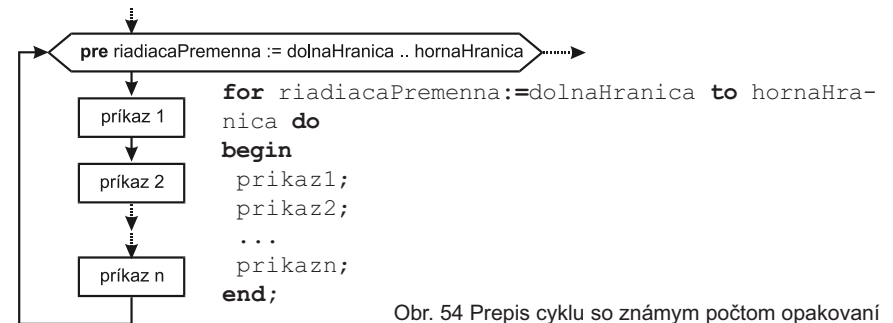
1. Napíšte program, ktorý vypočíta obsah a obvod kruhu.
2. Zistite absolútnu hodnotu reálneho čísla pomocou príkazu vetvenia.
3. Napíšte program, ktorý vypočíta BMI index a vypíše, či máte alebo nemáte nadváhu. BMI (telesný hmotnostný index) sa vyráta ako podiel hmotnosti v kilogramoch a druhej mocniny výšky v metroch, $BMI < 18,5$ podváha, $18,5 \leq BMI < 25$ normálna hmotnosť, $25 \leq BMI < 30$ nadváha, $BMI > 30$ obezita.

Cykly

Silnou zbraňou počítačových systémov a vo všeobecnosti automatov je schopnosť opakovať tú istú činnosť neúnavne prakticky ľubovoľný počet krát. Prostriedkom programovacieho jazyka, ktorý túto činnosť zabezpečuje, je cyklus.

Cyklus so známym počtom opakovaní

Ako prvý sme si predstavili cyklus so známym počtom opakovaní, ktorého prepis vyzerá nasledovne:



Obr. 54 Prepis cyklu so známym počtom opakovaní

Cyklus sa opakuje pre premennú riadiacaPremenna od hodnoty dolnaHranica po hodnotu hornaHranica tak, že hodnota riadiacej premennej na začiatku nadobudne hodnotu dolnaHranica a po každom vykonaní príkazov v tele cyklu sa automaticky zvýši o 1. Ak je už na začiatku dolnaHranica > hornaHranica, cyklus vôbec neprebehne. Ak sú rovné, prebehne práve raz.

Okrem cyklu, v ktorom sa riadiaca premenná mení od menšej k väčšej hodnote, existuje aj cyklus opačný, v ktorom kľúčové slovo to nahradíme downto – hodnota riadiacej premennej sa potom po každom zopakovaní cyklu o 1 zmenší. Rovnako ako v ostatných prípadoch je možné použiť v cykle jeden príkaz bez begin – end alebo viac príkazov uzavretých medzi nimi.

Napíšte program na zistenie súčtu prvých n prirodzených čísel, ktorých počet zadáte na vstupe.

Na zistenie súčtu použijeme premennú sucet, do ktorej budeme postupne pripočítavať hodnoty, ktoré bude nadobúdať riadiaca premenná. Cyklus bude bežať od 1 po zadanú hodnotu.


```

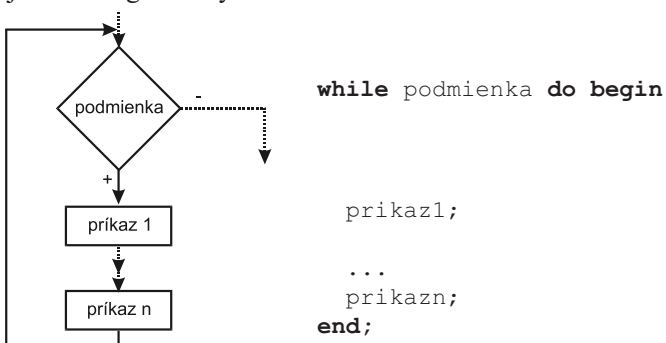
var i,n,sucet:integer;
begin
  n:=StrToInt(Edit1.Text);
  sucet:=0;
  for i:=1 to n do sucet:=sucet+i;
  ShowMessage(IntToStr(sucet));
end;

```

1. Zistite pre zadané číslo faktoriál ($n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$).
2. Napíšte program na zistenie súčinnu všetkých celých čísel nachádzajúcich sa medzi dvoma zadanými hodnotami.
3. Napíšte algoritmus na zistenie súčinnu dvoch celých čísel pre zariadenie, ktoré nepozná operáciu násobenia (nahradte ju kombináciou cyklu a sčítania), napr. namiesto $4 * 3$ sa výsledok získa ako $3 + 3 + 3 + 3$.

Cyklus s neznámym počtom opakovaní

Zo štúdia vývojových diagramov vieme, že nie vždy dokážeme vopred určiť, koľkokrát má cyklus prebehnúť. Na zápis úloh v takomto prípade môžeme použiť **cyklus s podmienkou na začiatku**, ktorý sme si už tiež predstavili – nepoužíva kontrolu hodnoty riadiacej premennej, ale obsahuje podmienku umiestnenú pred telom cyklu, ktorá sa postará o jeho ukončenie. Prepis vývojového diagramu vyzerá nasledovne:



Obr. 55 Prepis cyklu s podmienkou na začiatku

Pri tomto cykle nesmieme zabudnúť meniť hodnoty premenných, od ktorých závisí splnenie či nespĺnenie podmienky, vo vnútri cyklu tak, aby bolo možné dosiahnuť nespĺnenie podmienky. V opačnom prípade sa môže stať, že cyklus pobeží donekonečna.

Prepíšte do programovacieho jazyka algoritmus, ktorý zistí zvyšok pri delení dvoch čísel, ak máte povolenú len operáciu rozdielu.

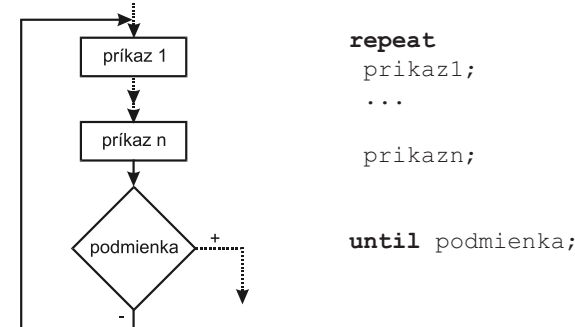
Na základe vývojového diagramu postup prepíšeme do pascalu (odlišnosť s *Delphi* je stále len vo vstupoch a výstupoch). Odčítavanie bude prebiehať dovtedy, kým bude od čoho odčítavať. Ak číslo, od ktorého odčítavame, je menšie ako to, ktoré odčítavame, treba skončiť. Ak sú rovnaké, odpočítame ich a v ďalšom kroku cyklus ukončíme (pokiaľ máte nejasnosti vo fungovaní algoritmu, pozrite si sledovaciu tabuľku vo vývojových diagramoch).

```

var a,b:integer;
begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);
  while a>=b do begin
    a:=a-b;
  end;
  ShowMessage('Zvyšok je'+ IntToStr(a));
end;

```

Alternatívou k predchádzajúcemu cyklu je **cyklus s neznámym počtom opakovaní a podmienkou na konci**. Už vieme, že pre tento cyklus je charakteristické minimálne jedno vykonanie tela cyklu.



Obr. 56 Prepis cyklu s podmienkou na konci

Nie je nutné používať `begin - end`, pretože telo cyklu je ohraničené dvojicou `repeat - until`. Druhou odlišnosťou voči predchádzajúcemu cyklu je, že príkazy sa vykonávajú dovtedy, kým je podmienka na konci nespĺnená – keď sa splní, cyklus skončí.

Prepíšme do tejto podoby predchádzajúci algoritmus.

Algoritmus sa mierne upraví – pokiaľ bude už na začiatku $a < b$, nie je potrebné, aby cyklus vôbec začal a okamžite sa môže vypísať výsledok. V opačnom prípade bude bežať dovtedy, kým nenastane stav, že $a < b$.

```
var a,b:integer;
begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);
  if a<b then ShowMessage('Zvysok je' + IntToStr(a))
  else begin
    repeat
      a:=a-b;
    until a<b;
    ShowMessage('Zvysok je' + IntToStr(a));
  end;
end;
```

1. Pridajte do algoritmu výpočet podielu (napr. pri každom odčítaní zväčšiť hodnotu premennej podiel o 1).
2. Vypočítajte ciferný súčet čísl daného prirodzeného čísla N (využite fakt, že poslednú cifru z čísla viete „odtrhnúť“ prostredníctvom operácie mod).
3. Prostredníctvom Euklidovho algoritmu nájdite najväčšieho spoločného deliteľa dvoch čísel.

5 Práca s textom

predpoklady na zvládnutie lekcie:

- schopnosť algoritmizovať problémy
- skúsenosti s prácou v prostredí Borland Delphi
- znalosť údajových typov string a integer

obsah lekcie:

- práca s textovými reťazcami
- spektrum funkcií na podporu práce s textom
- ordinálne a neordinálne typy
- ASCII tabuľka, logický typ, priorita operátorov

cieľ:

- zoznámenie sa s paletou údajových typov pascalu (Delphi)
- schopnosť riešiť typické problémy v prostredí Delphi

Textové reťazce

Doposiaľ sme uviedli tri údajové typy: `integer`, `real` a `string`. S číselnými typmi sme pracovali pomerne aktívne, typ `string` sme používali zatiaľ pri zadávaní vstupov a zobrazovaní výstupov programu.

V pascle dokážeme do premennej tohto typu umiestniť maximálne 255 znakov, *Delphi* nám ponúka pre jednu premennú až 2^{32} znakových pozícií.

Vieme, že výsledkom operácie zretazovania dvoch textových reťazcov, ktoré označujeme symbolom `+`, je tretí reťazec, obsahujúci spojené reťazce v tom poradí, v akom do zretazovania vstupovali (napr. `'Jožo' + 'Mara' = 'JožoMara'` alebo `'1'+'2'+'3'='123'` a pod.).

Do premennej typu `string` dokážeme prečítať ľubovoľný textový reťazec, čo môžeme využiť pri vytváraní aplikácií, ktoré pracujú s číselnými premennými a majú byť odolné voči chybám. V prostredí *Delphi* sice načítavame údaje prostredníctvom textových polí, no bolo by vhodné, aby sme prostredníctvom kontroly vstupných hodnôt informovali používateľa o chybe.

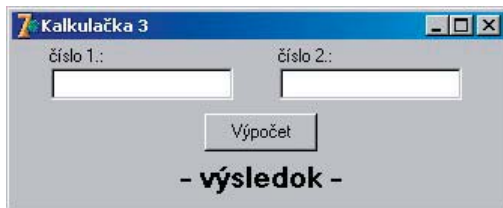
Na zistenie, či zadaná hodnota je skutočne číslo, máme k dispozícii procedúru `Val` (pozri ďalej). Táto prevádza textovú hodnotu na číselnú a v prípade, že sa prevod nepodarí, informuje nás prostredníctvom premennej o pozícii, ktorá bola príčinou neúspešného prevodu. Má podobu:

```
Val(textová premenná alebo hodnota, číselná premenná, kod)
```

Ak je prevod úspešný, textovú hodnotu zmení na číselnú podľa toho, akého je číselná premenná typu (`integer`, `real`) a premenná `kod` nadobudne hodnotu 0. V prípade neúspechu do premennej `kod` vloží chybový kód informujúci o pozícii, na ktorej chyba nastala.

Vytvorte aplikáciu na sčítavanie dvoch čísel, ktorá bude odolná voči zadávaniu nekorektných hodnôt.

V Delphi sa výpočet vykonáva ako reakcia na stlačenie tlačidla, ktorej súčasťou nie je zadávanie vstupných hodnôt premenných. Tie sa zadávajú v editovacích riadkoch (komponenty `Edit`). Tlačidlo môžeme ľubovoľne veľakrát stlačiť a výpočet opakovať a medzi stlačeniami prepisovať hodnoty vstupov v editovacích riadkoch. Aplikácia sa ukončí až po zatvorení okna:



Obr. 57 Formulár pre výpočet súčtu

```
procedure TForm1.Button1Click(Sender: TObject);
var a,b,vysledok,kod: integer;
begin
  Val(Edit1.Text,a,kod);
  if kod>0 then begin
    ShowMessage('Zle 1. cislo na pozicii '+IntToStr(kod));
    exit; // nestrukturovany, no uzitocny prikaz ukoncujucci proceduru
  end;
  Val(Edit2.Text,b,kod);
  if kod>0 then begin
    ShowMessage('Zle 2. cislo na pozicii '+IntToStr(kod));
    exit;
  end;
  vysledok := a + b;
  Label3.Caption:=StrToInt(vysledok);
end;
```

Príkaz `exit` zabezpečí ukončenie procedúry. Vedeli by sme síce algoritmus s použitím úplných vetvení prepísať do podoby nevyužívajúcej tento neštruktúrovaný príkaz, no jeho čitateľnosť by sa zhoršila.

Ak teda aplikácia narazí na nesprávne zadanú číselnú hodnotu, upozorní na to používateľa v okne so správou a vráti sa do stavu čakania (napr. na prepísanie hodnoty alebo na udalosť kliknutia na tlačidlo).

Zatiaľ čo v klasickom pascali máme k dispozícii na vkladanie poznámok, ktoré kompilátor neberie do úvahy dvojicu `{}`, v Delphi pribudli znaky `//` umožňujúce používateľovi vloženie poznámky napravo od znakov do konca riadku.

V nasledujúcich príkladoch bude pre nás užitočné vedieť zistiť počet znakov uložených v reťazci. Služí na to funkcia `length`, ktorá v príkaze

```
dlzka:=Length(mojString)
```

vráti počet znakov, ktoré premenná `mojString` obsahuje a tento sa vloží do premennej `dlzka` (musí byť celočíselného typu).

Premenná typu `string` umožňuje prístup k textu nielen ako k celku, ale aj k jeho jednotlivým znakom (písmenám) prostredníctvom poradového čísla znaku (indexu) v hranatých zátvorkách `[]` (napr. `mojString[5]` obsahuje piaty znak reťazca).

Pozor: prvý znak reťazca získame ako `mojString[1]`, a nie `mojString[0]`.

Číselné versus textové hodnoty

Nasledujúce úlohy sú príkladom toho, že niektoré problémy, ktoré sú formulované pre čísla, sa dajú jednoduchšie riešiť, keď sa na čísla pozeráme ako na reťazec cifier a namiesto číselných premenných použijeme textové.

Zistite, koľko ráz sa v zadanom čísle opakuje cifra 3.

```
var retazec:string;
    i,vyskytov:integer;

begin
  retazec:=Edit1.Text; {precitame retazec}
  vyskytov:=0; {na zaciatku mame 0 vyskytov}
  {prejdeme od 1 po posledny znak}
  for i:=1 to length(retazec) do
    {ak je na i-tej pozicii 3, zvyssime pocet vyskytov}
    if retazec[i]='3' then vyskytov:=vyskytov+1;
  {vypis}
  Label1.Caption:='Pocet vyskytov je '+IntToStr(vyskytov);
end;
```

Jednoducho sme prostredníctvom cyklu prešli po všetkých cifrách rovnakým spôsobom akoby sme ich mali napísané na papieri (a keď sme zbadali 3, urobili sme si čiarku).

Ak použijeme premennú typu `integer`, je postup iný: z čísla dokážeme jednoducho získať len jeho poslednú cifru – prostredníctvom zvyšku pri delení desiatimi (napr. $423 \bmod 10$ je 3). Po prečítaní poslednej cifry číslo vydáme desiatimi ($423 \div 10$ je 42), aby sme sa poslednej cifry „zbavili“ a dostali sa k nasledujúcej. Postup opakujeme dovtedy, kým všetky cifry neprečítame a spracúvané číslo neklesne na 0.

```
var mojeCislo, vyskytov: integer;

begin
  mojeCislo:=StrToInt(Edit1.Text); {precitame cislo}
  vyskytov:=0; {na zaciatku mam 0 vyskytov}
  while mojeCislo>0 do begin {kym nespracujeme cely vstup}
    {ak je zvysook po deleni 3, je na poslednom mieste 3}
    if mojeCislo mod 10=3 then vyskytov:=vyskytov+1;
    mojeCislo:=mojeCislo div 10; {odstranime poslednu cifru}
  end;
  Label1.Caption:='Pocet vyskytov je '+IntToStr(vyskytov);
end;
```

Rovnako by sme oboma spôsobmi mohli pracovať aj pri riešení ďalších úloh, vyberieme si však podľa nás názornejšie postupy.

Napište program, ktorý zistí ciferný súčet zadaného čísla.

Cifru čísla môžeme oddeliť, alebo s ňou manipulovať priamo vo výpočte:

```
var retazec:string;
    sucet,i:integer;

begin
  retazec:=edit1.Text;
  sucet:=0;
  for i:=1 to length(retazec) do
    sucet:=sucet+StrToInt(retazec[i]);
  ShowMessage('Ciferny sucet je '+IntToStr(sucet));
end;
```

Napište program, ktorý v zadanom čísle nájde maximálnu cifru.

Podstata riešenia spočíva v tom, že postupne prezeráme všetky cifry čísla a porovnávame ich s dovtedy nájdeným maximom. Ak nájdeme väčšiu cifru, uložíme si ju ako nové maximum. Na začiatku zvolíme za maximum 0, čo je určite menej, resp. pri najhoršom rovné ako najväčšia cifra v čísle.

```
var mx,i,cifra:integer;
    retazec:string;

begin
  retazec:=Edit1.Text;
  mx:=0; {zatiaľ najväčšia najdená cifra}
  for i:=1 to length(retazec) do begin
    cifra:=StrToInt(retazec[i]);
    {ak je aktuálna cifra väčšia ako najväčšia doposiaľ
    najdená, tak sa táto stáva najväčšou}
    if cifra>mx then mx:=cifra;
  end;
  ShowMessage('Maximálna cifra je '+IntToStr(mx));
end;
```

Upravte program tak, aby dokázal pracovať aj v prípade zadania desiatinného čísla (desatinnú bodku alebo čiarku môže ignorovať).

Napište program, ktorý vypíše zrkadlový obraz zadaného čísla.

V tomto prípade nepotrebujeme použiť žiadne funkcie prevádzajúce reťazec na číslo, lebo nebudeme vykonávať žiadne aritmetické operácie. Reťazec budeme prechádzať od začiatku do konca a každý „skúmaný“ znak umiestnime vždy na začiatok premennej uchovávajúcej si zrkadlový obraz.

```
var retazec,zrkadlo:string;
    i:integer;

begin
  retazec:=Edit1.Text;
  zrkadlo:=''; // do premennej vložíme prázdny reťazec - nič
  for i:=1 to length(retazec) do
    zrkadlo:=retazec[i]+zrkadlo;
  ShowMessage('Zrkadlom '+retazec+' je '+zrkadlo);
end;
```

Vytvorte pre riešenie úlohu sledovaciu tabuľku.

Riešte nájdenie zrkadlového obrazu prostredníctvom celočíselných premenných, operácií \div a \bmod . Aký problém sa pre určitú skupinu čísel môže vyskytnúť?

Na záver môžeme už len podotknúť, že výber vhodného typu v programe závisí od problému, ktorý chceme riešiť. Pokiaľ chceme pristupovať k

jednotlivým cifrám čísla, je rýchlejšie (ako z programátorského, tak i zo systémového hľadiska) a zrozumiteľnejšie použiť typ `string`. Navyše, ak sa zamyslíme nad predchádzajúcimi úlohami, zistíme, že po drobnej úprave sú riešenia funkčne nielen pre čísla, ale pre ľubovoľný textový reťazec.

Ak máme v programe vykonávať s číslami aritmetické operácie, je potrebné zvoliť číselný typ (ako napríklad v úlohách, kde sme číslo zadané v *Delphi* ako reťazec previedli na číselný typ).

Podporné funkcie

Vzhľadom na svoju univerzálnosť existuje pre prácu s premennými typu `string` mnoho funkcií podporujúcich a zrýchľujúcich prácu. Základné sú uvedené v tabuľke.

Funkcia	Popis	Syntax príklad
Length	vráti dĺžku reťazca	<ul style="list-style-type: none"> • <code>dlzka:=length(mojString);</code> • napr.: <code>mojString:='aha svet';</code> <code>dlzka:=length(mojString);</code> <code>=> dlzka obsahuje hodnotu 8</code>
Val	<ul style="list-style-type: none"> • skonvertuje reťazec (<code>string</code>) na číslo • v prípade neúspechu nepreruší vykonávanie programu, ale uloží pozíciu nepreložiteľného znaku do premennej 	<ul style="list-style-type: none"> • <code>Val(stringSCislo,ciselnaPremenna,kod);</code> <code>ciselnaPremenna</code> - obsahuje konvertovanú hodnotu, <code>kod</code> - vracia pozíciu v reťazci, na ktorej došlo k chybe, ak bol priebeh bezchybový nadobúda 0, • napr.: <code>Val('22',i,kod);</code> <code>=> i obsahuje číselnú hodnotu 22, kod údaj 0</code>
Str	konvertuje číslo na <code>string</code>	<ul style="list-style-type: none"> • <code>Str(zadaneCislo,mojString);</code> do premennej <code>mojString</code> vloží číselnú hodnotu alebo obsah premennej konvertovaný na <code>string</code>, • napr.: <code>cislo:=10;</code> <code>str(cislo,mojString);</code> <code>=> mojString obsahuje hodnotu '10'</code>
Concat	<ul style="list-style-type: none"> • zlučuje reťazce • analógia s operáciou + 	<code>zlučeny:=Concat(prvy,druhy,treti);</code> to isté ako <code>zlučeny:=prvy + druhy + tretí;</code>

Pos	zistí či sa jeden reťazec nachádza v inom	<ul style="list-style-type: none"> • <code>pozicia:=Pos(hladany,retazec);</code> ak sa obsah premennej <code>hladany</code> nachádza v obsahu premennej <code>retazec</code>, do premennej <code>pozicia</code> sa uloží poradové číslo (index) znaku, na ktorom začína zadaný hľadaný reťazec v zadanom prehľadávanom reťazci, pokiaľ sa nevyskytuje, <code>Pos</code> vráti hodnotu 0, • napr.: <code>poz:=Pos('ma','moja mama má maslo');</code> <code>=> poz nadobudne hodnotu prvého výskytu „ma“, t.j. 6</code>
Copy	vráti (skopíruje) časť zo zadaného reťazca	<ul style="list-style-type: none"> • <code>vysledok:=Copy(zdroj,indexZac,pocZnakov);</code> z premennej <code>zdroj</code> skopíruje reťazec s počtom znakov od pozície zadanej v <code>indexZac</code>, • napr.: <code>s:= Copy('značka',3,4);</code> <code>=> vráti 4 znaky od 3. pozície, t.j. 'ačka'</code>
Delete	vymaže z reťazca podreťazec	<ul style="list-style-type: none"> • <code>Delete(retazec,zaciatocnaPozicia,pocetZnakovNaVymazanie);</code> z premennej <code>retazec</code> vymaže časť začínajúcu na danej začiatkovej pozícii a pozostávajúci zo zadaného počtu znakov, • napr.: <code>s:='kobyła';</code> <code>Delete(s,3,2);</code> <code>=> vymaže od 3. pozície 2 znaky, t.j. zostane 'kola'</code>
Insert	zadaný reťazec vloží do iného reťazca	<ul style="list-style-type: none"> • <code>Insert(vkladanyRetazec,cielovyRetazec,poziciaVlozenia);</code> - <code>vkladanyRetazec</code> vloží do premennej <code>cielovyRetazec</code> tak, že ho vsunie na zadanú pozíciu a ostatné znaky posunie, • napr.: <code>s:='kola';</code> <code>Insert('by',s,3)</code> <code>=> do reťazca s vloží od 3. pozície reťazec 'by' - výsledkom bude 'kobyła'</code>
UpperCase	• prevedie znaky reťazca na veľké písmená, nefunguje pre diakritiku	<ul style="list-style-type: none"> • <code>velkeZnaky:=UpperCase(retazec);</code> • napr.: <code>s:=UpperCase('TeSt');</code> <code>=> s nadobudne hodnotu 'TEST'</code>
LowerCase	• prevedie znaky reťazca na malé písmená, nefunguje pre diakritiku	<ul style="list-style-type: none"> • <code>maleZnaky:=LowerCase(retazec);</code> • napr.: <code>s:=LowerCase('TeSt');</code> <code>=> s nadobudne hodnotu 'test'</code>

Tab. 5 Funkcie pre prácu s reťazcami

Zistíte, koľkokrát sa nachádza zadaný reťazec v inom zadanom reťazci.

