1. prednáška: úvod do prostredia, jednoduchá grafika

čo sa na tejto prednáške naučíme:

- čo je to Delphi - prečo sa ho ideme učiť
- ako vyzerá programátorské prostredie
- ako začneme robiť prvý program - aplikáciu, ako ju spúšťame a zastavujeme
- tlačidlá: tvar, umiestnenie a popis
- priradenie akcií ku tlačidlám
- grafická plocha (Image): umiestnenie, grafické príkazy - pero, farby, obdĺžnik
- celočíselné premenné, deklarovanie premennej
- for-cyklus, premenná cyklu, Integer, aritmetické výrazy
- zhrnutie grafických príkazov
- nastavenie prostredia Delphi, súbory, prípony

Čo je to Delphi - prečo sa ho ideme učiť

- Delphi je pomenovanie programátorského prostredia, v ktorom sa programuje v jazyku Objektový Pascal
- samotný jazyk Pascal je komunitou informatikov na celom svete považovaný za jazyk
    - vytvorený na to aby sa v ňom dalo učiť programovanie, princípy tvorby algoritmov, algoritmické rozmýšľanie, programátorská disciplína a pod.
    - v ktorom sa zapisujú algoritmy - tieto sú potom zrozumiteľné pre profesionálov bez ohľadu na to, v akom jazyku programujú
    - je odporúčaný ako jeden z prvých jazykov programovania napr. na ZŠ a SŠ (na rozdiel od C)
    - jazyk je natoľko jednoduchý a jednoznačný, že neskorší prechod na ľubovoľný iný programovací jazyk je veľmi príjemný
- Pascal podobne ako C vznikol začiatkom 70-tych rokov - oba boli postavené na základe štruktúrovaných konštrukcií jazyka Algol a oba sa inšpirovali pravdepodobne vtedajšími jazykmi ako Fortran a PL/1
    - hlavnou prioritou Pascalu bola akademická pôda, na rozdiel od C, ktoré vzniklo pre systémových programátorov - teda už veľmi skúsených programátorov
- postupom času (v polovici 80-tych rokov) sa z Pascalu vyvinul moderný Objektový Pascal a C sa zmodernizovalo na objektové C++
- tiež aj v súčasnosti vzniká množstvo nových programovacích jazykov, ktoré sa veľmi silne inšpirujú Pascalom aj jazykom C, resp. C++, takže v budúcej vašej praxi možno budete pracovať s jazykmi, ktoré buď zatiaľ ešte neexistujú, alebo sú u nás zatiaľ málo známe - ale s najväčšou pravdepodobnosťou sa vám programátorský štýl, ktorý získate v týchto úvodných programátorských predmetoch veľmi zíde...
- vo vyšších ročníkoch sa zoznámite aj s inými jazykmi, pričom práve Pascal vám umožní veľmi prirodzený prechod (napr. v 3. semestri táto prednáška pokračuje v jazyku Java, v 2. semestri je to úvod do PHP, v ďalších semestroch sa budete môcť zoznámiť s C++, SmallTalk ale aj deklaratívnym programovaním)

Ako vyzerá programátorské prostredie

Programátorské prostredie (IDE - integrated development environment) označuje, že programátor v jednom balíku môže

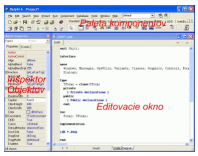
- programy navrhovať, písať, upravovať, vyvíjať
- kompilovať

- testovať, ladiť
- lokalizovať
- ...

Súčasné moderné programátorské prostredia, ktoré umožňujú vyvíjať nielen aplikácie pre grafické rozhranie (napr. Windows), ale aj aplikácie pre web - sú založené na vizuálnom princípe: všetko, čo bude mať v bežiacej aplikácii vizuálne znázornenie, sa už počas návrhu bude dať vizuálne poskladať z nejakých predpripravených častí. Programátor potom veľmi často "iba" doprogramováva správanie týchto komponentov v rôznych situáciách a grafická nadstavba mu zabezpečí celkové správne fungovanie.

Podobne funguje aj Delphi. Jeho celé prostredie sa skladá z viacerých častí:

- panel ovládacích tlačidiel: napr. načítaj, zapíš, skompiluj a spusti, a pod.
- editovacie okno, v ktorom v pascale popisujeme správanie programu v rôznych situáciách
- formulár: vizualizácia nášho budúceho programu, t.j. okno - tu budeme vkladať a upravovať rôzne, väčšinou vizuálne komponenty
- paleta komponentov: ponuka predpripravených "súčiastok", ktoré môžeme vkladať do nášho okna, napr. tlačidlá, grafické a textové plochy, editovacie okienka, posúvače a pod.
- objektový inšpektor: špeciálne okienko, v ktorom môžeme upravovať parametre komponentov vo formulári
- prostredie obsahuje ešte aj ďalšie časti - s niektorými sa zoznámime neskôr



Ako začneme robiť prvý program - aplikáciu, ako ju spúšťame a zastavujeme

Skôr ako do detailov pochopíme princípy tvorby nového programu (aplikácie, projektu) v Delphi, bude dobre si zautomatizovať nejaký jednoduchý postup, ktorý nám na začiatku pomôže vyvarovať sa niektorých začiatočníckych chýb. Takže poďme na našu prvú aplikáciu:

1. naštartujeme **Delphi**
2. v menu **File** zvolíme **New** a potom **Application** (alebo v ovládacom paneli tlačidlo **New** a potom **Application**)
3. Delphi je teraz už pripravené na to, aby sme mohli začať tvoriť náš nový program - editovacie okno obsahuje len zopár základných predpísaných konštrukcií a formulár je prázdny
4. v tomto momente (práve pred samotným začiatkom tvorby programu) je dobre si zvyknúť takýto zatiaľ "prázdny" program uložiť - samozrejme, že to môžme aj neskôr, ale začiatočník pri tom môže robiť množstvo chýb - takže, z ponuky **File** vyberieme **Save Project As...** (sú tam 4 rôzne **Save**, tento je teraz pre nás najlepší):
    - zvolíme priečinok, do ktorého budeme projekt ukladať - niektorí programátori preferujú ukladať každý projekt do nového priečinku - záleží od vás ako si to budete organizovať na disku
    - každý projekt, kým ho ešte programujeme, sa na disku skladá z minimálne 5 súborov a tieto by mali byť spolu v jednom priečinku a nemali by sa prekrývať s inými projektmi
    - pri ukladaní projektu sa Delphi pýtajú na uloženie dvoch súborov: **Unit1.pas** - to je samotný program aj s formulárom a **Project1.dpr** - to je projektový súbor, ktorý sa vytvára automaticky a obsahuje dôležité informácie o celkovej štruktúre nášho projektu
    - tieto dva súbory môžete premenovávať (zatiaľ to neodporúčame), ale dôležité je aby boli spolu v rovnakom priečinku

5. teraz už predpokladáme, že máme projekt úspešne uložený a môžeme ho spustiť:
    - hoci sme ešte nič neprogramovali, Delphi nám s novou aplikáciou všetko pripravia tak, aby sme už mali funkčné jedno windows okno
    - stlačíme kláves **F9** (alebo tlačidlo na ovládacom paneli so zeleným trojuholníkom) a ak je program bez chýb, tak sa spustí: objavilo sa prázdne šedé okno s titulovým modrým pásom, s textom **Form1** a s malými systémovými tlačidlami na minimalizovanie, maximalizovanie a ukončenie aplikácie
    - spustený program môžeme na pracovnej ploche windows posúvať, meniť mu veľkosť, prípadne sa prepnúť do inej bežiacej aplikácie, napr. prostredie Delphi (to že naša aplikácia práve beží, vidíme v Delphi, napr. tak, že tlačidlo so zeleným trojuholníkom je zablokované - zašedené)
    - bežiaci program ukončíme buď tlačidlom **Close** (v pravom hornom rohu) alebo klávesmi Alt+F4
6. po zastavení nášho bežiaceho prvého programu, ho môžeme teraz začať naozaj programovať - ale budeme na to potrebovať nejaké komponenty - začneme komponentom tlačidlo

Tlačidlá: tvar, umiestnenie a popis

Ukladanie komponentov do formulára je veľmi jednoduché:

- najprv treba mať v Delphi okno s formulárom vpredu - buď môže byť skryté, alebo prekryté editovacím oknom - stlačíme kláves **F12**, aby sa okno dostalo navrch
- z palety komponentov klikneme na komponent, ktorý budeme chcieť položiť do formulára - keďže paleta má viac záložiek s menami, napr. **Standard**, **Additional**, **System** a pod. - niekedy musíme najprv zvoliť správnu skupinu a až potom kliknúť na komponent
- takže začíname komponentom tlačidlo - jeho názov je **Button** a nachádza sa v štandardnej záložke palety - v palete má tvar malého tlačidla s textom **OK** - kliknite na tento komponent
- kliknite niekde do formuláru - objaví sa tu tlačidlo s popisom **Button1**
    - toto tlačidlo môžete presunúť na ľubovoľnú pozíciu, prípadne mu zmeniť veľkosť ťahaním za čierne štvorčeky
    - veľmi jednoducho zmeníme aj popis na tlačidle: všimnite si, že keď je naše nové tlačidlo označené (má okolo seba čierne štvorčeky), v objektovom inšpektore (**Object Inspector**) sú nejaké informácie, ktoré sa týkajú práve tohto tlačidla. Inšpektor sa skladá z dvoch stĺpcov: ľavý - šedý obsahuje meno nejakého nastavenia (napr. **Caption** označuje popis na tlačidle) a v príslušnom pravom je momentálna hodnota (teraz je tam **Button1**) - informácie v pravom stĺpci môžeme meniť a tým sa bude meniť vzhľad tlačidla a niekedy aj jeho správanie
    - zmeňte v Inšpektore nastavenie **Caption** na text, napr. **Tlačidlo** - zároveň sa zmenil popis na tlačidle
- vytvorte na formulári viac tlačidiel rôznych rozmerov, zmeňte im popisy a vyskúšajte, ako sa správajú, keď takúto aplikáciu spustíme (**F9**)
- ak potrebujete nejaké tlačidlo z formuláru odstrániť, stačí aby bolo označené (kliknite naň) a stlačte kláves **Del**

Priradenie akcií ku tlačidlám

Program s tlačidlami je už zaujímavejší, ako čisté okno, ale tlačidlá zatiaľ nerobia žiadne akcie. Každé tlačidlo môže mať priradenú nejakú akciu, ktorá sa spustí, keď naň počas behu programu klikneme - spustí sa zakaždým,

keď naň klikneme. Priradenie akcie sa robí nasledovne:

- na dané tlačidlo vo formulári dvojklikneme (nech to bolo tlačidlo Button1)
- objaví sa editovacie okno, v ktorom sú automaticky pripísané tieto riadky:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

- pre nás je dôležité si všimnúť slovo Button1Click - toto označuje, že to čo naprogramujeme medzi slová begin a end sa spustí vždy, keď sa klikne na tlačidlo Button1

Pre otestovanie tlačidiel sa naučíme jednoduchý príkaz na zmenu popisu na tlačidle:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Caption := 'zmena';
end;
```

Program spustíme (F9) a keď naňho teraz klikneme myšou, jeho popis sa zmení na slovo **zmena**. Ak máme v programe viac tlačidiel, každému môžeme naprogramovať inú akciu - dvojkliknutím sa predpripraví časť programu, do ktorej dopíšeme nové akcie. Ďalším jednoduchým príkazom je volanie štandardnej procedúry Close - spôsobí zatvorenie našej aplikácie a teda koniec programu, napr.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

POZOR! ak chceme akciu na tlačidle zrušiť, nikdy nezmažeme celú konštrukciu, ale len príkazy medzi begin a **end** - najbližšie uloženie projektu (napr. **Ctrl+S**), takéto zbytočné konštrukcie korektne odstráni.

Kompletný Unit1.pas teraz vyzerá takto:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Caption := 'zmena';
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

a po spustení:

Grafická plocha (Image): umiestnenie, grafické príkazy - pero, farby, obdĺžnik

Pre nás bude asi najdôležitejším komponentom "grafická plocha" - je to taký obdĺžnik vo formulári, do ktorého môžeme rôznymi nástrojmi kresliť, môžme ju jednoducho uložiť na disk, resp. zo súboru do nej prečítať nejaký obrázok. Naučíme sa tu robiť aj animácie, vizualizovať rôzne algoritmy a samozrejme naučíme túto plochu reagovať na klikania a ťahania myšou.

Začneme položením komponentu **Image** do formulára (tlačidlá z predchádzajúcich pokusov tu môžete nechať). Nájdeme ho v palete komponentov v záložke Additional (malý farebný obrázok). Komponent položme do plochy kliknutím (dostáva rozmer približne 100x100) a môžeme ho zväčšiť ťahaním za čierne štvorčeky, rovnako ako tlačidlá. Image ale môžeme prispôsobiť veľkosti formulára aj inak - v inšpektore tomuto komponentu zmeníme jeho nastavenie Align na hodnotu alClient - toto nastavenie označuje, že grafická plocha pokryje kompletný formulár a bude sa meniť, ak budeme meniť veľkosť formuláru. Všimnite si, že tlačidlá sú vždy nad grafickou plochou - môžme ich posunúť niekde ku okraju, aby nám nezavadzali pri kreslení.

Grafickú plochu teda už máme. Kresliť do nej budeme najčastejšie ako akcie pri zatlačení nejakých tlačidiel - teda napr. do procedúry Button1Click budeme zapisovať grafické príkazy medzi begin a end. Skôr ako začneme kresliť, vysvetlime si základné pojmy:

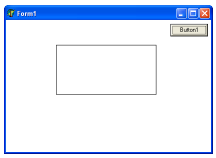
- naša grafická plocha má meno Image1 (podobne ako tlačidlo Button1 a aj formulár Form1)
- každá grafická plocha má svoje plátno a kreslíme práve do tohto plátna (podobné plátno, ako je v obrazoch) - plátnu sa povie Canvas,
- na plátno kreslíme pomocou pera - Pen alebo vyfarbujeme nejaké plochy pomocou štetca Brush,
- pero aj štetec majú svoje farby - Color, pero má aj svoju hrúbku - Width,
- príkazy, ktoré pohybujú a kreslia perom väčšinou vyžadujú zadávanie súradníc: súradná sústava je ale vo Windows trochu inak natočená, ako ju poznáme z matematiky - ľavý horný roh má súradnice (0,0) - teda je to počiatok; x-ová súradnica ide zľava doprava na hornom okraji plochy a y-ová súradnica ide zhora nadol pri ľavom okraji plochy; niekedy majú zmysel aj záporné súradnice, ale tieto popisujú body mimo plochu.

Prvý najjednoduchší príkaz je asi obdĺžnik - zadáme mu dva protiľahlé vrcholy nejakého obdĺžnika a ten ho nakreslí, pričom strany sú rovnobežné s osami. Vysvetlime si príkaz na nakreslenie obdĺžnika:

```
Image1.Canvas.Rectangle(100, 50, 300, 150);
```

Vidíme, že príkaz sa skladá z viacerých slov oddelených bodkami - poradie týchto slov a samozrejme aj presný

zápis je veľmi dôležitý a vyjadruje toto: ideme pracovať s grafickou plochou Image1 - budeme kresliť na jej plátno Canvas - konkrétne použijeme nástroj obdĺžnik - Rectangle. Štyri čísla v zátvorkách vyjadrujú súradnice nejakých dvoch protiľahlých vrcholov - prvé dve sú x-ová a y-ová súradnice prvého vrcholu a druhé dve opäť x-ová a y-ová súradnice druhého vrcholu. Treba si zapamätať, že tieto súradnice musia byť vždy celé čísla - ak je niektorá z nich mimo veľkosť grafickej plochy, tak zrejme aj nejaká časť obdĺžnika bude mimo plochy. Po spustení programu (F9) sa vo formulári sa objaví

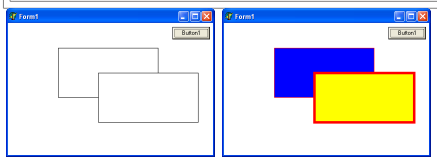


Program ukončime, aby sme mohli sledovať ďalšie pokusy. Teraz nakreslime ešte jeden obdĺžnik a to tak, aby sa oba navzájom prekrývali, napr.

```
begin
  Image1.Canvas.Rectangle(100, 50, 300, 150);
  Image1.Canvas.Rectangle(180, 100, 380, 200);
end;
```

Po spustení vidíme, že druhý obdĺžnik je vyplnený bielou farbou, lebo cez neho nie je vidieť ten prvý - povedzme si teda, ako je to s farbami: pri štarte programu má pero vždy čiernu farbu a hrúbku 1, štetec, ktorý vypĺňa vnútro obdĺžnika má pri štarte bielu farbu. Toto všetko sa dá zmeniť špeciálnymi priraďovacími príkazmi (už sme ho použili pri zmene popisu tlačidla Button1.Caption := 'xxx';) - najprv ich ukážeme a potom vysvetlíme:

```
begin
  Image1.Canvas.Pen.Color := clRed;
  Image1.Canvas.Brush.Color := clBlue;
  Image1.Canvas.Rectangle(100, 50, 300, 150);
  Image1.Canvas.Pen.Width := 5;
  Image1.Canvas.Brush.Color := clYellow;
  Image1.Canvas.Rectangle(180, 100, 380, 200);
end;
```



Vidíme, že prvý obdĺžnik má tenký červený obvod a je vyplnený modrou farbou, druhý obdĺžnik má hrubý červený obvod a je vyfarbený žltou farbou. Farby a hrúbky meníme priraďovacími príkazmi, napr.

```
Image1.Canvas.Pen.Color := clRed;
```

Znamená, že v grafickej ploche Image1, na jeho plátne Canvas, zmeníme pre pero Pen jeho farbu Color na červenú clRed. Úplne rovnako je to s farbou štetca a aj hrúbkou pera. S farbami sa naučíme pracovať dôkladnejšie niekedy neskôr, dnes si zapamätajme, že farby zadávame špeciálnymi pomenovanými konštantami clRed, clBlue, clYellow a pod. - kompletný zoznam týchto mien môžeme nájsť v Helpe (kliknite na niektoré meno farby v programe a stlačte **F1**).

Nasledovný program nakreslí vedľa seba 3 rôzne veľké štvorce:

```
begin
  Image1.Canvas.Pen.Width := 5;
  Image1.Canvas.Rectangle(100, 200, 150, 150);
```

```
  Image1.Canvas.Rectangle(150, 200, 250, 100);
  Image1.Canvas.Rectangle(250, 200, 400, 50);
end;
```

Všimnite si, že hrúbku sme nastavili len raz a odvtedy platí pre všetky kresby. Je jasné, že pomocou Rectangle kreslíme aj štvorce. Niekedy sa musíme so súradnicami "trochu pohrať", aby sme dostali útvary podľa našich predstáv - tu sú štvorce nakreslené tesne vedľa seba a ich spodná strana leží na jednej priamke.

Celočíselné premenné, deklarovanie premennej

Budeme riešiť takúto úlohu: treba nakresliť štvorec so stranou a = 100, ktorého ľavý horný roh má súradnice x = 50 a y = 70. Radi by sme v tejto úlohe použili premenné, aby sme s nimi mohli ďalej experimentovať:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, x, y: Integer;
begin
  a := 100;
  x := 50;
  y := 70;
  Image1.Canvas.Pen.Width := 5;
  Image1.Canvas.Rectangle(x, y, x+a, y+a);
end;
```

V našom programe sme najprv zadeklarovali tri celočíselné premenné - za slovo var sme najprv vymenovali nové premenné a, x, y a za dvojbodku zapísali, že budú celočíselné - Integer. Môžeme si ich predstaviť ako pomenované vyhradené pamäťové miesto veľkosti 4 bajty. Zmestí sa do nich celé číslo z intervalu **<** –2 147 483 648, 2 147 483 647 **>** čo je približne + alebo - 2^{31}. Treba si zapamätať, že takéto premenné sa v pamäti vyhradia vždy pri štarte procedúry (napr. pri slove begin) - vtedy majú ešte nedefinovanú hodnotu a treba im nejakú priradiť - to sú tie priraďovacie príkazy. Od tohto momentu môžeme s nimi pracovať pomocou ich mien, môžeme tieto hodnoty aj viackrát meniť, ale zapamätáme si, že na konci procedúry (pri slove end) sa tieto premenné zrušia a ich hodnota sa nenávratne zabudne.

Vždy keď sa táto procedúra vyvolá, t.j. keď klikneme na tlačidlo Button1 a vyvolá sa táto akcia, postupne sa vykoná postupnosť príkazov v tele procedúry - medzi slovami begin a end. V našom príklade sa vždy vykoná rovnaká postupnosť príkazov s rovnakým výsledkom - štvorcom so stranou 100 na súradniciach (50,70). To znamená, že ak budeme toto tlačidlo tlačiť viackrát za sebou, nepostrehneme žiadnu zmenu, lebo stále sa prekresľuje ten istý štvorec.

Použijeme náhodný generátor - funkciu Random, ktorá zakaždým vráti nejakú inú (možno aj tú istú) hodnotu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, x, y: Integer;
begin
  a := 100;
  x := Random(500);
  y := Random(400);
  Image1.Canvas.Pen.Width := 5;
  Image1.Canvas.Rectangle(x, y, x+a, y+a);
end;
```

Príkaz x := Random(500); znamená, že vždy keď sa ide vykonať, počítač si "vymyslí" nejaké náhodné číslo z intervalu **<**0, 499**>** a to priradí do premennej **x**, podobne je to aj s premennou **y** - tej sa priradí náhodná hodnota z intervalu **<**0, 399**>**. Teda Random(n) vráti hodnotu z intervalu **<**0, n-1**>**. Takáto procedúra po každom zavolaní

nakreslí štvorec na iných súradniciach.

Ešte sa vráťme k premenným. Meno premennej môže byť ľubovoľné slovo, ktoré môže obsahovať aj číslice (nie na začiatku), ale aj znak _ (podčiarkovník). Malo by sa zvoliť tak, aby už jej názov vyjadroval, čo bude v tejto premennej uchované. Mená premenných nemôžu obsahovať písmená s diakritikou. V jednoduchých krátkych programoch často používame jednopísmenové mená, ale v reálnych veľkých programoch budeme niektoré komplikovanejšie mená pomenúvať aj viac ako 10 znakovými reťazcami.

Často sa nám bude hodiť zmazanie grafickej plochy:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Image1.Canvas.Brush.Color := clWhite;
  Image1.Canvas.FillRect(Image1.ClientRect);
end;
```

Prvý riadok, v ktorom nastavujeme bielu farbu štetca, je tu preto, lebo príkaz FillRect zmaže celú plochu farbou štetca a najčastejšie sa nám hodí práve biela - samozrejme, že by sme mohli zmazávať ľubovoľnou farbou.

For-cyklus, premenná cyklu, integer, aritmetické výrazy

Naučíme sa novú konštrukciu, ktorá umožní, aby sme nejaké príkazy opakovali veľa krát, ale napíšeme ich iba raz. Napr. chceme nakresliť naraz 10 štvorcov na náhodných pozíciách. Všetky príkazy, ktoré kreslia jeden náhodný štvorec, vložíme do konštrukcie tzv. **for-cyklu**:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, x, y, i: Integer;
begin
  for i := 1 to 10 do
  begin
    a := 100;
    x := Random(500);
    y := Random(400);
    Image1.Canvas.Pen.Width := 5;
    Image1.Canvas.Rectangle(x, y, x+a, y+a);
  end;
end;
```

Veľmi dôležitý je prvý riadok konštrukcie:

- začína slovom for, za ktorým nasleduje priradenie pomocnej premennej **i** (tzv. premenná cyklu) počiatočnou hodnotou - u nás 1, a za slovom to je číslo 10, ktoré vyjadruje, že sa budú opakovať nejaké príkazy, pričom premenná cyklu bude pritom nadobúdať hodnoty od **1** do **10** - teda opakovaní bude 10. Za číslom 10 je ešte slovo do a slovo begin, ktoré označuje, že nasleduje postupnosť príkazov na opakovanie.
- premenná cyklu - u nás premenná i - musí byť tiež deklarovaná ešte pred prvým begin v časti var.

Ďalej nasledujú riadky, ktoré sa budú opakovane vykonávať 10-krát - my sme ich úmyselne trochu odsunuli vpravo, aby lepšie vyniklo slovo end - toto je koniec postupnosti opakovaných príkazov.

Premennú cyklu i môžeme veľmi užitočne využívať aj medzi príkazmi, ktoré sa budú opakovať (tzv. telo cyklu) - zrejme bude pri každom ďalšom opakovaní mať ďalšiu nasledujúcu hodnotu, teda postupne 1, 2, 3, ... 10. Napr.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, x, y, i: Integer;
begin
  for i := 1 to 10 do
```

```
  begin
    a := 10 * i;
    x := Random(500);
    y := Random(400);
    Image1.Canvas.Pen.Width := 5;
    Image1.Canvas.Rectangle(x, y, x+a, y+a);
  end;
end;
```

Každý z desiatich štvorcov bude mať veľkosť strany desaťnásobok poradového čísla, t.j. postupne 10, 20, 30, ... 100. Aby sme si lepšie uvedomili, čo všetko a prečo sa v cykle opakuje, ukážme si upravený variant tohto istého programu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  x, y, i: Integer;
begin
  Image1.Canvas.Pen.Width := 5;
  for i := 1 to 10 do
  begin
    x := Random(500);
    y := Random(400);
    Image1.Canvas.Rectangle(x, y, x+10*i, y+10*i);
  end;
end;
```

Príkaz, ktorým sa nastavovala hrúbka čiar štvorca na 5 nemusí byť vo vnútri cyklu a tým sa vykoná 10-krát, ale ak bude pred cyklom, vykoná sa len raz. Zrušili sme premennú a a tam kde sa používala, sme miesto nej dali vzorec 10*i. Hoci sme ušetrili 4 bajty na tejto premennej, niekedy ale takéto šetrenie nemusí pomôcť čitateľnosti programu.

Ukážme si použitie for-cyklu pri kreslení 12 štvorcov so spoločným smerom:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, x, y, i: Integer;
begin
  Image1.Canvas.Pen.Width := 5;
  x := 250;
  y := 200;
  for i := 1 to 12 do
  begin
    a := 150-8*i;
    Image1.Canvas.Rectangle(x-a, y-a, x+a, y+a);
  end;
end;
```

V tomto príklade v premenných (**x**,**y**) nie je ľavý horný roh štvorca, ale stred štvorca. Podobne v premennej **a** nie je veľkosť strany, ale polovica veľkosti strany. Na výpočet tejto veľkosti sme použili vzorec 150-8*i a preto pre prvý štvorec je to 142, pre druhý 134, pre tretí 126, ... pre posledný dvanásty 54. Opäť je tu veľa poučného:

- pri programovaní môžeme používať aj komplikovanejšie vzorce, pričom sa vypočítavajú použitím bežných matematických vlastností (napr. násobenie má prioritu pred odčitovaním) - okrem základných operácií **+**, **-**, *****, môžeme používať celočíselné delenie div a zvyšok po celočíselnom delení mod;
- pred cyklus sme presunuli aj priradenia do x a y, lebo aj tie stačí spočítať len raz a viac sa nemenia;
- pri kreslení štvorcov ich vedome kreslíme od najväčšieho po najmenší - keby sme ich kreslili v opačnom poradí, ten nasledujúci väčší by úplne prekryl ten predchádzajúci - na koniec by bol vidieť len ten posledný - najväčší. Skúste zmeniť priradenie do premennej a := 46+8*i; - tiež bude teraz kresliť 12 štvorcov so

stranami 54, 62, 72, ... 142 ale vidieť bude len posledný.

Ďalšou dvojicou príkazov na kreslenie v grafickej ploche sú MoveTo a LineTo. Oba presúvajú grafické pero na nové pozície, len MoveTo pritom nekreslí čiary a LineTo kreslí - nastavenou farbou a hrúbkou. Ak chceme nakresliť niekoľko úsečiek pod sebou, môžeme to zapísať takto:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Image1.Canvas.Pen.Width := 3;
  Image1.Canvas.Pen.Color := clGreen;
  Image1.Canvas.MoveTo(50, 20);
  Image1.Canvas.LineTo(300, 20);
  Image1.Canvas.MoveTo(50, 40);
  Image1.Canvas.LineTo(250, 40);
  Image1.Canvas.MoveTo(50, 60);
  Image1.Canvas.LineTo(200, 60);
  Image1.Canvas.MoveTo(50, 80);
  Image1.Canvas.LineTo(150, 80);
end;
```

alebo pomocou for-cyklu takto:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  i: Integer;
begin
  Image1.Canvas.Pen.Width := 3;
  Image1.Canvas.Pen.Color := clGreen;
  for i := 1 to 4 do
  begin
    Image1.Canvas.MoveTo(50, 20*i);
    Image1.Canvas.LineTo(350-i*50, 20*i);
  end;
end;
```

Konštrukcia for-cyklu je veľmi užitočná a budeme ju často používať, ale mali by sme si dobre zapamätať tieto pravidlá:

- premennú cyklu môžeme v príkazoch v tele cyklu používať, ale **nesmieme** ju tu meniť, t.j. nesmieme do nej nič priraďovať,
- premenná cyklu po skončení cyklu je už voľná na ďalšie použitie - hovoríme, že má nedefinovanú hodnotu, ak ju chceme používať, musíme jej nejakú hodnotu priradiť,
- premenná cyklu postupne nadobúda hodnoty od počiatočnej **A** až po koncovú **B**:
    - ak je **A ≤ B**, tak sa cyklus vykoná presne **B-A+1** krát,
    - ak **A = B**, tak sa vykoná práve raz,
    - ak **A > B**, tak sa nevykoná ani raz
- počiatočná aj koncová hodnota nemusí byť zadaná len konštantou, ale aj zložitejším aritmetickým výrazom s premennými

Zhrnutie grafických príkazov

Zhrnieme všetky grafické príkazy, s ktorými sme sa už zoznámili, resp. ich doplníme o niekoľko ďalších základných. Vo všetkých príkazoch predpokladáme, že začínajú napr. Image1.Canvas. Ak chcete o niektorých nastaveniach alebo príkazoch vedieť viac, pozrite sa do Helpu. Nastavenie pera a štetca:

- Pen.Color := clRed; // ďalšie farby sú, napr. clBlack, clGreen, clBlue, clYellow, clWhite ...
- Pen.Width := 3; // hrúbka pera od 1 vyššie

- Pen.Style := psSolid; // typ čiary - rôzne bodkované a čiarkované čiary, napr. psDash, psDot, ...
- Brush.Color := clWhite; // farba výplne
- Brush.Style := bsSolid; // štýl výplne, užitočný je bsClear - žiadna výplň, ďalšie sú napr. bsVertical, bsHorizontal, ...

Kreslenie čiar a útvarov:

- MoveTo(x, y); // presunie pero bez kreslenia čiar
- LineTo(x, y); // od momentálnej pozície pera kreslí úsečku do bodu (x, y)
- Rectangle(x1, y1, x2, y2); // obdĺžnik - teda aj štvorec
- Ellipse(x1, y1, x2, y2); // elipsa a teda aj kruh - parametre sú ako pri obdĺžniku a znamenajú elipsu vpísanú do obdĺžnika

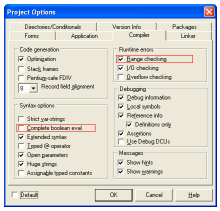
Písanie textov do grafickej plochy:

- TextOut(x, y, 'text'); // text v apostrofoch vypíše od súradnice (x, y) - písmo sa dá nastaviť pomocou Font
- Font.Height := 20; // veľkosť písma
- Font.Name := 'Arial'; // meno fontu
- Font.Color := clRed; // farba písma
- Font.Style := [fsBold]; // do hranatých zátvoriek sa vymenujú atribúty písma - napr. aj fsItalic, fsUnderline

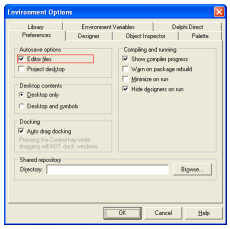
Nastavenie prostredia Delphi, súbory, prípony

Časom prídete na to, ktoré nastavenia prostredia vám vyhovujú najlepšie a zrejme, keď budete musieť pracovať na cudzom počítači, tak si najprv zmeníte nastavenia podľa svojich predstáv. Tu vás upozorníme len na dve dôležité nastavenia, ktoré sa predpokladajú nielen pri odovzdávaní vašich projektov, ale aj na skúške.

Prvým dôležitým nastavením je kontrola pretečenia indexov polí - **Range checking**. Nastavuje sa v menu **Project**, cez **Options...** Toto nastavenie nájdeme v záložke **Compiler**:



Ďalšie nastavenie je dôležité nie pre spúšťanie projektu, ale pre prácu v prostredí Delphi - je to automatické ukladanie súborov pred spúšťaním projektu (F9). Ak by sa náhodou stal nejaký problém s Delphi počas spúšťania (a niekedy sa to môže vyskytnúť, vďaka vašim chybám) a váš program by nebol uložený, nenávratne by sa vám stratili všetky posledné zmeny. Nastavenie môžme urobiť napr. takto: v menu **Tools** vyberieme **Environment Options...** a v záložke **Preferences** zaškrtneme **Editor Files**.



2. prednáška: grafický robot

čo už vieme:

- vytvárať programy, ktoré kreslia do grafickej plochy štandardnými príkazmi
- jednoduché celočíselné premenné, priradenie a for-cyklus

čo sa na tejto prednáške naučíme:

- idea robota v grafickej ploche - príkazy, nastavenia, relatívna geometria
- kreslenie geometrických útvarov, kružnice, špirály, časti kružníc
- farby
- viac robotov
- objekt Robot

Idea robota v grafickej ploche - príkazy, nastavenia, relatívna geometria

Na minulej prednáške sme sa naučili pracovať s grafickým perom v grafickej ploche. Dnes si ukážeme iný spôsob kreslenia: využijeme na to špeciálne "vycvičených" robotov, ktorí sa vedia v grafickej ploche nejako pohybovať a pritom môžu za sebou zanechávať čiaru. Nakoľko tieto roboty nie sú štandardnou súčasťou Delphi, bude treba do nášho programu špeciálne niečo kvôli tomu pridávať:

1. predpokladáme, že máme novú aplikáciu s grafickou plochou a aspoň jedným tlačidlom,
2. za riadok {$R *.dfm} pridáme:

```
uses
  RobotUnit;
```

3. tento zápis predpokladá, že na disku už máme súbor s definíciou správania robotov - súbor sa volá **RobotUnit.pas** a tu si ho môžete stiahnuť do svojho počítača - je ale veľmi dôležité, aby sa nachádzal presne v tom istom priečinku, ako aj váš program (ak budete mať viac programov, ktoré pracujú s týmito robotmi, vo viacerých priečinkoch, do všetkých treba tento súbor nakopírovať)
4. teraz už môžme zadefinovať nových robotov a aj s nimi pracovať

Do procedúry, ktorá spracováva kliknutie na tlačidlo, zapíšeme:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to 4 do
  begin
    r.fd(100);
    r.lt(90);
  end;
end;
```

Postupne vysvetlíme všetko, čo sme zapísali:

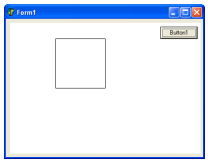
- zadeklarovali sme premennú r, pomocou ktorej budeme ovládať robota - premenná typu TRobot
- **cs** - príkaz na zmazanie obrazovky - skratka z anglického **Clear Screen**
- vytvorenie nového robota: TRobot.Create označuje, že chceme aby sa vytvoril (narodil) nový robot a tento sa priradí do premennej r - robot sa vytvoril v strede grafickej plochy

- vo for-cykle sa štyrikrát vykonajú dva príkazy robota:
  - r.fd - skratka z anglického **forward** - robot prejde požadovaný počet krokov, teda 100
  - r.lt - skratka z anglického **left** - robot sa otočí vľavo o zadaný uhol, teda 90 stupňov

Robot vo všeobecnosti pracuje na takomto princípe:

- nachádza sa niekde v grafickej ploche a je natočený nejakým smerom - nie je ho ale vidieť - nezobrazuje sa
- ak sa mu povie, aby prešiel nejakú vzdialenosť, tak v momentálnom smere natočenia sa presunie dopredu
- ak sa mu povie, aby sa o nejaký uhol otočil, tak nemení svoju pozíciu, len sa na svojom mieste otočí o zadaný uhol
- robot má k dispozícii pero, ktorým vie počas svojho chodenia zanechávať za sebou čiaru - toto pero môže byť spustené - vtedy kreslí, alebo zdvihnuté - vtedy sa len presúva
- tomuto peru môžme zmeniť farbu aj hrúbku

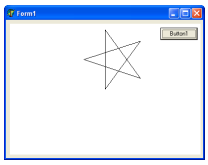
Nie je ťažké pochopiť, že robot, ak má spustené pero, daným programom nakreslí v ploche štvorec so stranou 100. Keďže sa narodil v strede plochy otočený na sever, štvorec bude mať strany rovnobežné s osami.



Aby sme si lepšie vedeli predstaviť, kde sa pri kreslení robot nachádza, môžeme toto kreslenie trochu spomaliť volaním procedúry wait. Pridali sme druhé tlačidlo Button2:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to 5 do
  begin
    r.fd(120);
    r.rt(144);
    wait(300);
  end;
end;
```

Neskôr počas semestra si ukážeme, prečo takýto spôsob spomaľovania výpočtu môže robiť pod Windows problémy, ale pre tieto jednoduché školské príklady je to v poriadku. Všimnite si, že okrem volania wait (parametrom je počet milisekúnd, teda v našom príklade je to 0,3 sekundy) sme cyklus nechali prejsť 5-krát a na otočenie sme použili príkaz robota r.rt - skratka z anglického slova **right** - robot sa otočí o zadaný uhol vpravo, t.j. v smere hodinových ručičiek.



V ďalšom programe ukážeme, ako spomaliť samotné kreslenie čiary:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  i: Integer;
```

```
begin
  cs;
  r := TRobot.Create(20, 250, 60);
  r.PW := 3;
  r.PC := clBlue;
  for i := 1 to 80 do
  begin
    r.fd(5);
    wait(50);
  end;
end;
```

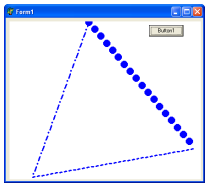
Aby robot nakreslil úsečku dĺžky 400, v cykle prešiel 80-krát dĺžku 5. Všimnite si, že sme použili iný spôsob vytvorenia nového robota - do konštrukcie TRobot.Create sme pridali 3 parametre - tieto označujú miesto v grafickej ploche, kde sa má robot "narodiť" (v našom prípade na súradniciach (20, 250)). Tretie číslo v tejto trojici označuje počiatočné natočenie robota: 0 označuje hore, 90 vpravo atď.

Ďalšími novinkami je nastavovanie parametrov pera robota:

- r.PW := 3; znamená, že nastavujeme hrúbku pera (z anglického **Pen Width**) - princíp je rovnaký ako nastavenie hrúbky pera v Canvase grafickej plochy
- r.PC := clBlue; nastavuje farbu pera robota (z anglického **Pen Color**)

Nasledujúci príklad ilustruje ďalšie príkazy robota:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create(50, 340, 80);
  r.PW := 3;
  r.PC := clBlue;
  for i := 1 to 36 do
  begin
    r.pd;
    r.fd(5);
    r.pu;
    r.fd(5);
    wait(20);
  end;
  r.lt(120);
  for i := 1 to 18 do
  begin
    r.fd(20);
    r.point(15);
    wait(50);
  end;
  r.lt(120);
  for i := 1 to 15 do
  begin
    r.pd;
    r.fd(10);
    r.pu;
    r.fd(7);
    r.point;
    r.fd(7);
    wait(50);
  end;
end;
```



V príklade vidíme tieto nové príkazy:

- r.pd; - robot dá pero dolu - teraz budú všetky pohyby zanechávať za robotom čiaru - keď sa robot "narodí", tak má hneď pero dolu
- r.pu; - robot dá pero hore (z anglického **Pen Up**)
- r.point(15); - robot na mieste, kde sa práve nachádza nakreslí bodku zadanej veľkosti, napr. 15
- r.point; - keď nezadáme veľkosť bodky, kreslí sa bodka podľa hrúbky pera, t.j. v našom prípade 3 - všimnite si, že bodka sa nakreslí, aj keď je zdvihnuté pero a teda čiary sa nekreslia.

Kreslenie geometrických útvarov, kružnice, špirály, časti kružníc

Nasledujúci príklad využije vnorené cykly a nakreslí 8 otočených rovnostranných trojuholníkov:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  i, j: Integer;
begin
  r := TRobot.Create;
  for j := 1 to 8 do
  begin
    for i := 1 to 3 do
    begin
      r.fd(100);
      r.lt(120);
    end;
    r.rt(45)
  end;
end;
```

V predchádzajúcich príkladoch sme videli, ako sa kreslí štvorec, päť-cípa hviezda a rovnostranný trojuholník. Pri kreslení pravidelného n-uholníka si treba uvedomiť, že robot sa vo vrchole nemá otočiť o vnútorný uhol (napr. 60 pre rovnostranný trojuholník) ale o vonkajší (teda 120). Jednoduchou matematickou úvahou sa dá ukázať, že vo všeobecnosti sa pri kreslení pravidelného n-uholníka treba otáčať o **360/n** stupňov. Tento aritmetický výraz ale nie je celočíselný - nemôžeme ho priradiť do celočíselnej premennej. Neskôr sa naučíme, ako sa pracuje s tzv. reálnym číselným typom. Našťastie všetky príkazy robota akceptujú ako svoje parametre aj nie celé čísla a teda sa dá dosť presne nakresliť aj pravidelný sedem-uholník:

```
procedure TForm1.Button1Click(Sender: TObject);
const
  n = 7;
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to n do
  begin
    r.fd(100);
    r.rt(360 / n);
  end;
```

```
end;
```

V tomto príklade sme opäť použili jednu malú novinku: n sme nedefinovali ako premennú, do ktorej by sme potom museli priradiť, napr. číslo 7, ale zadefinovali sme ju ako konštantu programu. Konštanta má tiež svoje meno, môžeme ju používať vo výrazoch, ale nesmieme jej už túto hodnotu v programe ďalej meniť, napr. priraďovacím príkazom. Takto napísaný program má tú výhodu, že pomocou neho vieme nakresliť ľubovoľný pravidelný n-uholník: stačí zmeniť konštantu n a po spustení nakreslí nový n-uholník.

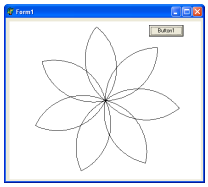
Zaujímavým je aj pravidelný 360-uholník s malou dĺžkou strany. Tento 360-uholník môžeme na ploche považovať za kružnicu.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to 360 do
  begin
    r.fd(2);
    r.rt(1);
  end;
end;
```

Ak by for-cyklus nebežal 360-krát, ale napr. 180 alebo 90, robot by nakreslil pol-kružnicu alebo štvrť-kružnicu. Podobne, ak by robot nezatáčal vpravo ale vľavo, vytváral by rôzne otočené časti kružníc - z nich by sa elegantne dal nakresliť, napr. kvet zo 7 lupeňov:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  i, j, k: Integer;
begin
  cs;
  r := TRobot.Create;
  for k := 1 to 7 do
  begin
    for j := 1 to 2 do
    begin
      for i := 1 to 90 do
      begin
        r.fd(2);
        r.rt(1);
      end;
      r.rt(90);
    end;
    r.rt(360/7);
  end;
end;
```

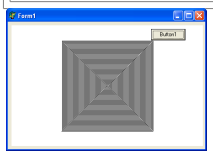
V tomto programe sú zaujímavé aj navzájom vnorené 3 for-cykly - pozorne ich preštudujte.



Špirála - vznikne, keď pri kreslení v cykle budeme meniť dĺžku čiary. V nasledujúcich príkladoch nebudeme

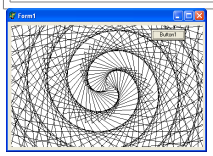
uvádzať prvý riadok definície procedúry Button1Click ...

```
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to 200 do
  begin
    r.fd(i);
    r.rt(90);
    wait(5);
  end;
end;
```



Zaujímavé obrazce môžeme dostať, keď budeme experimentovať s rôznymi uhlami, napr.

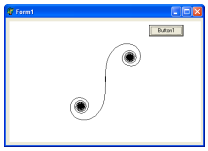
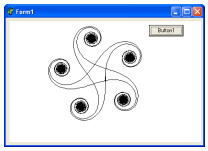
```
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to 200 do
  begin
    r.fd(4*i);
    r.rt(123);
    wait(5);
  end;
end;
```



alebo veľmi zaujímavé typy špirál

```
var
  r: TRobot;
  i: Integer;
begin
  cs;
  r := TRobot.Create;
  for i := 1 to 2000 do
  begin
    r.fd(8);
    r.rt(i);      // r.rt(i+0.1);
    wait(1);
  end;
end;
```

Text v programe, ktorý je za // je považovaný za komentár a nevykoná sa. Poexperimentujte s rôznymi obmenami otáčania sa robota.

Viac robotov

Ak chceme naraz pracovať s viacerými robotmi, zadeklarujeme viac premenných typu TRobot. Potom každého robota zvlášť vytvoríme pomocou Create, prípadne, aby sme ich odlíšili, zmeníme im farbu a hrúbku pera. Pred každý príkaz musíme zapísať meno robota, ktorému je tento príkaz určený, napr.

```
var
  r1, r2, r3: TRobot;
  i: Integer;
begin
  cs;
  r1 := TRobot.Create(50,170);
  r1.PC := clRed;
  r1.PW := 3;

  r2 := TRobot.Create(150,200,30);
  r2.PC := clBlue;
  r2.PW := 1;

  r3 := TRobot.Create(250,200,60);
  r3.PC := clGreen;
  r3.PW := 5;

  for i := 1 to 360 do
  begin
    r1.fd(1);
    r1.rt(1);

    r2.fd(100);
    r2.rt(143);

    r3.fd(150);
    r3.rt(173);
    wait(10);
  end;
end;
```

Farby

Funkciu Random - generátor náhodných čísel môžeme použiť aj pre nastavenie náhodnej farby pera robota. Vtedy musíme ako parameter funkcie Random poslať hodnotu **256*256*256**, napr. ako je v tomto príklade

```
var
  r: TRobot;
  i, j: Integer;
begin
  cs;
  r := TRobot.Create;
  r.pu;
  for i := 1 to 4 do
  begin
    for j := 1 to 10 do
    begin
      r.fd(15);
      r.PC := Random(256*256*256);
      r.point(12);
```

```
    end;
    r.rt(90);
  end;
end;
```

V skutočnosti je to skrátený zápis inej konštrukcie. Môžeme predpokladať, že všetky farby v počítači sú namiešané z troch základných farieb: červenej, zelenej a modrej (tzv. model **RGB**). Farba závisí od toho, koľko je v nej zastúpená každá z týchto troch farieb. Zastúpenie jednotlivej farby vyjadrujeme číslom od 0 do 255 (zmestí sa do jedného bajtu), napr. žltá farba vznikne, ak namiešame 255 červenej, 255 zelenej a 0 modrej. Ak budeme zastúpenie každej farby trochu meniť, napr. 250 červenej, 240 zelenej a hoci 100 modrej, stále to bude žltá, ale iného odtieňa. Na skladanie farieb máme k dispozícii funkciu RGB, ktorej zadáme tri čísla od 0 do 255 a ona vytvorí príslušnú farbu, napr. známe preddefinované farby majú takéto vyjadrenie:

clBlack	RGB(0,0,0)	clRed	RGB(255,0,0)
clMaroon	RGB(128,0,0)	clLime	RGB(0,255,0)
clGreen	RGB(0,128,0)	clYellow	RGB(255,255,0)
clOlive	RGB(128,128,0)	clBlue	RGB(0,0,255)
clNavy	RGB(0,0,128)	clFuchsia	RGB(255,0,255)
clPurple	RGB(128,0,128)	clAqua	RGB(0,255,255)
clTeal	RGB(0,128,128)	clLtGray	RGB(192,192,192)
clGray	RGB(128,128,128)	clDkGray	RGB(128,128,128)
clSilver	RGB(192,192,192)	clWhite	RGB(255,255,255)

Náhodnú farbu môžeme teda namiešať napr. takto

```
  r.PC := rgb(Random(256),Random(256),Random(256));
                  // to je to isté ako r.PC := Random(256*256*256);
  r.PC := rgb(Random(256),0,Random(256));
  r.PC := rgb(150+Random(106),0,200);
```

Môžete experimentovať s týmto príkladom

```
var
  r: TRobot;
  i, j: Integer;
begin
  r := TRobot.Create;
  r.pu;
  for i := 1 to 40 do
  begin
    for j := 1 to 10 do
    begin
      r.fd(15);
      r.PC := RGB(0,0,Random(256));
      // cs;
      r.point(20);
      wait(200);
    end;
    r.rt(90);
  end;
end;
```

Kreslia sa bodky rôznych odtieňov modrej: od čiernej až po modrú.

Ak zakomentovaný príkaz **cs** v cykle "odkomentujete", teda v cykle sa bude mazať obrazovka, budeme vidieť len jednu bodku na pozícii robota.

Objekt Robot

V doterajších príkladoch sme zadefinovali a používali buď celočíselné premenné (Integer) alebo premenné typu TRobot. Tieto premenné robota sa líšia od "obyčajných" premenných tým, že

- si pamätajú svoj momentálny stav v svojich tzv. **stavových premenných** (napr. pozícia, farba, ...)
- majú svoje *súkromné* príkazy, pomocou ktorých ich nejako riadime, resp. meníme ich stavové premenné - takýmto príkazom - sú to procedúry - hovoríme **metódy** a "rozumejú" im len roboty (premenné typu TRobot)
- musia byť vytvorené (nie deklarované) špeciálnym spôsobom (TRobot.Create(...);) a kým sa takto nevytvoria, nesmú sa vôbec používať
- takýmto premenným hovoríme **OBJEKT** a typom, z ktorých vytvárame objekty (napr. TRobot) hovoríme **TRIEDA** (po anglicky **object** a **class**); niekedy sa objektu hovorí aj **inštancia triedy**.
- okrem robotov sme sa už stretli aj s inými objektmi, napr. Form1, Image1, Button1
- zatiaľ si o objektoch treba zapamätať, že sú to premenné, ktoré môžu v sebe obsahovať veľa stavových premenných a tiež "v sebe" obsahujú nejaké svoje procedúry (metódy) => tomuto hovoríme **zapuzdrenie** (enkapsulácia), lebo v jednom "puzdre" sú aj údaje (stavové premenné) aj algoritmy (metódy), ktoré vedia s týmito údajmi pracovať.

Zhrňme všetky doteraz známe príkazy a nastavenia robota:

- vytvorenie nového robota:
  - Create; // robot vznikne v strede plochy
  - Create(x, y); // robot vznikne na súradniciach **(x, y)**
  - Create(x, y, uhol); // robot vznikne na súradniciach **(x, y)** s počiatočným uhlom **uhol**

- pohyby a kreslenie:
  - fd(dĺžka); // **forward** - robot prejde vzdialenosť **dĺžka**
  - setxy(x, y); // robot sa premiestni na súradnicu **(x, y)**
  - movexy(x, y); // robot sa premiestni na súradnicu **(x, y)** so zdvihnutým perom
  - point; // robot nakreslí bodku veľkosti hrúbky pera
  - point(veľkosť); // robot nakreslí bodku veľkosti **veľkosť**

- otáčanie sa:
  - rt(uhol); // **right** - robot sa namieste otočí o **uhol** stupňov vpravo
  - lt(uhol); // **left** - robot sa namieste otočí o **uhol** stupňov vľavo
  - seth(uhol); // **set heading** - robot sa namieste otočí do absolútneho uhla **uhol** stupňov

- nastavovanie:
  - PC := farba; // **pen color** - nastavíme robotovi farbu pera na **farba**
  - PW := hrúbka; // **pen width** - nastavíme robotovi hrúbku pera na **hrúbka**
  - pu; // **pen up** - robotovi zdvihneme pero
  - pd; // **pen down** - robotovi spustíme pero

Všimnite si, že PC a PW sú stavové premenné a môžeme s nimi pracovať skoro ako s obyčajnými premennými, napr.