Na nájdenie výskytu reťazca v inom reťazci máme k dispozícii funkciu `Pos`, ktorá vráti pozíciu prvého výskytu v reťazci, ak sa tam hľadaný reťazec nachádza alebo hodnotu 0, ak sa nevyskytuje. Pokiaľ chceme nájsť prvý výskyt, funkcia je postačujúca, ak však chceme zistiť druhý alebo ďalší výskyt, nevystačíme s ňou.

Aby sme sa boli schopní skontrolovať viacnásobný výskyt reťazca, potrebujeme vždy po jeho nájdení z reťazca „preskúmanú časť“ odstrániť – vtedy sa bude pri prehľadávaní používať časť, ktorú sme ešte neprehľadali.

Ak máme napr. v reťazci `hopsarasa` nájsť počet výskytov `sa`, postupujeme nasledovne:

`sa` v `hopsarasa` nájdeme na 4. pozícii, výskyt bude 1 a vymažeme prvé 4 znaky,
`sa` v `arasa` nájdeme zasa na 4. pozícii, zvýšime výskyt a vymažeme 4 znaky,
`sa` v `a` už nenájdeme, `Pos` vráti hodnotu 0 a skončíme.

```
var hladany, celyText:string;
    pocet, pozicia:integer;

begin
    celyText:=Edit1.text;
    hladany:=Edit2.text;
    pocet:=0;           {pocet vyskytov}
    while pos(hladany,celyText)>0 do begin {kym sa vyskytuje}
        {vrati poziciu prveho vyskytu reťazca}
        pozicia:=pos(hladany,celyText);
        pocet:=pocet+1;           {zvysi pocet vyskytov}
        delete(celyText,1,pozicia); {vymaze zaciatok reťazca}
    end;
    ShowMessage(IntToStr(pocet));
end;
```

Napište program, ktorý vypíše zo zadaného slova každý druhý znak.

Údajový typ Char

`String` bol do pascalu implementovaný až v jeho novších verziách, na uchovávanie textových údajov sa spočiatku využíval typ `char`. Premenná tohto typu umožňuje uchovanie jediného znaku (v predchádzajúcich úlohách sme prístup k jedinému znaku mali zabezpečený v podobe napr. `mojStr[i]`) a `string` vzniká vlastne až spojením viacerých znakov.

Ak porovnáme v pascali (či *Delphi*) dva znaky uložené v premenných (pozor, netýka sa to názvov premenných, ale ich obsahu), ktoré sa bežnému smrteľníkovi zdajú rovnaké, nemusí to byť celkom pravda: 'A' totiž nie je to isté čo 'a'. Dôvodom je kódovacia tabuľka (v našom prípade vystačíme s ASCII tabuľkou), ktorá má každý zo znakov umiestnený na inej pozícii.

	0	1	2	3	4	5	6	7	8	9
30			medzera	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	?	'	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	del		

Obr. 58 ASCII tabuľka

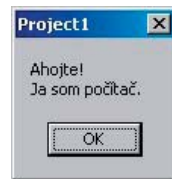
Napr. znak `\a` je umiestnený na pozícii 97, znak `\A` na 65. Znak s kódom 0–31 nemajú vizuálnu reprezentáciu, sú označované ako riadiace (reprezentujú napr. pohyb kurzora doprava, vymazanie znaku, odriadkovanie, zrušenie ESC a pod.)

Zobraziť pozíciu ľubovoľného znaku v ASCII tabuľke dokážeme prostredníctvom funkcie `Ord`, napr. `Ord('A')` vráti hodnotu 65 a `Ord('a')` hodnotu 97.

Opačnou funkciou k `Ord` je funkcia `Chr`, ktorú použijeme na zobrazenie znaku uloženého na konkrétnej pozícii, napr. `Chr(65)` vráti `'A'` alebo `Chr(97)` je `'a'`.

Zaujímavým (riadiacim) znakom je znak ukrytý na pozícii 13, ktorý v texte spôsobí odriadkovanie (Enter), napr. okno s odriadkovaním zobrazí v *Delphi* príkaz:

```
ShowMessage('Ahojte!' + chr(13) + 'Ja som počítač.');
```



Obr. 59 Odriadkovanie v Delphi

Ordinálne a neordinálne typy

Vďaka kódovaniu znakov v ASCII tabuľke dokážeme pre každý znak určiť jeho nasledovníka i predchodcu. Typ, v ktorom dokážeme o každej jeho hodnote povedať, za akou nasleduje a akú predchádza, označujeme ako **ordinálny typ**.

Okrem typu `char` je ordinálnym typom i typ `integer`. To, čo platí pre celé čísla, však už nemusí byť pravdivé pre čísla reálne – pre hodnotu typu `real` nasledovníka ani predchodcu určiť nedokážeme, veď na základe čoho by sme povedali, či za 8.1 nasleduje 8.11, 8.101 alebo 8.2? Takéto typy označujeme ako **neordinálne**.

Pre ordinálne typy máme v pascali definované funkcie:

- `Succ` – nasledovník, vráti nasledujúcu hodnotu – v prípade `integer` číslo o jedna väčšie, napr. `Succ('A') = 'B'`, `Succ(19) = 20`.
- `Pred` – predchodca, vráti predchádzajúcu hodnotu – v prípade `integer` číslo o jedna menšie, napr. `Pred('A') = '@'`, `Pred(19) = 18`.
- `High` – najväčšia možná hodnota, akú môže nadobudnúť typ, z ktorého pochádza argument funkcie, napr.:

```
var c:integer;
begin
  ShowMessage(inttostr(high(c)));
end;
```

- `Low` – najmenšia možná hodnota, akú môže nadobudnúť typ, z ktorého pochádza argument funkcie.

Pre celočíselné typy sú navyše definované procedúry:

- `Inc` – zvýši hodnotu premennej o 1 (ak `c=10`, tak po `Inc(c)` nadobudne hodnotu 11),
- `Dec` – zníži hodnotu premennej o 1 (ak `c=10`, tak po `Dec(c)` nadobudne hodnotu 9)

Napište program, ktorý zakóduje text tak, že posunie jednotlivé písmená abecedy o 3 pozície, napr. Ahoj bude: A – D, h – k, o – r, j – m, teda Dkrm.

```
var slovo,vysledok:string;
    znak:char;
    i,pozicia:integer;
begin
  slovo:=Edit1.Text;
```

```
vysledok:='';
for i:=1 to length(slovo) do begin
  znak:=slovo[i];
  pozicia:=ord(znak); {ord vrati poziciu v ASCII tabulke}
  pozicia:=pozicia+3; {posunie sa o tri pozicie}
  znak:=chr(pozicia); {precita znak z novej pozicie v ASCII}
  vysledok:=vysledok+znak; {prida sa do noveho slova}
end;
ShowMessage(vysledok);
end;
```

Postupnosť príkazov v tele cyklu možno zapísať i jediným riadkom:

```
vysledok:=vysledok+chr(ord(slovo[i])+3)
```

Upravte program tak, aby v prípade, že príde na koniec abecedy, začal odznova, napr. Z zakóduje ako C, y zakóduje ako b atď.

Užitočnou vlastnosťou ordinálnych typov je možnosť ich použitia v riadiacej premennej cyklu.

Napište program, ktorý nájde znak vyskytujúci sa v zadanom reťazci pozostávajúcom z malých písmen najviac rás.

```
var slovo:string;
    znak,i:char;
    j,pocet,mx:integer;
begin
  slovo:=Edit1.Text;
  znak:=''; {najvyskytovanejsi znak}
  mx:=0; {maximalny pocet vyskytov}
  for i:='a' to 'z' do begin {prejde znaky abecedy}
    pocet:=0; {pre pocet vyskytov skumaneho znaku}
    for j:=1 to length(slovo) do {prejde po slove}
      {ak najde v slove znak zhodny so skumanym zvysi pocet}
      if slovo[j]=i then pocet:=pocet+1;
      if pocet>mx then begin {ak pocet je vyssi ako dopo-}
        mx:=pocet; {sial maximalny, zapamata si}
        znak:=i; {znak i pocet jeho vyskytov}
      end;
    end;
  ShowMessage('Najcastejsie sa vyskytuje' + znak);
end.
```

Myšlienka programu využíva cyklus s riadiacou premennou *i* na „prejdenie“ po znakoch *a-z*. Prostredníctvom cyklu s riadiacou premennou *j* sa skúma počet výskytov aktuálneho znaku v textovom reťazci uloženom v premennej *slovo*. Ak je počet vyšší ako doposiaľ najväčší, zmení sa obsah premennej *mx* podľa neho a takisto sa zapamätá *i* znak, ktorý tento najväčší počet dosiahol.

Logický typ

Typ `boolean` predstavuje jednoduchý ordinálny typ schopný nadobúdať len dve hodnoty. Tieto sú reprezentované pravdivosťnými hodnotami `True` a `False` (prípadne `pravda` a `nepravda` alebo `áno` a `nie`). V niektorých iných programovacích jazykoch sa tento typ nepoužíva a v `pascal` ho tiež možno veľmi jednoducho nahradiť inými typmi (`integer`), no jeho použitie zlepšuje čitateľnosť kódu.

Pravdivosťná hodnota nám už v predchádzajúcich príkladoch poslúžila pri rozhodovaní prostredníctvom podmienky. Zápis

```
if a>b then...
```

by sme mohli prepísať i nasledovne:

```
var podmienka:boolean;
    a,b:integer;

begin
    podmienka:=a>b;
    if podmienka then...
end.
```

Zistite či sa v zadanom neprázdnom reťazci nachádza nula.

Ak sa nad zadaním zamyslíme, zistíme, že nie je potrebné prechádzať celý reťazec, ale stačí ho skúmať dovtedy, kým sa nezistí, že sa v ňom našla 0. V prípade, že sa nenájde, treba skončiť pri poslednom znaku.

Premenná `bolaNula` nadobudne na začiatku hodnotu `false`, ktorá nás informuje o tom, že nula doposiaľ nájdená nebola, pokiaľ sa pri prehládávaní vyskytne, bude hodnota premennej zmenená na `true` (pravda). Cyklus bude bežať dovtedy, pokiaľ premenná `bolaNula` nenadobudne hodnotu `true` (t.j. kým nebude nájdená nula) alebo kým nebude hodnota premennej *i* reprezentujúca poradové číslo skúmaného znaku väčšia ako dĺžka reťazca.

```
var retazec:string;
    i:integer;
    bolaNula:boolean;

begin
    retazec:=Edit1.Text;
    bolaNula:=false;
    i:=1;
    repeat
        if retazec[i]='0' then bolaNula:=true;
        inc(i);
    until bolaNula or (i>length(retazec));
    if bolaNula then ShowMessage('Nula tu je')
        else ShowMessage('Nula tu nie je');
end;
```

Na testovanie ukončenia cyklu sme použili podmienku zloženú z dvoch jednoduchších podmienok (logických výrazov) spojených logickou spojkou `or` (alebo).

Logický výraz obsahujúci ďalšie operácie sme **uzavreli do samostatných zátvoriek**, pretože v opačnom prípade by nám kompilátor generoval chybové hlásenie kvôli nesúladu typov (pozri *Priority operátorov*).

Na prácu s logickými hodnotami typu `boolean` má programovací jazyk `Pascal` k dispozícii logické operácie `and` (a súčasne), `or` (alebo) a `not` (negáciu, opak pravdivostnej hodnoty).

Pravdivosťná tabuľka vyjadrujúca pravdivosťnú hodnotu logických operácií vyzerá nasledovne:

X	Y	X and Y	X or Y	not (X)
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Tab. 6 Pravdivosťná tabuľka

Niekedy sa používa `i xor` označovaný ako `exclusive or` alebo `non-ekvivalencia`. Výsledkom je `true` vtedy, keď práve jeden z operandov má hodnotu `true` a druhý `false`, teda sú rôzne.

Priorita operátorov

Dôležitou témou, ktorej sme sa doposiaľ úspešne vyhýbali, je priorita operátorov – t.j. ktorú matematickú alebo logickú operáciu uprednostniť pred inou. Štandardné pravidlá pre aritmetické operácie (násobenie má prednosť

pred sčítaním a pod.) poznáme zo základnej školy, priority operátorov programovacieho jazyka nám určuje nasledujúca tabuľka.

V prípade, že vytvoríme zápis bez zátvoriek sa ako prvý bude aplikovať operátor s najvyššou prioritou.

V prípade operátorov rovnakej priority sa vyhodnocovanie výrazu realizuje zľava doprava.

Priorita	Operátor
4	not
3	*,/,div,mod,and
2	+,-,or
1	=,<>,>,<,<=,>=

Tab. 7 Priorita operátorov

Aké budú výsledky a ktoré výrazy sú nesprávne zapísané.

$4*3+5*8-1$, $5>2$ and $4<8$, not $8>5$,
not $(2>1)$, $3>2$ and $2>1$ or $8>9$

Zistite počet výskytov párných číslíc v reťazci a zistite, či sa v ňom nachádza aj nula.

```
var retaz:string;
    i,pocet:integer;
    nula:boolean;

begin
  retaz:=Edit1.Text;
  nula:=false;
  pocet:=0;
  for i:=1 to length(retaz) do begin
    if (retaz[i]='0') or (retaz[i]='2') or (retaz[i]='4') or
      (retaz[i]='6') or (retaz[i]='8') then pocet:=pocet+1;
    if retaz[i]='0' then nula:=true;
  end;
  ShowMessage('Pocet: '+IntToStr(pocet));
  if nula then ShowMessage('Nula tu je')
    else ShowMessage('Nula tu nie je');
end;
```

Premenná `nula` bola na začiatku nastavená na hodnotu `false`, v prípade ak sa 0 vyskytla, zmenila hodnotu na `true`. Podmienka testujúca párnosť čísel zabezpečí zvýšenie premennej `pocet` ak sa vyskytne 0 alebo 2 alebo 4 atď.

Zápis `if nula then` predstavuje štandardný zápis vychádzajúci z nasledovných faktov:

- v prípade, že `nula` obsahuje hodnotu `true`, je podmienka splnená a nepotrebujeme na to zápis `nula=true`, t.j. porovnanie `true=true` s výsledkom `true`,

- pokiaľ je v premennej `nula` hodnota `false`, opäť máme k dispozícii pravdivostnú hodnotu bez potreby porovnania, t.j. `nula=true`, čiže `false=true` s výsledkom `false`.

1. Porovnajzte možnosti poskytované údajovými typmi `integer` a `string` pri riešení problémov.
2. Porovnajzte údajové typy `real` a `integer` z hľadiska rozsahu, presnosti a ordinálnosti.
3. Vysvetlite pojem ordinálny a uveďte príklad ordinálnych a neordinálnych údajových typov.
4. Čo je ASCII tabuľka a aký má význam?
5. Vymenujte a popíšte údajové typy, s ktorými ste sa doposiaľ stretli.
6. Načrtnite a vysvetlite pravdivostnú tabuľku.
7. Napíšte program, ktorý vypíše zo zadaného slova `len` spoluhlásky.
8. Zistite, či sa v zadanom slove nachádza „y“. Ak áno, zistite koľkokrát a vymeňte ho za „i“.

6 Chyby v programe

predpoklady na zvládnutie lekcie:

- znalosť základov práce v prostredí Borland Delphi
- znalosť údajových typov string a integer

obsah lekcie:

- typy chýb a ich odhaľovanie
- debugovanie programu
- zložitosť a efektívnosť

cieľ:

- získať základné informácie o možnosti debugovania programu
- zoznámiť sa so základnými pravidlami pre tvorbu efektívnych programov

Každý program, ktorý napíšeme by mal byť funkčný a mal by vrátiť správne výsledky. Pretože program písal človek (tvor omylný) však tomu tak vždy nie je. Pri písaní nielen prvých programov sa veľmi často vyskytujú chyby, ktoré môžeme rozdeliť do dvoch kategórií: syntaktické a sémantické.

Syntaktické chyby sú tie, ktoré zapríčiňujú slabšiu znalosť programovacieho jazyka alebo nepozornosť pri písaní programu. Vznikajú vtedy, keď nedodržíme pravidlá, ktoré určujú skladbu (syntax) príkazov a celého programu v programovacom jazyku. Medzi najčastejšie patrí napr. vynechanie bodkočiarky, používanie premennej bez toho, aby sme ju vopred deklarovali, viackrát použitý `begin` ako `end` a pod. Sú to chyby, ktoré nájde počítač ešte pred samotným spustením programu a upozorní nás na ne – bez ich odstránenia nie je možné program skompilovať a spustiť.

Horšia situácia nastane v prípade **sémantických chýb**, keď je program syntakticky správne zapísaný v programovacom jazyku, no napriek tomu pomocou neho správny výsledok nezískame. Túto kategóriu chýb možno rozdeliť na chyby počas behu programu a logické chyby.

Chyby vznikajúce počas behu programu sa objavujú vtedy, keď počítač nie je schopný pokračovať v práci, pretože nastal stav, ktorý môže zapríčiniť nesprávny výsledok. Patrí sem napr. delenie nulou, otváranie neexistujúceho súboru, výpočet väčšieho čísla, aké je počítač schopný spracovať, atď.

Napr. výraz `c:=a/b` je správny, avšak ak je pred výpočtom v premennej `b` nula, program sa zastaví a upozorní na to. Podobne pri výpočtoch s celými

číslami: ak sčítame v *Turbo pascal* 32 000 s 50 000, buď program vyhlási chybu, alebo nám ako správny výsledok ponúkne 16 464. Táto chyba súvisí s obmedzením celých čísel v rozsahu od hodnoty -32 768 po 32 767.)

Logické chyby sa hľadajú najťažšie. Nie sú zapríčinené nesprávnym prepisom algoritmu do programovacieho jazyka, ale nesprávnosťou samotného algoritmu. Program zvyčajne beží bezproblémovo, no v niektorých (prípadne všetkých) prípadoch vracia zlé hodnoty. Na odhalenie tohto typu chýb potrebujeme mať určité skúsenosti alebo aspoň dobrého poradcu, pretože inak by ich nájdenie mohlo trvať veľmi dlho. Zvyčajne je potrebné program analyzovať a krokovat' po jednotlivých príkazoch.

Testovanie a ladenie

Činnosť, ktorou zisťujeme správnosť algoritmu (napr. zadávaním rozličných vstupných hodnôt) a jeho správanie sa v hraničných situáciách, hovoríme **testovanie**. Proces, počas ktorého zistené chyby hľadáme a odstraňujeme, sa nazýva **ladenie**.

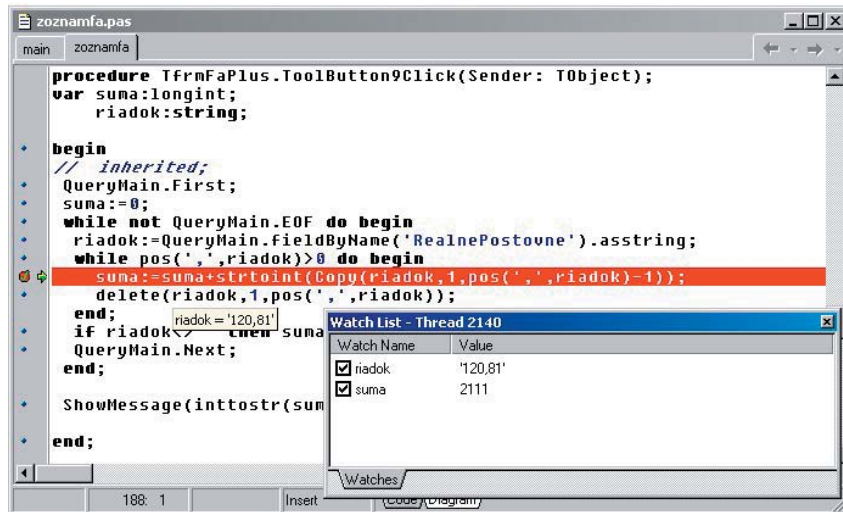
Na to, aby sme mohli program testovať, musíme pre zadané vstupné hodnoty poznať správny výsledok. Pokiaľ ho od počítača nezískame, musíme chybu nájsť. Samotné hľadanie niekedy (často) býva problémom, pretože chyba sa nemusí prejavíť na tom mieste, kde vznikla, ale niekde úplne inde.

Jedným z najpoužívanejších spôsobov hľadania chýb je **krokovanie** (trasovanie, debugovanie) programu. Pri krokovaní nespustíme program tak, aby sa vykonal celý, ale postupujeme po jednotlivých príkazoch. Okrem toho, že vidíme postup a jednotlivé vetvy, cez ktoré sa pri vykonávaní postupuje, môžeme zobrazit' aj hodnoty premenných a tak zistiť, v ktorom príkaze nadohľadnú „nehodný“ obsah.

Krokovanie je možné vykonávať na papieri (a činnosť počítača na základe programu len simulovať) prostredníctvom sledovacích a trasovacích tabuliek alebo priamo v počítači – každé prostredie programovacieho jazyka na to ponúka svoje nástroje.

Borland Delphi disponuje nástrojmi:

- zastavenie programu na ľubovoľnom mieste (*breakpoint* - dosiahnuteľný napr. prostredníctvom stlačenia klávesu `F5` na riadku, kde sa má vykonávať program zastaviť),
- krokovanie programu (`F7` krokuje kód a ak narazí na podprogram, vnorí sa doň a krokuje i ten, `F8` krokuje kód, no nevára sa do volaných podprogramov),
- sledovanie hodnôt premenných nastavením kurzora myši na premennú, ktorej hodnotu chceme zobrazit' alebo pridaním premennej do zoznamu



Obr. 60 Debugovanie

premenných prostredníctvom *Ctrl+F7*.

Pri testovaní je vhodné vyskúšať správanie sa programu po zadaní hraničných hodnôt (napr. 0, veľké čísla, veľmi malé čísla a pod.). Pokiaľ sú niektoré čísla neprípustné, treba zabezpečiť ošetrovanie (napr. pri zadávaní alebo pri výpočte vypísať, že riešenie pre zadanú hodnotu nie je možné). Testovať zložitejší program až po ukončení celej práce môže byť veľmi náročné, preto sa testovanie zvyčajne vykonáva po dokončení každej uzavretejšej časti.

Pozor: ak v programe nenájdeme chybu, neznamená to, že ju neobsahuje!

Zložitosť a efektívnosť

Úlohu na zostrojenie algoritmu riešiaceho daný problém možno riešiť spravidla mnohými spôsobmi. Aby sme si z viacerých správne fungujúcich algoritmov mohli vybrať ten najlepší, potrebujeme mať kritérium, podľa ktorého by sme mohli algoritmy porovnávať. Algoritmy je možné hodnotiť z rôznych hľadísk, napríklad aj podľa toho, aké ľahké je pre nás takýto algoritmus napísať. Kritériá, ktoré sa obvyčajne používajú pri hodnotení kvality algoritmov sú však trochu iného rázu. Najčastejšie sa vyžaduje minimálna časová a pamäťová výpočtová zložitosť algoritmov.

Vezmime si jednoduchý program na súčet prvých n prirodzených čísel...

```

var i,n,sucet:integer;
    s,o:real;

begin
  n:=StrToInt(Edit1.Text);
  sucet:=0;
  for i:=1 to n do sucet:=sucet+i;
  ShowMessage('Sucet cisel je: ' + IntToStr(sucet));
end;

```

...a porovnajme ho s inou verziou...

```

Begin
  n:=StrToInt(Edit1.Text);
  sucet:=n*(n+1) div 2;
  ShowMessage('Sucet cisel je: ' + IntToStr(sucet));
end;

```

V prvom prípade postupujeme síce správne, no čím väčšiu hodnotu n zadáme, tým dlhšie čakáme na výsledok. V druhom prípade sme použili fintu (vzorec na súčet prvých n čísel) a výpočet bude takmer rovnako rýchly pre všetky n .

V prvom prípade je zložitosť závislá od n priamoúmerne, hovoríme o zložitosti $O(n)$, v druhom prípade je konštantná (pre každé n rovnaká), hovoríme o zložitosti $O(1)$. Efektívnejší je jednoznačne druhý algoritmus, ktorý je okrem rýchlosti menej náročný aj pamäťovo, pretože pri ňom používame o jednu premennú menej.

Hľadanie čo najefektívnejších algoritmov patrí v súčasnosti medzi hlavné problémy a činnosti programátorov, ktorí vyvíjajú prekladače z programovacieho jazyka do strojového kódu. Je jasné, že ak by svoju prácu odflákli a vytvorili prekladač, ktorý by vytváral neefektívny preklad, nemohli by ani nami napísané programy preložené do strojového jazyka pracovať rýchlo. Prednosť sa obvyčajne dáva skráteniu času a hľadajú sa čo najrýchlejšie algoritmy i za cenu potreby dodatočnej pamäte.

Kde a ako môžeme efektívne pracovať my? Napísať pravidlá o tom, čo a kedy je efektívne, sa asi nedá. Spomaľovanie majú najčastejšie na svedomí nesprávne navrhnuté algoritmy, ignorovanie matematických vzorcov (často z cyklu urobia sekvenčný výpočet) alebo nerozmyslené implementovanie prvej myšlienky.

Časté je opakovanie rovnakých výpočtov v cykle.

Vypočítajte hodnoty funkcie $y=2*\pi*r$ pre r od 1 do 20 a vypíšte ich.

V prvom programe sa v cykle zbytočne opakovane vykonáva násobenie konštant $2*\pi$, v druhom programe sa vykoná pred cyklom len raz. Ďalšiu úsporu času získame nahradením operácie násobenia v cykle jednoduchším sčítaním.

neefektívne	efektívne
...	...
pi:=3.1415;	pi:=3.1415;
	pom:=2*pi;
	y:=0;
for r:=1 to 20 do begin	for r:=1 to 20 do begin
y:=2*pi*r;	y:=y+pom;
Vypis(r, '- ', y);	Vypis(r, '- ', y);
end;	end;
...	...

1. Uveďte príklady syntaktických chýb a popíšte spôsob ich odstraňovania.
2. Uveďte príklady sémantických chýb a tiež uveďte možnosti ich odstránenia.
3. Vysvetlite pojem debugovanie a popíšte nástroje, ktorými na tento účel disponuje prostredie Borland Delphi.

7 Zoznamy

predpoklady na zvládnutie lekcie:

- znalosť základov práce v prostredí Borland Delphi
- znalosť údajových typov string a integer
- schopnosť riešiť úlohy s cyklami

obsah lekcie:

- viacnásobné vetvenie
- reprezentácia údajových typov v pamäti
- údajový typ pole
- Listbox ako vizuálna reprezentácia poľa v Delphi
- náhodné čísla

cieľ:

- získanie zručností pri práci so zoznamom údajov
- priblíženie sa k „mysleniu programátora“

Viacnásobné vetvenie

Napište algoritmus, ktorý pre zadané číslo vypíše počty výskytov jednotlivých párnych čísel.

Na jednotlivé počty by sme využili premenné s mnemotechnickými názvami (napovedajúcimi význam premennej) napr. nula, dva, styri atď. a časť algoritmu testujúca i -tu cifru čísla by potom vyzerala ako postupnosť :

```
if retazec[i]='0' then nula:=nula+1;
if retazec[i]='2' then dva:=dva+1;
if retazec[i]='4' then styri:=styri+1;
if retazec[i]='6' then sest:=sest+1;
if retazec[i]='8' then osem:=osem+1;
```

Pokiaľ by sme chceli algoritmus sprehľadniť, môžeme použiť príkaz zabezpečujúci viacnásobné vetvenie. Má tvar:

```

case premenna of
  h1 : prikaz1; {pripadne postupnost prikazov medzi begin a end}
  h2 : prikaz2; {pripadne postupnost prikazov medzi begin a end}
  ...
  hn : prikazn {pripadne postupnost prikazov medzi begin a end}
else prikazx {pripadne postupnost prikazov medzi begin a end}
end;

```

Príkaz `case` nám dovoľuje reagovať na rôzne hodnoty premennej ($h_1 \dots h_n$) rôznym spôsobom a ošetriť situáciu i pre hodnoty, ktoré nie sú v zozname vymenované (prostredníctvom vetvy `else`).

Kompletné riešenie nášho príkladu by potom mohlo vyzerat' nasledovne:

```

var retazec:string;
    i,nula,dva,styri,sest,osem:integer;

begin
  retazec:=Edit1.Text;
  nula:=0; dva:=0; styri:=0; sest:=0; osem:=0;
  for i:=1 to length(retazec) do {prejdeme po celom retazci}
    case retazec[i] of {ak je na i-tej pozicii zadany znak}
      '0' : nula:=nula+1; {zvysi sa prislusna premenna}
      '2' : dva:=dva+1;
      '4' : styri:=styri+1;
      '6' : sest:=sest+1;
      '8' : osem:=osem+1;
    end; {pre case}
  ShowMessage('vyskyty: 0-` + IntToStr(nula)+chr(13)+
    '2-` + IntToStr(dva)+chr(13)+
    '4-` + IntToStr(styri)+chr(13)+
    '6-` + IntToStr(sest)+chr(13)+
    '8-` + IntToStr(osem));

end;

```

V štruktúre `case` sme *i*-ty znak reťazca porovnávali s párnymi číslicami rovnako ako v prvom riešení s vnorenými binárnymi vetveniami a zvyšovali príslušnú premennú. Hodnoty sú v apostrofoch, pretože skúmaná premenná `retazec[i]` je typu `char`.

Napište program, ktorý pre zadaný mesiac vypíše, koľko má dní (nepredpokladajte priestupný rok).

```

var mesiac,pocet:integer;

begin
  mesiac:=StrToInt(Edit1.Text);
  pocet:=0;

```

```

case mesiac of
  1:pocet:=31;
  2:pocet:=28;
  3:pocet:=31;
  4:pocet:=30;
  5:pocet:=31;
  6:pocet:=30;
  7:pocet:=31;
  8:pocet:=31;
  9:pocet:=30;
  10:pocet:=31;
  11:pocet:=30;
  12:pocet:=31
  else ShowMessage('Zly mesiac');
end;
if pocet>0 then
  ShowMessage('pocet dni: '+IntToStr(pocet));
end;

```

V štruktúre `case` máme vymenované všetky mesiace a priradený k nim počet dní. V prípade, že na vstupe je iná ako akceptovateľná hodnota, vypíše sa upozornenie a v premennej `pocet` zostane priradená hodnota 0. Tento fakt nám slúži aj pri kontrole výpisu – počet dní sa vypíše len v prípade, ak je premenná nenulová, t.j. bola zmenená v niektorej vetve `case`.

Premenná `mesiac` je celočíselná a teda aj hodnoty vo vetvách `case` sú zapísané ako celé čísla – bez apostrofov.

Náš príklad dokážeme vyriešiť aj stručnejším zápisom:

V každej vetve môžeme vymenovať i viac hodnôt, pre ktoré sa má vykonať ten istý príkaz.

```

var mesiac,pocet:integer;

begin
  mesiac:=StrToInt(Edit1.Text);
  pocet:=0;
  case mesiac of
    1,3,5,7,8,10,12 :pocet:=31;
    2                :pocet:=28;
    4,6,9,11        :pocet:=30;
  end;
  if pocet=0 then Showmessage('zly mesiac')
  else ShowMessage('pocet dni: '+IntToStr(pocet));
end;

```

Premenná, ktorá sa používa na porovnávanie hodnôt v štruktúre `case` musí byť ordinálna – nie je teda možné používať hodnoty typu `real` alebo `string` (v príklade s ciframi bol porovnávaný len jeden znak z premennej typu `string` – teda `char`). V prípade potreby porovnávania hodnôt neordinálneho typu budeme musieť vystačiť len s klasickým príkazom vetvenia (`if-then`).

Napriek (alebo vďaka) tomu nám `case` ponúka ešte jeden silný nástroj – používanie **intervalu**. V prípade ordinálnych typov totiž vieme vymenovať všetky hodnoty vyskytujúce sa medzi začiatkom a koncom intervalu, napr. medzi 5 a 8, čo v pascale zapíšeme ako 5..8, budú hodnoty 5,6,7,8. Využitie ilustruje nasledovný príklad.

Zistite koľkokrát sa v zadanom reťazci nachádzajú malé, koľkokrát veľké písmená a koľko obsahuje čísiel.

```
var i,male,velke,cislice:integer;
    retazec:string;

begin
  retazec:=Edit1.Text;
  male:=0; velke:=0; cislice:=0;
  for i:=1 to length(retazec) do begin
    case retazec[i] of
      'a'..'z':inc(male);{uvazuje male pismena medzi a a z}
      'A'..'Z':inc(velke);{uvazuje velke pismena medzi A a Z}
      '0'..'9':inc(cislice);{uvazuje cislice medzi 0 a 9,
        nie ako cele cisla, ale znaky z ASCII tabulky}
    end;
  end;
  ShowMessage('male: '+IntToStr(male));
  ShowMessage('velke: '+IntToStr(velke));
  ShowMessage('cislice: '+IntToStr(cislice));
end;
```

1. Napište program, ktorý na základe zadania matematickej operácie (+,-,*,/) zistí súčet, rozdiel, súčin a podiel medzi dvoma zadanými číslami.
2. Pre mesiac zadaný slovnou výpoveďou vypíšte počet dní v ňom.
3. Napište program, ktorý načíta poradové číslo mesiaca. Ak je to číslo z intervalu <3..5>, vypíšte 'JAR', ak je číslo z intervalu <6..8>, vypíšte 'LETO', ak je číslo z intervalu <9..11>, vypíšte 'JESEN', ak je to číslo 12, 1 alebo 2, vypíšte 'ZIMA'. Ak je číslo menšie ako 1, alebo väčšie ako 12, vypíšte 'ZLY MESIAC'.

4. Napište program pre colníkov, ktorí budú pri prechode cez hranice zisťovať, či turistu môžu pustiť bez kontroly. Rozhodujúcim údajom bude národnosť. Pokiaľ turista pochádza zo štátov EÚ môžu ho pustiť, pokiaľ nie, treba ho prevetrať. Ak je Američan, Austráľčan alebo Japonec, stačí ho skontrolovať len mierne. (Prečo nemožno použiť štruktúru `case`?)

Reprezentácia údajových typov

Údajové typy, ktoré sme doposiaľ používali (okrem typu `string`) patria medzi jednoduché – ich hodnota je zakódovaná prostredníctvom jedného alebo niekoľko málo bajtov, ku ktorým pristupujeme ako k celku:

- **integer** bol zrejme našim najpoužívanejším typom. V *Turbo pascale* je jeho hodnota zakódovaná do 2 bajtov, v prípade *Delphi* sa využívajú už 4 bajty. Už vieme, že do jedného bajtu možno zakódovať 256 rôznych hodnôt, do dvoch bajtov $256 \cdot 256 = 65536$, do štyroch bajtov 256^4 – viac ako 4 mld. rôznych hodnôt.

V prípade celých čísel je jeden bit vyhradený pre znamienko, z čoho vyplýva, že pri 2 bajtovom type máme k dispozícii hodnoty v rozsahu -32 768..32 767 a pri 4 bajtovom je to -2 147 483 648..2 147 483 647.

Pokiaľ tento rozsah (napr. pri výpočte) prekročíme, môže sa podľa nastavení kompilátora vyvolať chyba počas behu programu alebo sa bez výstrahy pokračuje vo vykonávaní programu – samozrejme s tým, že výsledok už správny nebude (napr. $32\,000 + 1\,000$ je $-32\,536$).

Okrem typu `integer` existujú aj ďalšie celočíselné typy napr. `byte`, ktorý nepoužíva znamienka a umožňuje uchovávať hodnoty 0..255 alebo `word` s hodnotami 0..65 535, či `longword` s intervalom 0..4 294 967 295.

- **char** je reprezentovaný jedným bajtom vďaka čomu dokáže zakódovať 256 hodnôt. Transformácia číselných hodnôt na znaky sa realizuje prostredníctvom ASCII tabuľky, kde každej numerickej hodnote zodpovedá jeden znak,
- **boolean** potrebuje pre zakódovanie svojich hodnôt (`true`, `false`) len 1 bit, no obvyčajne v prostredí *Windows* využíva celý bajt,
- **real** je neordinálnym typom, ktorý vnútorne pozostáva z dvoch častí: **mantisy** a **exponenta**. Napr. pre číslo $1,4587E20$, ktoré predstavuje hodnotu $1,4587 \cdot 10^{20}$ je mantisou $1,4587$ a exponentom 20. Platí, že čím presnejšiu hodnotu požadujeme, tým viac bitov má mať k dispozícii mantisa a čím väčšie (menšie) číslo chceme uchovávať, tým viac bitov má byť vyhradených pre exponent. Typ `real` má v prípade *Delphi* k dispozícii 8 bajtov, z toho pre mantisu je určených 15-16 bitov (z toho plynúci rozsah je $5.0 \cdot 10^{-324} \dots 1.7 \cdot 10^{308}$).

Štruktúrovaný typ pole

Doposiaľ nám na riešenie úloh postačovalo, ak sme od používateľa získali niekoľko málo hodnôt a s nimi sme pracovali. Prax je však omnoho komplikovanejšia a s tak jednoduchými úlohami sa stretávame veľmi zriedka. Oveľa častejšie sa vyžaduje spracovanie väčšieho počtu údajov rovnakého typu. Ako údajová štruktúra sa v takomto prípade používa typ pole – array.

Možno si ho predstaviť ako zoznam, ktorý obsahuje hodnoty rovnakého typu. Tieto sú jednoznačne určené indexom, z ktorého sa dá určiť ich umiestnenie v zozname. S poľom sme sa stretli už v prípade údajového typu `string`, ktorý vlastne predstavuje pole znakov (pole údajov typu `char`).

Pred použitím poľa treba najprv kompilátoru túto požiadavku oznámiť. Môžeme to urobiť dvoma spôsobmi – pri deklarácii premennej alebo definíciou nového typu.

V prvom prípade uvedieme údaje o poli v deklaračnej časti za menom premennej a dvojčipkou. Kľúčové slovo `array` hovorí o tom, že premenná bude typu pole, nasleduje typu indexu (v hranatých zátvorkách) a napokon typ údajov uložených v poli uvedený za kľúčovým slovíčkom `of`. Ako príklady môžu slúžiť nasledovné deklarácie:

```
var pole1:array[1..10] of char;
var pole2:array[1..20] of integer;
var pole3:array[0..10] of real;
var pole4:array[-10..15] of string;
var pole5:array['a'..'z'] of real;
```

Podmienkou pre index poľa je použitie ordinálneho typu, na základe ktorého je možné určiť poradie a počet prvkov v poli, ako aj veľkosť potrebnej pamäte už pri kompilácii programu.

Každý prvok poľa predstavuje samostatnú premennú, do ktorej je možné priradovať alebo z nej čítať hodnoty príslušného typu. Prístup je zabezpečený prostredníctvom indexu, rovnako ako tomu bolo v prípade premenných typu `string`, napr.:

```
pole1[3]:=10;      alebo   pole3[0]:=10.5+pom;
                   alebo   pole5['c']:=pole5['d']-3;
```

V prípade, že rovnaký typ poľa budeme používať v programe častejšie, je vhodné definovať preň typ:

```
type TPole=array[1..10] of char;
```

a potom **deklarovať** premenné typu `TPole`:

```
var pole1,pole2:TPole;
```

Prvé písmeno „T“ je zaužívaným štandardom pre vyjadrenie toho, že `TPole` je typ (z anglického `Type`). Nie je nutné ho používať, ale dodržiavanie štandardov zlepšuje čitateľnosť programov.

Pole môžeme zadávať viacerými spôsobmi – v tomto prípade sme zvolili priradenie hodnôt prvkom poľa v zdrojovom kóde. Výhodou je okrem iného najmä fakt, že kým program testujeme, nemusíme pri každom spustení zadávať vždy znova a znova tie isté hodnoty.

Napište program, ktorý pre zadané pole čísel nájde maximum.

```
var pole:array[1..10] of integer;
    i,max:integer;

begin
  {vloženie prvkov do pola}
  pole[1]:=10; pole[2]:=5; pole[3]:=15; pole[4]:=17;
  pole[5]:=12; pole[6]:=4; pole[7]:=1; pole[8]:=40;
  pole[9]:=11; pole[10]:=99;
  max:=pole[1]; {priradi do max hodnotu prveho prvku pola}
                {pri prezerani pola je po prvok kroku urcite}
                {to najvacsie, ktore sme zatiaľ nasli}
  for i:=2 to 10 do {staci teraz skumat od druheho prvku}
    if pole[i]>max then max:=pole[i];
  ShowMessage('Maximum je ' + IntToStr(max));
end;
```

Pri hľadaní maxima sme využili fakt, že každý prvok v poli bude určite väčší (prinajhoršom rovný) ako prvý prvok a tak sa prvotne nastavená hodnota priradená do premennej `max` v prípade potreby zmení.

Priradenie hodnôt do poľa možno zapísať i priamo v deklarácii poľa nasledovne:

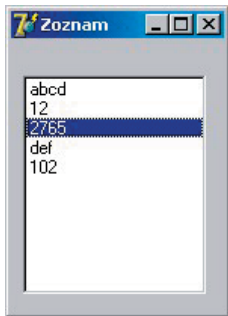
```
var pole:array[1..10] of integer = (10,5,15,17,12,4,1,40,
11,99)
```

Ostatným premenným možno priradiť počiatočnú hodnotu podobným spôsobom:

```
var a:integer = 20;
```


Listbox

Pre prácu v udalost'ami riadenom programovaní platí iná filozofia ako v prípade štruktúrovaného jazyka. S určitými úpravami by sme síce mohli vkladať do poľa údaje prostredníctvom komponentu `Edit` prvky do poľa klikaním na tlačidlo, no tento postup určite pretromfne možnosť vidieť naraz všetky vkladané prvky, mazať a upravovať ich predtým ako sa odštartuje samotné spracovanie.



Obr. 61 Listbox a jeho položky (Items)

Na prácu so zoznamom je jedným z najvhodnejších komponentov komponent `Listbox`, ktorý predstavuje vizuálnu reprezentáciu poľa obsahujúceho hodnoty typu `string`.

Do tohto zoznamu môžeme pridávať položky typu `string` buď priamo v návrhovom prostredí *Delphi* (uložiť pod seba položky cez vlastnosť `Items`), alebo prostredníctvom kódu.

Vytvorte aplikáciu, ktorá dokáže prostredníctvom tlačidla pridávať do listboxu text, ktorý zadáte do Editu a mazať položku, na ktorej ste v Listboxe nastavení.

V prvom rade vytvoríme rozhranie, prostredníctvom ktorého sa budú do zoznamu vkladať hodnoty.

Po kliknutí na tlačidlo sa na koniec `Listboxu` vloží hodnota umiestnená v `Edit` prostredníctvom kódu, ktorý hovorí o tom, že pre `Listbox1` je potrebné k prvkom zoznamu (`Items`) pridať (`Add`) hodnotu uvedenú v zátvorke. Hodnota je textová (typ `string`), v prípade vloženia konkrétneho textu musí byť v apostrofoch.