```
r.PW := r.PW + 1;
```

Ďalšími stavovými premennými robota sú:

- **X**, **Y** - pozícia robota - uvedomte si rozdiel medzi príkazmi v týchto dvoch riadkoch:

```
r.X := 100;
r.Y := 170;
r.setxy(100, 170);
```

- **H** - absolútny smer momentálneho otočenia, napr. robota môžeme otočiť vpravo o 30 stupňov aj takto:

```
r.H := r.H + 30;
```

Ďalšie príkazy, ktoré súvisia s robotmi, ale sú "globálne":

- cs; // **clear screen** - zmažeme grafickú plochu bielou farbou
- cs(farba); // zmažeme grafickú plochu farbou **farba**
- wait(ms); // program sa pozdrží na **ms** milisekúnd

Pozrite si **RobotUnit.pas** a pokúste sa pochopiť ďalšie metódy a stavové premenné. V niektorých nasledujúcich prednáškach sa s nimi stretneme.

3. prednáška: podmienky a procedúry

čo už vieme:

- pracovať s objektom robot
- jednoduché celočíselné premenné, priradenie a for-cyklus

čo sa na tejto prednáške naučíme:

- podmienky, podmienený príkaz, cyklus s podmienkou
- princíp fungovania procedúr, parametre
- globálne a lokálne premenné

Cyklus s podmienkou

Na prvej prednáške sme sa naučili opakovať postupnosť nejakých príkazov pomocou for-cyklu. S takýmto cyklom sa pracuje dobre, ak dopredu viem počet opakovaní, resp. viem určiť počiatočnú a koncovú hodnotu, teda nejaký celočíselný interval hodnôt. V takomto cykle sme mali tzv. "premennú cyklu", ktorá automaticky postupne nadobúdala hodnoty zo zadaného intervalu a pre každú z nich sa vykonalo tzv. "telo cyklu".

Väčšina štandardných programovacích jazykov má aj iný typ cyklu: postupnosť príkazov (telo cyklu) sa bude opakovať pokým je splnená nejaká podmienka.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  d: Integer;
begin
  r := TRobot.Create;
  d := 5;
  while d < 200 do
  begin
    r.fd(d);
    r.lt(90);
```

```
    d := d+2;
    wait(10);
  end;
end;
```

Vysvetlime, čo sme zapísali:

- program začína vytvorením robota;
- cyklus s podmienkou za slovom while má nejakú matematickú podmienku (niečo, čo má hodnotu pravda alebo nepravda). Kým je táto podmienka pravdivá, bude sa vykonávať celé telo cyklu;
- po vykonaní všetkých príkazov tela cyklu sa podmienka opätovne skontroluje a ak je ešte stále pravdivá, telo cyklu sa vykoná znovu;
- ak sa raz pri kontrole zistí, že podmienka neplatí, cyklus končí a výpočet pokračuje za ním - v našom prípade za end;
- Ak je podmienka nepravdivá už pri prvom testovaní, cyklus sa nevykoná ani raz.
- Ak by bola podmienka splnená stále, cyklus nikdy neskončí - hovoríme tomu **nekonečný cyklus**.

Náš prvý program nakreslil štvorcovú špirálu, pričom podmienkou pri kreslení bolo to, aby postupne zväčšujúce sa úsečky, z ktorých sa špirála skladá, boli menšie ako 200. Podmienka testuje premennú d, ktorá sa v tele cyklu zväčšuje a teda zrejme niekedy presiahne hranicu 200 - a cyklus skončí.

Aby sme vedeli zapisovať aj komplikovanejšie podmienky, ktoré by boli poskladané z viacerých podpodmienok a spojené logickými operátormi, musíme dobre rozumieť mechanizmu, ako takéto podmienky vyhodnocuje Pascal. Napr. sme si z matematiky zvykli, že vo výraze

```
3 + 4 * 5 - 2
```

najprv vypočítame **3*4** a až potom to pripočítame k **3** a odpočítame **2**. Toto platí vďaka tomu, že operácia ***** má vyššiu prioritu (hovoríme aj precendenciu) ako operácie **+** a **-**. Podobne je to aj s relačnými a logickými operátormi - všetky operátory so zoradené do skupín priorít a najprv sú vyhodnocované tie, ktoré majú vyššiu prioritu. Ukážme si tabuľku priorít operácií (zatiaľ tu nie sú všetky operácie):

unárne	**not**
multiplikatívne	*** / div mod and**
aditívne	**+ - or**
relačné	**= < <= > >= <>**

Najvyššiu prioritu má logická negácia not. Trochu nižšiu majú násobenie, delenie a logický súčin and. Ak je v jednom výraze viac operátorov z rovnakej úrovne, tak sa väčšinou vyhodnocujú zľava doprava. Samozrejme, že budeme využívať okrúhle zátvorky, aby sme toto poradie vyhodnocovania ovplyvnili podľa našich predstáv. Pozrime sa na zložitejší výraz:

```
x < 100 or y >= 50
```

Podľa tabuľky priorít vidíme, že ako prvé sa bude vyhodnocovať or, t.j. 100 or y, potom sa výsledok tejto operácie porovná, či je väčší ako x a ďalej by sa to porovnávalo s **50** ... Pravdepodobne sme pri tomto výraze neočakávali, že by sa mal takto vyhodnocovať (Delphi by aj tak hlásili chybu, že sa to nedá vyhodnotiť). Správne to teda malo vyzerať takto:

```
( x < 100 ) or ( y >= 50 )
```

Aby sme ďalej mohli experimentovať s while-cyklom, vylepšíme program tak, že sa robot bude otáčať o náhodný uhol:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  d, uhol: Integer;
begin
  Randomize;
  r := TRobot.Create;
  cs;
  d := 5;
  uhol := Random(90) + 90;      // náhodné číslo od 90 do 179
  while d < 200 do
  begin
    r.fd(d);
    r.lt(uhol);
    d := d+2;
    wait(1);
  end;
end;
```

Vždy, keď zatlačíme tlačidlo **Button2**, vykoná sa táto procedúra, t.j. nakreslí sa špirála s náhodným uhlom. Ak by sme chceli, aby sa opakovalo toto kreslenie náhodných špirál v nejakých časových intervaloch, väčšiu časť programu opäť "zabalíme" do nového cyklu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  d, uhol: Integer;
begin
  Randomize;
  r := TRobot.Create;
  while True do
  begin
    cs;
    r.movexy(200, 130);
    d := 5;
    uhol := Random(90) + 90;
    while d < 200 do
    begin
      r.fd(d);
      r.lt(uhol);
      d := d+2;
    end;
    wait(500);
  end;
end;
```

Použili sme tu nekonečný cyklus - podmienkou opakovania je vždy splnená podmienka, teda slovo True. Ak by sme namiesto toho použili slovo False, tak telo cyklu by sa nevykonalo ani raz. Po naštartovaní tohto programu (po zatlačení tlačidla Button1) sa budú kresliť rôzne špirály, až kým aplikáciu neukončíme.

Nakoľko sú náhodné uhly "slušné", t.j. z intervalu <90, 180), špirála sa bude kresliť v blízkom okolí štartu špirály. Ak by sme chceli cyklus ukončiť vtedy, keď sa robot vzdiali od bodu (200, 130) aspoň o 120 (r.X a r.Y je spôsob ako zistiť jej momentálne súradnice), môžeme to zapísať aj takto:

```
while Sqrt((r.X-200)*(r.X-200)+(r.Y-130)*(r.Y-130)) < 120 do ...
while Sqrt(Sqr(r.X-200)+Sqr(r.Y-130)) < 120 do ...
```

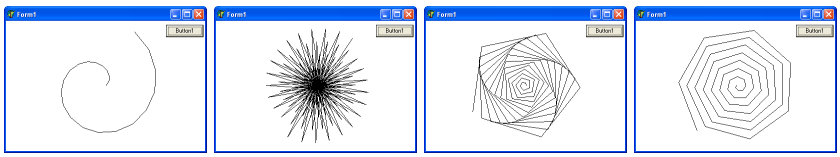
alebo využijeme funkciu robota dist, ktorá vypočíta vzdialenosť robota od zadaného bodu v rovine:

```
while r.dist(200, 130) < 120 do ...
```

Teraz môžeme prepísať vnútorný cyklus, ktorý skončí, keď sa robot priveľmi vzdiali od štartového bodu:

```
  r.movexy(200, 130);
  d := 5;
  uhol := Random(180);
  while r.dist(200, 130) < 120 do
  begin
    r.fd(d);
    r.lt(uhol);
    d := d+2;
  end;
```

Postupne sa budú objavovať rôzne špirály, napr.



Polohu robota môžeme nastaviť do stredu grafickej plochy pomocou hodnôt: Image1.Width a Image1.Height napr. takto:

```
uses
  RobotUnit, Math;

procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
  d, uhol, stredX, stredY, vzd: Integer;
begin
  Randomize;
  r := TRobot.Create;
  stredX := Image1.Width div 2;
  stredY := Image1.Height div 2;
  vzd := Min(stredX, stredY)-5;
  while True do
  begin
    cs;
    r.movexy(stredX, stredY);
    d := 5;
    uhol := Random(180);
    while r.dist(stredX, stredY) < vzd do
    begin
      r.fd(d);
      r.lt(uhol);
      d := d+2;
    end;
    wait(500);
  end;
end;
```

Nakoľko sme tu použili sme tu funkciu Min, do uses sme pripísali aj knižnicu Math.

Náhodné prechádzky

Necháme robota do "nekonečna" sa prechádzať po obrazovke: náhodne sa otočí o nejaký uhol a vykročí o nejakú konštantnú vzdialenosť. Nekonečný cyklus robíme pomocou stále pravdivého logického výrazu True:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
begin
  r := TRobot.Create(400, 300);
```

```
  while True do
  begin
    r.rt(Random(360));
    r.fd(5);
    wait(1);
  end;
end;
```

Zamyslite sa nad tým, čo sa stane a prečo, ak z nášho programu vyhodíme (zakomentujeme) príkaz wait(1).

Robot veľmi rýchlo ujde z plochy preč. Pokúsime sa preto strážiť robota tak, aby nevyšiel z nejakej oblasti, t.j. ak aj vyjde, tak ho ihneď vrátime späť. Použijeme nový príkaz vetvenia if (má dva varianty, zatiaľ ten jednoduchší):

if *podmienka* **then** *príkaz*

podobne ako while, ak je podmienka splnená vykoná sa vnorený príkaz, ak je hodnota podmienky nepravda, tak sa vnorený príkaz nevykoná ale preskočí.

Náhodná prechádzka - robot urobí krok späť, keď sa priveľmi vzdiali od štartového bodu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;
begin
  r := TRobot.Create(400, 300);
  while True do
  begin
    r.rt(Random(360));
    r.fd(5);
    if r.dist(400, 300) > 50 then
      r.fd(-5);
    wait(1);
  end;
end;
```

Vyskúšajte niekoľko variantov podmienky a trénujte aj kombinácie podmienok pomocou logických operátorov and a or (treba si uvedomiť, že podmienka v týchto príkladoch vyjadruje, že robot opustil stráženú oblasť, t.j. že je zle a treba sa vrátiť).

- kosoštvorec:

  ```
  if Abs(r.X-400)+Abs(r.Y-300) > 100 then r.fd(-5);
  ```

- obdĺžnik so stranami 2*100 a 2*50 a stredom v (400,300):

  ```
  if (Abs(r.X-400)>100) or (Abs(r.Y-300)>50) then ...
  ```

- zjednotenie troch kružníc (snehuliak):

  ```
  procedure TForm1.Button2Click(Sender: TObject);
  var
    r: TRobot;
  begin
    cs(clLtGray);
    r := TRobot.Create(400, 300);
    r.PC := clWhite;
    while True do
    begin
      r.rt(Random(360));
      r.fd(5);
      if (r.dist(400, 400) > 100) and
         (r.dist(400, 220) > 80) and
         (r.dist(400,80) > 60) then
  ```

```
      r.fd(-5);
    if Random(1000) = 0 then
      wait(1);
  end;
end;
```

Všimnite si, že aby sme urýchlili náhodné prechádzanie sa robota, príkaz wait(1); vykonávame približne len po každom 1000-om kroku robota. Mohli by sme na to zaviesť špeciálne počítadlo krokov robota a presne po jeho 1000 krokoch urobiť jeden wait(1), ale výsledok by bol prakticky rovnaký ako s if Random(1000) then ...

Robot sa môže pohybovať vo vnútri jednej kružnice ale tak, aby zároveň bol mimo druhej kružnice. Ak týmto dvom kružniciam dobre nastavíme stredy tak, aby sa prekrývali, môže vzniknúť mesiac (poexperimentujte s rôznymi polomermi kružníc):

```
procedure TForm1.Button3Click(Sender: TObject);
var
  r: TRobot;
begin
  cs(clNavy);
  r := TRobot.Create(400, 300);
  r.PC := clYellow;
  while True do
  begin
    r.rt(Random(360));
    r.fd(5);
    if (r.dist(440, 300) > 200) or (r.dist(600, 300) < 200) then
      r.fd(-5);
    if Random(1000) = 0 then
      wait(1);
  end;
end;
```

Posledná ukážka náhodnej prechádzky bude pre dvoch robotov: obaja sa pohybujú v tom istom kruhu, ale majú nastavenú inú farbu pera. Tiež sme im nastavili väčšiu hrúbku pera. Všimnite si, že oba roboty robia presne to isté, ale s iným náhodným uhlom:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  r1, r2: TRobot;
begin
  cs;
  r1 := TRobot.Create(400, 300);
  r1.PC := clGreen;
  r1.PW := 10;
  r2 := TRobot.Create(400, 300);
  r2.PC := clBlue;
  r2.PW := 10;
  while True do
  begin
    r1.rt(Random(360));
    r1.fd(5);
    if r1.dist(400, 300) > 200 then
      r1.fd(-5);
    r2.rt(Random(360));
    r2.fd(5);
    if r2.dist(400, 300) > 200 then
      r2.fd(-5);
    if Random(100) = 0 then
      wait(1);
  end;
end;
```

Ďalšie námety:

- náhodnú prechádzku upravte tak, aby sa robot pohyboval len v oblasti:
  - medzikružie - kružnice so spoločným stredom a rozdielnymi polomermi
  - slovenský dvojkríž ako zjednotenie troch obdĺžnikov
  - elipsa
- preprogramujte predchádzajúce príklady tak, aby robot nevystúpil zo stráženej oblasti a to tak, že so zdvihnutým perom otestuje krok dopredu a potom sa vráti späť. Ak to bolo OK, tak spraví krok so spusteným perom, inak nerobí nič a pokúša sa ísť iným smerom.

Procedúry - podprogramy

Podprogramom nazývame pomenovanie nejakého algoritmu, nejakej časti programu s tým, že túto časť môžme vyvolať pomocou tohto mena. Pomocou podprogramov môžeme riešiť problém jeho rozdelením na podúlohy - hovoríme tomu, že úlohu riešime zhora nadol. V pascale môžu byť podprogramy dvoch typov:

- procedúry - postupnosť príkazov
- funkcie - postupnosť príkazov - výsledkom je nejaká hodnota

Podprogramy sa najčastejšie používajú vtedy, keď

- nejaká časť programu sa opakuje na viacerých miestach - my ju zapíšeme len raz a používame viackrát;
- nejaká časť programu kvôli čitateľnosti alebo logike algoritmu je vytiahnutá do samostatnej časti - podprogramu - čím ju môžeme napr. lepšie odladiť;
- nejakú časť programu naprogramoval nejaký iný programátor - zabalil to do procedúry - a my to len používame;
- ak budeme potrebovať nejaký problém riešiť pomocou rekurzie, musíme na to použiť podprogram.

Deklarácie procedúr

```
procedure meno;
... // lokálne deklarácie pre procedúru
begin
... // telo procedúry
end;
```

Najlepšie to uvidíme na príklade: vytvoríme procedúru na kreslenie štvorca – všimnime si, že var r: TRobot musí byť deklarované ešte pred procedúrou, aby táto mohla pracovať s robotom. Teda procedúru stvorec sme vnorili do TForm1.Button1Click:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;

  procedure stvorec;
  var
    i: Integer;
  begin
    for i := 1 to 4 do
    begin
      r.fd(100);
      r.rt(90);
    end;
  end;

begin
  cs;
```

```
  r := TRobot.Create;
  stvorec;
  wait(500);
  r.setxy(500, 200);
  stvorec;
  wait(500);
  r.movexy(150, 150);
  stvorec;
end;
```

Podprogram sme zadefinovali raz a použili - volali - sme ho trikrát. Je veľmi dôležité správne pochopiť, ako pracuje počítač pri volaní podprogramu (pri vykonávaní programu sa príde na riadok s menom podprogramu):

1. zapamätá sa návratová adresa (kam sa bude treba vrátiť)
2. vytvoria sa **lokálne premenné** procedúry (so zatiaľ nedefinovanou hodnotou) - u nás je to premenná i - neskôr uvidíme, že tu sa vytvoria aj formálne parametre
3. prenesie sa riadenie programu do tela podprogramu (za príslušný begin)
4. vykonajú sa všetky príkazy podprogramu (až po koncový end)
5. zrušia sa lokálne premenné - "zabudne" sa premenná i
6. riadenie sa vráti za miesto v programe, odkiaľ bol podprogram volaný - napr. na riadok wait(500)

Pre pascal totiž platia tieto pravidlá:

- identifikátor (premennej, podprogramu, typu, objektu, ...) sa môže používať, až potom, keď bol zadeklarovaný
- štandardné deklarácie (štandardné typy, funkcie, podprogramy) – sú definované ako keby ešte pred programom, hovoríme, že sú v 0. úrovni
- prekrytie identifikátora s rovnakým menom je zakázané na tej istej úrovni, ale povolené vo vnorených úrovniach
    - napr. var Real: Integer; - čo je veľmi škaredé a robiť to radšej nebudeme
- identifikátor je viditeľný len na svojej úrovni a na všetkých pre neho vnorených úrovniach, kde nie je predefinovaný
- definované procedúry sú na vyšších úrovniach (sú vnorené) ako náš "program" – oni vidia deklarácie tohto programu (ak boli pred ich definíciou - napr. inštancia r) ale samotný program nevidí dovnútra procedúr
    - procedúra stvorec vidí premennú r (robot) ale program nevidí premennú i definovanú v tejto procedúre stvorec

V celom našom programe:

- na 0. úrovni sú všetky štandardné identifikátory a unit RobotUnit (teda aj typ TRobot, procedúry cs, wait a pod.)
- na 1. úrovni je definícia formuláru TForm1 ale aj procedure TForm1.Button1Click(Sender: TObject); do ktorej píšeme náš program
- na 2. úrovni je premenná r a procedúra stvorec
- na 3. úrovni je lokálna premenná i

Formálne parametre

Pri volaní podprogramu mu môžeme poslať nejaké hodnoty, aby sme nemuseli rôzne špeciality nastavovať v globálnych premenných (napr. veľkosť kresleného štvorca): v podprograme zadefinujeme špeciálne lokálne premenné, tzv. formálne parametre, do ktorých sa pred samotným volaním priradia (inicializujú) hodnoty skutočných parametrov. Procedúra stvorec s parametrom velkost - veľkosť strany štvorca:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  r: TRobot;

  procedure stvorec(velkost: Integer);  // velkost – formálny parameter
  var
    i: Integer;
  begin
    for i := 1 to 4 do
    begin
      r.fd(velkost);
      r.rt(90);
    end;
  end;

var
  j: Integer;
begin
  cs;
  r := TRobot.Create;
  j := 10;
  while j <= 400 do
  begin
    stvorec(j);
    r.lt(8);
    wait(50);
    inc(j, 5);          // j := j+5;
  end;
end;
```

Podprogram môže mať aj viac parametrov, buď rovnakého alebo rôznych typov:

```
procedure TForm1.Button12Click(Sender: TObject);
var
  r: TRobot;

  procedure poly(n, s, u: Integer);
  var
    i: Integer;
  begin
    for i := 1 to n do
    begin
      r.fd(s);
      r.rt(u);
    end;
  end;

begin
  cs;
  r := TRobot.Create;
  poly(5, 100, 144);
  r.movexy(100, 100);
  poly(360, 1, 1);
  r.movexy(200, 200);
  poly(90, 90, 90);
  r.movexy(300, 200);
  poly(3, -100, -120);
  r.movexy(200, 300);
  poly(90, 2, 1);
  r.movexy(200, 300);
  poly(90, 2, -1);
end;
```

Môžeme opraviť mechanizmus pri volaní procedúry:

1. zapamätá sa návratová adresa (kam sa bude treba vrátiť)

2. vytvoria sa **lokálne premenné** procedúry (s nedefinovanou hodnotou) a vytvoria sa aj **formálne parametre** ako lokálne premenné, ktoré sú v tele podprogramu naozaj obyčajnými lokálnymi premennými, ale majú inicializovanú hodnotu podľa skutočného parametra
3. prenesie sa riadenie programu do tela podprogramu
4. vykonajú sa všetky príkazy podprogramu
5. zrušia sa lokálne premenné (teda aj formálne parametre - samozrejme, že sa tu nezrušia skutočné parametre)
6. riadenie sa vráti za miesto v programe, odkiaľ bol podprogram volaný

Viac tlačidiel a tlačidlo s viac stavmi

Niekedy sa nám hodí, aby objekt robot bol v globálnej premennej: vytvoríme ho raz pri štarte programu a neskôr bude tento robot prístupný všetkým nasledujúcim akciám, napr. pod rôznymi tlačidlami:

```
uses
  RobotUnit;

var
  r: TRobot;  // robot je tu globálny, aby ho "videli" všetky procedúry

procedure TForm1.FormCreate(Sender: TObject);
begin
  r := TRobot.Create;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  cs;
end;

procedure poly(n, d, u: Integer);
begin
  while n > 0 do
  begin
    r.fd(d);
    r.rt(u);
    dec(n);
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  poly(360, 2, 1);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  poly(5, 150, 144);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  poly(4, 100, 90);
end;

...
```

Ešte ukážeme, ako využiť jediné tlačidlo, ktoré bude mať postupne rôzne funkcie - po každom zatlačení sa bude meniť stav tohto tlačidla:

```
var
```

```
  stav: Integer = 0;    // premenná stav je inicializovaná na 0
  r: TRobot;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Caption := 'štart';
end;

procedure poly(n, d, u: Integer);
begin
  while n > 0 do
  begin
    r.fd(d);
    r.rt(u);
    dec(n);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if stav = 0 then
  begin
    r := TRobot.Create;
    cs;
    Button1.Caption := 'kružnica';
    stav := 1;
  end
  else if stav = 1 then
  begin
    poly(360, 2, 1);
    Button1.Caption := 'hviezda';
    stav := 2;
  end
  else if stav = 2 then
  begin
    cs;
    poly(5, 150, 144);
    Button1.Caption := 'štvorec';
    stav := 3;
  end
  else if stav = 3 then
  begin
    cs;
    poly(4, 200, 90);
    stav := 4;
  end
  else if stav = 4 then
  begin
    poly(4, -200, 90);
    stav := 5;
  end
  else if stav = 5 then
  begin
    poly(4, 200, -90);
    stav := 6;
  end
  else if stav = 6 then
  begin
    poly(4, -200, -90);
    Button1.Caption := 'zmaž';
    stav := 7;
  end
  else if stav = 7 then
  begin
    cs;
    Button1.Caption := 'kružnica';
    stav := 1;
```

```
  end;
end;
```

Všimnite si spôsob formátovania vnorených if-príkazov. Tiež sme použili novinku: menili sme text na tlačidle pomocou

```
Button1.Caption := 'nový text';
```

Podobne môžeme nejaké tlačidlo aj zablokovať (nedá sa zatlačiť) alebo spätne odblokovať:

```
Button1.Enabled := False;       // zablokuje
Button1.Enabled := True;        // odblokuje
```

To isté môžeme zapísať oveľa elegantnejšie pomocou zloženého príkazu case - podľa nejakej hodnoty sa vykoná jeden z možných príkazov (alebo aj žiadny):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  case stav of
    0:
      begin
        r := TRobot.Create;
        cs;
        Button1.Caption := 'kružnica';
        stav := 1;
      end;
    1:
      begin
        poly(360, 2, 1);
        Button1.Caption := 'hviezda';
        stav := 2;
      end;
    2:
      begin
        cs;
        poly(5, 150, 144);
        Button1.Caption := 'štvorec';
        stav := 3;
      end;
    3:
      begin
        cs;
        poly(4, 200, 90);
        stav := 4;
      end;
    4:
      begin
        poly(4, -200, 90);
        stav := 5;
      end;
    5:
      begin
        poly(4, 200, -90);
        stav := 6;
      end;
    6:
      begin
        poly(4, -200, -90);
        Button1.Caption := 'zmaž';
        stav := 7;
      end;
    7:
      begin
        cs;
```

```
        Button1.Caption := 'kružnica';
        stav := 1;
      end;
  end;
end;
```

Všeobecný tvar príkazu case je

```
case výraz of
  konštanta1: príkaz1;
  konštanta2: príkaz2;
  ...
  konštantaN: príkazN;
  else príkazy;
end;
```

Pričom vetva else (podobne ako pri if) môže chýbať. Samotný výraz aj všetky konštanty môžu byť len celočíselného typu (mohli by byť aj logického typu, ale toto zrejme veľký význam nemá; neskôr uvidíme aj iné prípustné typy). Bodkočiarky treba dať za každú vetvu - tu ju odporúčame dávať aj pred else.