Obr. 62 Rozhranie s Editom a Buttonom

```
Listbox1.Items.Add(Edit1.Text);
```

Ak sa chceme pustiť do manipulácie s prvkami `Listboxu`, mali by sme poznať základné operácie na prístup k prvkom:

operácia	zápis	poznámka
pridať položku	<code>Listbox1.Items.Add(text)</code>	text je typu <code>string</code>
počet položiek v Listboxe	<code>pocet:=Listbox1.Items.Count</code>	pocet je typu <code>integer</code>
získať text i-tej položky	<code>polozka:=Listbox1.Items.Strings[i];</code> alebo jednoduchšie <code>polozka:=Listbox1.Items[i];</code>	polozka je typu <code>string</code>
odstrániť i-tu položku	<code>Listbox1.Items.Delete(i);</code>	
vymazať všetky položky z Listboxu	<code>Listbox1.Items.Clear</code> alebo <code>Listbox1.Clear</code>	
zistiť na ktorej položke Listboxu je nastavený kurzor	<code>cislo:=Listbox1.ItemIndex;</code>	číslo je typu <code>integer</code> pri výbere prvej položky je hodnota 0, druhej 1 atď. Pokiaľ nie je vybraná žiadna položka, je vrátená hodnota (-1)
vypísať text na vybranej položke	<code>mytext:=Listbox1.Items[Listbox1.ItemIndex];</code> jednoduchšie <code>mytext:=Listbox1.SelectedText;</code>	ide o kombináciu dvoch predchádzajúcich: <code>ItemIndex</code> vráti poradové číslo položky, ktorá je aktívna a <code>Items[Listbox1.ItemIndex]</code> vráti text, ktorý je v aktívnej položke uložený
zistiť, či sa daná položka v Listboxe už nachádza a ak áno, vrátiť jej poradové číslo	<code>porcislo:=Listbox1.Items.IndexOf(text);</code>	<code>porcislo</code> obsahuje poradové číslo položky, ktorá obsahuje zadaný text. V prípade, že sa takáto položka v listboxe nenachádza, <code>IndexOf</code> vráti -1

Tab. 8 Podpora práce s Listboxom

Pri `Listboxe` sa stretne s vlastnosťou typickou aj pre ďalšie komponenty - index prvého prvku je 0, posledného `Listbox1.Items.Count-1` - počítanie nezačína od 1, ale od 0.

Pri mazaní položky z `Listboxu` by sme mali najprv zistiť, či je vôbec nejaká aktívna, a ak áno zistiť jej pozíciu a vymazať ju. Postupnosť budeme realizovať pri kliknutí na druhé tlačidlo.

```

procedure TForm1.Button2Click(Sender:TObject);
var index:integer;

begin
  {ak je nejaka položka aktivna}
  if ListBox1.ItemIndex>-1 then begin
    index:=ListBox1.ItemIndex;      {zistim ktora}
    ListBox1.Items.Delete(index);   {a odstranim ju}
  end;
end;

```

Napište program, ktorý umožní do Listboxu vkladať len čísla a potom prostredníctvom osobitných tlačidiel nájdite minimum a maximum zo zoznamu.

Najprv vytvoríme rozhranie známe už z predchádzajúcej úlohy, pri ktorom však doplníme kontrolu, či vkladaná hodnota je skutočne číslo. Využijeme na to známu procedúru Val.

```

procedure TForm1.Button1Click(Sender:TObject);
var cislo,kod:integer;

begin
  val(Edit1.Text,cislo,kod); // prevadza string na cislo,
  if kod=0 then begin      // ak k chybe nedoslo kod je 0,
                          // inak <>0
    ListBox1.Items.Add(Edit1.Text);
    Edit1.Text:=''; // patri sa vyprazdnit pre nove cislo
  end else ShowMessage('Zadajte radšej číslo');
end;

```

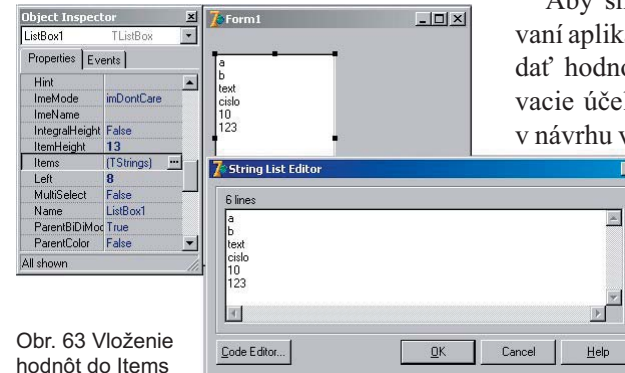
Pre tlačidlo, ktoré spúšťa hľadanie minima, môže kód vyzerat' nasledovne (a pre maximum to bude analógia):

```

procedure TForm1.Button2Click(Sender:TObject);
var i,min:integer;

begin
  {za minimum oznacime prvý prvok zoznamu s indexom 0}
  min:=StrToInt(Listbox1.Items[0]);
  {a porovnavame s dalsimi pricom nezabudame}
  {ze posledny prvok ma index celkovy pocet minus 1}
  for i:=1 to ListBox1.Items.Count-1 do
    if min>StrToInt(Listbox1.Items[i]) then
      min:=StrToInt(Listbox1.Items[i]);
  ShowMessage('Minimum je: '+IntToStr(min));
end;

```



Obr. 63 Vloženie hodnôt do Items

Aby sme nemuseli pri testovaní aplikácie vždy nanovo vkladať hodnoty, je dobré na testovacie účely nastaviť ich priamo v návrhu vo vlastnosti Items pre zoznam položiek Listboxu. Po doladení aplikácie sa však patrí tieto hodnoty odstrániť.

Úprava prvku

V profesionálnych aplikáciách sa úpravy hodnôt zvyčajne realizujú dvojklikom na prvok, ktorý chceme meniť. V našom prípade bude najefektnejšie zabezpečiť prenos prvku z Listboxu do Editu (priamo v Listboxe totiž nedokážeme robiť úpravy) s tým, že sa súčasne z Listboxu odstráni.

Keďže operácia bude realizovaná na dvojklik, použijeme udalosť OnDblClick (záložka Events v Object Inspectore). Reakcia na ňu bude pozostávať z prenosu aktuálneho prvku do Editu a z jeho vymazania v Listboxe.

```

procedure TForm1.ListBox1DblClick(Sender: TObject);

begin
  {ak su v listboxe nejake prvky}
  if ListBox1.Items.Count>0 then begin
    Edit1.Text:=ListBox1.Items[ListBox1.ItemIndex];{prenos}
    ListBox1.Items.Delete(ListBox1.ItemIndex); {vymazanie}
  end;
end;

```

Hľadanie v Listboxe

V rozsiahlejších zoznamoch je niekedy vhodné zisťovať, či sa v nich už zadávaný prvok nenachádza, prípadne ak áno, tak na ktorej pozícii. Služí na to funkcia ListBox1.Items.IndexOf(mojText), ktorá vráti index prvého z riadkov, na ktorom sa nachádza hľadaná hodnota mojText. V prípade nenájdenia vracia -1.

Napište program, ktorý bude v Listboxe evidovať zoznam návštevníkov kina podľa čísla ich vstupenky. Ak návštevník vojde do kinosály, pridá sa jeho číslo do zoznamu, ak ju z nejakého dôvodu opustí, zo zoznamu sa vyhodí. Pri vstupe kontrolujte, či sa vstupenka so zadaným číslom v zozname už nenachádza a ak áno, upozorníte na falšovateľa.

Štandardné operácie už nebudeme opakovať, zameriame sa na zistenie existencie vkladanej hodnoty.

```
var index:integer;

begin
  index:=ListBox1.Items.
  IndexOf(Edit1.text);
  if index>-1 then
    ShowMessage('číslo '+Edit1.Text+' je na r.'+IntToStr(index))
  else
    ListBox1.Items.Add(Edit1.Text);
end;
```



Obr. 64 Možný vzhľad formulára

Listbox a pole

Ak máme porovnať pole a Listbox, tak na základe doteraz známych skutočností Listbox poskytuje viac možností na pohodlnú prácu s prvkami zoznamu – je vizuálny, prvky zoznamu dokážeme odstraňovať, pridávať relatívne neobmedzene, vyhľadávať.

Nevýhodou Listboxu je, že dokáže uchovávať len údaje typu string a pokiaľ ich potrebujeme spracúvať iným spôsobom (napr. ako čísla), musíme ich neustále konvertovať.

Naproti tomu pole môže obsahovať údaje ľubovoľného typu, a to nielen jednoduchého, teda napríklad aj pole alebo inú údajovú štruktúru.

Napište program, ktorý do poľa celých čísel preniesie číselné údaje z Listboxu a v opačnom poradí ich umiestni do druhého Listboxu.

Konštanty

Napište program, ktorý bude načítavať prvky poľa dovtedy, kým sa nezadá hodnota 0. Následne vypíše najprv kladné a potom záporné hodnoty.

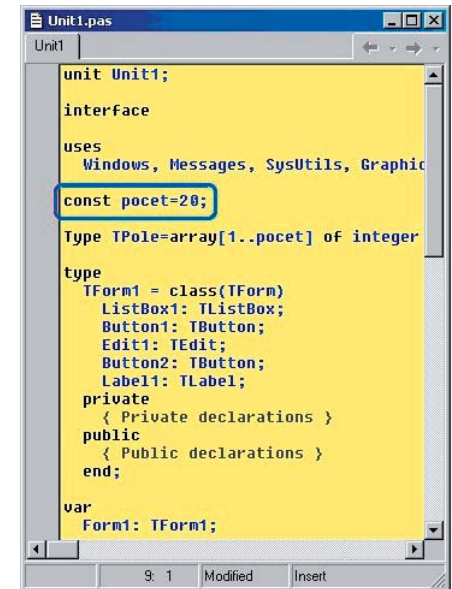
Doposiaľ sme sa diskretné vyhýbali možnosti, že počas behu programu prekročíme počet prvkov v poli, pre ktoré sme vyhradili pamäť v deklaračnej (resp. v definičnej) časti. Správanie sa programu závisí od nastavení kompilátora, ktorý nás môže na chybu upozorniť, prípadne neupozorní a môže pokračovať s rizikom, že výsledky už nebudú korektné.

Zmeniť veľkosť poľa počas behu programu nedokážeme, pretože potrebná veľkosť pamäte sa určí pri kompilácii. Ako najvhodnejší spôsob zadania veľkosti poľa sa v programátorskej praxi javí odhadnúť potrebu používateľa a v prípade potreby dokázať čo najjednoduchším a najrýchlejším spôsobom zmeniť v zdrojovom kóde počet vyhradených prvkov – samozrejme prepísaním parametrov poľa.

Na tento účel sa najčastejšie používajú konštanty. **Konštantu** je pomenovaná hodnota, ktorá sa počas behu programu nemení a pre nás predstavuje spôsob, ktorým môžeme prepísaním jedinej hodnoty rýchlo a jednoducho zmeniť hodnoty na viacerých miestach programu.

Predstavte si, že vytvárate program, v ktorom používate niekoľko polí s rovnakou veľkosťou a od používateľa príde požiadavka túto veľkosť zmeniť. Pokiaľ sú rozmery poľa zadané číselne, musíte túto hodnotu prepísať na všetkých miestach, no pokiaľ použijete symbolickú konštantu, zmeníte jej hodnotu na jednom mieste a táto sa pri kompilácii aplikuje na všetkých miestach výskytu príslušnej konštanty. Konštanty definujeme prostredníctvom kľúčového slova const.

V Delphi sa konštanty uvádzajú tiež na začiatku kódu, musia byť však umiestnené za kľúčovým slovom interface, napr. tak ako na obrázku.



Obr. 65 Definícia konštanty

Ako prostriedok na načítavanie i výpis údajov použijeme Listbox, pričom údaje budeme čítať dovtedy, kým v čítanom riadku nebude hodnota 0:

```
const veľkostPola=20;
var pole:array[1.. veľkostPola] of integer;
    i,pocet,hodnota:integer;
```

```

begin
pocet:=0; {sluzi na pamatanie si poctu nacitanych prvkov}
repeat
  {precitaj hodnotu, zacina od riadku 0}
  hodnota:=ListBox1.Items[pocet];
  if hodnota<>0 then begin {ak je nenulova}
    inc(pocet);{zvys pocet a prirad do pola}
    pole[pocet]:=hodnota;
  end;
until hodnota=0; {opakuj pokiaľ na vstupe nebude 0}

ListBox1.Clear; {vymaze obsah Listboxu}
for i:=1 to pocet do
  if pole[i]>0 then ListBox1.Item.Add(IntToStr(pole[i]));
for i:=1 to pocet do
  if pole[i]<0 then ListBox1.Item.Add(IntToStr(pole[i]));
end;

```

V premennej `pocet` evidujeme počet prečítaných prvkov a na základe nej vieme, do ktorého prvku v poli máme priradiť novonacítanú hodnotu. Premennú využijeme i pri výpise, keď je v nej uložený počet všetkých prečítaných hodnôt.

Využitie poľa

Napište program, ktorý pre zadané číslo zistí počet výskytov jednotlivých cifier. Výsledok vypíše do Listboxu.

Na zapamätanie výskytu cifier využijeme pole `p`, pričom jeho indexy môžeme považovať za cifry, t.j. v `p[1]` bude počet jednotiek, v `p[7]` počet sedmičiek a samozrejme nesmieme zabudnúť ani na počet výskytov núl. Pole, ktoré použijeme teda bude mať rozsah 0..9 (ako cifry).

```

var i,cifra:integer;
    p:array[0..9] of integer;
    cislo:string; {aby sme mohli zadavat dlhsie cisla,}
                {vyuzijeme na ich vstup premennu typu
                string}

begin
  cislo:=Edit1.Text;
  {vynulujeme pocty vyskytov jednotlivych cifier}
  for cifra:=0 to 9 do p[cifra]:=0;

```

```

for i:=1 to length(retazec) do begin
  {zistime, aka cifra je na aktualnej pozicii}
  cifra:=StrToInt(cislo[i]);
  p[cifra]:=p[cifra]+1; {zvysime pocet jej vyskytov v poli}
end;

ListBox1.Clear;
for cifra:=0 to 9 do
  ListBox1.Items.Add(IntToStr(cifra)+' - '+ {a vypis}
                    IntToStr(p[cifra]));

end;

```

Napište program, ktorý pre zadaný text zistí počet výskytov jednotlivých znakov (a-z) a vypíše ich. Nulové výskyty vynechajte.

Úloha je len miernou obmenou predchádzajúcej. Vzhľadom na to, že znaky a-z sú súčasťou ordinálneho typu `char`, môžeme ich použiť ako indexy poľa, čím sa náš problém značne zjednoduší (a skúste problém vyriešiť len pomocou Listboxu...).

```

var p:array['a'..'z'] of integer; {pole pre pocty znakov}
    retazec:string;
    i,znak:char;
    j:integer;

begin
  retazec:=Edit1.Text;
  {vynulujeme pocty vyskytov}
  for i:='a' to 'z' do pocet[i]:=0;
  for j:=1 to length(retazec) do begin
    {zistim aky znak je na aktualnej pozicii}
    znak:=retazec[j];
    {zvysim pocet jeho vyskytov v poli}
    p[znak]:=p[znak]+1;
  end;
  for i:='a' to 'z' do if p[i]>0 then
    ListBox1.Items.Add(IntToStr(i)+' - '+IntToStr(p[i]));
end;

```

Telo cyklu určeného na počítanie znakov by sa mohlo nahradiť i zápisom `inc(p[retazec[j]])`, kde `retazec[j]` reprezentuje písmeno na `j`-tej pozícii, t.j. vráti nám znak, ktorého počet výskytov máme zvýšiť, hodnota `p[...]` predstavuje doterajší počet výskytov a `inc` zabezpečí samotné zvýšenie hodnoty.

Náhodné čísla

Ladenie aplikácií, v ktorých vystupuje pole, môže byť v niektorých prípadoch náročné na čas najmä z dôvodu neustálej potreby zadávania vstupných hodnôt. O možnosti priradenia hodnôt priamo v zdrojovom kóde už vieme, programátori v *Delphi* dokážu vložiť potrebné hodnoty do `Listboxu` v návrhu aplikácie, avšak všetky tieto hodnoty sú nemenné. Na zjednodušenie a pritom určité „znáhodnenie“ vkladania hodnôt využívame **generátor náhodných čísel**.

Vygenerovanie náhodného čísla nám dokáže zabezpečiť funkcia `random`, ktorá v tvare `random(n)` vráti celé číslo z intervalu $\langle 0, n-1 \rangle$ a v tvare `random` bez parametra generuje reálne číslo z intervalu $\langle 0, 1 \rangle$. Ak chceme generovať čísla z iného intervalu, musíme výraz na generovanie náhodného čísla mierne upraviť.

Pri generovaní náhodného celého čísla parameter funkcie `random` vyjadruje počet hodnôt, z ktorých sa vyberá (0, 1, 2, ..., n-1).

Ak chceme napr. generovať číslo z intervalu 5 až 11, náhodne vyberáme jednu zo 7 hodnôt (5, 6, 7, 8, 9, 10, 11). Operáciou `random(7)` dostaneme číslo z intervalu 0 až 6 a pričítaním čísla 5 určite dostaneme číslo najmenej 5 a najviac 11. Náhodné celé číslo z intervalu 5..11 teda vyjadruje výraz `5+random(7)`.

Pri generovaní náhodného reálneho čísla použijeme funkciu `random` bez parametra, ktorá náhodne vyberá číslo z intervalu $\langle 0, 1 \rangle$. Ak chceme napr. generovať reálne číslo z intervalu $\langle 5, 11 \rangle$, interval musí mať šírku 7. Preto najprv vynásobíme vygenerované náhodné číslo siedmimi, jeho hodnota bude najmenej 0 a nepresiahne 7. Pričítaním čísla 5 sa jeho hodnota zväčší na najmenej 5 a nepresiahne 11. Náhodné reálne číslo z intervalu $\langle 5, 11 \rangle$ teda vyjadruje výraz `5+7*random`.

Ak dlhšie pracujete s generátorom, po čase si určite všimnete, že po spustení aplikácie sa generujú vždy tie isté hodnoty. Dôvodom je spôsob generovania, ktorý používa na získavanie čísel algoritmus, v ktorom je nasledujúca hodnota závislá od predchádzajúcej a tá je po spustení aplikácie vždy rovnaká.

Riešením je použitie príkazu `randomize`, ktorý nastaví generátor na náhodnú východziu hodnotu. Postačí ho použiť v aplikácii raz (napr. pri jej štartovaní, resp. udalosti `OnCreate`, či `OnShow` formulára).

Napište program, ktorý vygeneruje a do poľa uloží náhodné čísla.

```
var i:integer;
    pole:array[1..20] of integer;

begin
  for i:=1 to 20 do
    pole[i]:=-50+random(101); {generuje hodnoty od -50 do 50}
  end;
```

1. *Napište program, ktorý bude s používateľom hrať hru na hádanie čísel. Program vygeneruje náhodné číslo z nastaveného intervalu (napr. 0-100) a používateľ ho háda. Pri hádaní mu program vypisuje, či hľadané číslo je väčšie alebo menšie ako to, čo zadal.*
2. *Program upravte tak, aby používateľ disponoval na začiatku určitou sumou, z ktorej bude môcť časť (alebo celú) vsadiť. V každom kroku hádania sa z nej určitá časť odpočíta, v prípade uhádnutia sa zvyšná suma zdvojnásobí.*
3. *Napište program, ktorý bude simulovať hádzanie tromi kockami. Používateľ bude disponovať určitou sumou, z ktorej časť alebo celú môže vsadiť na ľubovoľné číslo. V prípade, že toto číslo bude zodpovedať súčtu náhodne vygenerovaných hodnôt na kockách, používateľ získava dvojnásobok vlozenej sumy, v opačnom prípade o ňu príde.*
4. *Ošetrte program tak, aby v prípade prehrania celej sumy automaticky skončil.*

Lekcia 8

8 Súborny

predpoklady na zvládnutie lekcie:

- znalosť základov práce v prostredí Borland Delphi
- znalosť údajových typov string a integer
- schopnosť práce s údajovým typom pole

obsah lekcie:

- ukladanie údajov do súboru
- typové súborny
- textový súbor
- dialógy systému Windows na podporu komunikácie so súborovým systémom

cieľ:

- získanie zručností na ukladanie a čítanie údajov zo súboru
- schopnosť využívania komponentov typu „Dialog“

Uloženie údajov

Údaje, s ktorými pracujeme, sú často jedinečné a výsledky, ktoré pri práci s nimi získavame, v praxi obyčajne odkladáme na ďalšie použitie. Bolo by nemysliteľné, aby sme zoznam adres, textovú žiadosť alebo fotografiu vkladali do programu vždy, keď s ňou chceme pracovať. Na ukladanie údajov mimo operačnej pamäte sú určené **súborny**. Pracovať s nimi na úrovni operačného systému, resp. na úrovni hotových aplikácií už dokážeme, vytvárať a manipulovať s uloženými údajmi prostredníctvom programu sa pokúsime na nasledujúcich riadkoch.

Napište program, ktorý od používateľa načíta do pola zoznam čísel a uloží ich do súboru 'mojedata'.

Na to, aby sme údaje mohli ukladať, potrebujeme určiť **typ súboru**, do ktorého budeme zapisovať. V pascali sa typ súboru logicky odvíja od typu údajov, ktoré doň ukladáme. Podľa potreby možno pre súbor typ definovať alebo ho postačí len deklarovať:

```
type TSubor=file of integer;    {subor celych cisel}
...
var subor:file of integer;
```

Napišme teda samotný program (vysvetlenie jednotlivých krokov je k dispozícii v komentároch):

```
const pocet=10;
var pole:array[1..pocet] of integer;
    subor:file of integer;
    i:integer;

begin
    {pre zjednodusenie naplnime pole nahodnymi hodnotami}
    for i:=1 to pocet do pole[i]:=random(1000);
    {premenna subor sa inicializuje (nastavi) tak,}
    {aby ukazovala na udaje v subore mojeData}
    AssignFile(subor, 'mojeData');
    {vytvori subor (ak existuje, tak vymaze jeho obsah) a
    pripraví ho na zapis, pokiaľ nie je zadana cesta subor
    sa vytvori v aktualnom adresari}
    Rewrite(subor);
    {vypise (zapise) udaje z pola na miesto, kam ukazuje
    premenna subor u nas ukazuje na subor mojeData, takže
    sa zapisuje don}
    for i:=1 to 10 do Write(subor,pole[i]);
    {ukonci subor, potvrdi zapis a zatvori subor}
    CloseFile(subor);
end;
```

Dôležité kroky predstavuje:

- nastavenie premennej `subor` prostredníctvom príkazu `AssignFile`, ktoré do nej umiestni adresu miesta na disku, na ktorom bude vytvorený súbor so zadaným menom,
- samotné vytvorenie súboru na disku prebehne až pri použití príkazu `Rewrite`. Pokiaľ existoval súbor so zadaným menom, tak ho tento príkaz prepíše – nastaví jeho veľkosť na 0 a tým pádom zničí všetky údaje, ktoré predtým obsahoval,
- na zapisovanie údajov sa používa príkaz `Write`, ktorý v prípade, ak má na prvej pozícii umiestnenú premennú odkazujúcu na súbor, presmeruje zápis na miesto, kam ukazuje,
- napokon príkaz `CloseFile` uzavrie súbor a ukončí prácu s ním – v prípade neukončenia sa údaje doposiaľ vložené do súboru spravidla stratia.

Po vykonaní programu by sme mali mať k dispozícii v aktuálnom adresári súbor `mojeData` s veľkosťou 40 bytov (10 položiek 4 bytového typu `integer`).

Údaje do súboru zrejme nevkladáme len na to, aby boli uložené, ale zrejme na to, aby sme ich v prípade potreby dokázali prečítať a používať.

Napište program, ktorý zo súboru vytvoreného predchádzajúcim programom údaje načíta do poľa a vypíše od posledného po prvý.