4. prednáška: textové súbory

čo už vieme:

- základný typ Integer - celé číslo
- podmienky v podmienenom príkaze if a cykle while

čo sa na tejto prednáške naučíme:

- nový jednoduchý typ Char a nový komplexnejší typ textový súbor
- ukážeme, ako sa pracuje s textovým súborom, aké tu platia pravidlá, ako sa riešia jednoduché úlohy
- ukážeme zopár úloh, v ktorých sa pracuje s robotmi a s textovými súbormi

Vráťme sa k procedúre poly z predchádzajúcej prednášky:

```
procedure poly(n, s, u: Integer);
var
  i: Integer;
begin
  for i := 1 to n do
  begin
    r.fd(s);
    r.rt(u);
  end;
end;

...
  poly(7, 100, 360 div 7);
```

a nakreslime pravidelný sedemuholník: keďže všetky parametre sú len celé čísla, tento nakreslený útvar sa v skutočnosti neuzavrie, lebo namiesto 51.43 uhla sa robot otáča o uhol 51 stupňov a po siedmich natočeniach je to len 357 namiesto očakávaných 360. S celočíselnou aritmetikou nevystačíme. Potrebujeme čísla s desatinnou časťou: v programovacích jazykoch ich nazývame reálne čísla a pracuje sa s nimi rovnako ako v matematike. Treba si ale uvedomiť, že reálna aritmetika v princípe nie je presná a teda často dáva nepresné výsledky - musíme sa s

tým naučiť pracovať a v citlivých situáciách počítať s chybami. Procedúru poly môžeme teraz zapísať pomocou reálneho typu (len parameter n musí byť celočíselný, dĺžka s a uhol u môžu byť reálne):

```
procedure poly(n: Integer; s, u: Real);
var
  i: Integer;
begin
  for i := 1 to n do
  begin
    r.fd(s);
    r.rt(u);
  end;
end;

...

  poly(7, 100, 360 / 7);
```

Jednoduché typy

V tejto prednáške zhrnieme, resp. sa zoznámime s týmito jednoduchými typmi:

- celočíselný typ – Integer
- logický typ – Boolean
- reálny typ – Real
- znakový typ - Char

Integer

S týmto typom pracujeme od prvej prednášky a vieme, že

- hodnoty sú z intervalu -2147483648 .. 2147483647
- zaberá 4 bajty = 32 bitov, 1bit znamienko
- definované sú aritmetické operácie +, -, *, div, mod – obidva operandy musia byť celé čísla
- priraďovací príkaz: na oboch stranách musia byť celé čísla
- relačné operácie <, <=, >, >=, <>, =
- je to **ordinálny typ** (skoro každá hodnota má nasledovníka a predchodcu):
  - funkcie: Pred, Succ - predchodca, nasledovník
  - funkcie High a Low – minimálna a maximálna hodnota typu
  - príkazy: Inc, Dec - zväčši, zmenši
  - môže sa použiť pre riadiacu premennú for-cyklu a v príkaze case
- ďalšie štandardné funkcie: Sqr, Abs, Sign
- šestnástkové konštanty majú predponu $, napr. $1e=30

Real

Ďalší číselný typ, ktorý sa trochu podobá na celočíselný:

- konštanty obsahujú desatinnú bodku a/alebo exponenciálnu časť, napr.
  - 3.14, -0.5, 3000000000.0, 3E9, 4.7E-3
- exponent v môže byť približne v rozsahu <-300, 300>
- presnosť výpočtov je približne na 15 desatinných miest, ale niektoré operácie nie sú až tak presné ...
- premenné typu Real zaberajú v počítači 8 bajtov,
- základné operácie sú +, -, *, /, pričom ak jeden z operátorov je celé číslo, automaticky sa konvertuje na reálne,

    - napr. ak zapíšeme
      1 + 1.5 // bude sa počítať 1.0 + 1.5
      1 / 3 // bude sa počítať 1.0 / 3.0
- automatická konverzia sa robí aj pri priradení: ak do reálneho čísla chceme priradiť celé číslo, toto sa automaticky konvertuje na reálne,
    - napr.
      rcislo := 2*5; // sa prekonvertuje na rcislo := 10.0;
- opačné priradenie (do celočíselnej premennej reálnu hodnotu) nie je povolené! Ak také niečo potrebujeme musíme použiť niektorú zaokrúhľovaciu funkciu.
- fungujú porovnávania <, <=, >, >=, <>, =
- reálny typ nie je ordinálny a preto ho nemôžeme použiť
    - ako riadiacu premennú for-cyklu (môžeme to zapísať while-cyklom),
    - ako hodnotu v podmienenom príkaze case,
    - ako parameter v Inc a Dec.
- existuje množstvo štandardných funkcií, napr.
    - Sqrt – odmocnina
    - Sqr – druhá mocnina
    - Sin, Cos, Tan – trigonometrické funkcie
    - Abs – absolútna hodnota
    - Round – zaokrúhľovanie – vráti celé číslo
    - Trunc – odtrhne desatinnú časť – vráti celé číslo
    - FloatToStr – vráti reťazec
    - StrToFloat – z reťazca spraví reálne číslo

Boolean

Logické hodnoty môžeme použiť nielen ako podmienky v príkazoch while a if, ale ich výsledok môžeme uložiť aj do premenných typu Boolean. Tieto premenné potom môžeme použiť v logických výrazoch a v rôznych podmienkach.

- konštanty - predefinované identifikátory konštanty True, False
- v pamäti zaberajú 1 bajt
- operácie: and, or, not, xor
- všetky relácie z predchádzajúcich typov dávajú výsledok logickú hodnotu, napr. výraz 5 < 10/3 má hodnotu False
- fungujú aj relácie, napr. platí
    - False < True, False <= True, True <= True
- priradenie – do logických premenných len logické hodnoty
- je to ordinálny typ a preto:
    - štandardné funkcie: Ord, Boolean(0 alebo 1)
      napr. Boolean(0) = False
    - fungujú aj Pred, Succ
      ale Pred(False) aj Succ(True) spôsobí chybu
    - funkcie High a Low – minimálna a maximálna hodnota
    - môžeme ho použiť pre premennú cyklu vo for-cykle

Char

Nový typ, ktorý slúži na prácu s jednotlivými znakmi.

- ordinálny typ, ktorý obsahuje sadu 256 znakov (tzv. **ASCII**), tieto sú usporiadané
- vnútorne sú reprezentované číselným kódom <0, 255>, t.j. zaberajú 1 bajt (8 bitov)
- znakové konštanty zapisujeme medzi apostrofy alebo pomocou #kód
- v Pascale nie sú žiadne znakové operácie (iba relácie)
- treba si pamätať:
  - ' ' < ... < '0' < '1' < ... < '9' < ... < 'A' 'A' < 'B' < ... < 'Z' < ... < 'a' < 'b' < ... < 'z'
- je to ordinálny typ a preto fungujú:
  - znakové funkcie: Pred, Succ, Ord, Char (to isté aj ako Chr)
  - príkazy Inc, Dec fungujú aj pre znaky (napr. c := 'A'; Inc(c, 32);)
  - dá sa robiť for-cyklus
  - funkcie High a Low – minimálna a maximálna hodnota
- treba si pamätať:
  - Ord(' ')=32;
  - Ord('0')=48;
  - Ord('1')=49;
  - ... #32=' '; #48='0'; #49='1';
  - ... Ord('A')=65; Ord('a')=Ord('A')+32=97;
  - ... #65='A'; #97='a'; ...
- ak c obsahuje znak cifry ('0'..'9'), tak Ord(c)-Ord('0') = cifra (podobne pre písmená)

Použitie znakového typu ukážeme na príklade výpisu do grafickej plochy:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  z: Char;
  x: Integer;
begin
  Image1.Canvas.Font.Height := 30;
  Image1.Canvas.Font.Name := 'Verdana';
  z := 'Q';
  Image1.Canvas.TextOut(50, 50, z);
  z := 'f';
  Image1.Canvas.TextOut(70, 50, z);
  x := 10;
  for z := 'a' to 'z' do
  begin
    Image1.Canvas.TextOut(x, 100, z);
    Inc(x, 16);
  end;
end;
```

Použili sme grafický príkaz TextOut, ktorý od nejakej súradnice (prvé dva parametre) vypíše text, ktorý je v treťom parametri.

Ďalší príklad využíva vlastnosť textov (znakových reťazcov - budeme sa učiť neskôr): texty môžeme spájať (zreťazovať) pomocou operácie +. Príklad vypíše prvých niekoľko písmen abecedy aj s ich zodpovedajúcimi kódmi ASCII.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  z: Char;
  y: Integer;
begin
  Image1.Canvas.Font.Height := 30;
  Image1.Canvas.Font.Name := 'Verdana';
  y := 10;
  for z := 'A' to 'K' do
```

```
  begin
    Image1.Canvas.TextOut(50, y, z + ' = #' + IntToStr(Ord(z)));
    Inc(y, 24);
  end;
end;
```

Pre všetky ordinálne typy môžeme premenné použiť ako premenné for-cyklu (počítadlo), napr.

```
var
  i, x: Integer;
  b: Boolean;
  z: Char;
begin
  for i := 1 to 100 do ...             // cyklus prejde 100-krát
  for b := False to True do ...        // cyklus prejde 2-krát
  for b := x=1 to x=2 do ...           // cyklus prejde 0, 1 alebo 2-krát
  for z := 'A' to 'Z' do ...           // cyklus prejde 26-krát
  for z:= Pred('#') to Succ('#') do ...// cyklus prejde 3-krát
```

Textová plocha

- aby sme mohli experimentovať so znakmi, s textovou informáciou, naučíme sa v Delphi pracovať s textovou plochou
- do prázdneho formulára položíme komponent Memo (trieda TMemo) - podobne ako grafickú plochu Image budeme ťahať aj túto plochu - vytvorí sa komponent s menom Memo1 (v prvom riadku sa objaví text Memo1)
- zatiaľ sa naučíme len niekoľko príkazov na prácu s textovou plochou a rozumieť im budeme až neskôr. Podobne ako sme písali Image1.Canvas. a za tým príkaz, budeme teraz písať Memo1.Lines. a za to príkaz:
  - Memo1.Lines.Clear; - vyčistí všetky riadky
  - Memo1.Lines.Append('nejaký text'); - pridá ďalší riadok s daným textom na koniec
  - Memo1.Lines.LoadFromFile(súbor); - do textovej plochy zapíše obsah súboru z disku
  - Memo1.Lines.SaveToFile(súbor); - obsah textovej plochy zapíše do súboru

Nasledujúca ukážka s jedným komponentom Memo1 vypisuje kódy znakov písmen:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  z: Char;
begin
  Memo1.Lines.Clear;
  for z := 'a' to 'z' do
    Memo1.Lines.Append(z + ' ' + IntToStr(Ord(z)));
    // k písmenu prilepí medzeru a číselný kód znaku
end;
```

Textové súbory TEXT

Súbor je postupnosť (sekvencia) prvkov rovnakého typu - najčastejšie sa nachádza na nejakom vonkajšom zariadení, napr. na disku.

Textový súbor je postupnosť riadkov (aj prázdna), pričom riadok textového súboru je postupnosť znakov (aj prázdna) ukončená špeciálnou značkou <Eoln> (z anglického **end of line**). Všetky údaje sú v textovom súbore zapísané ako ASCII znaky.

Vo všeobecnosti sa so súbormi dá pracovať sekvenčne alebo priamo:

- priamy prístup do súboru - určíme pozíciu (index) niektorého prvku v súbore a s týmto ďalej pracujeme,

napr. ho prečítame do pamäte alebo prepíšeme novým obsahom,
- sekvenčný prístup do súboru - s prvkami súboru môžeme pracovať len postupne od prvého až po posledný: postupne vieme údaje čítať, alebo postupne ich vieme do súboru zapisovať.

Textové súbory v pascale majú iba sekvenčný prístup a preto

- musí sa rozlišovať, či zo súboru čítame alebo do neho zapisujeme údaje
  - vstupný súbor - len čítať pripravené údaje
  - výstupný súbor - len zapisovať nové údaje na koniec súboru
- do textového súboru sa nedá aj zapisovať aj súčasne z neho čítať

Práca so súborom:

- najprv treba súbor "**otvoriť**"
- potom môžeme **pracovať** s jeho obsahom
- na záver ho treba "**zatvoriť**"

Ako pracujeme s textovým súborom

1. zadeklarovanie premennej typu TextFile:

```
var
  t: TextFile;     // premenná t je typu textový súbor
```

2. priradenie konkrétneho súboru:

```
AssignFile(t, meno_súboru);
```

   - dávajte si pozor na uvádzanie plnej cesty na disku k nejakému súboru - ak sa takýto program prenesie na iný počítač, s veľkou pravdepodobnosťou tieto absolútne cesty k súborom nebudú fungovať - používajte radšej relatívne cesty od momentálnej adresy projektu (tam, kde sa nachádza EXE súbor)

3. otvoriť súbor

   - Reset(t);   // otvorenie súboru na čítanie - súbor už musí existovať
   - ReWrite(t); // otvorenie súboru na zápis - ak už existuje, tak sa najprv vyprázdni

4. práca so súborom, t.j. samotné čítanie alebo zapisovanie:

   - Read(t, premenná); alebo Readln(t, ...);   // čítanie zo súboru - z pozície ukazovateľa
   - Write(t, hodnota); alebo Writeln(t, ...);   // zápis do súboru - na koniec

5. zatvorenie súboru, t.j. ukončenie práce so súborom:

```
CloseFile(t);
```

Ukazovateľ = pozícia v súbore - na začiatku je na 1. znaku; po každom Read aj Write sa posunie o 1 znak vpravo.

Čítanie zo súboru

Príkaz Read(t, znaková_premenná); prečíta jeden znak zo súboru z pozície ukazovateľa, priradí sa jeho hodnota do premennej a ukazovateľ sa posunie na nasledujúci znak.

Testovanie konca súboru a konca riadka

- štandardná logická funkcia Eof(t) = skratka z **End Of File**
  - vráti True, ak je ukazovateľ nastavený za posledným znakom súboru
- štandardná logická funkcia Eoln(t) = skratka z **End Of LiNe**
  - vráti True, ak je ukazovateľ na značke <Eoln> alebo aj za posledným znakom súboru, t.j. ak platí Eof(t)=True, tak platí Eoln(t)=True
  - značka <Eoln> sa vnútorne kóduje dvomi znakmi #13 a #10 (hovoríme im **CR** a **LF**)
  - príkazom Readln(t) preskočíme všetky znaky v súbore až za najbližšiu značku <Eoln> (na konci súboru nerobí nič)
  - POZOR, pomocou Read(t, z) je možné čítať aj značku <Eoln>, lenže táto sa potom chápe ako 2 znaky #13 a #10 a nie ako nejaký špeciálny znak
  - príkaz Readln(t,z) je skrátený tvar pre dvojicu príkazov Read(t, z); Readln(t);
- čítanie na konci súboru (za posledným znakom) môže vyvolať vstupno-výstupnú chybu
- znak #26 má v pascale (z historických dôvodov) niekedy špeciálny význam: čítanie textového súboru si na ňom "myslí", že je na konci súboru (Eof(t)=True) a nedovolí čítať ďalšie znaky za ním...
  - môžete to otestovať tak, že si vytvoríte textový súbor, ktorý bude niekde v strede obsahovať znak #26, potom tento súbor vypíšte pomocou Memo1.Lines.LoadFromFile(...) a tiež ho čítajte a vypisujte pomocou while not Eof(t) do begin Read(t, z); ...end;

V nasledujúcom príklade zistíme počet medzier v textovom súbore medzery.txt:

```
var
  t: TextFile;
  z: Char;
  pocet: Integer;
begin
  AssignFile(t, 'medzery.txt');
  Reset(t);
  pocet := 0;
  while not Eof(t) do
  begin
    Read(t, z);
    if z = ' ' then
      Inc(pocet);
  end;
  CloseFile(t);
  // výsledok vypíšeme do textovej plochy:
  Memo1.Lines.Append('Počet medzier v súbore ' + IntToStr(pocet));
end;
```

Zistíme, počet riadkov textového súboru text.txt:

```
var
  t: TextFile;
  pocet: Integer;
begin
  AssignFile(t, 'text.txt');
  Reset(t);
  pocet := 0;
  while not Eof(t) do
  begin
    Readln(t);
    Inc(pocet);
  end;
  CloseFile(t);
  Memo1.Lines.Append('Počet riadkov v súbore ' + IntToStr(pocet));
```

```
end;
```

Zistíme, dĺžku najdlhšieho riadka súboru text.txt

```
var
  t: TextFile;
  z: Char;
  max, dlzka: Integer;
begin
  AssignFile(t, 'text.txt');
  Reset(t);
  max := 0;
  while not Eof(t) do
  begin
    dlzka := 0;
    while not Eoln(t) do    // zistí dĺžku riadka
    begin
      Read(t, z);
      Inc(dlzka);
    end;
    Readln(t);              // Readln sa tu nesmie zabudnúť
    if dlzka > max then
      max := dlzka;
  end;
  CloseFile(t);
  Memo1.Lines.Append('Dĺžka najdlhšieho riadka ' + IntToStr(max));
end;
```

Zápis do súboru

- súbor treba otvoriť pomocou ReWrite(t)
- ak súbor už existoval, tak hneď po ReWrite sa zruší jeho obsah
- ak súbor ešte neexistoval, vytvorí sa s prázdnym obsahom
- ukazovateľ v súbore je vždy nastavený na jeho koniec
- príkaz Write(t, znak) zapíše do súboru jeden znak; Write(t, reťazec) zapíše kompletný reťazec
- príkaz Writeln(t) zapíše do súboru značku <Eoln>, t.j. robí to isté ako Write(t, #13#10)
- príkaz Writeln(t, reťazec) je skrátený tvar pre Write(t, reťazec); Writeln(t); alebo aj Write(t, reťazec, #13#10);
- v nasledujúcom príklade využijemee
  - Memo1.Lines.LoadFromFile(meno_súboru)
- pomocou ktorého zapíšeme obsah celého súboru do textovej plochy

Vytvoríme súbor text.txt z písmen 'a' až 'z' a potom jeho obsah vypíšeme do textovej plochy:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  z, z1: Char;
begin
  AssignFile(t, 'text.txt');
  ReWrite(t);
  for z := 'a' to 'z' do
  begin
    for z1 := z to 'z' do
      Write(t, z1);
    Writeln(t);
  end;
  CloseFile(t);

  Memo1.Lines.LoadFromFile('text.txt');
```

```
end;
```

Vytvoríme kópiu súboru unit1.pas do súboru text.txt

```
var
  t1, t2: TextFile;
  z: Char;
begin
  AssignFile(t1, 'unit1.pas');
  Reset(t1);
  AssignFile(t2, 'text.txt');
  ReWrite(t2);
  while not Eof(t1) do
    if Eoln(t1) then
    begin
      Readln(t1);
      Writeln(t2);
    end
    else
    begin
      Read(t1, z);
      // spracuj prečítaný znak
      Write(t2, z);
    end;
  CloseFile(t1);
  CloseFile(t2);
  Memo1.Lines.LoadFromFile('text.txt');
end;
```

iný variant kopírovania súboru - nevšímame si konce riadkov - pritom prerábame malé písmená na veľké:

```
var
  t1, t2: TextFile;
  z: Char;
begin
  AssignFile(t1, 'unit1.pas');
  Reset(t1);
  AssignFile(t2, 'text.txt');
  ReWrite(t2);
  while not Eof(t1) do
  begin
    Read(t1, z);
    if (z >= 'a') and (z <= 'z') then
      z := Char(Ord(z)-Ord('a')+Ord('A'));          // alebo dec(z,32);
    Write(t2, z);
  end;
  CloseFile(t1);
  CloseFile(t2);
  Memo1.Lines.LoadFromFile('text.txt');
end;
```

Pozn:

- na prerábanie malých písmen na veľké môžeme použiť štandardnú znakovú funkciu z := Upcase(z);

Ďalšie námety:

- postupnosť medzier nahraď 1 medzerou
- vyhoď riadky, ktoré sú prázdne alebo obsahujú len medzery
- vyhoď medzery na konci riadkov
- postupnosť znakov 'end' nahraď '***'

Reset a ReWrite na ten istý súbor na disku

```
var
  f, g: TextFile;
begin
  AssignFile(f, 'a.txt');
  Reset(f);
  AssignFile(g, 'a.txt');
  ReWrite(g);
  ...
```

je to nepredvídateľné - rôzne verzie pascalu reagujú rôzne, napr. Delphi hlási I/O chybu

Čítanie čísel

- pomocou Read môžeme čítať aj čísla (celé aj reálne), ale v súbore musia byť tieto čísla ukončené medzerou, koncom riadka alebo tabulátorom (znak s kódom #9)
- príkaz Read(t, číselná_premenná); najprv preskočí všetky medzerové znaky (medzera, <Eoln> alebo #9), potom prekonvertuje znaky zo vstupu na číslo a ak je číslo ukončené nemedzerovým znakom (napr. ',' alebo ';'), tak vyhlási chybu **Invalid numeric format**
- čítanie čísla na konci súboru (t.j. platí Eof, ale aj ak sú tam len medzery) vráti hodnotu 0 - treba sa tohoto vyvarovať!
- profesionálny softvér takýto Read na čítanie čísel nepoužíva, lebo chyba v súbore spôsobí chybovú správu (výpočet je ďalej nekorektný) - vy môžete používať takéto čítanie, len ak je v zadaní výslovne povedané, že je súbor korektný a číslo je ukončené medzerovým znakom
- neskôr uvidíme aj iný (bezpečný) spôsob čítania čísel

Program nájde maximálne číslo v súbore celých čísel:

```
var
  t: TextFile;
  cislo, max: Integer;
begin
  AssignFile(t, 'text.txt');
  Reset(t);
  max := -MaxInt;          // lepšie by bolo Low(Integer)
  while not Eof(t) do      // vyskúšajte SeekEof
  begin
    Read(t, cislo);
    if cislo > max then
      max := cislo;
  end;
  CloseFile(t);
  if max = -MaxInt then
    Memo1.Lines.Append('súbor je prázdny')
  else
    Memo1.Lines.Append('maximálne číslo v súbore je ' + IntToSTr(max));
end;
```

- ak súbor obsahoval napr. len prázdny riadok (alebo len medzery), tak program vypíše, že maximum bolo 0
- toto isté sa stane, ak súbor obsahuje len záporné čísla a za posledným číslom sú ešte nejaké medzerové znaky - funkcia Eof(t) za posledným číslom vráti False - ešte nie je koniec súboru, ale už tam nie je žiadne číslo, teda prečíta sa hodnota 0
- v takejto situácii môžeme namiesto Eof(t) použiť štandardnú funkciu SeekEof(t), ktorá skôr ako odpovie na stav súboru, odfiltruje všetky medzerové znaky
- podobne existuje SeekEoln(t), ktorá testuje koniec riadka, ale najprv odfiltruje medzery a tabulátory (znaky s kódom #9)
- Pozn. Pri použití funkcie SeekEoln(t) a SeekEof(t) sa odignorujú všetky medzery, tabulátory a v prípade ; i

konce riadkov. Preto tieto funkcie nie sú veľmi vhodné pri čítaní znakov, používame ich hlavne pri čítaní čísel z textového súboru.

Zápis čísel

- pomocou Write(t, ...) môžeme do textového súboru zapisovať aj čísla (hodnoty číselných výrazov)
- celé čísla sa zapíšu bez medzery pred číslom aj za číslom, t.j. Write(t, i,i+1) pre i=17 zapíše 1718
- reálne čísla sa do súboru zapisujú v semilogaritmickom tvare s medzerou pred číslom

Príklad: Zo súboru text1.txt budeme kopírovať všetky čísla do súboru text2.txt, pričom ich budeme zaraďovať po troch do riadka:

```
var
  t1, t2: TextFile;
  cislo, pocet: Integer;
begin
  AssignFile(t1, 'text1.txt');
  Reset(t1);
  AssignFile(t2, 'text2.txt');
  ReWrite(t2);
  pocet := 0;
  while not SeekEof(t1) do
  begin
    Read(t1, cislo);
    if pocet=3 then
    begin
      Writeln(t2);
      pocet := 1;
    end
    else
    begin
      if pocet>0 then
        Write(t2, ' ');
      Inc(pocet);
    end;
    Write(t2, cislo);
  end;
  CloseFile(t1);
  CloseFile(t2);
  Memo1.Lines.LoadFromFile('text2.txt');
end;
```

V súbore sú reálne čísla, máme zistiť počet čísel, ktoré majú hodnotu menšiu ako je priemer všetkých čísel v súbore

```
var
  t, tt: TextFile;
  cislo, suma, priemer: Real;
  pocet: Integer;
begin
  AssignFile(t, 'text1.txt');
  Reset(t);
        // priradenie vstupného súboru a jeho otvorenie na čítanie
  suma := 0;
  pocet := 0;
  while not SeekEof(t) do begin
    Read(t, cislo);
    suma := suma+cislo;
    Inc(pocet);
  end;
  Reset(t);        // !!! nastavenie pozície na začiatok súboru !!!
  AssignFile(tt, 'text2.txt');
  ReWrite(tt);
```

```
  priemer := suma/pocet;
  pocet := 0;
  // premennú pocet využijeme na zistenie počtu podpriemerných čísel
  while not SeekEof(t) do
  begin
    Read(t, cislo);
    if cislo < priemer then
    begin
      Inc(pocet);
      Writeln(tt, cislo:0:2);
    end;
  end;
  CloseFile(t);
  CloseFile(tt);
  Memo1.Lines.Append('počet podpriemerných=' + IntToSTr(pocet));
end;
```

Formátovací parameter vo Write

formátovací parameter za znakom alebo znakovým reťazcom

Write(t, '*':10);

- označuje, že znak sa vypíše na šírku 10, t.j. najprv 9 medzier a potom '*'

Write(t, 'delphi':3);

- nakoľko reťazec je dlhší ako formátovací parameter, zapíše sa kompletný reťazec, t.j. ignoruje sa formát

formátovací parameter za celým číslom označuje šírku, do ktorej sa má zapísať číslo, ak by nevošlo do danej šírky, formát sa ignoruje

Write(t, 25*25:5);

- zapíše dve medzery, za ktoré dá číslo 625

formátovací parameter za reálnym číslom tiež označuje šírku, číslo sa vypíše v semilogaritmickom tvare; druhý formátovací parameter označuje počet desatinných miest

Write(t, Sin(2):15);

- zapíše 9.092974E-0001

Write(t, Cos(2):7:4);

- zapíše -0.4161

Textové súbory a roboty

Pripomeňme príkaz case, ktorý pre ordinálnu hodnotu zabezpečí vykonanie nejakej vetvy podľa príslušnej konštanty – všeobecne:

```
case ordinálny_výraz of
  konštanta1: príkaz1;
  konštanta2: príkaz2;
  ...
  else príkazy;   // táto vetva môže chýbať
end;
```

Predpokladajme, že máme daný súbor, v ktorom sú príkazy pre robota: f (pre fd), l (pre lt), r (pre rt) pričom za

každým písmenom je jedno celé číslo, napr.

f 100 r 120 f 100 r 120 f 100

pričom písmená a čísla sú navzájom oddelené aspoň jednou medzerou alebo novým riadkom

robot interpretuje tento súbor:

```
var
  t: TextFile;
  r: TRobot;
  z: Char;
  p: Integer;
begin
  AssignFile(t, 'robot1.txt');
  Reset(t);
  r := TRobot.Create;
  while not Eof(t) do
  begin
    if Eoln(t) then
    begin
      Readln(t);
      z := ' ';
    end
    else
      Read(t, z);
    if z<>' ' then
      Read(t, p);
    case z of
      'f': r.fd(p);
      'r': r.rt(p);
      'l': r.lt(p);
    end;
  end;
  CloseFile(t);
end;
```

vylepšená verzia s použitím SeekEof(t):

```
var
  t: TextFile;
  r: TRobot;
  z: Char;
  p: Integer;
begin
  AssignFile(t, 'robot1.txt');
  Reset(t);
  r := TRobot.Create;
  while not SeekEof(t) do
  begin
    Read(t, z, p);
    case z of
      'f': r.fd(p);
      'r': r.rt(p);
      'l': r.lt(p);
    end;
  end;
  CloseFile(t);
end;
```

v súbore sú slová forward, right, left, pu, pd (z každého aspoň prvé dve písmená) – za niektorými nasleduje číslo:

```
var
  t: TextFile;
  r: TRobot;
```

```
  z, z1, z2: Char;
  p: Integer;
begin
  AssignFile(t, 'robot1.txt');
  Reset(t);
  r := TRobot.Create;
  while not SeekEof(t) do
  begin
    Read(t, z, z1, z2);
    while (z2 <> ' ') and not Eoln(t) do
      Read(t, z2);
    if (z = 'f') and (z1 = 'o') then
    begin
      Read(t, p);
      r.fd(p);
    end
    else if (z = 'r') and (z1 = 'i') then
    begin
      Read(t, p);
      r.rt(p);
    end
    else if (z = 'l') and (z1 = 'e') then
    begin
      Read(t, p);
      r.lt(p);
    end
    else if (z = 'p') and (z1 = 'u') then
      r.pu
    else if (z = 'p') and (z1 = 'd') then
      r.pd;
  end;
  CloseFile(t);
end;
```

Pozn.

- riešenie tejto úlohy môžeme zjednodušiť, ak si budeme všímať len 2. písmeno slov - v tomto prípade by sa dal použiť príkaz case

Príklad

- Počas kreslenia nejakého obrázka pomocou grafického robota (napr. kvetinky) si budeme do textového súboru zapamätávať momentálne súradnice robota. Súradnice robota sú v stavových premenných r.X a r.Y a zrejme ich stačí ukladať len po príkaze fd
- Najprv vytvoríme súbor s postupnosťou súradníc:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  r: TRobot;

  procedure poly(n, d, u: Integer);
  begin
    while n > 0 do
    begin
      r.fd(d);
      Writeln(t, r.X:0:2, ' ' , r.Y:0:2);
      r.rt(u);
      Dec(n);
    end;
  end;

var
  i: Integer;
```

```
begin
  AssignFile(t, 'robot2.txt');
  ReWrite(t);
  r := TRobot.Create;
  poly(1, 100, 0);
  for i:=1 to 7 do
  begin
    poly(9, 5, 10);
    r.rt(90);
    poly(9, 5, 10);
    r.rt(90 + 360/7);
  end;
  CloseFile(t);
end;
```

A teraz robot interpretuje súbor z predchádzajúceho príkladu:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  t: TextFile;
  r: TRobot;
  nx, ny: Real;
begin
  cs;
  AssignFile(t, 'robot2.txt');
  Reset(t);
  r := TRobot.Create;
  while not Eof(t) do
  begin
    Readln(t, nx, ny);
    r.setxy(nx, ny);
  end;
  CloseFile(t);
end;
```

Tento program prepíšeme tak, že do súboru sa ukladajú relatívne posuny robota, t.j. postupnosť vektorov:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  r: TRobot;
  x0, y0: Real;

  procedure poly(n, d, u: Integer);
  begin
    while n > 0 do
    begin
      r.fd(d);
      Writeln(t, r.X-x0:0:2, ' ' , r.Y-y0:0:2);
      x0 := r.X;
      y0 := r.Y;
      r.rt(u);
      Dec(n);
    end;
  end;

var
  i: Integer;
begin
  AssignFile(t, 'robot3.txt');
  ReWrite(t);
  r := TRobot.Create;
  x0 := r.X;
  y0 := r.Y;
  poly(1, 100, 0);
```

```
  for i:=1 to 7 do
  begin
    poly(9, 5, 10);
    r.rt(90);
    poly(9, 5, 10);
    r.rt(90 + 360/7);
  end;
  CloseFile(t);
end;
```

Robota vygenerujeme na náhodnej pozícii a určíme náhodnú veľkosť obrázka (náhodný interval <0.3, 2.5>):

```
procedure TForm1.Button2Click(Sender: TObject);
var
  t: TextFile;
  r: TRobot;
  nx, ny, koef: Real;
  sirka, vyska: Integer;
begin
  AssignFile(t, 'robot3.txt');
  Reset(t);
  Randomize;
  sirka := Image1.Width;
  vyska := Image1.Height;
  r := TRobot.Create(Random(sirka), Random(vyska-100)+100);
  koef := Random(23)/10+0.3;
  while not Eof(t) do
  begin
    Readln(t, nx, ny);
    r.setxy(r.X+nx*koef, r.Y+ny*koef);
  end;
  CloseFile(t);
end;
```

Ak je koef = 1, tak sa obrázok nakreslí v pôvodnej veľkosti, ak je koef < 1, tak bude zmenšený a ak koef > 1, tak bude zväčšený.

5. prednáška: funkcie, znakové reťazce

čo sme sa doteraz naučili

- poznáme procedúry s parametrami - vieme mechanizmus volania podprogramu
- okrem jednoduchých typov Integer, Real, Boolean a Char poznáme aj zložený typ **textový súbor**, ktorý je reprezentuje postupnosť znakov

čo sa budeme dnes učiť

- zoznámime sa s rôznymi typmi formálnych parametrov
- naučíme sa nový zložený typ **znakový reťazec**, ktorý tiež reprezentuje postupnosť znakov

Parametre procedúr

Začneme takouto úlohou: napíšeme program, v ktorom sa 2 roboty budú stále pohybovať po svojich kruhoch. Súčasne s nimi sa tretí robot bude stále snažiť byť presne v strede medzi nimi (ako keby bol v strede gumenej nite, ktorá je pripevnená na týchto dvoch robotoch).

- Vytvoríme si pomocnú procedúru pocitajStred, ktorá z dvoch robotov r1 a r2 vypočíta súradnice stredu (x, y) pre tretieho robota:

Tretí robot je vždy medzi ďalšími dvoma:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2, r3: TRobot;
  x, y: Real;

  procedure pocitajStred;
  begin
    x := (r1.X+r2.X)/2;
    y := (r1.Y+r2.Y)/2;
  end;

begin
  r1 := TRobot.Create(300, 250, 27);
  r2 := TRobot.Create(200, 200, 0);
  pocitaj_stred;
  r3 := TRobot.Create(x, y, 0);
  r3.PC := clRed;
  while True do
  begin
    r1.fd(4);
    r1.rt(3);
    r2.fd(3);
    r2.lt(2);
    pocitajStred;
    r3.setxy(x, y);
    wait(10);
  end;
end;
```

V tomto príklade sme vytvorili procedúru pocitajStred bez formálnych parametrov, lebo táto "vidí" aj na roboty r1, r2 aj na premenné x a y. Tieto premenné sú pre ňu **globálne** (nie sú to jej **lokálne** premenné). Tento spôsob, keď procedúra používa globálne premenné, nie je vždy použiteľný, lebo niekedy môžeme potrebovať vypočítať stred iných dvoch robotov a výsledok dať do iných dvoch premenných, alebo samotnú procedúru chceme mať "**globálnu**" a táto nebude vidieť na potrebné premenné (lokálne v našej procedúre Button1Click).

Úlohu preprogramujeme tak, aby sa nepoužili "globálne" premenné, ale formálne parametre:

```
procedure TForm1.Button1Click(Sender: TObject);

  procedure pocitajStred(r1, r2: TRobot; xx, yy: Real);
  begin
    xx := (r1.X+r2.X)/2;
    yy := (r1.Y+r2.Y)/2;
  end;

var
  r1, r2, r3: TRobot;
  x, y: Real;
begin
  r1 := TRobot.Create(300, 250, 27);
  r2 := TRobot.Create(200, 200, 0);
  pocitajStred(r1, r2, x, y);
  r3 := TRobot.Create(x, y, 0);
  r3.PC := clRed;
  while True do
  begin
    r1.fd(4);
    r1.rt(3);
    r2.fd(3);
    r2.lt(2);
    pocitajStred(r1, r2, x, y);
    r3.setxy(x, y);
    wait(10);
  end;
end;
```

Takto zapísaný **program nebude fungovať**, lebo formálne parametre xx a yy sú "lokálne" premenné, ktoré sú inicializované hodnotami skutočných parametrov a po skončení procedúry sa ich hodnoty "zabúdajú". Doteraz známy mechanizmus formálnych parametrov nám v tomto prípade nepomôže - potrebujeme niečo nové.

Formálne parametre v pascale

Existujú rôzne prístupy k informáciám - predstavme si napr. diár, pre ktorý majú rôzne osoby rôzne prístupové práva:

- vlastník - zápis, čítanie
- 1. sekretárka - prístup k duplikátu - čítanie, modifikácia kópie
- 2. sekretárka - len čítanie
- ostatní - žiadny prístup

Typy prístupov k informáciám:

- úplný - čítanie, zápis
- len čítanie
- len zápis
- úplný prístup k duplikátu
- žiaden

Poznáme štandardné procedúry na prácu so súbormi:

- procedúra Read(t, x)
  - prístup k súboru t na čítanie
  - k premennej x na zápis (pôvodný obsah premennej nás nezaujíma)
- procedúra Write(t, x)
  - prístup k súboru t na zápis
  - k premennej x na čítanie

3 typy parametrov v Pascale

- prístup k duplikátu - hodnotové parametre, volanie hodnotou (mali sme doteraz)
- úplný prístup - premenné parametre, volanie adresou (var parametre)
- prístup len na čítanie - konštantné parametre (hodnota formálneho parametra sa nezmení) (const parametre)

V predchádzajúcom príklade sme procedúre pocitajStred poslali všetky parametre ako hodnotové, t.j. použili **volanie hodnotou**. Čo sa teda stane:

- vypočítajú sa hodnoty skutočných parametrov, s ktorými je procedúra volaná (v tomto prípade r1, r2 a momentálne hodnoty x a y)
- odovzdá sa riadenie procedúre, pričom sa pre ňu vytvoria lokálne premenné r1, r2, xx, yy a do týchto sa priradia hodnoty, ktoré sa pri volaní vypočítali
- s robotmi (grafickými perami) nebude problém, lebo nepotrebujeme meniť hodnoty robotov, problém bude s premennými x a y:
    - tu by sa nám viac hodil úplný prístup k týmto premenným, t.j. potrebovali by sme volanie adresou
    - v Pascale sa to zapíše rezervovaným slovom var pred príslušným parametrom

korektné riešenie predchádzajúcej úlohy:

```
procedure TForm1.Button1Click(Sender: TObject);

  procedure pocitajStred(r1, r2: TRobot; var xx, yy: Real);
  begin
    xx := (r1.X+r2.X)/2;
    yy := (r1.Y+r2.Y)/2;
  end;

...
```

Teraz zhrňme všetky doterajšie poznatky o tom ako pracuje počítač pri volaní procedúry. Keď sa objaví v programe meno procedúry (musela byť definovaná v deklaráciách), ide o volanie tejto procedúry, t.j.

1. zapamätá sa návratová adresa (kam sa bude treba vrátiť)
2. vytvoria sa **lokálne premenné** procedúry:
    - naozajstné lokálne premenné dostávajú nedefinovanú hodnotu
    - aj **hodnotové formálne parametre** sú lokálne premenné - len sú inicializované hodnotami skutočných parametrov (duplikát)
    - **premenné parametre** sa nikde nevytvárajú - sú to len dočasné nové mená premenných - skutočných parametrov
3. prenesie sa riadenie programu do tela podprogramu
4. vykonajú sa všetky príkazy podprogramu
5. zrušia sa lokálne premenné - teda aj nové hodnoty hodnotových formálnych parametrov
6. riadenie sa vráti za miesto v programe, odkiaľ bol podprogram volaný

Ukážme si teraz inú verziu tejto úlohy, kde jeden z robotov chodí po obvode štvorca a druhá po kružnici. Všimnite si, že procedúru pocitajStred sme vytiahli pred Button1Click, takže sa stala globálnou:

```
procedure pocitajStred(r1, r2: TRobot; var xx, yy: Real);
begin
  xx := (r1.X+r2.X)/2;
  yy := (r1.Y+r2.Y)/2;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2, r3: TRobot;
  x, y: Real;
  i: Integer;
begin
  r1 := TRobot.Create(300, 250, 27);
  r2 := TRobot.Create(200, 200, 0);
  pocitajStred(r1, r2, x, y);
  r3 := TRobot.Create(x, y, 0);
  r3.PC := clRed;
  while True do
  begin
    for i := 1 to 30 do    // skúste iný počet opakovaní, napr. 20
    begin
      r1.fd(4);
      r2.fd(3);
      r2.lt(2);
      pocitajStred(r1, r2, x, y);
      r3.setxy(x, y);
      wait(10);
    end;
    r1.rt(90);   // skúste iné uhly: 120, 144 a pod.
  end;
end;
```

Funkcie

Máme procedúru, ktorá vráti pomocou parametra minimum z dvoch čísel:

```
procedure min(a, b: Integer; var m: Integer);
begin
  if a < b then
    m := a
  else
    m := b;
end;
```

Podprogram môžeme definovať v tvare funkcie:

- v hlavičke podprogramu musíme okrem formálnych parametrov zadať aj typ funkcie
- v tele funkcie sa musí do špeciálnej lokálnej premennej Result priradiť nejaká hodnota - táto bude výsledkom funkcie
- lokálna premenná Result je automaticky už zadeklarovaná (my ju **nesmieme** deklarovať) a je rovnakého typu ako je typ funkcie
- POZOR! táto premenná má na začiatku **nedefinovanú** hodnotu, t.j. ak do nej nič nepriradíme, výsledkom bude nezmysel...

a teraz min ako funkcia:

```
function min(a, b: Integer): Integer;
begin
  if a < b then
    Result := a
  else
    Result := b;
  // v starom pascale bolo min := a ... ale my toto používať nebudeme!
end;
```

Príklady:

Logická funkcia jecifra, ktorá vracia True, ak je vstupný parameter cifrou:

```
function jecifra(z: Char): Boolean;
begin
  if (z >= '0') and (z <= '9') then
    Result := True
  else
    Result := False;
end;
```

túto funkciu zapíšeme elegantnejšie:

```
function jecifra(z: Char): Boolean;
begin
  Result := (z >= '0') and (z <= '9');
end;
```

Celočíselná funkcia, ktorá vracia hodnotu načítanej cifry (inak -1):

```
function cifra(z: Char): Integer;
begin
  if jecifra(z) then
    Result := Ord(z)-Ord('0')  // Ord('0') = 48
  else
    Result := -1;
end;
```

Logická funkcia, ktorá vracia True, ak je vstupný parameter písmenom:

```
function pismeno(z: Char): Boolean;
begin
  Result := (z >= 'a') and (z <= 'z') or (z >= 'A') and (z <= 'Z');
end;
```

Logická funkcia Odd, ktorá vracia True, ak je celé číslo, ktoré je jej parametrom, nepárne (pozn.: Odd je už preddefinovaná ako štandardná funkcia):

```
function Odd(x: Integer): Boolean;
begin
  Result :=  x mod 2 = 1;
end;
```

Celočíselná funkcia, ktorá vracia súčet prvých N celých čísel:

```
function suma(N: Integer): Integer;
begin
  Result := 0;    // zrejme by fungovalo aj Result := n*(n+1) div 2;
  while N > 0 do
  begin
    Result := Result+N;
    Dec(N);
  end;
end;
```

Funkcia, ktorá vracia znak zo vstupu, pričom vracia ' ' ak bol Eoln, '#' ak bol Eof (znaková premenná z je globálna):

```
var
  z: Char;
```

```
function znak(var t: TextFile): Char;
begin
  if eof(t) then
    z := '#'
  else if eoln(t) then
  begin
    z := ' ';
    Readln(t);
  end
  else
    Read(t, z);
  Result := z;
end;
```

Poznámka:

- všimnite si použitie textového súboru ako formálneho parametra - zapamätajte si, že textový súbor ako parameter musí byť vždy typu **var-parameter**!

Zaujímavá je potom časť programu, ktorá preskakuje všetky medzery a konce riadkov:

```
while znak(t) = ' ' do;
```

Príklad s počítaním mocniny čísla. Najprv uvedieme chybné riešenie:

```
function mocnina(x, K: Integer): Integer;
begin
  Result := 1;
  while K > 0 do
  begin
    Result := Result*x;
    Dec(K);
  end;
end;
```

Táto funkcia občas vráti nesprávny výsledok, pričom my nevieme, či momentálny výsledok je dobrý. Preto pridáme ďalší parameter (typu var), v ktorom vrátime informáciu, či nepretiekol výsledok:

```
function mocnina(x, K: Integer; var ok: Boolean): Integer;
var
  hranica: Integer;
begin
  Result := 1;
  hranica := MaxInt div x;
  while (K > 0) and (Result <= hranica) do
  begin
    Result := Result*x;
    Dec(K);
  end;
  ok := K=0;
end;
```

Funkcia random2 – náhodné číslo zo zadaného intervalu:

```
function random2(a, b: Integer): Integer;
begin
  Result := Random(b-a+1)+a;
end;
```

Funkcia dist2 – vzdialenosť dvoch robotov:

```
function dist2(r1, r2: TRobot): Real;
begin
  Result := Sqrt(Sqr(r1.X-r2.X) + Sqr(r1.Y-r2.Y));
  // alebo Result := r1.dist(r2.X, r2.Y);
end;
```

Ďalšie námety:

- celočíselná funkcia, ktorá počíta NSD dvoch prirodzených čísel Euklidovým algoritmom
- funkcia zistí, či dané číslo je prvočíslo
- procedúra SKRAT(a, b, p, q) s celočíselnými parametrami (b<>0), ktorá skráti zlomok a/b na základný tvar p/q
- dve prirodzené čísla sa nazývajú "priateľské", ak je každé z nich rovné súčtu všetkých deliteľov druhého bez neho samého (také sú napr. čísla 220 a 284); vypíšte všetky páry "priateľských" čísel, ktoré nepresahujú zadané prirodzené číslo

Znakové reťazce

Znakový reťazec je údajový typ, ktorý obsahuje nejakú **postupnosť znakov** (Char). Je to niečo veľmi podobné textovému súboru (TextFile). Lenže v znakovom reťazci sú údaje uložené v premennej (v pamäti) a môžem ich jednoducho modifikovať - naproti tomu v súbore sú údaje najčastejšie uložené na disku a ich zmena je algoritmicky trochu náročnejšia.

Znaky v postupnosti sú očíslované od 1 až po momentálnu dĺžku reťazca. Túto dĺžku zistíme pomocou štandardnej funkcie Length. Delphi si uchováva dĺžku v 4 bajtoch, t.j. teoreticky by maximálna dĺžka mohla byť 4 gigabajty - v skutočnosti je obmedzená možnosťami Windows: možno len 1 gigabajt - otestujte to na vašom počítači.

Premennú typu znakový reťazec deklarujeme takto:

```
var s: String;
```

Premenná s môže mať zatiaľ nedefinovanú hodnotu a preto jej musíme niečo priradiť, napr. reťazcovú konštantu:

```
s := 'reťazec';
```

Premenná s teraz obsahuje postupnosť znakov dĺžky 7: prvý znak je 'r', druhý 'e', atď. T.j. hodnota funkcie Length(s) je teraz 7. Reťazcové konštanty sú uzavreté rovnako ako znakové konštanty v apostrofoch. Premenná typu String môže obsahovať aj prázdny reťazec:

```
s := '';
```

Vtedy je jej dĺžka 0.

Môžeme pracovať aj s jednotlivými prvkami postupnosti, t.j. s jednotlivými znakmi v reťazci. Napr.

```
s := 'abcdef';
s[3] := '*';
```

Najprv sme do s priradili 5-znakový reťazec a potom sme v ňom zamenili 3-tí znak (znak 'c') za hviezdičku '*'. **Nesmieme** sa pritom odvolávať na znaky mimo rozsahu <1, momentálna dĺžka>, napr.

```
s[10] := '+';   // reťazec má zatiaľ dĺžku len 6
```

spôsobí chybovú správu.

Okrem priradení môžeme reťazce zapisovať do textového súboru a tiež ich môžeme zo súboru čítať. Napr.

```
Write(t, premenná_typu_String );   // výpis hodnoty premennej
Read(t, premenná_typu_String );    // prečítanie riadka zo vstupu do premennej
```

Znakové reťazce môžeme porovnávať pomocou relačných operácií (=, <>, <, >, <=, >=). Reťazce sú usporiadané pomocou tzv. **lexikografického usporiadania**. Napr. platí

```
'abc'  > 'ABC'
'Adam' < 'Eva'
'jana' < 'jano'
'Jana' < 'jana'
```

Postupne sa pri tom porovnáva znak za znakom. Kým sú v oboch reťazcoch rovnaké, pokračuje sa v porovnávaní. Keď sa narazí na rozdielne znaky, tak sa tieto dva porovnajú navzájom a podľa toho sa nastaví celkový výsledok porovnania. Porovnávanie dvoch znakov sa robí podľa pravidiel Char: t.j. menší je ten znak, ktorý má menší **ascii**-kód.

Operácia '+' slúži na zreťazenie reťazcov (hovoríme, že + je polymorfný operátor, lebo jeho funkčnosť závisí od typu operandov: iný význam má pre čísla a iný pre reťazce). Napr.

```
s := 'abc'+'def';      // s='abcdef'
s := '';
for i := 1 to 10 do    // s='**********'
  s := s + '*';
```

Krátke znakové reťazce

V Turbo pascale ste sa mohli stretnúť s typom String, pri ktorom sme do hranatých zátvoriek zapisovali nejaké číslo od 1 do 255. Tým sme označili maximálnu možnú dĺžku reťazca. Napr.

```
var s: String[20];
```

Označuje, že premenná s bude môže mať maximálnu dĺžku 20 znakov. Priradenie dlhšieho reťazca spôsobí odhodenie všetkých znakov za 20. znakom. Aj v Delphi fungujú takéto reťazce, ale my ich budeme používať veľmi zriedka. Napr. môžeme zapísať

```
var
  s: String[10];
begin
  s := 'abcd'+'efgh'+'ijkl';
  ...
```

Do premennej s sa ale dostalo len prvých 10 znakov, t.j. s='abcdefghij'.

Preddefinované (štandardné) podprogramy s reťazcami

- Length(*reťazec*)
  - táto funkcia vráti momentálnu dĺžku reťazca
- SetLength(*reťazcová_premenná*, *dĺžka*)
  - táto procedúra nastaví dĺžku reťazca
  - pre krátke reťazce nesmie presiahnuť deklarované maximum
  - ak je menšia ako momentálna dĺžka, znaky na konci sa strácajú
  - ak je väčšia ako momentálna, tak pridané znaky (od konca) majú nedefinovanú hodnotu
- Copy(*reťazec*, *od*, *koľko*)
  - táto funkcia vráti nový reťazec - podreťazec pôvodného
  - ak je od mimo rozsahu, vráti prázdny reťazec, t.j. '', napr.

```
s := Copy('abcde', 2, 3);    // s='bcd'
s := Copy('abcde', 7, 3);    // s=''
s := Copy('abcde', 3, 5);    // s='cde'
```

  - ak potrebujeme podreťazec od nejakého indexu až do konca, nemusíme vypočítať presný počet, ale môžeme použiť konštantu MaxInt, napr.