Načítanie predstavuje opačný tok údajov – pokiaľ sme v predchádzajúcom prípade zapisovali, v tomto budeme čítať. Popis je opäť súčasťou zdrojového kódu.

```
const pocet=10;
var pole:array[1..pocet] of integer;
    subor:file of integer;
    i:integer;

begin
    {opat pripravime premennu subor tak,}
    {aby ukazovala na udaje v subore mojedata}
    AssignFile(subor, 'mojeData');
    Reset(subor);           {otvori subor na citanie}

    {precita zo suboru udaj, vlozi ho do pole[i] a posunie
    ukazovatel o poziciu dalej}
    for i:=1 to pocet do Read(subor,pole[i]);
    CloseFile(subor);       {zatvori subor}
end;
```

Príkaz `Reset` má okrem otvorenia súboru na starosti i nastavenie ukazovateľa na prvý záznam. Ukazovateľ sa po každom prečítaní posunie na ďalší údaj, ktorého veľkosť je daná typom súboru (`file of integer`).

Pokiaľ by sme zo súboru chceli prečítať viac hodnôt, ako je v ňom uložených, program skolabuje a vyvolá sa chyba počas behu programu.

Na ošetrenie takejto situácie máme k dispozícii funkciu `EOF` (*End Of File*), ktorá nás v každom momente dokáže informovať o tom, či sme už prečítali všetky údaje a sme na konci súboru. Jej aplikovaním sa program mierne modifikuje:

```
var pole:array[1..pocet] of integer;
    subor:file of integer;
    i:integer;

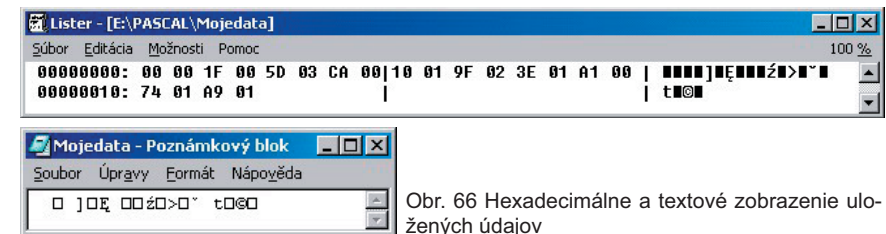
begin
    AssignFile(subor, 'mojeData'); {inicializacia suboru}
```

```
Reset(subor);           {otvorenie suboru na citanie}
i:=0;                  {index prvku, do ktoreho sa nacitava}
{kym nie je koniec (EOF) suboru}
while not EOF(subor) do begin
    inc(i);             {ak ideme citat, index sa posunie}
                        {mozno ho pouzit aj na zistenie}
                        {poctu ulozenych zaznamov}
    Read(subor,pole[i]); {udaj sa precita do pola}
end;
CloseFile(subor);      {zatvorenie suboru}
end;
```

Podmienka cyklu `not EOF(subor)` je splnená a telo cyklu sa vykoná, keď nie sme na konci súboru, teda keď `EOF(subor)` je `false` a jej negácia `not EOF(subor)` je `true`.

Údaje v súbore typu integer

Ak sa pozrieme do súboru obsahujúceho naše uložené pole čísel, uvidíme len akési nezmyselné znaky, ktoré pre nás nič nevyjadrujú, no systém ich z nejakého dôvodu načíta ako čísla, ktoré sme zadávali.



Obr. 66 Hexadecimálne a textové zobrazenie uložených údajov

Keď údaje zobrazíme hexadecimálne, vidíme, že súbor obsahuje 10 štvoric bajtov, t.j. pre každé zapísané číslo práve 4 bajty, ktorými je reprezentovaný typ `integer`. Pokiaľ by sme údaje chceli zrekonštruovať, urobíme to prevodom čísel zo 16-kovej do desiatkovej sústavy.

Výhodou používania typového súboru je rýchle a jednoduché ukladanie hodnôt daného typu, aj ich čítanie priamo do premenných bez potreby pretypovania, nevýhodou je nemožnosť (resp. presnejšie sťaženie) úpravy uložených údajov mimo programu.

V súčasných pomeroch by bolo o mnoho efektnejšie a vzhľadom na prácu používateľa efektívnejšie, keby sme údaje videli a mohli prepísať v súbore aj manuálne.

Typ súboru, ktorý definujeme v programe v prvom rade určuje, koľko bajtov sa do súboru zapíše pri vkladaní jednej hodnoty – typ `integer` zapíše svoje hodnoty do štyroch bajtov, `char` do jedného bajtu atď.

Vzhľadom na tento fakt nie je možné vytvoriť súbor s hodnotami typu `string`. Problémom je variabilná (nie pevná) dĺžka (a tým aj kapacita použitej pamäte) pre premenné typu `string` – nie je jasné koľko bajtov by sa pri jednom zápise zapísalo a naopak koľko by sa ich malo načítať.

Existuje možnosť vytvoriť súbor, ktorý bude mať veľkosť zapisovaného `stringu` vopred určenú, napr.:

```
subor:file of string[50];
```

t.j. pri každom zápise sa zapíše a pri každom prečítaní načíta 50 znakov, ale používanie tohto postupu je jednak neefektívne (vždy zapisuje 50 znakov bez ohľadu na to, či ich využijeme 49 alebo len jeden) a jednak by nedovolilo používateľovi zmeniť údaje úplne voľne (musel by presne dodržiavať počty znakov uložené v súbore, pretože ak by napr. prepísal znak reprezentujúci dĺžku `stringu`, mohol by sa tento správať nekontrolovateľne).

Obr. 67 `String[50]` v súbore – pokiaľ nie je ukladaný reťazec naplnený všetkými 50 znakmi, môže sa stať, že nevyužitý rozsah bude zaplnený náhodnými (pozostalými) hodnotami z pamäte.

Textový súbor

Špeciálnym typom súboru je `text`. Môžeme doň zapisovať a z neho čítať znaky (`char`), ale aj celý reťazec prakticky ľubovoľnej dĺžky (`string`). operácie s údajmi prebiehajú prostredníctvom príkazov:

`Write` - realizuje zápis,

`WriteLn` - realizuje zápis a zapisovací kurzor zapíše do nového riadku, takže ďalší údaj bude zapísaný do nového riadku,

`Read` - prečíta údaj,

`ReadLn` - prečíta údaj a čítací kurzor presunie do nového riadku.

Typ `text` je riešením, ktoré možno zvoliť ako vhodnú alternatívu k typovým súborom. Deklarácia má podobu:

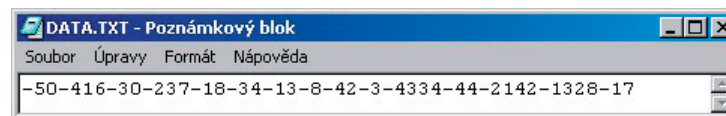
```
var subor:text;           nie           var subor:file of text;
```

V Delphi je potrebné deklarovať `subor` v časti `public` alebo `private` (nepostačí ho použiť v deklaračnej časti procedúry) prípadne definovať najprv vlastný typ v sekcii `type` (potom ho možno použiť i v deklarácii procedúry).

Napište program, ktorý uloží údaje z poľa čísel do textového súboru „data.txt“ tak, aby boli zrozumiteľné aj po otvorení v textovom editore.

```
const pocet=20;
var i:integer;
    pole:array[1..pocet] of integer;
    subor:text;
    hodnota:string;

begin
  {vygeneruju sa nahodne hodnoty <-50,50>}
  for i:=1 to pocet do pole[i]:=-50+random(101);
  AssignFile(subor, 'data.txt');
  Rewrite(subor);
  for i:=1 to pocet do begin
    Write(subor,pole[i]); {zapise obsah premennej pole[i]
                          ako text}
  end;
  CloseFile(subor);
end;
```



Obr. 68 Výsledok zápisu

Výsledkom našej činnosti je textový súbor, ktorý je síce na prvý pohľad zrozumiteľný, no na pohľad druhý ním príliš nadšený asi nebudeme – na viacerých miestach je totiž problematické posúdiť, kde jedno číslo končí a druhé začína, pričom vzhľadom na to, že nemáme presne určené koľkokciferné hodnoty sme zapisovali, nedokážeme spätne skonštruovať pôvodné pole.

Mohli by sme jednotlivé čísla oddeliť čiarkami alebo vkladať nové číslo vždy do nového riadku:

```
AssignFile(subor, 'data.txt');
Rewrite(subor);
for i:=1 to pocet do begin
  {zapise udaj z premennej a nasledne odriadkuje}
  WriteLn(subor,pole[i]);
end;
CloseFile(subor);
```

Napište program, ktorý načíta číselné údaje z textového súboru do poľa. Pred spustením operácie nech sa program opýta na meno súboru, z ktorého má čítať.

Požiadavka na parametrizovanie názvu súboru je v zadaní z toho dôvodu, aby sme si uvedomili, že program nemusí údaje načítavať vždy z rovnakého súboru a prostredníctvom premennej dokážeme ovplyvniť nielen názov súboru v aktuálnom adresári, ale pokiaľ zadáme kompletnú cestu k súboru, dokážeme ho načítať i z iného adresára (napr. `C:\Test\data.txt`).

```
var i,pocet:integer;
    pole:array[1..20] of integer;
    nazov,riadok:string;
    subor:text;

begin
    nazov:=Edit1.Text;
    pocet:=0;
    AssignFile(subor,nazov);
    Reset(subor);
    while not EOF(subor) do begin {kym nie je koniec suboru}
        inc(pocet); {zvysi pocet}
        ReadLn(subor,pole[pocet]); {vlozi do premennej hodnoty
            bez potreby konverzie}

    end;
    CloseFile(subor);
    // mozu prebehnúť napr. vypočty
    for i:=1 to pocet do {a na zaver vypis do Listboxu}
        Listbox1.Items.Add(IntToStr(pole[i]));
    end;
```

Vzhľadom na to, že každý údaj bol zapísaný do samostatného riadku, je potrebné i pri čítaní „odriadkovať“.

Pokiaľ si nie sme istý, že budeme zo súboru načítavať len čísla, je vhodné načítavať obsah súboru ako `string`, riadky konvertovať na čísla napr. prostredníctvom `Val` a v prípade chyby používateľa upozorniť.

Vylepšenie práce so súbormi

Bežný používateľ je pri práci s *Windows* zvyknutý pracovať so súbormi prostredníctvom dialógových okien (otváranie, ukladanie atď.). *Delphi* takúto činnosť podporuje a ponúka možnosť veľmi jednoducho integrovať dialógy do vytváranej aplikácie.

Dialógové okná máme k dispozícii na záložke *Dialogs*. Zaujímavé budú pre nás spočiatku len



Obr. 69 Záložka Dialogs

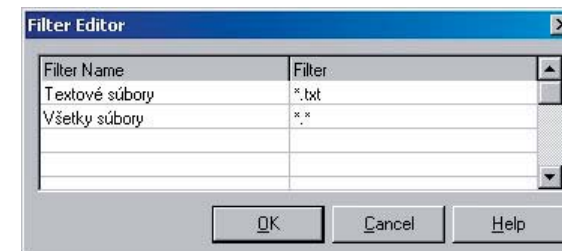
`OpenDialog` a `SaveDialog`. Pokiaľ ich chceme využívať, potrebujeme ich umiestniť na príslušný formulár. Samotným vložením sa však nič nezabezpečí – po spustení aplikácia nevykonáva o nič viac ani menej, ako keď na formulári dialógy umiestnené neboli. Tieto komponenty patria medzi **neviditeľné** – umiestnením na formulár dostaneme k dispozícii ich funkcie, no po spustení ich v aplikácii nevidíme.



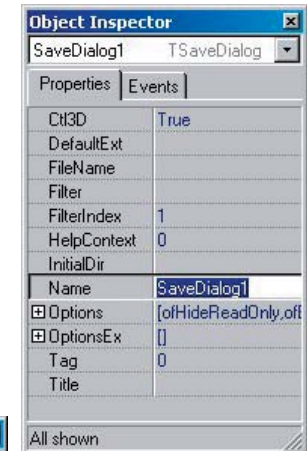
Skôr ako ich začneme používať, pozrime sa prostredníctvom *Object Inspector* na komfort, ktorý nám ponúkajú. Začnime napr. `SaveDialog`-om.

Parameter `DefaultExt` určuje koncovku (príponu), ktorá sa automaticky pridá súboru pri uložení (napr. vložením `txt` zabezpečíme, že súbor dostane automaticky túto koncovku a bude v systéme vystupovať ako textový). Pokiaľ ponecháme túto položku prázdnu, budeme odkázaní na ručné zadávanie koncovky pri vkladaní názvu súboru alebo pri jej vynechaní budeme ochudobnení o asociovanú ikonu a automatické otvorenie po dvojkliku.

Parameter `Filter` určuje zoznam typov súborov v dialógovom okne. Prvý stĺpec obsahuje



Obr. 71 Filter pre textové súbory



Obr. 70 Vlastnosti `SaveDialog`

text, ktorý bude v časti okna *Typ súboru* zobrazovaný, druhý masku súborov, ktoré sa budú zobrazovať.

Posledným pre nás zaujímavým nastavením je `Title`, kam môžeme umiestniť text, ktorý chceme zobrazovať v hlavičke ukladacieho dialógu.

Po realizácii nastavení môžeme prikráčať k napísaniu kódu a využitiu `SaveDialog` na získanie mena súboru.

Kód nepíšeme do udalosti zobrazenej po dvojkliku – častá chyba vo verziách nižších ako *Delphi 7.0*!!

SaveDialog funguje ako čierna skrinka, ktorú vyvoláme prostredníctvom príkazu `SaveDialog.Execute`. Vyvolaním sa zobrazí okno SaveDialogu a preberie riadenie, ktoré vráti až po svojom uzatvorení. K tomuto môže dôjsť viacerými spôsobmi:

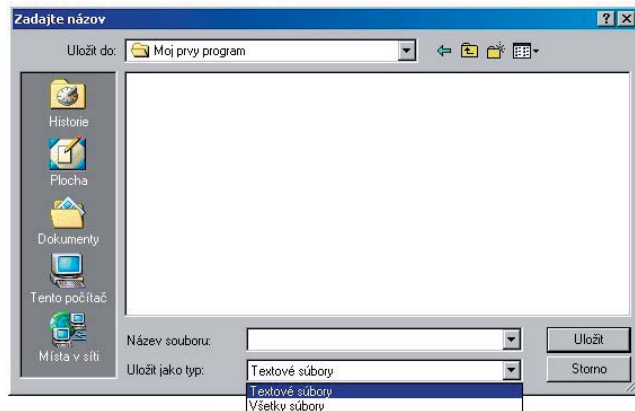
- kliknutím na ikonu „x“ a zatvorením okna alebo kliknutím na tlačidlo *Zrušiť*,
- zadaním názvu súboru a potvrdením cez *Uložiť*.

V prvom prípade potrebujeme informáciu o tom, že čierna skrinka sa zavrela a nič sme si nevybrali, v druhom o tom, že zadávanie súboru dopadlo úspešne a názov súboru. Spôsob zatvorenia okna je uložený priamo v metóde `Execute` – táto vráti v prípade zrušenia okna hodnotu `false`, v prípade zadania a potvrdenia hodnotu `true`. Názov súboru je pri pozitívnom ukončení uložený v parametri `FileName`.

Na odštartovanie operácie ukladania môžeme použiť tlačidlo, ktorému ako reakciu na stlačenie (udalosť `onClick`) vložíme nasledovný kód:

```
var nazov:string;
    i:integer;

begin
  {ak sa savedialog ukonci potvrdenim (kliknutim na ulozit),}
  {prebehne kladna vetva podmienky - ak sa zrusi, nic sa neudeje}
  if SaveDialog1.Execute then begin
    {ak bol teda zadany nazov suboru}
    {do premennej precitame nazov suboru zadany
     v SaveDialogu ulozená je kompletná cesta k súboru!!!}
    nazov:=SaveDialog1.FileName;
    {a nasleduje už známe ukladanie}
    AssignFile(subor,nazov);
    Rewrite(subor);
```



Obr. 72 Výsledok nastavení Savedialogu

```
{napr. hodnot z listboxu do suboru}
for i:=0 to Listbox1.Items.Count-1 do
  WriteLn(subor,Listbox1.Items.Strings[i]);
CloseFile(subor);
end;
end;
```

Je potrebné si uvedomiť, že `SaveDialog` rieši len získanie názvu súboru – ukladanie si musí programátor zabezpečiť sám.



Na získanie názvu súboru pre otvorenie slúži `OpenDialog`, ktorý má podobné parametre ako `SaveDialog`, možno mu nastaviť `Title` i `Filter` (ako prednastavený sa berie typ súboru zadaný vo filtri ako prvý). Riešenie pre načítanie údajov z textového súboru do `Listboxu` môže vyzerať nasledovne:

```
begin
  if OpenDialog1.Execute then begin
    nazov:=OpenDialog1.FileName;
    Listbox1.Items.Clear;
    AssignFile(subor,nazov);
    Reset(subor);
    while not EOF(subor) do begin
      ReadLn(subor,riadok);
      Listbox1.Items.Add(riadok);
    end;
    CloseFile(subor);
  end;
end;
```

Niekedy môže byť pre nás užitočné zistiť, či sa súbor so zadaným menom už na zadanom mieste nenachádza. Určitú podporu nám dokáže poskytnúť i `SaveDialog`, ktorý v prípade zadania názvu, ktorý sa už v aktuálnom priečinku vyskytuje, upozorní a opýta sa, či chceme daný súbor prepísať. Univerzálnejší je však test prostredníctvom `FileExist` v podobe:

```
if FileExists(nazov) then ... prípadne
if FileExists(SaveDialog1.FileName) then ...
```

pričom `nazov` reprezentuje kompletnú cestu k súboru. Pokiaľ obsahuje len názov súboru bez cesty, prehľadáva sa aktuálny adresár.

Na odstránenie súboru môžeme použiť príkaz

```
DeleteFile(nazov);
```


Ak pracujeme zväčša s tým istým súborom (napr. pri testovaní aplikácie), dokážeme zabezpečiť, aby sa otváral automaticky pri štarte aplikácie, čím ušetríme čas potrebný na jeho vyhľadanie a potvrdenie prostredníctvom `OpenDialogu`. Postačí nám otvorenie vložiť napr. do udalosti zobrazenia formulára – `FormShow`.

Pokiaľ máme otváranie súboru realizované už na inom mieste kódu, môžeme problém vyriešiť elegantne za pomoci podprogramov...

1. *Napište program, ktorý vypočíta aritmetický priemer čísel v textovom súbore. Čísla sú v súbore uložené tak, že v každom riadku je jedno číslo.*
2. *Je daný textový súbor `zoznam.txt`, o ktorom predpokladáme, že každé meno je uložené v osobitnom riadku. Vypíšte:*
 - a) mená začínajúce písmenom B,
 - b) meno žiakov, ktorí sú prvý a posledný podľa abecedy,
 - c) meno prvého a posledného žiaka v zozname podľa poradia v súbore,
 - d) najdlhšie meno,
 - e) meno, ktoré sa vyskytuje najviac ráz,
 - f) mená, ktoré sa vyskytujú práve raz.
3. *Sú dané textové súbory `prvy.txt` a `druhy.txt`. Napište program, ktorý spojí tieto dva súbory do jedného súboru s názvom `spolu.txt`, tak, že v ňom budú uložené všetky údaje z prvého súboru a za nimi údaje z druhého súboru.*
4. *Napište program, ktorý upraví zadaný textový súbor tak, že veľké písmená zamení za malé, malé za veľké a ostatné znaky ponechá bez zmeny.*
5. *Textový súbor obsahuje text telegramu. Napište program, ktorý zistí jeho cenu, ak každé písmeno v telegrame stojí 1 Sk, pričom medzery sa do ceny nerátajú.*
6. *Zistite počet slov v zadanom textovom súbore. Predpokladajte, že slová neprechádzajú medzi riadkami. Ošetríte voči výskytu viacerých medzier za sebou.*
7. *V aktuálnom adresári je uložený súbor s matematickými príkladmi. V každom riadku obsahuje matematický príklad s výsledkom, napr. $11+23=44$, v druhom riadku $15*2=31$ atď., pričom výpočty predstavujú súčty, súčiny a rozdiely. Do nového súboru príklady „opravte“: ak je výsledok správny, uveďte za príklad text „OK“, ak nie je uveďte „ERROR:“ a správny výsledok. Na koniec súboru doplňte riadok, v ktorom bude uvedený počet správnych, počet nesprávnych riešení a percentuálna úspešnosť.*

9 Podprogramy

predpoklady na zvládnutie lekcie:

- znalosť základov práce v prostredí Borland Delphi
- znalosť údajových typov `string` a `integer`

obsah lekcie:

- pojem podprogram
- typy podprogramov
- globálne a lokálne premenné
- parametre a typy parametrov podprogramov

cieľ:

- naučiť sa používať podprogramy
- použiť výhody používania podprogramov a parametrov

Dôvody používania podprogramov

Väčšina programovacích jazykov umožňuje vytvárať určité ucelené a relatívne samostatné časti programov, ktoré sa označujú ako podprogramy. Správne používanie dokáže výrazne zrýchliť ako tvorbu, tak i ladenie programu.

Podprogramy používame, ak:

- sa v programe vyskytuje úplne **rovnaká postupnosť príkazov** viackrát na rôznych miestach. Je bez diskusie, že opakovanie tých istých postupností príkazov vedie k nevhodnému využívaniu prostriedkov (zbytočne sa zväčšuje veľkosť výsledného súboru, a to ako aplikácie, tak i zdrojového kódu) a na druhej strane predstavuje nočnú moru pre tých, ktorí majú zdrojový kód neskôr ladiť – opakovaním rovnakého kódu sa program stáva neprehľadným a každú úpravu i opravu chyby treba realizovať na všetkých miestach, kde sa táto postupnosť vyskytuje (častým javom je, že raz sa zabudne zapísať jedna zmena na jednom mieste, potom ďalšia na inom a napokon sa programátor čuduje, prečo sa program správa „divne“ a náhodne),
- v programe sa viackrát vyskytuje **podobná postupnosť príkazov odlišujúca sa iba parametrami**,
- potrebujeme **zvýšiť prehľadnosť programu** a priblížiť jeho zápis „ľudskému“ riešeniu. Veľmi často programátor postupuje pri vytváraní

programu tak, že najprv určí postupnosť vykonania jednotlivých podprogramov, ktorým dá názvy vystihujúce činnosť, ktorú popisujú a až potom ich naprogramuje, napr.:

```
VycistiMriezku;
NacitajHodnotyZoSuboru;
NajdiNajmensiPrvok;
Vypis;
```

Výhodou takéhoto postupu je, že okrem zrozumiteľného zápisu algoritmu si ujasní vlastné myšlienkové pochody ešte predtým, ako začne písať samotný kód,

- riešenie je formulované **rekurzívne**, t.j. je opísané pomocou “samého seba”, napr. $n! = n * (n-1)!$ alebo $a_n = a_{n-1} + d$. Tejto problematike sa budeme podrobnejšie venovať v samostatnej kapitole.