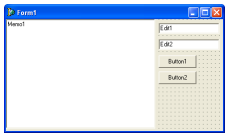
```
s := Copy('abcde', 3, MaxInt);
```

- Pos(*podreťazec*, *reťazec*)
  - táto funkcia vráti začiatočný index **prvého** výskytu podreťazca v reťazci alebo 0, ak sa v reťazci podreťazec nenachádza, napr.

```
i := Pos('d', 'abcde');     // i=4
i := Pos('x', 'abci := Pos('x', 'i := Pos('ba', 'abababab'); //
i=2
```

komponent editovací riadok TEdit

Aby sme mohli lepšie experimentovať s o znakovými reťazcami, ukážeme použitie komponentu, pomocou ktorého môžeme veľmi jednoducho zadávať vstupné znakové reťazce do nášho programu. Komponent TEdit (v štandardnej palete komponentov) sa trochu podobá textovej ploche ale len s jedným riadkom. Položme do formulára okrem textovej plochy Memo1, tlačidiel Button1, Button2 aj dva editovacie riadky Edit1 a Edit2:



Prvý podprogram vypíše index výskytu druhého reťazca (Edit2) v prvom reťazci (Edit1):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1, s2: String;
begin
  s1 := Edit1.Text;
  Memo1.Lines.Append('s1 = "' + s1 + '"');
  s2 := Edit2.Text;
  Memo1.Lines.Append('s2 = "' + s2 + '"');
  Memo1.Lines.Append('Pos(s2, s1) = ' + IntToStr(Pos(s2, s1)));
end;
```

Druhý podprogram zistí počet všetkých výskytov podreťazca p v reťazci s:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  s, p: String;
  i, j, poc: Integer;
begin
  s := Edit1.Text;
  p := Edit2.Text;
  poc := 0;
  j := 0;
  repeat
    i := Pos(p, Copy(s, j+1, MaxInt));
    if i > 0 then
    begin
      Inc(poc);
      j := j+i;
```

```
    end;
  until i = 0;
  Memo1.Lines.Append('poèet výskytov = ' + IntToStr(poc));
end;
```

Poznámka:

- všimnite si použitie inej konštrukcie cyklu a to cyklu repeat:
- tento cyklus opakovane vykonáva telo cyklu (príkazy medzi slovami repeat a until) a po každom takomto vykonaní týchto príkazov, otestuje podmienku cyklu (logický výraz za slovom until) - keď je podmienka splnená (hodnota je True), cyklus **končí** a pokračuje sa príkazmi za cyklom; keď je podmienka nepravdivá (hodnota je False), cyklus pokračuje - hovoríme tomu podmienka ukončenia cyklu
- tento cyklus sa teda správa "opačne" ako cyklus while
- uvedomte si, že telo cyklu sa vykoná vždy aspoň raz, aj keď je podmienka hneď na začiatku splnená
- program predpokladá, že funkcia Pos je dostatočne rýchla a teda sa snaží čo najviac urýchliť počítanie - program by mohol byť dostatočne rýchly aj pre veľmi dlhé reťazce ...

Ďalší príklad na repeat-cyklus: všetky výskyty znaku ' ' nahraď '***':

```
repeat
  i := Pos(' ', s);
  if i > 0 then
    s := Copy(s, 1, i-1) + '***' + Copy(s, i+1, MaxInt);
until i=0;
```

Ďalšie námety:

- vyriešte aj pre prípad, že všetky výskyty ' ' sa nahradia '* *' - vsunutú medzeru už nenahrádza

Ďalšie preddefinované (štandardné) procedúry/funkcie s reťazcami:

Najprv ukážeme, ako by sme mohli sami naprogramovať už pre nás známu funkciu Pos:

```
function Pos(const p, s: String): Integer;
var
  nasiel: Boolean;
begin
  nasiel := False;
  Result := 0;
  while (Result < Length(s)) and not nasiel do
  begin
    Inc(Result);
    nasiel := p=Copy(s, Result, Length(p));
  end;
  if not nasiel then
    Result := 0;
end;
```

Poznámka:

- samozrejme, že štandardná funkcia je naprogramovaná trochu inak a zrejme je rýchlejšia - toto je len ukážka,
- vo while podmienke začiatočníci ľúbia zapisovať nasiel = False, je to škaredé - radšej zapisujeme not nasiel
- priradenie nasiel := p=Copy(s, Result, Length(p)); začiatočníci sú niekedy schopní zapísať:

```
if p = Copy(s, i, Length(p)) then
  nasiel := True
else
  nasiel := False;
```

- vo while-podmienke by sa dalo skončiť aj skôr:

```
while (Result <= Length(s)-Length(p)) and ...
```

Funkcia Delete

Táto funkcia má tento tvar: **Delete**(s, od, kolko), kde

- s je znaková premenná
- od a kolko sú celočíslené hodnoty
- funkcia zmení reťazec s, tak, že z neho vypustí kolko znakov od pozície od

Vedeli by sme ju zapísať napr. takto:

```
procedure Delete(var s: String; od, kolko: Integer);
begin
  s := Copy(s, 1, od-1) + Copy(s, od+kolko, MaxInt);
end;
```

Procedúra Insert

Táto procedúra má tento tvar: **Insert**(co, s, od), kde

- co je znakový reťazec (napr. premenná, konštanta alebo reťazcový výraz)
- s je znaková premenná
- od je celočíslená hodnota
- procedúra vloží podreťazec co do reťazca s od pozície od

Vedeli by sme ju zapísať napr. takto:

```
procedure Insert(const co: String; var s: String; od: Integer);
begin
  s := Copy(s, 1, od-1) + co + Copy(s, od, MaxInt);
end;
```

Procedúra Str

Táto procedúra má tento tvar: **Str**(číslo, s), kde

- číslo je celočíslená alebo reálna hodnota
- s je znaková premenná
- procedúra prevedie číslo do znakového reťazca, napr.

```
Str(123, s);        // s='123'
Str(-5.72, s);      // s='-5.72'
Str(123:5, s);      // s='  123'
Str(-5.72:7:3, s);  // s=' -5.720'
```

Zjednodušene by sme ju vedeli zapísať napr. takto:

```
procedure Str(c: Integer; var s: String);
var
  znam: Boolean;
begin
  znam := c<0;
  if znam then
    c := -c;
  s := '';
  repeat
    s := Chr(c mod 10+Ord('0')) + s;
```

```
    c := c div 10;
  until c = 0;
  if znam then
    s := '-' + s;
end;
```

Procedúra Val

Táto procedúra má tento tvar: **Val**(s, číslo, ok), kde

- s je znakový reťazec
- číslo je celočíslená alebo reálna premenná
- ok je celočíselná premenná - v nej nám vráti informáciu, či bol reťazec korektné číslo (0 je ok, inak pozícia v reťazci, kde nastala chyba)
- procedúra prevedie reťazec na číslo, napr.

```
Val('123', i, ok);        // i=123, ok=0 - prevod bol v poriadku
Val('123, 124', i, ok);   // i=?, ok=4 - pozícia chyby
Val('- 123', i, ok);      // i=?, ok=2
Val('123 ', i, ok);       // i=?, ok=4
```

Zjednodušene by sme ju vedeli zapísať napr. takto:

```
procedure Val(const s: String; var i, ok: Integer);
var
  p: Integer;
  znam: Boolean;
begin
  p := 1;
  i := 0;
  znam := s[1]='-';
  if znam then
    p := 2;
  while (p <= Length(s)) and (s[p] >= '0') and (s[p] <= '9') do
  begin
    i := i*10+Ord(s[p])-Ord('0');
    Inc(p);
  end;
  if p > Length(s) then
  begin
    ok := 0;
    if znam then
      i := -i;
  end
  else
    ok := p;
end;
```

Ďalšie námety:

- preprogramujte Str aj Val, aby pracovali pre reálne čísla

Ďalší príklad

Za každý znak v reťazci chceme pridať medzeru ' '. Najprv uvedieme chybné riešenie, ktoré sa ale na prvý pohľad môže zdať správne:

```
function pridaj(s: String): String;
var
  i: Integer;
begin
```

```
  for i := 1 to Length(s) do
    Insert(' ', s, i);
  Result := s;
end;
```

Možné správne riešenie:

```
function pridaj(s: String): String;
var
  i: Integer;
begin
  for i := 1 to Length(s) do
    Insert(' ', s, 2*i);
  Result := s;
end;
```

iné riešenie:

```
function pridaj(s: String): String;
var
  i: Integer;
begin
  Result := '';
  for i := 1 to Length(s) do
    Result := Result + s[i] + ' ';
end;
```

Ďalšie námety:

- Napíšte funkciu UpCaseStr, ktorá prerobí v reťazci malé písmená na veľké.
- Napíšte procedúru Center, ktorá centruje reťazec na udanú šírku:

  ```
  procedure Center(var s: String; sirka: Byte);
  ```

- Napíšte funkciu, ktorá pre vstupný reťazec s dvoma slovami oddelenými medzerou vytvorí reťazec, ktorý má tieto slová v opačnom poradí (napr. krstné meno a priezvisko)

Delphi má preddefinované tieto užitočné funkcie (v programe musí byť uses SysUtils;)

- function UpperCase(const S: String): String;
- function LowerCase(const S: String): String;
- function Trim(const S: String): String;
- function TrimLeft(const S: String): String;
- function TrimRight(const S: String): String;
- function AdjustLineBreaks(const S: String): String;
- function IntToStr(Value: Integer): String;
- function IntToHex(Value: Integer; Digits: Integer): String;
- function FloatToStr(Value: Extended): String;
- function StrToInt(const S: String): Integer;
- function StrToIntDef(const S: String; Default: Integer): Integer;
- function StrToFloat(const S: String): Extended;

oplatí sa s nimi zoznámiť a používať ich - môžete si ich cvične naprogramovať aj sami ako "Ďalšie námety"

Tiež Delphi ponúkajú tieto funkcie na prácu so súbormi:

- function FileExists(const FileName: String): Boolean;

- function DeleteFile(const FileName: String): Boolean;
- function RenameFile(const OldName, NewName: String): Boolean;
- function CreateDir(const Dir: String): Boolean;
- function RemoveDir(const Dir: String): Boolean;

tieto funkcie vrátia True, ak prebehli bez chyby - ak nás chybný výsledok nezaujíma, môžeme tieto funkcie volať ako keby to boli procedúry

6. prednáška: polia

čo už vieme:

- poznáme niektoré jednoduché typy - Integer, Char, Boolean a Real
- poznáme aj niektoré zložené typy - String a TextFile (pravdepodobne aj objekt robot je zložený typ)

čo sa na tejto prednáške naučíme:

- zoznámime sa s novým jednoduchým typom - interval a s novým zloženým typom - pole
- naučíme sa, ako sa pracuje s poľom, ktoré je parametrom podprogramu
- uvidíme príklady na pole robotov

Typ interval

Interval je taký typ, ktorý je odvodený z nejakého už existujúceho ale ordinálneho typu - hovoríme mu bázový typ. Typ definujeme tak, že určíme minimálnu a maximálnu konštantu tohto nového typu. Napr. ak zapíšeme

```
1 .. 10
```

znamená to, že vytvárame podtyp celých čísel (lebo sú to celočíselné konštanty) a tento nový typ obsahuje všetky konštanty z intervalu <1, 10>. Automaticky zo svojho bázového typu (Integer) preberá všetky vlastnosti a operácie. Napr.

```
type
  Interval = 1..10;
var
  x: Interval;
```

premenná x môže nadobúdať len hodnoty z tohto intervalu. Ak by sme sa pokúsili do nej priradiť nejakú inú hodnotu (napr. 0), pascal by hlásil chybu. Inak pracujeme s touto premennou rovnako ako s obyčajnou celočíselnou premennou. Nakoľko **typ interval je tiež ordinálny typ**, môžeme ho použiť napr. aj vo for-cykle aj v príkaze case. Ešte je tu jeden rozdiel od typu Integer - typ interval môže v pamäti zaberať menej miesta ako jeho bázový typ, napr. v našom príklade premenná x zaberá iba jeden bajt (8 bitov). Typ interval môžeme odvodiť aj od iných ordinálnych typov, napr.

```
type
  MaleCislo = -100..100;
  Roky = 1900..2100;
  CeleCislo = Integer;
  Bajt = 0..255;
  Cifry = '0'..'9';
  Pismena = 'A'..'Z';
```

Takejto definícii nového typu hovoríme **priama definícia**, lebo typ definujeme v odseku definície typov. Typ môžeme definovať aj **nepriamo** pri deklarovaní premennej, napr.

```
var
  malepis: 'a'..'z';
```

Niektoré jednoduché typy, ktoré sú preddefinované v pascale:

```
type
  Integer = -2147483648 .. 2147483647;
  Longint = -2147483648 .. 2147483647;
  Smallint = -32768 .. 32767;
  Shortint = -128 .. 127;
  Byte = 0 .. 255;
  Word = 0 .. 65535;
  Cardinal = 0 .. 4294967295;
  Int64 = -2^63 .. 2^63-1;

  Char = #0 .. #255;
```

Všimnite si, že typ Longint je identický s typom Integer - Borland ho tam nechal kvôli kompatibilite s Turbo Pascalom.

Štruktúrovaný (zložený) typ pole

Definujeme:

```
type
  Pole = array[typ_indexu] of typ_prvkov;
```

Pole sa skladá z veľa "jednoduchších" premenných, tzv. prvkov poľa. Všetky tieto prvky sú rovnakého typu. Pristupujeme k nim (selekcia) pomocou, tzv. indexu. Typom indexu môže byť ľubovoľný ordinálny typ. Typom prvkov poľa môže byť ľubovoľný typ. Štruktúra pole v pamäti zaberá toľko miesta, koľko zaberá jeden prvok krát počet prvkov poľa, t.j. počet rôznych hodnôt typu indexu. Napr.

```
type
  MojePole = array[1..100] of Integer;
```

definuje pole ktoré bude mať 100 prvkov (premenných) a keďže každý prvok zaberá 4 bajty, celé toto pole zaberá 400 bajtov. Delphi dovolí zadeklarovať maximálne 2 GB štruktúru, ale Windows má k dispozícii pre aplikáciu väčšinou len niekoľko desiatok až stovák MB (už neplatí obmedzenie z Turbo pascalu, že celá štruktúra musí zaberať menej ako 64 KB). Napr.

```
type
  Typ1 = array[Integer] of Byte;   // zaberá presne 4 GB - to je už veľa
  Typ2 = array[Byte] of Integer;   // 1 KB (1024 B)
  Typ3 = array[Char] of Boolean;   // 0,25 KB (256 B)
```

Zapamätajte si:

1 B = bajt (8 bitov)

1 KB = kilo bajt = 1024 bajtov (trochu viac ako tisíc bajtov)

1 MB = mega bajt = 1024 KB = 1024*1024 B = 1048576 bajtov (trochu viac ako milión bajtov)

1 GB = giga bajt = 1024 MB = 1024*1024 KB = 1024*1024*1024 B = 1073741824 bajtov (trochu viac ako miliarda bajtov)

Ak zadeklarujeme

```
var
```

```
p: MojePole;
```

označuje to jednu premennú p typu MojePole. Už vieme, že táto premenná zaberá 400 bajtov, lebo v sebe obsahuje 100 celočíselných premenných - prvkov. Ku každému prvku pristupujeme pomocou indexu, t.j. pomocou hodnoty z prípustného intervalu podľa deklarácie - v našom prípade indexom musí byť hodnota z intervalu 1..100. Ak chceme pracovať so samostatným prvkom, index zapisujeme do hranatých zátvoriek, napr.

```
p[5] := 37;
```

zmení obsah 5-teho prvku. S prvkami poľa pracujeme úplne rovnako ako s obyčajnými premennými (okrem toho, že ich nesmieme použiť ako premennú for-cyklu). Ak chceme do poľa prečítať nejaké hodnoty zo súboru, alebo ich chceme zapísať do súboru, použijeme cyklus, napr. ako v programe, ktorý o každom prvku poľa vypíše, či je jeho hodnota nad priemerom, pod priemerom všetkých alebo sa rovná priemeru všetkých čísel:

```
const
  max = 7;
type
  Index = 1..max;
  Pole = array[index] of Integer;
var
  p: Pole;
  i: Index;
  sucet: Integer;
  t: TextFile;
begin
  AssignFile(t, 'cisla.txt');
  Reset(t);
  for i := 1 to max do
    Read(t, p[i]);
  CloseFile(t);
  AssignFile(t, 'vypis.txt');
  Rewrite(t);
  sucet := 0;
  for i := 1 to max do
    Inc(sucet, p[i]);
  writeln(t, 'Priemer: ', sucet/max:0:2);
  for i := 1 to max do
    if p[i] < sucet/max then
      Writeln(t, p[i], ' pod priemer')
    else if p[i] > sucet/max then
      Writeln(t, p[i], ' nad priemer')
    else
      Writeln(t, p[i], ' priemer');
  CloseFile(t);
end;
```

Tu sme použili bežný spôsob práce s poľom: zadeklarovali sme konštantu max, v ktorej máme počet prvkov poľa. Program sme zapísali tak, aby len jednoduchou zmenou tejto konštanty napr. na 100 program korektne pracoval aj pre 100 čísel.

Dá sa to aj inak. Nebudeme si v konštante (max) pamätať počet prvkov poľa, a prepíšeme tento program tak, aby fungoval správne aj pre zmenený interval indexov poľa. Môžeme použiť pomocné funkcie pre pole (nech type Xp = array[-3..15] of Real;):

- funkcia High - vráti maximálny index poľa, napr. High(Xp) je 15
- funkcia Low - vráti minimálny index poľa, napr. Low(Xp) je -3
- funkcia Length - vráti počet prvkov (rovnako ako pre String vráti momentálnu dĺžku reťazca), napr. Length(Xp) je 19

Všimnite si, že v nasledujúcom príklade sme p[i] < sucet/max nahradili výrazom p[i]*Length(p) < sucet - tento

počíta to isté, ale pracuje len v celých číslach (nepoužíva reálnu aritmetiku). Funkcie Low a High pracujú aj pre ľubovoľné ordinálne typy a premenné:

- funkcia High - vráti maximálnu prípustnú hodnotu typu, napr. High(Byte) je 255
- funkcia Low - vráti minimálnu prípustnú hodnotu typu, napr. Low(Char) je #0.

Zmenený predchádzajúci program bez použitia reálnej aritmetiky:

```
type
  Pole = array[-1..5] of Integer;    // počet prvkov je 7
var
  p: Pole;
  i, sucet: Integer;
  t: TextFile;
begin
  AssignFile(t, 'cisla.txt');
  Reset(t);
  for i := Low(p) to High(p) do
    Read(t, p[i]);
  CloseFile(t);
  AssignFile(t, 'vypis.txt');
  Rewrite(t);
  sucet := 0;
  for i := Low(p) to High(p) do
    Inc(sucet, p[i]);
  Writeln(t, 'Priemer: ', sucet/Length(p):0:2);
  for i := Low(p) to High(p) do
    if p[i]*Length(p) < sucet then
      Writeln(t, p[i], ' pod priemer')
    else if p[i]*Length(p) > sucet then
      Writeln(t, p[i], ' nad priemer')
    else
      Writeln(t, p[i], ' priemer');
  CloseFile(t);
end;
```

Polia ako parametre podprogramov

Formálne parametre typu pole môžu byť troch typov:

- **var**-parameter - nevyhradzuje sa žiadna nová pamäť, ale podprogram priamo manipuluje so skutočným parametrom, teda s celým poľom
- **const**-parameter - podobne ako **var**-parameter, len je zakázané v podprograme toto pole meniť, t.j. do neho niečo priraďovať
- hodnotový parameter - pri volaní podprogramu sa vytvorí duplikát celého poľa (rovnakej veľkosti) a vďaka tomu môžeme modifikovať tento duplikát (lokálnu premennú) bez toho, aby sa menil skutočný parameter. Treba dávať pozor na použitie tohto typu formálneho parametra, ak je to pole. Okrem toho, že spomaľuje výpočet, veľmi míňa pamäť pre lokálne premenné a často program na tomto aj hlási chyby.

Pri definovaní formálnych parametrov nesmieme **nepriamo** definovať nový typ, t.j. môžeme uvádzať len identifikátory už zadeklarovaných typov.

Aj výsledkom funkcie môže byť typ pole ale tiež musí byť vždy zadaný identifikátorom typu. Vysvetlime si, ako funguje volanie funkcie, ktorej výsledkom je typ pole:

- vyhradia sa všetky lokálne premenné (aj hodnotové formálne parametre)
- vyhradí sa jedna špeciálna lokálna premenná Result, ktorá je rovnakého typu ako typ funkcie, teda pole - táto premenná má zatiaľ **nedefinovanú** hodnotu
- s touto premennou pracujeme vo funkcii ako s iným bežným poľom

- po skončení výpočtu funkcie sa "zabudnú", t.j. uvoľnia všetky lokálne premenné, len hodnota Result sa stane výsledkom celej funkcie.

Chybné použitie typov:

```
procedure a(b: array[1..10] of Char);
procedure x(y: 1..10);
function f(z: Char): array[1..10] of Char;
```

Príklad: dva varianty procedúry presun, ktorá presúva všetky prvky poľa b do poľa a:

```
type
  Index = 1..max;
  Pole = array[Index] of Integer;

procedure presun1(var a: Pole; const b: Pole);
var
  i: Integer;
begin
  for i := 1 to max do
    a[i] := b[i];
end;

procedure presun2(var a: Pole; const b: Pole);
begin
  a := b;
end;
```

Hoci obe procedúry robia to isté, presun2 je výrazne efektívnejšia a teda rýchlejšia. Uvedomte si použitie var a const v definícii formálnych parametrov.

Príklad: Napíšeme program, ktorý najprv náhodne vygeneruje do poľa X čísla z intervalu 0..n-1, vypíše pole X, presunie pole X do poľa Y, pričom prvky náhodne zamieša a vypíše pole Y. Na záver zvýši hodnoty v poli Y o 1 a vypíše pole Y.

```
type
  Pole = array[1..100] of Integer;

function nahodne(n: Integer): Pole;
var
  i: Integer;
begin
  for i := 1 to High(pole) do
    Result[i] := Random(n);
end;

procedure vypis(var t: TextFile; const p: Pole);
var
  i: Integer;
begin
  for i := 1 to High(p) do
    Write(t, p[i], ' ');
  Writeln(t);
  Writeln(t);
end;

procedure premiesaj(var a: Pole; b: Pole);
var
  i, j: Integer;
begin
  for i := 1 to High(a) do
  begin
    j := Random(High(b)-i+1)+1;
    a[i] := b[j];
```

```
    b[j] := b[High(b)-i+1];
  end;
end;

procedure Inc(var a: Pole);
var
  i: Integer;
begin
  for i := 1 to High(a) do
    a[i] := a[i]+1;                    // System.Inc(a[i]);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  x, y: Pole;
  t: TextFile;
begin
  AssignFile(t, 'vypis.txt');
  Rewrite(t);
  Randomize;
  x := nahodne(1000);
  vypis(t, x);
  premiesaj(y, x);
  vypis(t, y);
  Inc(y);
  vypis(t, y);
  CloseFile(t);
  Memo1.Lines.LoadFromFile('vypis.txt');
end;
```

Všimnite si, že sme vytvorili funkciu Inc na zvýšenie hodnôt prvkov v poli, tým sme prekryli pôvodný štandardný identifikátor funkcie Inc a preto ho už v pôvodnom význame používať nemôžeme - v komentári môžete vidieť, ako sa tento problém aj tak dá vyriešiť.

Pole znakov

Z historických dôvodov má pole znakov niekoľko výnimiek oproti iným poliam:

- polia znakov do 255 znakov je možné vypísať do súboru pomocou jedného volania Write, resp. Writeln
- do poľa môžeme jedným priradením priradiť aj konštantu znakový reťazec, ale musí byť rovnakej dĺžky ako je počet znakov v poli
- znakové pole môžeme aj prečítať zo súboru jedným príkazom Read, resp. Readln, ale pole musí byť deklarované s dolnou hranicou indexu 0 - vtedy to pascal pochopí ako "Null-terminated String" a s tým vie robiť aj iné špeciality - toto uvidíme neskôr

```
type
  Pole = array[1..10] of Char;
var
  s: Pole;
begin
  s := 'abcdefghij';    // ok
  s := 'abc';           // chyba
  Writeln(t,s);         // ok
end;
```

Polia robotov

Už predtým sme pracovali naraz s viac robotmi. Ešte zaujímavejšie je, keď roboty sú v poli a pracujeme s nimi pomocou cyklov. V príklade vygenerujeme vedľa seba na vodorovnej priamke n robotov, každému nastavíme iný uhol a všetky "naraz" nakreslia kružnicu. Najprv pomalá verzia, t.j. po každom pohybe každého robota pozdržíme

výpočet pomocou wait:

```
const
  n = 50;
var
  r: array[1..n] of TRobot;
  i, j: Integer;
begin
  for i := 1 to n do
    r[i] := TRobot.Create(10*i+10, 250, i*15);
  for j := 1 to 180 do
    for i := 1 to n do
    begin
      r[i].fd(4);
      r[i].rt(2);
      wait(1);
    end;
  for i := 1 to n do
    r[i].Free;
end;
```

Program urýchlime tak, že jeden wait urobíme až keď sa pohnú všetky roboty:

```
  ...
  for i := 1 to n do
  begin
    r[i] := TRobot.Create(10*i+10, 250, i*15);
    r[i].PW := 5;
  end;
  for j := 1 to 180 do
  begin
    for i := 1 to n do
      with r[i] do
      begin
        fd(4);
        rt(2);
      end;
    wait(1);
  end;
  ...
```

Na tomto príklade zároveň ukazujeme aj nový pascalovský príkaz with: vo vnútri neho sa "prednostne" pracuje s daným robotom.

Jednoduchá animácia pomocou robotov

Predchádzajúci príklad pozmeníme takto: for-cyklus nahradíme nekonečným while True do a pred samotným pohybom robotov zmažeme obrazovku. Vznikne efekt, v ktorom na obrazovke vidíme len posledne nakreslené čiary od všetkých robotov:

```
procedure TForm1.Button2Click(Sender: TObject);
const
  n = 50;
var
  r: array[1..n] of TRobot;
  i: Integer;
begin
  for i := 1 to n do
  begin
    r[i] := TRobot.Create(10*i+10, 250, i*15);
    r[i].PW := 10;
  end;
```

```
  while True do
  begin
    cs;
    for i := 1 to n do
      with r[i] do
      begin
        fd(4);
        rt(2);
      end;
    wait(1);
  end;
end;
```

S programom môžeme ďalej experimentovať: hoci robot pri krúžení prejde len vzdialenosť 4, nakreslí úsečku dĺžky 100; tiež si všimnite, že každý robot sa otáča o iný uhol:

```
const
  n = 50;
var
  r: array[1..n] of TRobot;
  i: Integer;
begin
  for i := 1 to n do
  begin
    r[i] := TRobot.Create(10*i+10, 250, i*15);
    r[i].PW := 5;
    r[i].PC := clYelLow;
  end;
  while True do
  begin
    cs(clNavy);
    for i := 1 to n do
      with r[i] do
      begin
        fd(100);
        fd(-96);
        rt(1+i/n);
      end;
    wait(1);
  end;
end;
```

V ďalšom príklade nevygenerujeme roboty na jednej priamke, ale rovnomerne na kružnici: najprv ich všetky vytvoríme v strede plochy, každý natočíme iným smerom a potom so zdvihnutým perom prejdú nejakú rovnakú vzdialenosť. Použili sme stavovú premennú (vlastnosť) robota H (heading - momentálny smer otočenia), pomocou ktorej môžeme nastavovať absolútne natočenie robotov:

```
const
  n = 60;
var
  r: array[1..n] of TRobot;
  i: Integer;
begin
  for i := 1 to n do
  begin
    r[i] := TRobot.Create;
    with r[i] do
    begin
      PW := 15;
      PC := clBlue;
      pu;
      H := 360/n*i;
      fd(100);
```

```
      pd;
      // H := H*2;
    end;
  end;
  while True do
  begin
    cs;
    for i := 1 to n do
      with r[i] do
      begin
        fd(4);
        rt(2);
      end;
    wait(1);
  end;
end;
```

Môžete experimentovať s rôznymi počiatočnými natočeniami robotov, napr. ak odkomentujete priradenie H := H*2; vznikajú veľmi zaujímavé efekty. Napr.

```
const
  n = 360;
var
  r: array[1..n] of TRobot;
  i: Integer;
begin
  for i := 1 to n do
  begin
    r[i] := TRobot.Create;
    with r[i] do
    begin
      // PW := 10;
      PC := clBlue;
      pu;
      H := 360/n*i;
      fd(100);
      pd;
      H := H*50;
    end;
  end;
  while True do
  begin
    cs;
    for i := 1 to n do
      with r[i] do
      begin
        fd(204);
        fd(-200);
        rt(2);
      end;
    wait(1);
  end;
end;
```

Aj iné konštanty vo výraze H := H*50; vytvárajú pekné obrazce - vyskúšajte, napr. 2, 3, 4, 9, 10, 15, 30, 45, 50, 56, 60, 90, 120, 151, 179, ...

Roboty sa naháňajú

Na náhodných pozíciách vygenerujeme n robotov. Potom budeme opakovať túto akciu: v každom kroku sa každý robot posunie o 1/100 vzdialenosti k svojmu nasledovníkovi (posledný sa posunie k prvému):

```
const
```

```
  n = 8;
var
  r: array[1..n] of TRobot;
  i, sirka, vyska: Integer;
  xx, yy: Real;
begin
  cs;
  Randomize;
  sirka := Image1.Width;
  vyska := Image1.Height;
  for i := 1 to n do
  begin
    r[i] := TRobot.Create(Random(sirka), Random(vyska));
    r[i].PW := 5;
    r[i].PC := Random(16777216);
  end;
  while True do
  begin
    for i := 1 to n do
      with r[i] do
      begin
        xx := r[i mod n+1].X;
        yy := r[i mod n+1].Y;
        towards(xx, yy);
        fd(dist(xx, yy)/100);
      end;
    wait(1);
  end;
end;
```

Všimnite si použite i mod n+1 - takto i-ty robot pristupuje ku svojmu nasledovníkovi, pričom posledný má nasledovníka prvého. Ďalej sme použili nový príkaz robota towards, ktorý ho otočí do smeru k bodu (x, y).

Keďže sa po nejakom čase všetky roboty stretnú v jednom bode, bolo by vhodné cyklus vtedy ukončiť. Premyslite si, ako zistiť, že všetky roboty sú už navzájom veľmi blízko.

Roboty môžu kresliť veľmi zaujímavé obrazce, keď okrem toho, že prejdú 1/10 vzdialenosti ku svojmu nasledovníkovi, nakreslia k nemu aj spojnicu:

```
const
  n = 8;
var
  r: array[1..n] of TRobot;
  i, j, sirka, vyska: Integer;
  xx, yy, d: Real;
begin
  cs(clBlack);
  Randomize;
  sirka := Image1.Width;
  vyska := Image1.Height;
  for i := 1 to n do
  begin
    r[i] := TRobot.Create(Random(sirka), Random(vyska));
    r[i].PW := 2;
    r[i].PC := Random(16777216);
  end;
  for j := 1 to 100 do
  begin
    for i := 1 to n do
      with r[i] do
      begin
        xx := r[i mod n+1].X;
        yy := r[i mod n+1].Y;
        d := dist(xx, yy);
```

```
        towards(xx,yy);
        fd(d);
        fd(d/10-d);
      end;
   wait(1);
  end;
  for i := 1 to n do
    r[i].Free;
end;
```

Z programu môžete vyhodiť wait(1).

7. prednáška: typ záznam, vyhľadávanie

čo už vieme:

- štruktúrovaný typ pole je zložený z prvkov rovnakého typu - pristupujeme k nim pomocou indexu

čo sa na tejto prednáške naučíme:

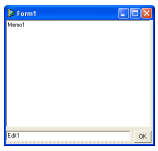
- komponent editovací riadok
- nový zložený typ - záznam, ktorý sa môže skladať s prvkov rôznych typov - pristupujeme k nim pomocou mena
- rôzne metódy hľadania informácie v (utriedenej/neutriedenej) tabuľke

Komponent editovací riadok

Už sme sa s ním stretli v predchádzajúcej prednáške pri interaktívnom zadávaní znakových reťazcov do programu. Používateľ počas behu programu môže do tohto riadka písať, resp. upravovať nejaký text. Program potom môže zistiť obsah tohto riadka, resp. ho meniť.

Tento komponent (podobne ako Button, Memo a Image) má viac stavových premenných - pre nás najdôležitejšou stavovou premennou bude Text. V nej sa nachádza momentálny obsah editovacieho riadka (je typu String). Z udalostí tohto komponentu nás bude zaujímať len onKeyPress. Táto udalosť vznikne vždy, keď používateľ, ktorý edituje tento riadok, stlačí nejaký kláves na klávesnici (písmená, číslice, špeciálne znaky ako Enter, Esc a pod.)

Editovací riadok otestujeme pomocou tohto projektu: do formulára vložíme textové okno (Memo1), tlačidlo (Button1) a editovací riadok (Edit1):



Zatlačenie tlačidla Button1 prekopíruje obsah editovacieho riadka do textovej plochy a editovací riadok sa potom vyčistí:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.Append(Edit1.Text);
  Edit1.Text := '';
```

```
  Edit1.SetFocus;
end;
```

Bolo by dobre, keby sme po každom zadaní textu nemuseli klikať na tlačidlo, ale fungovalo by aj zatlačenie klávesu **Enter**. Pridáme spracovanie udalosti onKeyPress pre Edit1: v Inšpektore objektov sa nastavíme na komponent Edit1, prepneme záložku Udalosti (Events) a dvojklikneme OnKeyPress - vytvorí sa metóda Edit1KeyPress - táto metóda sa zavolá vždy, keď sa bude niečo zapisovať do editovacieho riadka. Nás však bude zaujímať len kláves <Enter> s kódom #13:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
    Button1.Click
  else if (Key < '0') then
    Key := #0
  else
    Key := Upcase(Key);
end;
```

Zložený typ záznam - record

Už vieme, že pole je n-tica premenných (položiek), ktoré musia byť rovnakého typu. Naproti tomu **záznam** je n-tica premenných (položiek), ktoré nemusia byť rovnakého typu. Hovoríme, že premenná typu **záznam** sa skladá zo svojich súkromných premenných. Prístup k týmto premenným je realizovaný pomocou mena položky (selektor) - na rozdiel od polí, kde to bolo pomocou indexu.

S celým záznamom nie je povolený ani vstup ani výstup z/do textových súborov (ale len po položkách) a ani porovnávanie obsahov záznamov. Priradenie je povolené len ak sú obe premenné identického typu (rovnako ako pre polia). Záznamy môžu byť použité ako parametre procedúr, resp. ako výsledok funkcie.

napr.

```
type
  Zaznam = record
    meno: String;
    rocnik: 1..5;
    priemer: Real;
  end;
```

Mohli by sme si to predstaviť tak, že v pamäti sú položky uložené "tesne" za sebou (s výnimkou niektorých typov, napr. Real, ktoré sú v pamäti na adresách deliteľných 8).

So záznamom pracujeme takto:

```
var
  z, z1: Zaznam;
begin
  ...
  z.meno := 'Janko Hraško';
  z.rocnik := 1;
  z.priemer := 1.33;
  ...
  z1.meno := Copy(z.meno, 1, 4) + 'a Dadová';
  z1.rocnik := z.rocnik;
  z1.priemer := Random(4)+1;
  ...
end;
```

Pomocou záznamu reprezentujeme vektor ako dvojicu čísel (súradníc):

```
type
  Vektor = record
    x, y: Real;
  end;

function StrToVektor(const s: String): Vektor;
begin
  Result.x := StrToFloat(Copy(s, 1, Pos(' ', s)-1));
  Result.y := StrToFloat(Copy(s, Pos(' ', s)+1, MaxInt));
end;

function vektorToStr(v: Vektor): String;
begin
  Result := '(' + FloatToStr(v.x) + ',' + FloatToStr(v.y) + ')';
end;

function sucet(u, w: Vektor): Vektor;
begin
  Result.x := u.x + w.x;
  Result.y := u.y + w.y;
end;

function dlzka(v: Vektor): Real;
begin
  Result := Sqrt(Sqr(v.x)+Sqr(v.y));
end;

function otoc(v: Vektor; uhol: Real): Vektor;
begin
  uhol := uhol*pi/180;
  Result.x := cos(uhol)*v.x + sin(uhol)*v.y;
  Result.y := cos(uhol)*v.y - sin(uhol)*v.x;
end;
```

a otestovať to môžeme, napr. takto (vo formulári máme 2 editovacie riadky):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  v1, v2: Vektor;
  r: Real;
begin
  v1 := StrToVektor(Edit1.Text);
  Memo1.Lines.Append('vektor = ' + vektorToStr(v1));
  Memo1.Lines.Append('dĺžka = ' + FloatToStr(dlzka(v1)));

  r := StrToFloat(Edit2.Text);     // uhol
  v2 := otoc(v1,r);
  Memo1.Lines.Append('otočený vektor = ' + vektorToStr(v2));
  Memo1.Lines.Append(' dĺžka = ' + FloatToStr(dlzka(v2)));
  Memo1.Lines.Append('');
end;
```

Príklad s mestami

Postupne - na niekoľko krokov - budeme riešiť takýto príklad: v textovom súbore mesta.txt máme zadané nejaké informácie o mestách na mape. V každom riadku sú tri hodnoty: meno (reťazec ukončený ';') a súradnice x a y na obrazovke (oddelené medzerou).

```
type
  Info = record
    meno: String;
```

```
    x, y: Integer;
  end;

function citaj(var t: TextFile): Info;
var
  z: Char;
begin
  with Result do
  begin
    Read(t, z);
    meno := '';
    while z <> ';' do
    begin
      meno := meno + z;
      Read(t, z);
    end;
    Readln(t, x, y);
  end;
end;

var
  p: array[1..100] of Info;
  pocet: Integer;

procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  i: Integer;
begin
  AssignFile(t, 'mesta.txt');
  Reset(t);
  pocet := 0;
  while not SeekEof(t) do
  begin
    Inc(pocet);
    p[pocet] := citaj(t);
  end;
  CloseFile(t);
  for i := 1 to pocet do
    with p[i] do
      Memo1.Lines.Append(IntToStr(i) + '. ' + meno +
                         ' (' + IntToStr(x) + ',' + IntToStr(y) + ')');
end;
```

Pokračujme ďalej: namiesto textovej plochy Memo1 do formulára vložíme grafickú plochu Image1 (môžete do nej - property Picture v Inšpektore objektov - vložiť bitmapu mapa.bmp). Do tejto grafickej plochy pre každé mesto vykreslíme malý červený krúžok a napíšeme k nemu text - použijeme na to kresliacich robotov: ku každému mestu vytvoríme jedného robota:

```
uses
  RobotUnit;

type
  Info = record
    meno: String;
    x, y: Integer;
  end;

function citaj(var t: TextFile): Info;
var
  z:Char;
begin
  with Result do
  begin
```

```
    Read(t, z);
    meno := '';
    while z <> ';' do
    begin
      meno := meno + z;
      Read(t, z);
    end;
    Readln(t, x, y);
  end;
end;

var
  p: array[1..100] of Info;
  r: array[1..100] of TRobot;
  pocet: Integer;

procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  i: Integer;
begin
  AssignFile(t, 'mesta.txt');
  Reset(t);
  pocet := 0;
  while not seekeof(t) do
  begin
    Inc(pocet);
    p[pocet] := citaj(t));
  end;
  CloseFile(t);
  for i := 1 to pocet do
    with p[i] do
    begin
      r[i] := TRobot.Create(x, y);
      with r[i] do
      begin
        PC := clRed;
        point(15);
        PC := clBlack;
        Image1.Canvas.Brush.Color := clWhite;
        Image1.Canvas.Font.Style := [fsBold];
        text(meno);
      end;
    end;
end;
```

Pridáme editovací riadok: keď doňho napíšeme dve mená miest oddelené bodkočiarkou, spojí ich úsečkou:

```
function hladaj(s: String): Integer;
begin
  Result := pocet;
  while (Result > 0) and (p[Result].meno <> s) do
    Dec(Result);
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  s: String;
  i, i1, i2: Integer;
begin
  if Key <> #13 then
    Exit;
  s := Edit1.Text;
  i := Pos(';', s);
  if i = 0 then
    Exit;
```

```
  i1 := hladaj(Copy(s, 1, i-1));
  i2 := hladaj(Copy(s, i+1, MaxInt));
  if (i1 = 0) or (i2 = 0) then
    Exit;
  with r[i1] do
  begin
    PC := clYelLow;
    PW := 3;
    with p[i2] do
      setxy(x, y);
    with p[i1] do
      setxy(x, y);
  end;
  Edit1.Text := '';
end;
```

Všimnite si procedúru hladaj, ktorá v tabuľke miest vyhľadá pozíciu nejakého konkrétneho mesta. Tabuľku prezerá od konca a skončí vtedy, keď nájde hľadané mesto (vráti index), alebo keď prejde celú tabuľku a hľadané mesto nenájde (vráti hodnotu 0).

Program trochu vylepšíme: nakoľko polia p a k a premenná pocet (počet načítaných prvkov) spolu súvisia, vytvoríme z nich jedinú dátovú štruktúru – Tabulka:

```
uses
  RobotUnit;

type
  Info = record
    meno: String;
    x, y: Integer;
  end;

function citaj(var t: TextFile): Info;
var
  z: Char;
begin
  with Result do
  begin
    Read(t, z);
    meno := '';
    while z <> ';' do
    begin
      meno := meno + z;
      Read(t, z);
    end;
    Readln(t, x, y);
  end;
end;

type
  Tabulka = record
    p: array[1..100] of Info;
    r: array[1..100] of TRobot;
    pocet: Integer;
  end;

procedure pridaj(var t: Tabulka; i: Info);
begin
  with t do
  begin
    Inc(pocet);
    p[pocet] := i;
  end;
end;
```

```
var
  tab: Tabulka;

procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  i: Integer;
begin
  AssignFile(t, 'mesta.txt');
  Reset(t);
  tab.pocet := 0;
  while not seekeof(t) do
    pridaj(tab, citaj(t));
  CloseFile(t);
  for i := 1 to tab.pocet do
    with tab, p[i] do
    begin
      r[i] := TRobot.Create(x, y);
      with r[i] do
      begin
        PC := clRed;
        point(15);
        PC := clBlack;
        Image1.Canvas.Brush.Color := clWhite;
        Image1.Canvas.Font.Style := [fsBold];
        text(meno);
      end;
    end;
end;

function hladaj(s: String): Integer;
begin
  Result := tab.pocet;
  while (Result > 0) and (tab.p[Result].meno <> s) do
    Dec(Result);
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  s: String;
  i, i1, i2: Integer;
begin
  if Key <> #13 then
    Exit;
  s := Edit1.Text;
  i := Pos(';', s);
  if i = 0 then
    Exit;
  i1 := hladaj(Copy(s, 1, i-1));
  i2 := hladaj(Copy(s, i+1, MaxInt));
  if (i1 = 0) or (i2 = 0) then
    Exit;
  with tab, r[i1] do
  begin
    PC := clYelLow;
    PW := 3;
    with p[i2] do
      setxy(x, y);
    with p[i1] do
      setxy(x, y);
  end;
  Edit1.Text := '';
end;
```

Je veľmi dôležité správne chápať a aj používať príkaz with - všimnite si riadok with tab, r[i1] do - toto je skrátený

tvar konštrukcie: with tab do with r[i1] do - ak by tu nebola čiarka ale bodka with tab.r[i1] do, tak vo vnútri with by sme museli ku p[i1] a p[i2] písať tab**.** t.j. tab.p[i1] a tab.p[i1]. Nasledujúce zápisy znamenajú to isté:

```
with tab, r[i1] do
begin
  with p[i2] do
    setxy(x, y);
  with p[i1] do
    setxy(x, y);
end;
```

aj toto:

```
with tab.r[i1] do
begin
  with tab.p[i2] do
    setxy(x, y);
  with tab.p[i1] do
    setxy(x, y);
end;
```

aj toto:

```
with tab do
begin
  r[i1].setxy(p[i2].x, p[i2].y);
  r[i1].setxy(p[i1].x, p[i1].y);
end;
```

aj toto:

```
tab.r[i1].setxy(tab.p[i2].x, tab.p[i2].y);
tab.r[i1].setxy(tab.p[i1].x, tab.p[i1].y);
```

Vyhľadávanie v tabuľke

Algoritmus hľadania môžeme vylepšiť, ak budeme predpokladať, že tabuľka je vzostupne utriedená (abecedne - lexikograficky):

vylepšené hľadanie:

```
function hladaj(m: String): Integer;
begin
  Result := 1;
  while (Result <= tab.pocet) and (tab.p[Result].meno < m) do
    Inc(Result);
  if (Result > tab.pocet) or (tab.p[Result].meno <> m) then
    Result := 0;
end;
```

Zistite, čo bude funkcia hladaj vyhľadávať, ak z nej vyhodíme podčiarknutú časť.

Utriedenú tabuľku môžeme prehľadávať aj od konca:

```
function hladaj(m: String): Integer;
begin
  Result := tab.pocet;
  while (Result > 0) and (tab.p[Result].meno > m) do
    Dec(Result);
```

```
  if (Result <> 0) and (tab.p[Result].meno <> m) then
    Result := 0;
end;
```

Predpokladali sme, že tabuľka je utriedená – môžeme opraviť procedúru pridaj napr. takto:

```
procedure pridaj(var tab: Tabulka; const pp: Info);
var
  i: Integer;
begin
  with tab do
  begin
    i := pocet;
    while (i > 0) and (p[i].meno > pp.meno) do
    begin
      p[i+1] := p[i];
      Dec(i);
    end;
    p[i+1] := pp;
    Inc(pocet);
  end;
end;
```

Vnútorný while-cyklus posunie prvky tabuľky tak, aby sa nová položka mohla zasunúť na správne miesto. Zrejme predpokladáme, že zaraďovaný prvok sa ešte v tabuľke nenachádza.

Binárne vyhľadávanie

Naučíme sa nový typ algoritmu - **binárne vyhľadávanie** - podobný, ako keď hľadáme v telefónnom zozname:

- otvoríme v strede zoznamu: ak je hľadané slovo v abecede skôr ako slovo v strede, tak budeme pokračovať v prednej polovici zoznamu, inak v druhej polovici
- opäť vo vybranej polovici zoznamu sa pozrieme do jeho stredu a porovnáme s hľadaným slovom
- takto budeme postupne hľadať v menšom a menšom úseku zoznamu, až kým nenarazíme na hľadané slovo

Treba si zapamätať, že to funguje, len ak je pole / tabuľka utriedená:

```
function hladaj(m: String): Integer;
var
  z, k, s: Integer;
begin
  z:=1;                          // začiatok intervalu
  k:=tab.pocet;                  // koniec intervalu
  while z <= k do
  begin
    s := (z+k) div 2;            // stred intervalu
    if tab.p[s].meno < m then
      z := s+1
    else if tab.p[s].meno > m then
      k := s-1
    else
      z := k+1;
  end;
  if tab.p[s].meno = m then
    Result := s
  else
    Result := 0;
end;
```

alebo:

```
function hladaj(m: String): Integer;
var
  z, k, s: Integer;
begin
  z := 1;
  k := tab.pocet;
  while z < k do
  begin
    s := (z+k) div 2;
    if tab.p[s].meno < m then
      z := s+1
    else if tab.p[s].meno > m then
      k := s-1
    else
    begin
      z := s;
      k := s;
    end;
  end;
  if (z = k) and (tab.p[z].meno = m) then
    Result := z
  else
    Result := 0;
end;
```

Snažte sa pochopiť rozdiely medzi oboma riešeniami.

Testovanie rýchlosti vyhľadávania

Použitím štandardných podprogramov DecodeTime a Now môžeme dosť presne odmerať, ako dlho bežal nejaký výpočet. Budeme to organizovať napr. takto:

```
function myTimeToStr(cas: TDateTime): String;
var
  hod, min, sek, msek: Word;
begin
  DecodeTime(cas, hod, min, sek, msek);
  Result := IntToStr(hod) + ':' +
            IntToStr(min) + ':' +
            IntToStr(sek) + ':' +
            Copy(IntToStr(msek+1000), 2, 4);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  cas: TDateTime;
begin
  Screen.Cursor := crHourGlass;
  cas := Now;                               // zapamätá si momentálny čas
  // výpočet, ktorý potrebujeme odmerať
  Memo1.Lines.Append('čas = ' + myTimeToStr(Now-cas));  // výpis času
  Screen.Cursor := crDefault;
end;
```

Pomocou Screen.Cursor môžeme počas dlhšie trvajúcich výpočtov zapnúť kurzor myši na tvar "presýpacích hodín".

A otestovať niektoré vyhľadávanie môžeme napr. takto:

```
var
  tab: array[1..10000000] of Integer;

procedure TForm1.FormCreate(Sender: TObject);
```

```
var
  i: Integer;
begin
  tab[1] := 1;
  for i := 2 to High(tab) do
    tab[i] := tab[i-1]+1+Random(100);
end;

function hladaj(m: Integer): Integer;
begin
  Result := High(tab);
  while (Result > 0) and (tab[Result] <> m) do
    Dec(Result);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  cas: TDateTime;
  i: Integer;
begin
  Screen.Cursor := crHourGlass;
  cas := Now;
  for i := 1 to 1000 do
    hladaj(Random(MaxInt));
  Memo1.Lines.Append('čas = ' + myTimeToStr(Now-cas));
  Screen.Cursor := crDefault;
end;
```

Sekvenčné vyhľadávanie nahraďte binárnym vyhľadávaním a odmerajte, ako sa zmenil čas - možno bude treba zmeniť aj počet opakovaní vyhľadávania napr. na 1000000.

Všimnite si, že hoci je hladaj funkcia, volali sme ju ako keby bola procedúrou. Toto si môžeme dovoliť, keď nám nevadí zanedbať výsledok funkcie.

8. prednáška: dvojrozmerné polia, obrázky

čo už vieme:

- jednorozmerné polia prvkov rovnakého typu, napr. jednoduché typy alebo roboty
- štruktúrovaný typ záznam

čo sa na tejto prednáške naučíme:

- prvkami poľa môžu byť reťazce ale aj iné polia
- ako sa pracuje s obrázkami v grafickej ploche - budeme pracovať po jednotlivých pixeloch (farebné body obrázka)
- uvidíme nové vlastnosti komponentov
    - Memo - práca s riadkami pomocou Strings[]
    - Image - práca s farebnými bodkami pomocou Pixels
    - Button - text na tlačidle Title

- uvidíme niektoré jednoduché udalosti (onMouseMove, onMouseDown, onKeyPress)

Pole znakových reťazcov

Údajová štruktúra pole môže mať prvky ľubovoľného typu, teda aj znakový reťazec. S takýmito premennými, ktoré sú prvkami poľa, sa pracuje rovnako ako s jednoduchými premennými typu reťazec. Nasledujúci program ilustruje použitie poľa znakových reťazcov:

```
var
  t: TextFile;
  p: array [1..1000] of String;
  i, j, n: Integer;
begin
  AssignFile(t, 'unit1.pas');
  Reset(t);
  n := 0;
  while not Eof(t) and (n < High(p)) do
  begin
    Inc(n);
    Readln(t, p[n]);
  end;
  CloseFile(t);
  for i := 1 to n do
    for j := 1 to Length(p[i]) do
      if (j = 1) or (p[i][j-1] = ' ') then
        p[i][j] := Upcase(p[i][j]);
  Memo1.Clear;
  for i := 1 to n do
    Memo1.Lines.Append(p[i]);
end;
```

Tento program najprv prečíta riadky textového súboru a skôr ako ich vypíše do textovej plochy Memo1, každému slovu zmení prvé písmeno na veľké. Zápis p[i][j] označuje j-ty znak i-teho reťazca - môžeme to zapísať skrátene: p[i, j].

Nasledujúca časť programu filtruje medzery na začiatku a konci každého reťazca:

```
  ...
  for i := 1 to n do
  begin
    j := 1;
    while (j <= Length(p[i])) and (p[i, j] = ' ') do
      Inc(j);
    Delete(p[i], 1, j-1);
    j := Length(p[i]);
    while (j >= 1) and (p[i, j] = ' ') do
      Dec(j);
    SetLength(p[i], j);
  end;
  ...
```

Toto isté by sa dalo jednoduchšie zrealizovať pomocou štandardnej funkcie Trim.

Textová plocha

Ešte sa vrátime ku komponentu textová plocha - TMemo. Už poznáme metódy:

- Memo1.Lines.Clear; - vyčistí všetky riadky
- Memo1.Lines.Append('nejaký text'); - pridá ďalší riadok s daným textom na koniec
- Memo1.Lines.LoadFromFile('meno_súboru'); - obsah plochy nahradí obsahom zadaného súboru

Na riadky (Lines) textovej plochy sa môžeme pozerať ako na jednorozmerné pole znakových reťazcov Strings. Metóda Clear toto pole inicializuje, t.j. zruší všetky prvky poľa a prvý prvok (s indexom 0) vyprázdni. Metóda Append pridá na koniec tohto poľa nový reťazec. Momentálny počet riadkov zistíme pomocou Memo1.Lines.Count. Pomocou Strings môžeme modifikovať konkrétne riadky textovej plochy, napr.