Procedúry

Prvým typom podprogramu je **procedúra**. Spravidla obsahuje postupnosť príkazov, ktoré riešia v programe nejakú podúlohu. Procedúra pozostáva z rovnakých častí ako program – obsahuje povinnú hlavičku pozostávajúcu z kľúčového slova `procedure` a názvu procedúry, môže (nemusí) nasledovať deklaračná časť a po nej príkazy umiestnené medzi programovými zátvorkami `begin` a `end`. Všeobecný zápis má podobu:

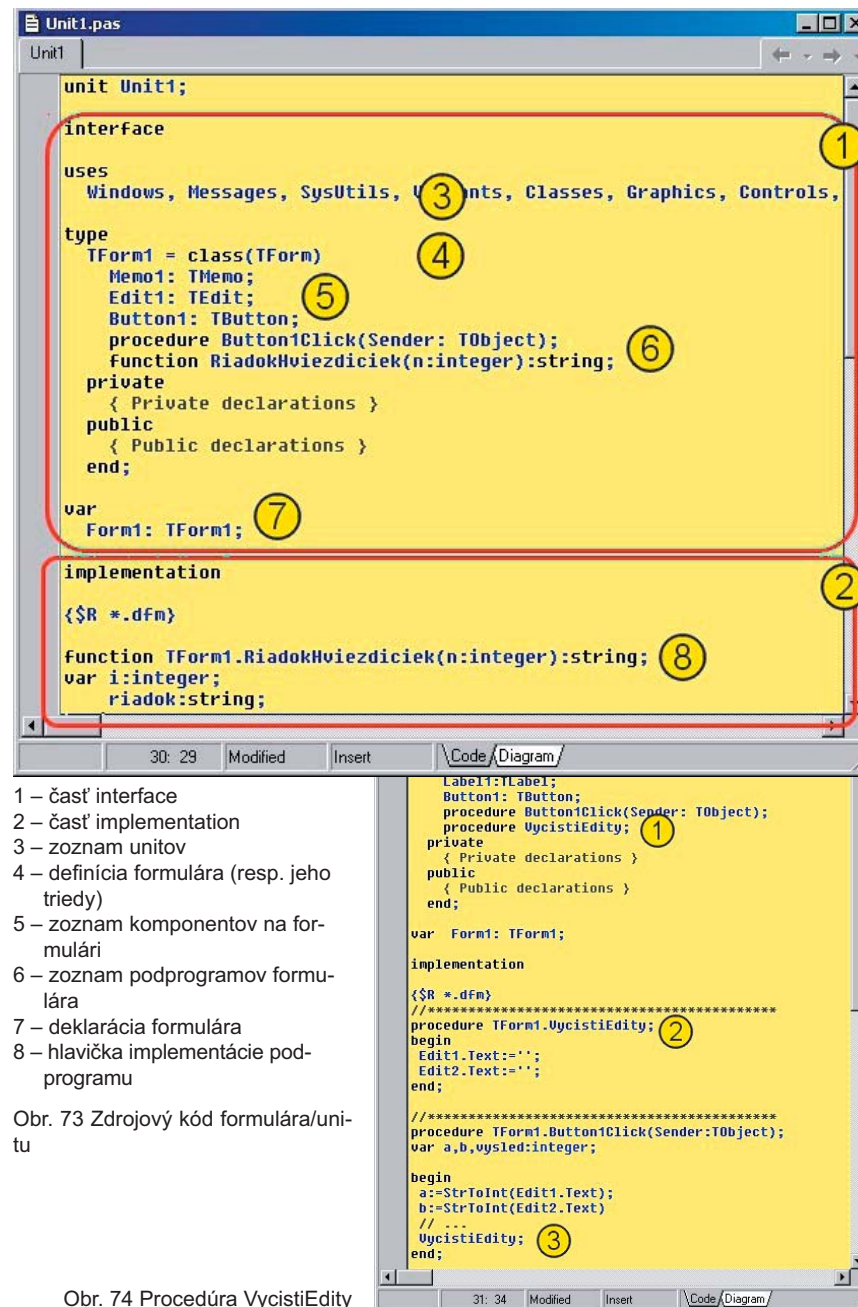
```
procedure NazovProcedury;
```

Napište program, ktorý bude prostredníctvom klikania na tlačidlá vykonávať matematické operácie pre zadané dvojice čísel. Po výpočte zobrazí výsledok a vyčistí obsah editovacích riadkov pre zadávanie vstupov. História výpočtov uchováva v Listboxe.

V prostredí *Delphi* pracujeme s podprogramami (a konkrétne s procedúrami) už od začiatku. Možno ste si to neuvedomili, no všetky príkazy, ktoré sme doposiaľ zapisovali, sa vkladali do tela udalostnej procedúry. Zvyčajne šlo o udalosť kliknutia na tlačidlo.

Procedúry patriace formuláru sú umiestnené v samostatnom súbore – **unit**. Každý unit pozostáva z dvoch základných častí:

- **interface** (rozhranie) obsahuje zoznam použitých unitov (iných), z ktorých sa v aktuálnom formulári využívajú komponenty i podprogramy, ďalej zoznam typov a v rámci definovaného typu pre formulár (`TFormX`)



i zoznam podprogramov, ktoré sú preň vytvorené. Sekciu končí deklaračná časť. Všetky prvky v časti `interface` sú prístupné aj pre iné programy alebo unity, ktoré by náš unit použili,

- **implementation** (implementačná časť) obsahuje zdrojový kód podprogramov definovaných v časti `interface`.

Pri vytváraní vlastného podprogramu ho budeme aspoň spočiatku viazať na formulár (unit), v ktorom ho vytvárame – zjednoduší sa nám prístup ku komponentom (pokiaľ sa na ne budeme v rámci podprogramu odvolávať) a zlepší sa prehľadnosť kódu. Pri zápise v implementačnej časti budeme pred názov vkladať jeho vlastníka (napr. `TForm1`), do `interface` ho budeme vkladať ako súčasť triedy `TForm1`, už bez vlastníka.

Procedúra na vyčistenie `Editov` potom môže vyzeráť podobne ako na obrázku na predchádzajúcej strane.

Samotný výpočet odohrávajúci sa napr. pri kliknutí na tlačidlo zabezpečujúce sčítanie môže vyzeráť nasledovne:

```
procedure TForm1.Button1Click(Sender:TObject);
var a,b,vys:integer;
    riadok:string;

begin
  a:=StrToInt(Edit1.Text);
  b:=StrToInt(Edit2.Text);
  vys:=a+b;
  Label1.Caption:=IntToStr(vys);
  riadok:=Edit1.Text+ ' + ' +Edit2.Text+' =' +IntToStr(vys);
  ListBox1.Items.Add(riadok);
  VycistiEdity;
end;
```

Globálne a lokálne premenné

Napište procedúru, ktorá nájde v poli prvok s najväčšou hodnotou.

V programe je v prvom rade potrebné hodnoty do poľa načítať, na to by však naše doterajšie vedomosti mali postačovať. V ďalšom kroku budeme v programe volať procedúru na výpočet maxima. Získanú hodnotu umiestnime do premennej `max` a v tele hlavného programu (v *Delphi* v procedúre obsluhujúcej udalosť kliknutia na tlačidlo) vypíšeme.

```
var max:integer;
    pole:array[1..20] of integer;
//*****
procedure NajdiMaximum;
var i:integer;

begin
  max:=pole[1];{na zaciatku za maximum prehlasime 1. prvok pola}
                {budeme prechadzat pole od dalsieho prvku}
  for i:=2 to 20 do {ak najdeme vacsiu hodnotu, prehlasime ju za max}
    if pole[i]>max then max:=pole[i];
end;
{***** hlavny program *****}
begin
  NacitajHodnotyDoPola;{nacita, prip. vygeneruje hodnoty do pola}
  NajdiMaximum;      {po vykonani tohto riadku bude v premennej max}
                      {umiestnena maximalne hodnota pola}
  ShowMessage(IntToStr(max));
end;
```

Ak sa lepšie zahľadíte na prezentovaný kód, určite si všimnete, že premenné sú deklarované na dvoch miestach – na začiatku programu sú samostatne umiestnené `max` a `pole`, v rámci procedúry `NajdiMaximum` zase premenná `i`.

Premenné deklarované na začiatku programu sú prístupné a použiteľné na ľubovoľnom mieste programu počas jeho behu a označujeme ich ako **globálne**. Premenné definované v tele podprogramu vznikajú až pri volaní podprogramu a môžeme k nim pristupovať len v rámci podprogramu, v ktorom sú deklarované. Po ukončení podprogramu sa z pamäte uvoľňujú – označujeme ich ako **lokálne**.

V *Delphi* sú globálne premenné deklarované v časti `interface`. Pokiaľ definujete premennú v udalostnej procedúre, je prístupná len v jej tele, iné podprogramy definované mimo procedúry k nej prístup nemajú.

Globálne premenné sa v rámci nášho programu využili na uloženie hodnôt poľa (pracovali sme s ním v procedúre pre načítanie hodnôt i pri samotnom vyhľadávaní) a na „vynesenie“ hodnoty `max` z procedúry `NajdiMaximum`.

Vo všeobecnosti platí, že v programe je potrebné používať minimálne množstvo globálnych premenných, pretože pri väčších programoch znižujú prehľadnosť a podprogramy strácajú nezávislosť. Ak chceme použiť podprogram, je potrebné mať k dispozícii naplnené globálne premenné, ktoré do podprogramu vstupujú i globálne premenné, prostredníctvom ktorých údaje z podprogramu vystupujú.

Funkcie

Okrem procedúr ponúkajú programovacie jazyky i podprogramy, ktoré ako výsledok svojej činnosti vrátia hodnotu. Označujeme ich ako **funkcie**. Funkcie ako také poznáme už z matematiky (*abs*, *sin*, *cos* a pod.), mnohé máme k dispozícii i v programovacích jazykoch (*Length*, *Copy*, *IntToStr* a pod.) a dokážeme vytvoriť i vlastné. Funkcia sa definuje kľúčovým slovom *function* a za jej názov sa uvádza typ výsledku.

```
function NazovFunkcie:typ;
```

pričom typ predstavuje niektorý z jednoduchých, prípadne i štruktúrovaných typov (pole).

Volanie funkcie je výraz, ktorého hodnotou je výsledok získaný vykonaním funkcie. V programe ho môžeme použiť všade tam, kde sa môže použiť hodnota rovnakého typu ako je výsledok funkcie. Napríklad v priradovacom príkaze môže byť vložená do premennej.

```
mojText:=IntToStr(cislo);
max:=VratMaximum;
```

Výsledok funkcie sa definuje v jej tele priradovacím príkazom, v ktorom sa výsledná hodnota priradí názvu funkcie. Najlepšie vysvetlenie poskytnete nasledujúci príklad.

Upravte úlohu na hľadanie maxima v poli tak, aby maximálny prvok pola vrátila funkcia.

```
var pole:array[1..20] of integer;
    max:integer;
//*****
function NajdiMaximum:integer; {vrati hodnotu typu integer}
var i,mm:integer;

begin
  mm:=pole[1];
  for i:=2 to 20 do if pole[i]>mm then mm:=pole[i];
  NajdiMaximum:=mm; {priradenie „vynesie“ hodnotu z funkcie}
end;
{***** hlavny program *****}
begin
  NacitajHodnotyDoPola;
  max:=NajdiMaximum; {do max sa priradi hodnota ziskana}
                    {vykonanim funkcie}
  ShowMessage (IntToStr (max) );
end;
```

Úlohu sme modifikovali, v tele funkcie sme do jej názvu priradili hodnotu, ktorú požadujeme vrátiť ako výsledok. Tento bude priradením do premennej *max* použitý v ďalšom behu programu.

V tele funkcie sme namiesto premennej *max* (z riešenia pomocou procedúry) použili premennú *mm*. Iný názov sme zvolili len s didaktických dôvodov – aby sa čitateľovi neplietli názvy. Ak by sme v tele funkcie *NajdiMaximum* deklarovali premennú s rovnakým názvom ako je premenná globálna (*max*), došlo by k **prekrytiu** – v tele funkcie by sa zmeny realizovali len na lokálnej premennej, globálna by zostávala nepovšimnutá a po ukončení podprogramu by obsahovala rovnakú hodnotu ako pred jeho spustením.

Globálnu premennú *max* by sme dokonca mohli v našom riešení i vynechať a telo programu upraviť na:

```
NacitajHodnotyDoPola;
ShowMessage (IntToStr (NajdiMaximum) );
```

Funkcia *NajdiMaximum* vráti hodnotu, ktorá sa použije ako argument pre *ShowMessage*.

Parametre podprogramov

Rovnako ako algoritmy vďaka možnosti zadávať rôzne vstupné údaje, dokážu riešiť úlohu pre rôzne vstupné hodnoty, i vhodne napísané podprogramy dokážeme využiť tak, aby úlohu riešili pre rôzne vstupy. Jeden zo spôsobov, akým možno do podprogramu „poslať“ hodnoty sme už prezentovali – spracúvali sme údaje uložené v globálnej premennej.

Ak však chceme, aby podprogram predstavoval samostatný a nezávislý celok (vďaka čomu by bolo jeho použitie univerzálne) a aby sme sa nemuseli zaoberať neustálym sledovaním globálnych premenných, potrebujeme použiť iný aparát.

Funkcie i procedúry môžu komunikovať (vymieňať si údaje) s ostatnými časťami programu pomocou špeciálnych premenných, tzv. **parametrov**.

Pri definícii podprogramu sa v hlavičke uvedie zoznam parametrov, ktoré budú v podprograme zastupovať hodnoty, ktoré doň pošleme. Spolu s názvom sa uvedie aj ich typ. Tieto parametre označujeme ako **formálne** – zastupujú hodnotu, s ktorou bude podprogram pri svojom vykonávaní pracovať. Napr.:

```
procedure Vypis(a,b:integer;c:string)
function Vypocet(a,b:integer;c:string):integer
```

pričom parametre rovnakého typu oddeľujeme čiarkou, v prípade použitia parametrov viacerých typov použijeme na oddelenie skupín bodkočiarku.

Pri volaní podprogramu sa formálnym parametrom priradia hodnoty – **skutočné parametre**, s ktorými bude skutočne pracovať. Volanie podprogramu potom môže vyzerat' napr. nasledovne:

```
Vypis(10, 20*x, 'test')
vysledok:=Vypocet(x, 10, 'test')
```

pričom ako skutočný parameter môžeme použiť konštantu, výraz alebo premennú.

Hodnotu (obsah) formálnych parametrov možno v tele podprogramu, ktorý ich používa, meniť, ich zmena však nemá vplyv na žiadne premenné deklarované mimo daného podprogramu.

Napište podprogram, ktorý pre zadané n vloží do prvého riadku Listboxu jednu hviezdičku, do druhého dve atď. až po n-tý riadok.

```
procedure Hviezdy(n:integer);
var i:integer;
    riadok:string;
begin
    riadok:=''; {obsahuje pocet hviezd, ktore treba vypisat}
    for i:=1 to n do begin
        riadok:=riadok+'*'; {v kazdom riadku sa prida jedna *}
        Listbox1.Items.Add(riadok); {vypis}
    end;
end;
```

Premenná n obsahuje počet riadkov, ktoré sa majú spracovať a využíva sa v podprograme len na čítanie. Vykonanie procedúry by sme mohli zabezpečiť napr. volaním `Hviezdy(4);`.

Napište podprogram, ktorý na základe procedúry z predchádzajúcej úlohy vykreslí polovicu vianočného stromu v podobe ako na obrázku:

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

Úlohu vyriešime dvoma spôsobmi:

- v hlavnom programe zavoláme procedúru na vykreslenie trojuholníkov postupne so zväčšujúcim sa parametrom,
- napíšeme podprogram, ktorý bude vytvárať strom na základe zadania počtu jeho „poschodí“.

```
...
begin
    for i:=2 to 4 do Hviezdy(i);
end;
```

Procedúra `Hviezdy` sa volá v cykle najprv s hodnotou 2 (vykreslí sa trojuholník s prvým riadkom obsahujúcim jeden znak a druhým dva znaky), potom s parametrom 3 (vykreslí sa trojuholník s jednou, dvoma a tromi hviezdami) atď.

Druhá možnosť je zaujímavejšia – v procedúre `Strom` budeme volat' procedúru `Hviezda` vďaka čomu získame pomerne univerzálnu procedúru:

```
procedure Strom(n:integer);
var i:integer;
begin
    for i:=2 to n do Hviezda(i);
end;
```

Rovnako ako do procedúr, môžeme i do funkcií posielat' parametre. Tento prístup je pre nás v určitej podobe známy už z matematiky.

Napište funkciu, ktorá pre dve zadané hodnoty vráti väčšiu z nich. Ak sú rovnaké, návratová hodnota bude jedna z nich.

```
function Max(a,b:integer):integer;
begin
    if a>b then Max:=a {ak je a vacsie, Max vrati a}
        else Max:=b; {inak je b vacsie a Max vrati b}
end;
```

Do funkcie posielame dve celé čísla a výsledok bude tiež celočíselný, vo funkcii dôjde k ich porovnaniu a na základe neho sa do názvu funkcie priradí väčšia hodnota.

Volanie môže mať podobu:

```
vysledok:=Max(10,20) alebo vysledok:=Max(a,b)
prípadne vysledok:=Max(Max(10,20),30)
```

V poslednom volaní sa najprv nájde hodnota vnorenej funkcie `Max(10,20)`. Výsledok z nej sa potom použije na ďalšej úrovni a zavolá sa `Max(20,30)`, ktorá vráti konečný výsledok.

1. Napište funkciu, ktorá pre zadané n vráti $n!$
2. Napište funkciu, ktorá pre zadané hodnoty a , n vráti a^n .
3. Je daný polomer kruhu. Napište funkcie na výpočet obvodu a obsahu kruhu.
4. Napište funkciu, ktorá zistí počet písmen najdlhšieho reťazca v poli.
5. Napište funkciu, ktorá zistí, či dané číslo je prvočíslo alebo nie. Využite ju v programe, ktorý pre zadané číslo n a vypíše prvých n prvočísel.
6. Napište funkciu, ktorá zistí, či dané číslo je párne a výsledok vráti v prostredníctvom typu boolean. Funkciu využite v hlavnom programe, ktorý načíta do poľa n čísel a vypíše všetky párne čísla.
7. Napište funkciu, ktorá zistí, či je dané číslo palindrom (odpredu aj odzadu sa číta rovnako). Funkciu využite v programe, ktorý pre zadaný interval $\langle a, b \rangle$ vypíše všetky číselné palindromy.

V praxi sa často stretáme s prípadmi, keď od podprogramu potrebujeme vrátiť viac ako jednu hodnotu alebo potrebujeme upraviť obsah premennej deklarovanej mimo podprogramu. Na tento účel môžeme síce použiť globálne premenné, no kvôli nim by podprogram mohol stratiť svoju nezávislosť od programu, z ktorého je volaný. V jazyku pascal máme na tento účel k dispozícii aparat, ktorým to vieme zabezpečiť prostredníctvom parametrov.

Dve triedy súťažia v zbere papiera. Evidujte odovzdané množstvá papiera jednotlivých žiakov prostredníctvom poľa.

- a. zistite, ktorá trieda nazbierala viac papiera,
- b. zistite, v ktorej triede bol vyšší priemer na žiaka.

Čítajte údaje z Listboxu a do procedúry pošlite názov Listboxu, pričom jeho typ bude TListBox.

Pri riešení zadania sa dostávame do situácie, keď pre načítanie hodnôt do oboch polí použijeme prakticky rovnaký postup (načítanie postupnosti ukončenej nulou), čo nám ponúka možnosť využiť podprogram. Okrem toho by sme v rámci podprogramu mohli už pri načítaní zistiť priemery i celkové nazbierané množstvo.

Vzhľadom na množstvo údajov, ktoré vyžadujeme, nám funkcia vracajúca ako výsledok jediná hodnotu, určite nepostačí. Navyše by bolo vhodné, aby sme si údaje uložené do poľa i zapamätali.

Pri volaní podprogramov, ktoré sme doteraz programovali, sa formálnym parametrom priradzovala (odovzdávala) určitá hodnota a tie sa potom správa-

li ako lokálne premenné podprogramu až kým po skončení výpočtu zanikli. Takémuto spôsobu odovzdávania skutočných parametrov formálnym hovorme **odovzdávanie hodnotou**. Parametre odovzdávané hodnotou slúžia na to, aby prostredníctvom nich do podprogramu vstúpili zvonka vstupné údaje.

Ak chceme z podprogramu získať výstupné údaje a pritom komunikovať s volajúcim programom prostredníctvom parametrov, musíme zabezpečiť, aby sa po skončení výpočtu ich obsah uchoval.

Pri volaní podprogramu odovzdáme formálnemu parametru meno (adresu) premennej z volajúceho programu. Všetky zmeny, ktoré sa s takýmto formálnym parametrom v tele podprogramu udejú sa premietajú do obsahu tejto premennej. Premenná po skončení výpočtu podprogramu nezaniká, lebo nevznikla pri jeho volaní a podprogram ju len použil. Takémuto odovzdávaniu parametrov hovorme **odovzdávanie adresou**.

S oboma spôsobmi odovzdávania parametrov sme sa už stretli v kapitole o údajovom type `string`, pre ktorý sme uviedli niektoré operácie na prácu s reťazcami. Napríklad procedúra `Val`, ktorá prevádza zadaný reťazec na číslo, má tri parametre.

```
Val(text, cislo, kod);
```

V prvom zadáme vstupný reťazec, ktorý sa má previesť. Je to parameter odovzdávaný hodnotou (hodnotou môže byť konkrétny reťazec v apostrofoch, obsah premennej alebo výraz, ktorého hodnotou je reťazec).

V druhom parametri určíme meno číselnej premennej, do ktorej sa má uložiť výsledné číslo a v treťom parametri uvedieme meno premennej, ktorá má po skončení procedúry obsahovať informáciu o úspešnosti alebo neúspešnosti prevodu. Sú to parametre odovzdávané adresou (adresu premennej v pamäti jednoznačne určuje jej meno). Príklad volania:

```
Val('1234', cislo, kod);
```

Určite, ktoré parametre vo volaniach podprogramov pre prácu s reťazcami sú odovzdávané hodnotou a ktoré adresou:

```
dĺzka:=length(mojString)
zluceny:=Concat(prvy, druhy, treti)
Delete(s, 3, 2)
Insert('by', s, 3)
poz:=Pos('ma', 'moja mama má maslo')
```

Kľúčovým slovom, ktoré definuje parameter ako odovzdávaný adresou je **var**. Použitie v podprograme môže vyzeráť ako v riešení zadaného príkladu:

```
{na to, aby sme mohli použiť pole ako parameter podprogramu,}
{potrebujeme ho definovať ako typ}
Type TZoznam=array[1..30] of integer;

var trieda1, trieda2:TZoznam;
    sucet1, sucet2:integer;      {sucty za triedu}
    priemer1, priemer2:real;    {priemery za triedu}
//*****
{parametre pred ktorými sa nachádza "var" menia v podprograme
hodnoty premenných, ktoré sú pri volaní umiestnené na rovnakých
miestach }
procedure Operacia(var trieda:TZoznam;var suc:integer;
    var priem:real;Lb:TListBox);
var pocet, hodnota:integer;

begin
    pocet:=0;
    suc:=0;                    {inicializácia premenných}
    repeat                      {opakuje, kým sa nezadá hodnota 0}
        hodnota:=StrToInt(Lb.Items[pocet]); {nacita sa}
        if hodnota>0 then begin      {ak je nenulová}
            inc(pocet);             {zvysi sa počet zapojených žiakov}
            trieda[pocet]:=hodnota; {zapamätá sa v poli}
            suc:=suc+hodnota;      {zvysi sa sucet za triedu}
        end;
    until hodnota=0;            {koniec nacistavacieho cyklu}
    priem:=suc/pocet;          {vypocita sa priemer}
end;
{***** startovací program *****}
procedure TForm1.Button1Click(Sender:TObject);
begin
    {do premennej trieda1 sa uloží zoznam hodnôt pre žiakov,}
    {do sucet1 hodnota, ktorá je v procedure vedena ako sucet}
    {ListBox1 predstavuje zoznam hodnôt z triedy}
    Operacia(trieda1, sucet1, priemer1, ListBox1);
    Operacia(trieda2, sucet2, priemer2, ListBox2); {analógia}
    if sucet1>sucet2 then ShowMessage('Vyhrala 1. trieda')
        else ShowMessage('Vyhrala 2. trieda');
    if priemer1>priemer2 then
        ShowMessage('Lepší priemer má 1. trieda')
    else
        ShowMessage('Lepší priemer má 2. trieda');
end;
```

Procedúra naplní polia pre prvú i pre druhú triedu, zároveň v premenných `sucet1`, `priemer1` a `sucet2`, `priemer2` získavame vypočítané hodnoty. Premenné `trieda1`, `sucet1`, `priemer1` sú svojím formálnym parametrom trieda, `suc`, `priem` odovzdané adresou (takisto pre druhú triedu).

Ošetrite program tak, aby bral do úvahy i rovnosť výsledkov.

Mechanizmus volania podprogramu

Keď sa objaví v programe volanie podprogramu:

1. zapamätá sa návratová adresa (kam sa bude treba vrátiť),
2. vytvorí sa lokálne premenné podprogramu (s nedefinovanou hodnotou) a formálne parametre s hodnotami nastavenými podľa vstupov,
3. prenesie sa riadenie programu do tela podprogramu,
4. vykonajú sa všetky príkazy podprogramu,
5. zrušia sa lokálne premenné,
6. riadenie sa vráti za miesto v programe, odkiaľ bol podprogram volaný.

Kvôli potrebe vykonania všetkých týchto operácií môže byť rýchlosť programu s podprogramami o čosi nižšia ako bez ich použitia, pri súčasných procesoroch však ide o mizivé a prakticky nezmerateľné zdržanie.

1. *Napište podprogram, ktorý vymení hodnoty dvoch zadaných premenných.*
2. *Napište podprogram, ktorý nájde maximálne hodnoty v dvoch poliach a porovná ich.*
3. *Napište podprogram, ktorý pre zadané pole zistí jedným volaním jeho minimum a maximum.*
4. *Napište podprogram, ktorý pre zadaný názov súboru a meno človeka prehľadá súbor a vypíše či a koľkokrát sa v ňom človek nachádza (súbor môže obsahovať len zoznam mien, napr. návštevníkov budovy).*
5. *Napište podprogram, ktorý zaokrúhli číslo na zadaný počet desatinných miest. Riešte ako funkciu i ako procedúru.*

Lekcia 10

10 Štruktúrované typy

predpoklady na zvládnutie lekcie:

- schopnosť práce s údajovým typom pole
- zručnosť pri práci s komponentom Listbox

obsah lekcie:

- záznam ako základný štruktúrovaný typ
- manipulácia s položkami záznamu na úrovni aplikácie
- kombinácie štruktúrovaných typov

cieľ:

- schopnosť vybrať pre vlastnú aplikáciu vhodné typy na prácu s údajmi
- využívanie záznamov v aplikáciách databázového typu

Záznam (record)

Značnú časť v praxi nasadených aplikácií tvoria najrozličnejšie evidencie, databázy a zoznamy. Udržiavanie a manipulácia s týmito údajmi predstavuje jednu z najčastejších požiadaviek zákazníkov.

Evidujte údaje o svojich učiteľoch (meno, priezvisko, počet detí, vek, triedny (áno/nie)) a v evidovanom zozname zabezpečte vyhľadávanie podľa ľubovoľného kritéria.

Na základe doterajších vedomostí by sme mohli pristúpiť k evidencii údajov v niekoľkých poliach – pre každý údaj by existovalo samostatné pole, pričom by platilo, že priezvisku uvedenom v 6. položke poľa zodpovedá meno, vek i ostatné údaje uložené v 6. položke ostatných polí. Polia by sme teda mohli deklarovať nasledovne:

```
var meno,priezvisko      :array[1..20] of string;
    vek, deti            :array[1..20] of integer;
    triedny              :array[1..20] of boolean;
```

a vkladanie údajov by sme zabezpečovali napr. nasledovným spôsobom:

```
meno[7]                 := 'Jozef';
priezvisko[7]           := 'Mrkva';
```

```
vek[7]                  :=44;
deti[7]                 :=5;
triedny[7]              :=true;
```

Na hľadanie by sme mohli vytvoriť samostatné funkcie, ktoré by pre zadanú hodnotu prehľadali príslušné pole a pokiaľ by ju našli, vrátili by jej index v rámci poľa. Na základe indexu by sme potom mohli zobrazit' aj ostatné údaje patriace príslušnému človeku. Pre zadané priezvisko by mohla funkcia vyzerať nasledovne:

```
function IndexPodlaPriezviska (hladany:string) :integer;
var i,index:integer;

begin
index:=-1; {pokial hodnotu nenajde, vrati -1}
for i:=1 to pocetLudi do {prehlada zoznam}
    {ak sa naslo zapamata si poradove cislo}
    if hladany=priezvisko[i] then index:=i;
IndexPodlaPriezviska:=index;{vrati ulozenu hodnotu alebo -1}
end;
```

1. Upravte funkciu tak, aby sa dala použiť univerzálne na prehľadanie ľubovoľného poľa typu string.
2. Daná funkcia vráti index posledného prvku, ktorý našla. Upravte ju tak, aby vracala index prvého nájdeného učiteľa.

Až do tohto okamžiku je naša práca s údajmi bezproblémová, ale...

Napište program, ktorý umožní usporiadanie evidovaných údajov podľa ľubovoľného kritéria (mena, priezviska, počtu detí, veku).

Na usporiadanie údajov by sme mohli použiť napr. bublinové triedenie, pri ktorom sa porovnávajú susedné hodnoty a ak je napr. hodnota napravo menšia ako hodnota naľavo, vymenia sa. A v tomto momente sa riešenie stáva zbytočne prácnym – na to, aby sme vymenili údaje o učiteľoch potrebujeme vymeniť údaje vo všetkých poliach (ak by sme vymenili napr. len vek, nezodpovedali by mu údaje na rovnakých indexoch v ostatných poliach).

Usporiadanie údajov podľa veku môže mať nasledovnú podobu:

```
procedure UsporiadajPodlaveku
var i,j,ipom,pocet:integer; {na vymenu hodnot je potrebné}
    bpom:boolean; {pouzit pomocne premenne}
    spom:string; {pre kazdy typ osobitnu}
```

```

begin
  for i:=1 to pocetLudi do {aby bol cely zoznam usporiadany,}
    {potrebujeme porovnanie opakovat}
    for j:=1 to pocetLudi-1 do {prechod po zozname}
      if vek[j]>vek[j+1] then begin {ak je lavy vacsi ako pravý}
        {prebehne vymena vsetkych poloziek}
        ipom:=vek[j]; vek[j]:=vek[j+1]; vek[j+1]:=ipom;
        ipom:=deti[j]; deti[j]:=deti[j+1]; deti[j+1]:=ipom;
        spom:=meno[j]; meno[j]:=meno[j+1]; meno[j+1]:=spom;
        spom:=priezvisko[j]; priezvisko[j]:=
          priezvisko[j+1]; priezvisko[j+1]:=spom;
        bpom:=triedny[j]; triedny[j]:= triedny[j+1];
        triedny[j+1]:=bpom;
      end;
    end;
end;

```

Samotná výmena je pomerne rozsiahla operácia, ktorá spomalí ako hľadanie chýb tak i samotné vykonávanie programu. A to máme len 5 charakteristík učiteľa (v praxi sa často pohybujeme v rozpätí desiatok a stoviek položiek). Ak by sme ďalej rozvíjali prácu s týmito údajmi, narazili by sme i na ďalšie komplikácie (vymazávanie údajov, ukladanie z polí rôzneho typu do súboru atď.).

Elegantné riešenie ponúka využitie údajovej štruktúry záznam (record). Predstavuje **heterogénnu štruktúru** umožňujúcu kombinovať položky rôzneho typu a poskytuje nástroje na jednoduchú manipuláciu s nimi.

Záznam rovnako ako iné objekty možno používať na základe definície alebo deklarácie, častejšie sa však využíva definícia, pretože v programe je jednoduchšie deklarovať premennú daného typu ako vždy nanovo rozpisovať položky záznamu.

Definícia pre náš pôvodný príklad by mala podobu:

```

Type TOsoba = record   alebo   Type TOsoba = record
  meno:string;         meno,priezvisko:string;
  priezvisko:string;   vek, deti:integer;
  vek:integer;         triedny:boolean;
  deti:integer;        end;
  triedny:boolean;
end;

```

Na základe definovaného typu by sa vykonala deklarácia:

```
var clovekl, clovek2:TOsoba;
```

a manipulácia s hodnotami premenných by vyzerala:

priradenie hodnoty	čítanie hodnoty
clovekl.meno:='Jano';	if clovekl.meno='Jano' then...
clovekl.vek:=55;	sucet:=sucet + clovekl.vek;
clovekl.triedny:=true;	

Položka záznamu sa od jeho názvu oddeľuje bodkou.

Ak je záznamov veľa, je lepšie uložiť ich do poľa, namiesto ukladania do samostatných premenných. Pole záznamov deklarujeme známym spôsobom:

```
var ludia:array[1..20] of TOsoba;
```

a zápis štandardných operácií sa tiež nemení – pristupujeme k *i*-tej položke poľa (ktorá predstavuje jeden konkrétny záznam) a v rámci nej k položke záznamu:

priradenie hodnoty	čítanie hodnoty
ludia[5].meno:='Jano';	if ludia[i].meno='Jano' then...
ludia[3].vek:=55;	sucet:= sucet + ludia[i].vek;

Zjednoduší sa i zápis pre usporiadanie údajov – pri rovnakom type triedenia bude postačujúci jednoduchý zápis výmeny:

```

procedure UsporiadajPodlaVeku;
var i,j:integer;
    pom:TOsoba;

begin
  for i:=1 to pocet do // pocet je globalna premenna
    for j:=1 to pocet-1 do
      if ludia[j].vek > ludia[j+1].vek then begin
        pom:=Ludia[j];
        Ludia[j]:=Ludia[j+1];
        Ludia[j+1]:=pom;
      end;
end;

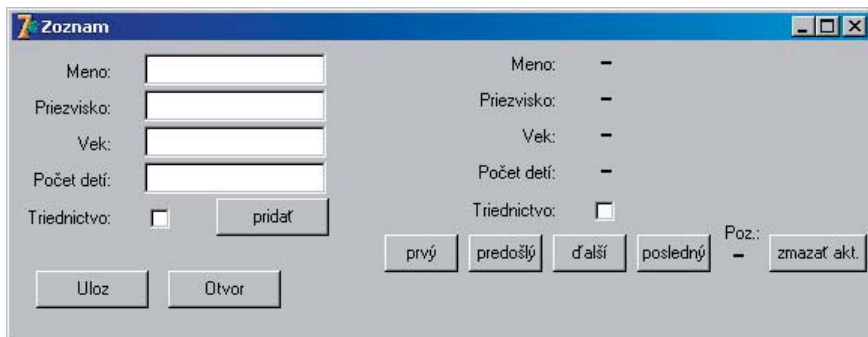
```

Prostredie pre prácu so záznamami

Nevyhnutnou operáciou, ktorú sme doposiaľ taktne zamlčali, je načítanie údajov do záznamov. Údaje možno získať z existujúceho súboru (pozri ďalej) alebo priamo od používateľa (čo bude určite potrebné vykonať minimálne raz).

Vytvorte prostredie, ktoré umožní načítať zoznam osôb do poľa záznamov.

Prepojenie údajov v záznamoch na komponenty v *Delphi* spočíva vo vytvorení formulára, ktorý nám umožní údaje pridávať, listovať v nich a prípadne aktuálny vymazať. Je síce možné údaje do niektorého zo známych komponentov i vypísať, no vhodnejšie bude použiť formulár ako na obrázku.

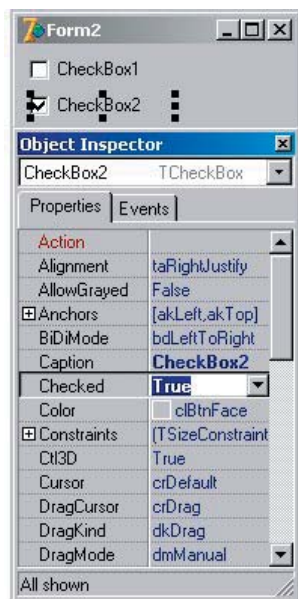


Obr. 75 Formulár pre manipuláciu so záznamami

Začneme s využívaním komponentov `Edit`, do ktorých budeme vkladať nové položky a `Label`, kde budeme aktuálny záznam zobrazovať. Navyše potrebujeme tlačidlo na posun po zozname.

Novinkou je komponent `Checkbox`, ktorý zostáva z textu uloženého vo vlastnosti `Caption` a zaškrťavacieho políčka. Zaškrtnutiu zodpovedá vlastnosť `checked` – ak je políčko zaškrtnuté obsahuje hodnotu `true`, v prípade nezaškrtnutia – `false`. Tieto hodnoty vieme nastavovať, samozrejme, i z programu a zaškrtnutie nám bude v tomto prípade zodpovedať triednictvu daného učiteľa.

So zoznamom budeme manipulovať prostredníctvom udalostných procedúr tlačidiel, preto budeme navyše potrebovať globálne premenné, v ktorých bude uchovaný celkový (zatiaľ naplnený) počet záznamov a aktuálny (zobrazovaný) záznam.



Obr. 76
Checkbox a jeho vlastnosti

Načítať naraz celý zoznam v *Delphi* efektívnym spôsobom nedokážeme, budeme preto potrebovať procedúru, ktorá do existujúceho zoznamu pridá nový záznam.

Zápis postupnosti príkazov, v ktorých sa vyskytuje odkaz na záznam, dokážeme zapísať i v zjednodušenej podobe prostredníctvom kľúčového slova `with`, ktoré v podobe

```
with zaznam do begin
  ...
end;
```

umožňuje vynechať názov záznamu pri odvolávaní sa na jeho položky. Túto možnosť hneď aj využijeme pri prístupe k záznamu v zozname našich učiteľov:

```
procedure TForm1.btnPridatClick(Sender: TObject);
begin
  inc(pocet); {zvysim pocet poloziek a do poslednej vlozim udaj}
  with ludia[pocet] do begin
    meno:=Edit1.Text;
    priezvisko:=Edit2.Text;
    vek:=StrToInt(Edit3.Text);
    deti:=StrToInt(Edit4.Text);
    triedny:=chbTriednictvo.Checked;
  end;
  {nastavime sa na pridany zaznam - bol pridany ako posledny,}
  aktualnaPozicia:=pocet;
  Zobraz(aktualnaPozicia); {a zobrazime ho}
end;
```

Premenná `ludia[i]` je typu `record`, preto ju možno v kombinácii s `with` použiť v tejto podobe. Systém pri spracúvaní každej premennej skúma, či je súčasťou záznamu uvedeného vo `with`. Pokiaľ premennú s daným menom v `recorde` nájde, prístupuje k nej ako k položke `recordu`, ak nie, považuje ju za „čistú“ premennú.

Pre listovanie po zozname, potrebujeme zabezpečiť pohyb po položkách. Tento pozostáva vždy z dvoch krokov:

- z posunu na príslušný záznam priradením hodnoty do premennej `aktualnaPozicia`,
- zo zobrazenia záznamu umiestneného na indexe `aktualnaPozicia`.

Prvý krok sa mení v závislosti od toho, kam sa presúvame, druhý (zobrazenie) je rovnaký pre všetky pohyby. Vzhľadom na tento fakt, môžeme pre zobrazenie na aktuálnej pozícii vytvoriť podprogram, ktorý potom využijeme pre každý posun.

```
procedure TForm1.Zobraz(aktualnaPozicia:integer);
begin
  lblPozicia.Caption:=IntToStr(aktualnaPozicia); {pozicia}
  Label1.Caption:=ludia[aktualnapozicia].meno;
  Label2.Caption:=ludia[aktualnapozicia].priezvisko;
  Label3.Caption:=IntToStr(ludia[aktualnapozicia].vek);
  Label4.Caption:=IntToStr(ludia[aktualnapozicia].pocetDeti);
  CheckBox1.Checked:=ludia[aktualnapozicia].triedny;
end;
```

Procedúry pre jednotlivé „pohybové“ tlačidlá budú vyzerať nasledovne:

```
procedure TForm1.btnPrvyClick(Sender: TObject);
begin
  aktualnaPozicia:=1; // nastavi sa na prvý záznam
  Zobraz(aktualnaPozicia);
end;
//*****
procedure TForm1.btnPoslednyClick(Sender: TObject);
begin
  aktualnaPozicia:=pocet; // nastavi sa na posledný záznam
  Zobraz(aktualnaPozicia);
end;
//*****
procedure TForm1.btnPredoslyClick(Sender: TObject);
begin
  if aktualnaPozicia>1 then dec(aktualnaPozicia);
  Zobraz(aktualnaPozicia);
end;
//*****
procedure TForm1.btnDalsiClick(Sender: TObject);
begin
  if aktualnaPozicia<pocet then inc(aktualnaPozicia);
  Zobraz(aktualnaPozicia);
end;
```

Mazanie záznamu spočíva v znížení počtu záznamov a posune nasledovníkov vymazaného o položku späť, t.j. ak vymažem 3. záznam, všetky s vyšším indexom sa posunú o 1 nazad – 4. prejde na 3. pozíciu, 5. na štvrtú atď. Po

presunoch zabezpečíme zobrazenie údajov, ktoré sa presunuli na miesto práve odstránenej položky.

```
procedure TForm1.btnMazatClick(Sender: TObject);
var i:integer;
begin
  dec(pocet);
  for i:=aktualnaPozicia to pocet do ludia[i]:=ludia[i+1];
  Zobraz(aktualnaPozicia);
end;
```

Častou operáciou pri práci so zoznamom je i vyhľadanie položky na základe zvoleného kritéria - napr. na základe priezviska učiteľa.

Napište funkciu, ktorá vráti záznam učiteľa so zadaným priezviskom.

V jednom z predchádzajúcich podprogramov sme ako identifikátor vracali index, teraz ho upravíme tak, aby výsledkom bol priamo záznam o hľadacom.

```
{v prípade nenájdenia záznamu je potrebné tento fakt oznámiť}
{v boolovskej premennej nasiel bude v prípade nenájdenia false}
function UdajePrePriezvisko(hladany:string):TOsoba;
var i:integer;
    nasiel:boolean;
begin
  nasiel:=false;
  for i:=1 to pocet do
    if hladany=ludia.priezvisko[i] then begin
      UdajePrePriezvisko:=Ludia[i];
      nasiel:=true;
    end;
  if nasiel=false then
    ShowMessage('Nenájdený, hodnota je nepoužiteľná.');
```

Volanie z programu bude vyžadovať premennú typu `TOsoba`, do ktorej sa výsledok vráti:

```
var clovek:TOsoba;
....
clovek:= UdajePrePriezvisko('Mrkva');
...
```

Upravte funkciu pre hľadanie tak, aby ako funkčnú hodnotu vracala `true` v prípade nájdania a `false` v prípade nenájdania človeka s daným priezviskom. V prípade nájdania požadovaného záznamu vráťte údaje o hľadanom učiteľovi prostredníctvom premennej volanej adresou.

Vkladanie údajov do programu po každom jeho spustení používateľa (i programátora) veľmi rýchlo omrzí, preto je vhodné uložiť údaje do súboru a neskôr ich odtiaľ načítavať.

Výhodou definovaného záznamu je, že po drobných úpravách dovoľí vytvoriť súbor typu `TOsoba`. Problém spočíva v neurčitej veľkosti premenných `meno` a `priezvisko` (typ `string`). Podobne, ako keď sme sa pokúšali o definovanie súboru s údajmi typu `string`, potrebujeme aj v tomto prípade zabezpečiť, aby mali premenné definovanú jednoznačnú veľkosť:

```
Type TOsoba = record
  meno, priezvisko: string[20];
  vek, deti: integer;
  tried: boolean;
end;

var subor: file of TOsoba;
```

Zložitejšie štruktúrované typy

Údajové typy, s ktorými sme sa doposiaľ stretli, môžeme rozdeliť na **jednoduché** a **štruktúrované**. Jednoduché sú tie, ktoré umožňujú uchovávať jedinou hodnotu (`integer`, `char`, `real`, `boolean`).

Štruktúrované typy predstavujú štruktúru – zjednotenie viacerých premenných. Podľa toho, či sú v štruktúre všetky prvky rovnakého typu alebo to nie je podmienkou, môžeme ich ďalej deliť na **homogénne** (`string`, `array`) a **heterogénne** (`record`).

Ako ste už určite v predchádzajúcej kapitole postrehli, prvkami údajovej štruktúry môžu byť údaje jednoduchého aj štruktúrovaného typu (stretli sme sa s poľom záznamov).

Záznam v zázname

Navrhňte štruktúru na evidovanie triedy, ktorá bude pozostávať z údajov o žiakoch, učiteľovi a nesmie chýbať názov triedy.

Na prvý pohľad vidíme, že osobné údaje bude potrebné evidovať ako pre žiakov, tak i pre učiteľa. Vytvoríme pre ne typ `TOsoba`:

```
Type TOsoba = record
  meno, priezvisko: string;
  vek: integer;
end;
```

Na základe požiadaviek vytvoríme pre triedu typ, ktorý bude obsahovať údaje o učiteľovi a zoznam žiakov:

```
Type TTrieda = record
  NazovTriedy: string;
  Ucitel: TOsoba;
  Ziaci: array[1..30] of TOsoba;
end;
```

Ak použijeme deklaráciu v podobe:

```
var MojaTrieda: TTrieda;
    Skola: array[1..20] of TTrieda;
```

Bude prístup napr. k učiteľovmu veku:

```
MojaTrieda.ucitel.vek
Skola[3].ucitel.vek
```

k priezvisku 8. žiaka:

```
MojaTrieda.ziaci[8].priezvisko
Skola[3].ziaci[8].priezvisko
```

1. Navrhňte štruktúru, pomocou ktorej by ste boli schopní (čo najprehľadnejšie) evidovať údaje o prospechu žiakov počas štyroch rokov štúdia na škole.
2. Navrhňte štruktúru schopnú evidovať zoznam oddelení veľkej firmy, ktoré bude obsahovať informácie o názve oddelenia, vedúcom a zamestnancoch a ich pracovných zaradeniach.
3. Vytvorte štruktúru zamestnanec (`meno`, `priezvisko`, `plat`) a vytvorte súbor `zoznam.dat`, do ktorého uložíte údaje v podobe záznamov. Napíšte podprogram, ktorý vytvorí nový súbor so zadaným názvom a uloží doň zamestnancov s platom väčším ako 20 000 Sk.
4. Napíšte procedúru na uloženie a na načítanie údajov. Ukladať môžete do súboru s pevne zadaným názvom. Pri načítavaní myslíte na to, že počet

uložených záznamov vopred nepoznáte. Pridajte podprogram, ktorý napokon zoznam evidovaných údajov vypíše (napr. do Listboxu).

5. Navrhните record, v ktorom budete evidovať knihy, a napíšte procedúry, ktoré budú schopné nájsť, pridávať, mazať a vyhľadávať a zisťovať cenu pre rovnaké knihy.
6. Navrhните štruktúru, ktorá dokáže reprezentovať klasické bludisko: možno sa pohybovať štyrmi smermi, medzi susednými miestnosťami môže alebo nemusí byť priechod.
7. Napíšte program, v ktorom bude definovaný typ zajazd s položkami: krajina, miesto, cena, dĺžka pobytu, mesiac odchodu. Deklarujte a naplňte pole zajazdy a vypíšte:
 - a) všetky zajazdy s dĺžkou pobytu aspoň 7 dní,
 - b) všetky zajazdy do Ruska alebo Poľska,
 - c) priemernú cenu zajazdov,
 - d) počet zajazdov do Grécka,
 - e) všetky augustové zajazdy.
8. Napíšte program, ktorý dokáže evidovať žoldnierov na pirátskej lodi, dokáže vypočítať ich plat podľa počtu akcií, ktorých sa zúčastnili, a v prípade ich smrti rozdelí majetok rovným dielom medzi ostatných.

11 Tabuľka - matica

predpoklady na zvládnutie lekcie:

- schopnosť práce s údajovým typom pole a záznam
- schopnosť práce so súbormi

obsah lekcie:

- pojem matica a jej použitie v prostredí Delphi
- vizuálna reprezentácia tabuľky - StringGrid
- ukladanie a načítanie údajov tabuľky

cieľ:

- využívanie vizuálneho komponentu StringGrid pre prácu s tabuľkovými údajmi
- zvládnutie práce s maticami vo všeobecnosti

Pole polí

Veľmi často sa pri programovaní môžeme stretnúť s poľom polí – maticou. Z používateľského pohľadu ju poznáme skôr pod pojmom tabuľka. Deklarácia i definícia sú si navlas podobné a môžeme ich zapísať v tvare:

```
Type TMatica=array[1..20] of array[1..20] of integer;
```

alebo jednoduchšie

```
Type TMatica=array[1..20,1..20] of integer;
```

V matici sú všetky prvky rovnakého typu (jednoduchého (integer, char a pod.) alebo pokojne i štruktúrovaného – array, record a pod.).

Dvozmerné polia (matice) sa často používajú na popis dvozmerných objektov (mapy, tabuľky a pod.), no často sa môžeme stretnúť aj s troj- a viac-rozmernými poliami (priestorové, časopriestorové zobrazenia a pod.).

Časté je využívanie matíc pri riešení matematických úloh alebo v teórii grafov.

1. Napíšte program, ktorý do dvozmerného poľa rozmeru $N*N$ zapíše čísla

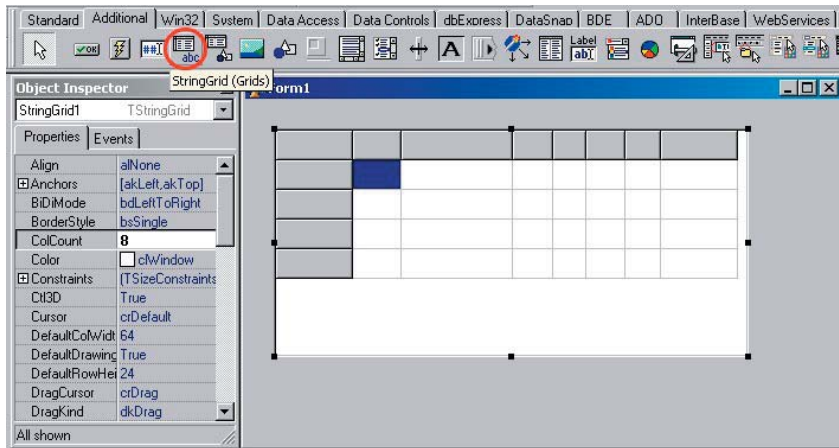
$1..N*N$. Čísla nech sú zapísané postupne v sprava doľava napr.

1	2	3
4	5	6
7	8	9

2. Napište program, ktorý načíta maticu s rozmermi $n \times n$ a vypíše prvky hlavnej i vedľajšej diagonály.
3. Napište program, ktorý načíta maticu čísel $n \times n$, preklopí ju okolo hlavnej diagonály, vypíše pôvodnú aj preklopenú maticu.

Vizuálna reprezentácia tabuľky

Delphi obsahuje niekoľko komponentov, ktoré možno využiť na zobrazenie údajov v podobe tabuľky. Najjednoduchším a zrejme spočiatku pre nás i najpoužívanejším bude `StringGrid`.



Obr. 77 StringGrid a jeho vlastnosti

Ide o tabuľku (maticu) pozostávajúcu z buniek, ktoré môžu obsahovať textové hodnoty. Podľa nastavení môže obsahovať záhlavie v hornej alebo ľavej časti tabuľky. Počet riadkov (stĺpcov) záhlavia určuje hodnota nastavená vo vlastnostiach `FixedCols` a `FixedRows`.

`StringGrid` disponuje relatívne neobmedzeným počtom stĺpcov i riadkov, ktoré možno pridávať i uberať ako v návrhovom zobrazení, tak i za behu programu prostredníctvom zmeny vlastnosti `ColCount` a `RowCount`.

Zvláštnosťou `StringGridu` je, že má prednastavený zákaz úpravy textu, a tým pádom nemožno doň písať. Túto vlastnosť však veľmi jednoducho zmeníme nastavením vlastnosti `Options-goEditing` na `True`.

Zrejme najdôležitejšou je pre nás schopnosť prístupit' ku konkrétnej bunke `StringGridu`. Zabezpečuje ju vlastnosť `Cells`, ktorej udávame stĺpec a riadok, ku ktorému chceme prístupit'. Prvý riadok i stĺpec majú index 0.

Do bunky [2,3] (prvý parameter udáva číslo stĺpca, druhý číslo riadku) tabuľky s názvom `StringGrid1` vložíme hodnotu priradením

```
StringGrid1.Cells[2,3]:='ahoj';
```

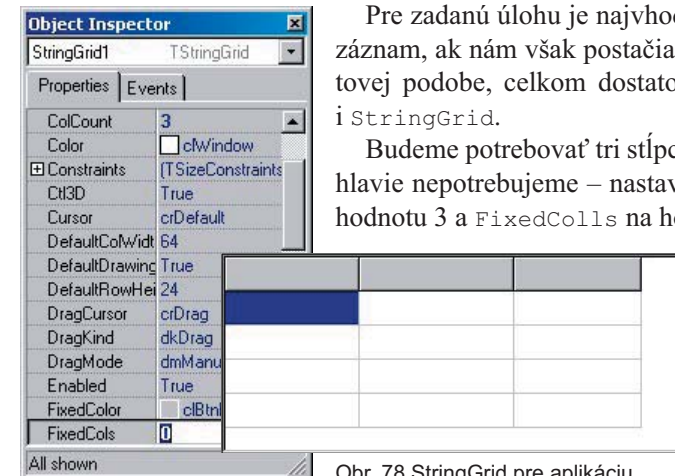
a rovnakým spôsobom z nej i čítame:

```
text:=StringGrid1.Cells[2,3];
```

Text do gridu dokážeme manuálne vkladať len počas behu programu, ak si teda chceme pripraviť pre tabuľku hlavičku, musíme tak urobiť v niektorej z úvodných udalostí formulára (napr. `OnShow` formulára). Pri tej istej udalosti môžeme nastaviť aj východziu šírku stĺpcov:

```
StringGrid1.Cells[0,0]:='Meno';
StringGrid1.ColWidths[0]:=50;
StringGrid1.Cells[1,0]:='Priezvisko';
StringGrid1.ColWidths[1]:=100;
StringGrid1.Cells[2,0]:='Rok nar.';
StringGrid1.ColWidths[2]:=40;
```

Vytvorte aplikáciu, ktorá bude evidovať údaje o žiakoch: meno, priezvisko a rok narodenia. Umožnite pridávanie, mazanie a vyhľadávanie údajov.



Pre zadanú úlohu je najvhodnejšou štruktúrou záznam, ak nám však postačia údaje v čistej textovej podobe, celkom dostatočným riešením je i `StringGrid`.

Budeme potrebovať tri stĺpce, pričom ľavé záhlavie nepotrebujeme – nastavíme `ColCount` na hodnotu 3 a `FixedCols` na hodnotu 0.

Obr. 78 StringGrid pre aplikáciu

Pridanie riadku na koniec `StringGridu` zabezpečíme zvýšením hodnoty `RowCount`:

```
StringGrid1.RowCount:= StringGrid1.RowCount+1
```

Veľmi často však potrebujeme pridať riadok nie na koniec, ale napr. nad pozíciu, na ktorej sa v tabuľke nachádza kurzor.

Pri zisťovaní pozície sa môžeme stretnúť s dvoma možnosťami:

- v prípade zapnutej vlastnosti `Options.RangeSelect` môžeme v `StringGrid1` označovať blok – obdĺžnik. Údaje o označení poskytujú:

<code>StringGrid1.Selection.Left</code>	– najľavejší stĺpec označenia
<code>StringGrid1.Selection.Right</code>	– najpravejší stĺpec označenia
<code>StringGrid1.Selection.Top</code>	– najvyšší stĺpec označenia
<code>StringGrid1.Selection.Bottom</code>	– najnižší stĺpec označenia

Riadok budeme vkladať zrejme nad pozíciu `StringGrid1.Selection.Top`.

- pokiaľ je vlastnosť `Options.RangeSelect` vypnutá, označená je vždy len jedna bunka tabuľky a jej pozíciu získame z hodnôt `StringGrid1.Selection.Left` a `StringGrid1.Selection.Top`.

`StringGrid` nedisponuje schopnosťou vloženia riadku na danú pozíciu, potrebujeme preto vložiť riadok na koniec a posunúť nadol všetky údaje od aktuálneho riadku (obsah aktuálneho treba vyprázdniť):

```
procedure TForm1.PridajNadAktualny;
var aktualny,i,j:integer;

begin
  {aktualna pozicia - vyska}
  aktualny:=StringGrid1.Selection.Top;
  {pridanie riadku}
  StringGrid1.RowCount:=StringGrid1.RowCount+1;
  {od posledneho riadku po aktualny sa budu brat hodnoty}
  for i:=StringGrid1.RowCount-1 downto aktualny+1 do begin
    for j:=0 to StringGrid1.ColCount-1 do{pre kazdy stlpec}
      {sa do bunky vlozi obsah bunky nad nou}
      StringGrid1.Cells[j,i]:=StringGrid1.Cells[j,i-1];
    end;
  {aktualny riadok sa vyprazdni}
  for j:=0 to StringGrid1.ColCount-1 do
    StringGrid1.Cells[j,aktualny]:='';
  end;
```

Rovnakým spôsobom možno riešiť i vymazanie riadku:

- zistí sa aktuálna pozícia,
- všetky riadky pod ňou sa posunú o jeden vyššie,

- vyprázdni sa posledný riadok a zníži sa počet riadkov `StringGrid1`.

Vyhľadávanie môžeme obmedziť na jeden stĺpec alebo prehľadávať celú tabuľku (okrem záhlavia), porovnávať môžeme hľadanú hodnotu s celým obsahom bunky alebo zisťovať, či sa nachádza v bunke hľadaný reťazec (prostredníctvom funkcie `Pos`).

Nasledujúca procedúra vypíše riadok a stĺpec, v ktorom sa nachádza hľadaná hodnota:

```
procedure TForm1.Hladaj(retazec:string);
var i,j:integer;

begin
  for i:=1 to StringGrid1.RowCount-1 do {prechod po riadkoch}
    for j:=0 to StringGrid1.ColCount-1 do {prechod po stlpcoch}
      if StringGrid1.Cells[j,i]=hladany then
        ShowMessage('Hodnota je na pozicii'
          +IntToStr(j)+' '+IntToStr(i));
    end;
```

Vzhľadom na nemožnosť vloženia údajov v návrhovom režime, je vhodné už pri vytváraní aplikácie na testovacie účely, údaje tabuľky uložiť do súboru. Univerzálnym riešením je formát `csv` – riadky tabuľky sú v nových riadkoch súboru a údaje zo stĺpcov sú oddelené bodkočiarkou (s týmto formátom dokáže pracovať i tabuľkový kalkulátor).

Názov súboru budeme zadávať štandardným spôsobom - cez `SaveDialog`:

```
procedure TForm1.btnUlozitClick(Sender: TObject);
var nazov,riadok:string;
    i,j:integer;
    ff:TextFile;

begin
  {ak sa odsuhlasi savedialog}
  if SaveDialog1.Execute then begin
    nazov:=SaveDialog1.FileName; {zisti sa nazov}
    {vytvori subor - ak existoval iny s rovnakym nazvom tak ho prepise}
    AssignFile(ff,nazov);
    Rewrite(ff);
```



```

{prechod po riadkoch}
for i:=0 to StringGrid1.RowCount-1 do begin
  riadok:='';{idu sa citat bunky v riadku, na zaciatku nema nic}
  for j:=0 to StringGrid1.ColCount-1 do
    {precita sa udaj z bunky, da sa zan bodkociarka}
    {a pokracuje sa, kym sa nepride na posledny stlpec}
    riadok:=riadok+StringGrid1.Cells[j,i]+' ';
  WriteLn(ff,riadok); {riadok zapise do suboru, ln odriadkuje}
end;
CloseFile(ff); {subor sa zatvori}
end;
end;

```

Otvorenie súboru zabezpečíme rovnakým spôsobom ako uloženie:

```

procedure TForm1.btnOtvoritClick(Sender: TObject);
var i,j,poz:integer;
    riadok:string;
    ff:TextFile;
begin
  if OpenFileDialog1.Execute then begin
    AssignFile(ff,OpenDialog1.FileName); {otvorenie suboru}
    reset(ff);
    j:=0; {aktualny riadok, do ktoreho sa bude zapisovat}
    StringGrid1.RowCount:=1; {pocet riadkov v StringGride}
    while not EOF(ff) do begin {kym nie je koniec suboru}
      readln(ff,riadok); {precita sa riadok}
      i:=0; {i predstavuje aktualny stlpec}
      {postupuje sa po texte, ak sa v nom nachadza bodkociarka,}
      {text pred nou sa vlozi do aktualnej bunky, postup skonci}
      {ak v texte ; nie je}
      while length(riadok)>0 do begin
        poz:=Pos(,;',riadok); {vrati poziciu bodkociarky}
        {do i-teho stlpca a j-teho riadku sa vlozi text od zaciatku
        textu v premennej riadok po bodkociarku}
        StringGrid1.Cells[i,j]:=Copy(riadok,1,poz-1);
        Delete(riadok,1,poz); {vlozeny text sa z riadok vymaze}
        inc(i); {posun na dalsi stlpec}
      end;
      inc(j); {posun na dalsi riadok}
      {navysi riadky}
      StringGrid1.RowCount:= StringGrid1.RowCount+1;
    end;
  end;
end;

```

1. Napíšte procedúru, ktorá usporiada údaje v tabuľke podľa zadaného stĺpca.
2. Je daná tabuľka a v jej bunkách náhodne umiestnené znaky „x“ a „o“ (v jednej bunke môže byť maximálne jeden znak). Zistite, ktorých je viac. Umožnite ukladanie a otváranie takýchto tabuliek.
3. Vytvorte aplikáciu na evidovanie zamestnancov. Evidujte meno, vek, funkciu a počet detí. Údaje evidujte v stringgride a umožnite ukladanie a otváranie súborov. Pridajte funkciu, ktorá zvýši vek o jedna a v prípade, že vek nadobudne hodnotu väčšiu ako 60, zamestnanca zo zoznamu automaticky vyhodí.
4. Vytvorte aplikáciu, ktorá dokáže sčítať dve matice do tretej. Rozmery výslednej matice nastavte podľa potreby.
5. Naprogramujte Puzzle v nasledovnej verzii: na uzavretej ploche je 4 x 4 miest, pričom 15 z nich obsahuje čísla 1-15, samozrejme, pomiešané. Úlohou hráča je posúvaním zoradiť čísla tak, aby išli za sebou od 1 až po 15 s tým, že posledné pole zostane prázdne. Ovládanie možno realizovať šípkami, tlačidlami alebo zadávaním príkazov (h (hore), d (dole), p (doprava), l (doľava)).

Záverečný test

1. Premenná X má hodnotu 4 a Y má hodnotu 5. Koľkokrát sa vykoná uvedený cyklus?

```
while X<=5 do begin
    X:=X+1;
    Y:=Y-1;
end;
```

Správna odpoveď: 2

2. Premenná A predstavuje údajovú štruktúru zobrazenú na obrázku nižšie. Ako by sme mohli takúto premennú deklarovať?

1	21	8
10	3	7
0	50	11

- a. var A: array [,a`.. ,c`, 1..3] of byte;
- b. var A: array [0..3,0..3] of char;
- c. var A: array [1..9] of integer;

Správna odpoveď: a

3. Premenná S je typu string. Akú hodnotu bude mať premenná S po vykonaní cyklu for?

```
S:= ,VIANOCE`;
For i:=1 to (Length(S) div 2) do S[i]:=S[Length(S)-i+1];
```

- a. VIANOCE
- b. ECONOCE
- c. ECONAIV

Správna odpoveď: ECONOCE

4. Pre aké N sa telo cyklu

```
for i:=3*N-5 to 4*N+2 do
```

vykoná práve 10-krát?

Správna odpoveď: 2

5. V programe používame funkciu Fun s takouto hlavičkou.

```
Function Fun(x:real):real;
```

Ktorý z uvedených zápisov nie je možné použiť?

- a. Showmessage (FloatToStr(fun(10.5)));
- b. If fun(a) > 0 then {postupnosť príkazov};
- c. Fun(a) :=8;

Správna odpoveď: c

6. Máme deklarovanú procedúru, ktorej hlavička je uvedená nižšie. Ktoré z parametrov sú volané adresou?

```
Procedure Waw(x,y: integer; var z:integer; ok:boolean)
```

- a. x, y
- b. z
- c. z, ok

Správna odpoveď: b

7. Premenné A, B sú typu string a obe majú hodnotu '10'. Aké hodnoty budú mať po vykonaní uvedených príkazov priradenia

```
A:= A + B;
B:= B + A;
```

- a. A=1010, B=101010
- b. A=1010, B=1010
- c. A=20, B=30

Správna odpoveď: a

8. Čo sme dosiahli nasledujúcim zápisom?

```
Type RIADOK = array [ 0..4 ] of char;
```

- Deklarovali sme konštantu RIADOK
- Deklarovali sme premennú RIADOK ako 5 prvkové pole prvkov typu char
- Definovali sme údajový typ RIADOK

Správna odpoveď: c

9. Pole A sme deklarovali nasledovne:

```
var X: array [1..4,1..4] of integer;
```

Akú hodnotu bude mať prvok X[4, 4] po vykonaní časti programu?

```
for i:= 1 to 4 do begin
  X[i,1]:= i + 1;
  for j:=2 to 4 do X[i,j]:= X[i,j-1]+1;
end;
```

Správna odpoveď: 8

10. V procedúre Cinnost zavoláme procedúru Akcia. Aké hodnoty budú mať premenné X, Y a Z na konci procedúry Cinnost?

```
Procedure Akcia(a,b:integer; var c:integer);
var pom:integer;
begin
  pom:=a;
  a:=b;
  b:=pom;
  c:=a*b;
end;

Procedure Cinnost;
var x,y,z:integer;
Begin
  x:=1;
  y:=2;
  z:=0;
  Akcia(x,y,z);
End;
```

Správna odpoveď: 1,2,2

Použitá literatúra:

- ČIČALA, D.: Programovanie v Delphi (alternatívna učebnica), AM, 1999 (materiál na webe: <http://am-skalka.sk/index.php?page=vydavatelstvo&product=528>)
- DRLÍK, P.: Turbo pascal I. AM Nitra 1998.
- HVORECKÝ, J. - KELEMEN, J.: Algoritmizácia. Alfa, Bratislava 1983, 1987.
- KAPUSTA, J.: Programové vyučovanie programovania. In V. vedecká konferencia doktorandov a mladých vedeckých pracovníkov : zborník z medzinárodnej konferencie. Nitra : FPV UKF, edícia Prírodovedec č. 126, 2004, s. 306-309. ISBN 80-8050-670-1.
- SKALKKA, J.: Turbo pascal II. AM Nitra 1997.
- STOFFOVÁ, V.: IKT vo vyučovaní programovania. In Žilinská didaktická konferencia 2005 (Zborník príspevkov). Žilina : ŽU v Žiline, 2005, s. 30. ISBN 80-8070-430-9.
- STOFFOVÁ, V.: Zbierka úloh z algoritmizácie a programovania I, 1. vyd. VŠPg v Nitre, Nitra 1993
- WIRTH, N.: Algoritmy a štruktúry údajov. Alfa, Bratislava 1988.

Autor: Skalka Ján
Cápay Martin
Lovászová Gabriela
Mesárošová Miroslava
Palmárová Viera

Názov diela: Algoritmizácia a úvod do programovania

Vydavateľ: UKF v Nitre
Edícia: Prírodovedec č. 276
Autor obálky: PaedDr. Katarína Zverková
Rok vydania: 2007
Poradie vydania: prvé
Počet strán titulu: 158
Počet výtlačkov: 100
Tlač: Vydavateľstvo Michala Vaška, Prešov

© UKF v Nitre 2007

ISBN 978-80-8094-217-5

EAN 978-80-8094-217-5