```
Memo1.Lines.Strings[0] := 'mmmmmm';
```

prepíše prvý riadok textovej plochy na mmmmmm. V nasledujúcom programe zmeníme každý riadok textovej plochy:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Memo1.Lines.LoadFromFile('unit1.pas');
  for i := 0 to Memo1.Lines.Count-1 do
    Memo1.Lines.Strings[i] :=

      '<' + UpperCase(Memo1.Lines.Strings[i]) + '>';
end;
```

Program najprv do textovej plochy zapíše obsah súboru unit1.pas a potom v celých riadkoch zmení malé písmená na veľké a pridá znaky < a > na začiatok a koniec každého riadka.

Viacrozmerné polia

Zovšeobecníme polia: prvkami poľa môže byť opäť pole -- ak je prvkom jednorozmerné pole, tak výsledný typ je dvojrozmerné pole. Napr. a[4][2] označuje druhý prvok v štvrtom poli - väčšinou si dvojrozmerné pole predstavujeme ako tabuľku, v ktorej každý prvok leží v nejakom riadku a nejakom stĺpci, napr. a[4][2] môže označovať prvok v 4-tom riadku a v 2-om stĺpci - skrátene to môžeme zapísať a[4, 2].

Príklad: Je dané dvojrozmerné pole znakov, ktoré je už zaplnené nejakým textom. Napíšte program, ktorý presunie prvý riadok do druhého, druhý do tretieho, atď., až posledný do prvého (napr. motivácia: rolovanie obrázka ako vo svetelných novinách). Najprv nie pekné riešenie:

```
var
  a: array[1..10] of array[1..25] of Char;
  r: array[1..25] of Char;
  i, j: Integer;
begin
  ... // zaplnenie poľa
  for i := 1 to 25 do     // odložíme posledný riadok do pomocného poľa
    r[i] := a[10, i];
  for i := 9 downto 1 do  // postupne kopírujeme všetky riadky
    for j := 1 to 25 do
      a[i+1, j] := a[i, j];
  for i := 1 to 25 do     // presunieme pomocné pole do prvého riadka
    a[1, i] := r[i];
  ...
```

A teraz zapíšeme to isté, ale využijeme to, že ak sú dve polia zadeklarované identicky, môžme ich kopírovať obyčajným priradením:

```
type
  Riadok = array[1..25] of Char;
  Pole = array[1..10] of Riadok;
var
  a: Pole;
  r: Riadok;
```

```
  i: Integer;
begin
  ...
  r := a[10];
  for i := 9 downto 1 do
    a[i+1] := a[i];
  a[1] := r;
  ...
```

Nasledujúci príklad ukazuje spôsob prezerania prvkov dvojrozmerného poľa (zisťujeme, či je matica NxN symetrická). Najprv ukážeme nie najlepší spôsob riešenia:

```
const
  n = 100;
type
  Matica = array[1..n, 1..n] of Integer;

function symetricka(const m: Matica): Boolean;
var
  i, j: Integer;
begin
  Result := True;
  for i := 2 to N do
    for j := 1 to i-1 do
      if m[i, j] <> m[j, i] then
      begin
        Result := False;
        Break;              // goto von;
      end;
end;
```

Toto riešenie sa snaží použiť príkaz Break, ktorý spôsobí vyskočenie z najvnútornejšieho cyklu - toto je nedostatočné, lebo aj tak sa budú kontrolovať ešte všetky nasledujúce riadky poľa. Ak by sme toto vyskočenie z cyklu break nahradili príkazom **goto**, možno by to fungovalo, ale tento príkaz je pre nás v tomto kurze absolútne **zakázaný**. Keďže po skončení cyklu táto funkcia končí, v tomto prípade môžeme použiť príkaz na vyskočenie z podprogramu - Exit. Pozrite sa ešte na riešenie, ktoré je z programátorského hľadiska správnejšie - ak potrebujeme vyskakovať z viacerých cyklov, nepoužijeme for-cyklus ale while-cyklus:

```
function symetricka(const m: Matica): Boolean;
var
  i, j: Integer;
begin
  Result := True;
  i := 2
  while Result and (i <= n) do
  begin
    j := 1;
    while Result and (j <= i-1) do
    begin
      if m[i, j] <> m[j, i] then
        Result := False;
      Inc(j);
    end;
    Inc(i);
  end;
end;
```

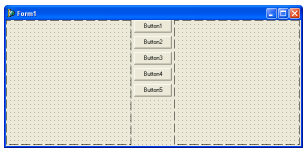
Kopírujeme a modifikujeme obrázky

Aby sa nám lepšie natrénovala práca s dvojrozmerným poľom, vytvoríme v Delphi nový projekt, v ktorom budú

dve grafické plochy: Image1 a Image2. Do prvej načítame z nejakého súboru bitmapu a túto potom prekopírujeme do druhej plochy. Pri kopírovaní ju môžeme rôzne meniť, prípadne meniť spôsoby kopírovania. S obrázkami budeme pracovať ako s dvojrozmerným poľom, pričom prvý index poľa bude vyjadrovať x-ovú súradnicu a druhý index y-ovú súradnicu farebného bodu (pixel). Bodky v grafickej ploche číslujeme od 0 do šírka-1, resp. 0 až výška-1.

Najprv pripravme nový projekt:

- do formulára vložíme dve grafické plochy Image - upravíme im rozmery tak, aby boli rovnako veľké 250x250
- do formulára vložíme niekoľko tlačidiel - postupne im budeme priraďovať rôzne funkcie (dvojkliknutím na príslušné tlačidlo)
- formulár teraz môže vyzerať takto:



Projekt uložíme do nejakého priečinku na disku a tiež sem do tohto istého priečinku prekopírujeme niekoľko obrázkov (súborov s príponou .BMP). Tieto súbory si môžete stiahnuť z **obrazky.zip**. Postupne dvojklikneme na tlačidlá Button1, Button2 a Button3 a priradíme im akcie prečítanie bitmapových obrázkov do prvej grafickej plochy Image1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('tiger.bmp');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('miri.bmp');
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('zem.bmp');
end;
```

Stavová premenná Pixels nám umožňuje pristupovať k jednotlivým pixelom (farebným bodkám) obrázka - funguje to na rovnakom princípe ako Strings v textovej ploche Memo - teda Pixels je ako dvojrozmerné pole prvkov typu TColor (farba). Treba si ale zapamätať, že na rozdiel od obyčajných dvojrozmerných polím, "pole" Pixels má prvý index x-ovú súradnicu (stĺpec) a druhý index y-ovú súradnicu (riadok). Postupne budeme kopírovať obrázok z Image1 do Image2 po riadkoch:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  x, y: Integer;
begin
  for y := 0 to Image1.Height-1 do
  begin
    for x := 0 to Image1.Width-1 do
      Image2.Canvas.Pixels[x, y] := Image1.Canvas.Pixels[x, y];
    Image2.Repaint;
  end;
end;
```

Image2.Repaint prekreslí druhú grafickú plochu po prekopírovaní každého riadka, aby sa postupne zobrazovali

kopírované riadky (inak by sa zobrazila až záverečná zmena). Ďalší variant programu sa hrá s farbami cez ich položky RGB:

```
procedure TForm1.Button5Click(Sender: TObject);
var
  x, y: Integer;
  c: TColor;
  r, g, b: Byte;
begin
  for y := 0 to Image1.Height-1 do
  begin
    for x := 0 to Image1.Width-1 do
    begin
      c := Image1.Canvas.Pixels[x, y];
      r := GetRValue(c);
      g := GetGValue(c);
      b := GetBValue(c);
      // r := (r+g+b) div 3;
      Image2.Canvas.Pixels[x, y] := RGB(r, g, b);    // RGB(r, r, r);
    end;
    Image2.Repaint;
  end;
end;
```

Funkcia GetRValue vráti červenú zložku farby, podobne GetGValue a GetBValue vrátia zelenú a modrú zložku. Konkrétne tento príklad neurobí nič iné, ako skopíruje obrázok z Image1 do Image2. Lenže tu môžete elegantne experimentovať s farbami, napr. niektorú zložku vynulovať, alebo vydeliť dvomi, alebo hoci všetky tri spriemerovať (vznikne čiernobiely obrázok).

Ak sa stane, že grafická plocha pri každom prekresľovaní veľmi bliká, vtedy pomôže riadok, ktorý pridáme do procedúry FormCreate (táto procedúra sa vytvorí dvojkliknutím do formulára na prázdne miesto bez komponentu):

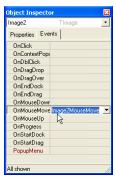
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := True;
end;
```

Môžete si stiahnuť kompletný projekt.

Práca s myšou

V Delphi sa s myšou pracuje pomocou udalostí, ktoré automaticky vznikajú pri každom pohybe myši na obrazovke. Pri pohybe myši túto udalosť dostáva ten komponent formulára, nad ktorým sa myš práve nachádza (napr. grafická plocha alebo tlačidlo). Procedúra, ktorá sa vyvolá ako reakcia na udalosť (event driver) pohyb myši, dostáva ako parametre aj súradnice myši (X, Y) - tieto sú vždy relatívne vzhľadom na ľavý horný roh komponentu - v grafickej ploche sú to normálne grafické súradnice bodu.

Napíšeme procedúru, ktorá sa zavolá vždy, keď pohneme kurzorom myši nad plochou Image2 - táto procedúra prekopíruje len jednu bodku z Image1 a to presne bodku z pozície myši. Najprv vo formulári klikneme na Image2 a potom v Inšpektore Objektov (F11) prepneme záložku (kartu) Events a dvojklikneme na pravú časť riadka s OnMouseMove:



Pripraví sa procedúra TForm1.Image2MouseMove, do ktorej napíšeme, čo všetko chceme urobiť pri každom pohnutí myši nad plochou Image2:

```
procedure TForm1.Image2MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  Image2.Canvas.Pixels[X, Y] := Image1.Canvas.Pixels[X, Y];
end;
```

Aby sa pri pohybe neobjavovala len malá bodka ale nejaké okolie bodu, môžeme zapísať:

```
procedure TForm1.Image2MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
const
  d = 3;
var
  xx, yy: Integer;
begin
  for xx := X-d to X+d do
    for yy := Y-d to Y+d do
      Image2.Canvas.Pixels[xx, yy] := Image1.Canvas.Pixels[xx, yy];
end;
```

Ďalšie námety:

- pri kopírovaní pixelov môžete posúvať začiatok každého riadka o nejakú vypočítanú hodnotu, napr. pomocou funkcie Sin
- obrázky môžete preklápať podľa rôznych osí
- obrázok sa dá zväčšovať/zmenšovať, resp. kopírovať viackrát na rôzne pozície; v nejakej časti zahusťovať a pod.
- je zaujímavé pohrať sa s farbami - robiť rôzne jasy, stmavieť obrázok ku okrajom plochy, zvýrazňovať len nejakú farebnú zložku (napr. "dozelena")
- z obrázka prekopírovať len tie pixely, ktoré sú od nejakého bodu (napr. (150,150)) nie ďalej ako napr. 100 -- ostatné pixely zafarbiť na bielo
- otočiť obrázok Image1 o 90 stupňov
- otočenie Image1 o 90 stupňov na svojom mieste tak, že sa pritom nepoužijete Image2 (ani iný TImage) a ani iná dátová štruktúra (napr. pomocné pole)

Príklad: hra LIFE

Ukážeme pre programátorov veľmi známy program, v ktorom sa simuluje život organizmov vo svete na štvorčekovom papieri. Na každom políčku sa môže nachádzať jedna bunka, prípadne sa tu nová bunka môže narodiť, resp. aj umrieť. Vývoj, keď zo starej generácie buniek vznikne nová generácia, funguje podľa týchto pravidiel:

- každé políčko má 8 susedných políčok (aj po uhlopriečke),
- ak má živá bunka 2 alebo 3 susedov (na susedných políčkach sú živé bunky), tak prežije aj do ďalšej generácie,
- ak má živá bunka 1 alebo žiadneho suseda, prípadne, ak ich má viac ako 3, tak do ďalšej generácie sa nedostáva - umiera,
- ak má prázdne políčko práve troch živých susedov, narodí sa tu nová bunka.

Na simuláciu života použijeme dvojrozmerné pole logických hodnôt: True bude označovať políčko s bunkou, False bude prázdne políčko. Toto pole zadeklarujeme ako globálnu premennú p, aby sme naňho videli z viacerých podprogramov. Pomocná procedúra kresli vykreslí toto pole do grafickej plochy, pričom plátno (Canvas) tejto

plochy dostane ako parameter - robíme to preto, aby sme videli akým spôsobom sa pracuje s takýmto parametrom. V budúcnosti sa nám to môže zísť. Funkcia pocetSusedov pre dané políčko zistí, koľko má susedov so živými bunkami - treba si dávať pozor, aby sme pritom pole neindexovali mimo rozsah indexov <1, n> a aby sme do toho nezapočítali aj samotné zisťované políčko.

Ako už vieme, udalosť FormCreate sa automaticky spúšťa pri štarte projektu a preto ju využívame ako inicializáciu premenných, resp. spustenie nejakých akcií. V našom prípade tu vygenerujeme pole s náhodnými hodnotami (do každého políčka dáme True alebo False podľa výsledku funkcie Random - všimnite komentár v tomto riadku, ktorý dáva rovnaký výsledok) a vykreslíme ho pomocou procedúry kresli. Zatlačenie tlačidla Button1 vygeneruje v štvorčekovej ploche nasledujúcu generáciu života - toto tlačidlo sa bude opakovane stláčať aj pri podržaní klávesu Enter. Tu si všimnite priradenie p := pom; , ktoré prekopíruje celé dvojrozmerné pole pom (nová generácia) do poľa p.

Rovnako, ako sme v predchádzajúcom príklade pridali udalosť onMouseMove, pridáme udalosť onMouseDown: táto udalosť vzniká pri klikaní do plochy ľavým tlačidlom myši, pričom vo formálnych parametroch X a Y získavame informáciu o súradniciach kliknutého miesta. Kliknutím do plochy budeme môcť zmeniť obsah ľubovoľného políčka.

```
const
  n = 50;        // veľkosť štvorčekovej plochy
  vel = 8;       // veľkosť políčka

type
  Pole = array[1..n, 1..n] of Boolean;

var
  p: Pole;

procedure kresli(c: TCanvas);
var
  i, j: Integer;
begin
  for i := 1 to n do
    for j := 1 to n do
    begin
      if p[i, j] then         // i-riadok, j-stĺpec
        c.Brush.Color := clBlack
      else
        c.Brush.Color := clWhite;
      c.Rectangle(j*vel, i*vel, j*vel+vel, i*vel+vel);
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  i, j: Integer;
begin
  Randomize;
  for i := 1 to n do
    for j := 1 to n do
      p[i, j] := Random(2) = 0;     // p[i, j] := Boolean(Random(2));
  kresli(Image1.Canvas);
end;

function pocetSusedov(x, y: Integer): Integer;
var
  i, j: Integer;
begin
  Result := 0;
  for i := y-1 to y+1 do
    for j := x-1 to x+1 do
      if (i >= 1) and (i <= n) and (j >= 1) and (j <= n) then
```

```
        if (x <> j) or (y <> i) then
          if p[i, j] then
            Inc(Result);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  pom: Pole;
  i, j, pocet: Integer;
begin
  for i := 1 to n do
    for j := 1 to n do
    begin
      pocet := pocetSusedov(j, i);
      pom[i, j] := (pocet = 3) or (pocet = 2) and p[i, j];
    end;
  p := pom;
  kresli(Image1.Canvas);
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  i, j: Integer;
begin
  i := Y div vel;
  j := X div vel;
  if (i < 1) or (i > n) or (j < 1) or (j > n) then
    Exit;
  p[i, j] := not p[i, j];
  kresli(Image1.Canvas);
end;
```

Môžete si stiahnuť kompletný projekt.

Príklad s robotom a editovacím riadkom

Vytvoríme novú aplikáciu - do formulára položíme grafickú plochu (Image1) a pod ňu editovací riadok (komponent Edit1 zo štandardnej palety komponentov). Naprogramujeme takéto správanie: v grafickej ploche sa bude nachádzať kresliaci robot a budeme ho riadiť príkazmi, ktoré sa zapisujú do editovacieho riadka. V Inšpektore objektov sa nastavíme na komponent Edit1, prepneme Udalosti (Events) a dvojklikneme OnKeyPress - vytvorí sa metóda Edit1KeyPress - táto metóda sa zavolá vždy, keď sa bude niečo zapisovať do editovacieho riadka. Nás však bude zaujímať len kláves <Enter> s kódom #13.

Program bude rozpoznávať túto množinu príkazov: fd lt rt pu pd setpc setpw. Parameter príkazu bude od neho oddelený aspoň jednou medzerou. Už vieme, že Edit1.Text obsahuje momentálny text v editovacom riadku. Použijeme štandardnú funkciu StrToIntDef, ktorá prevedie reťazec na číslo a ak reťazec nie je správne zadané celé číslo, tak vráti druhý parameter funkcie (tzv. náhradnú hodnotu - default).

Pomocou const môžeme definovať aj pole konštánt - za identifikátor konštanty napíšeme definíciu poľa a za znamienko rovnosti do zátvoriek postupne vymenujeme všetky konštanty.

```
uses
  RobotUnit;

var
  r: TRobot;

procedure TForm1.FormCreate(Sender: TObject);
begin
  r := TRobot.Create;
```

```
  Edit1.Text := '';
  cs;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
const
  tab: array[1..7] of String =
    ('fd', 'lt', 'rt', 'pu', 'pd', 'setpc', 'setpw');
  farba: array[0..7] of TColor =
    (clBlack, clBlue, clGreen, clRed,
     clYellow, clGray, clNavy, clWhite);
var
  s: String;
  i, p: Integer;
begin
  if Key <> #13 then
    Exit;
  s := Edit1.Text;
  i := Pos(' ', s+' ');
  p := StrToIntDef(Copy(s, i+1, MaxInt), 0);
  s := LowerCase(Copy(s, 1, i-1));
  i := 1;
  while (i <= High(tab)) and (s <> tab[i]) do
    Inc(i);
  case i of
    1: r.fd(p);
    2: r.lt(p);
    3: r.rt(p);
    4: r.pu;
    5: r.pd;
    6: if (p >= Low(farba)) and (p <= High(farba)) then
         r.setpc(farba[p]);
    7: r.setpw(p);
  end;
  Edit1.Text := '';
end;
```

Program vylepšíme tak, že zabezpečíme, aby bolo robota vidieť - nakreslíme ho malým trojuholníkom. Trojuholník bude natočený v smere natočenia robota - vždy, keď sa robot pohne alebo otočí, prekreslí sa celá grafická plocha najprv bez robota a na záver ho dokreslíme v novej polohe (pomocná procedúra kresli). V poli prikazy si budeme pamätať postupnosť všetkých doterajších príkazov. Príkaz cs vyprázdni túto zapamätanú postupnosť.

```
var
  r: TRobot;
  prikazy: array[1..1000] of record
    prikaz, param: Integer;
  end;
  pocet: Integer = 0;

procedure kresli;
const
  farba: array[0..7] of TColor =
    (clBlack, clBlue, clGreen, clRed,
     clYellow, clGray, clNavy, clWhite);
var
  i: Integer;
begin
  with r do begin
    PC := clBlack;
    PW := 1;
    setxy(Form1.Image1.Width/2, Form1.Image1.Height/2);
    H := 0;
    PD;
```

```
    cs;
    for i := 1 to pocet do
      with prikazy[i] do
        case prikaz of
          1: fd(param);
          2: lt(param);
          3: rt(param);
          4: PU;
          5: PD;
          6: if (param >= Low(farba)) and (param <= High(farba)) then
               PC := farba[param];
          7: PW := param;
        end;
    // nakreslí rovnoramenný fialový trojuholník
    PC := clPurple;
    PW := 3;
    PD;
    rt(90);
    fd(8);
    lt(105);
    fd(30.9);
    lt(150);
    fd(30.9);
    lt(105);
    fd(8);
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  r := TRobot.Create;
  Edit1.Text := '';
  kresli;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
const
  tab: array[0..7] of String =
    ('cs', 'fd', 'lt', 'rt', 'pu', 'pd', 'setpc', 'setpw');
var
  s: String;
  i, p: Integer;
begin
  if Key <> #13 then
    Exit;
  s := Edit1.Text;
  i := Pos(' ', s+' ');
  p := StrToIntDef(Copy(s, i+1, MaxInt), 0);
  s := LowerCase(Copy(s, 1, i-1));
  i := 1;
  while (i <= High(tab)) and (s <> tab[i]) do
    Inc(i);
  if i > High(tab) then
    Exit;
  if i = 0 then
    pocet:=0     // príkaz cs
  else
  begin
    Inc(pocet);
    prikazy[pocet].prikaz := i;
    prikazy[pocet].param := p;
  end;
  kresli;
  Edit1.Text := '';
end;
```

Všimnite si, že v procedúre kresli potrebujeme pracovať s rozmermi grafickej plochy, t.j. Width a Height. Keďže je to globálna procedúra, ktorá ale nepatrí formuláru TForm1, na to aby sme mohli pracovať s grafickou plochou, musíme uvádzať nielen Image1 ale aj Form1, t.j. Form1.Image1.Width.

Môžete si stiahnuť kompletný projekt.

Ďalší námet:

- namiesto editovacieho riadka použite textovú plochu, do ktorej sa zapisuje celá postupnosť príkazov robota a celá táto postupnosť sa spracováva až do konca alebo po prvý nedokončený, resp. nesprávny príkaz

9. prednáška: vymenovaný typ a typ množina

čo už vieme:

- základné ordinálne typy Integer, Char a Boolean a od nich vieme odvodiť nový typ interval

čo sa na tejto prednáške naučíme:

- naučíme sa nový ordinálny typ - vymenovaný typ
- naučíme sa pracovať s pascalovskými množinami
- ukážeme, ako si môžeme nasimulovať aj veľké množiny

vymenovaný typ (enumerated type)

Vymenovaný typ je taký ordinálny typ, ktorý definujeme vymenovaním všetkých prípustných hodnôt daného typu. Vymenované hodnoty sú identifikátory konštánt (ich ordinálne hodnoty sú čísla od 0 do počet-1), napr.

```
type
  Tyzden = (pon, uto, str, stv, pia, sob, ned);
```

definuje nový ordinálny typ - premenné tohto typu budú môcť nadobúdať hodnotu len z tohto zoznamu konštánt. Vymenovaný typ nie je kompatibilný so žiadnym iným typom (napr. interval preberá všetky vlastnosti aj konštanty nadradeného typu a tiež je s ním kompatibilný - dovoľuje navzájom sa priraďovať, miešať sa v operáciách a pod.) Deklarácia vymenovaného typu okrem samotného typu automaticky definuje aj identifikátory konštánt tohto typu, t.j. s deklaráciou typu Tyzden sa zadeklarovalo aj 7 nových identifikátorov konštánt. Nakoľko je vymenovaný typ ordinálny typ, fungujú s ním všetky štandardné funkcie a procedúry, ktoré pracujú s inými ordinálnimi typmi (Integer, Char, Boolean):

- štandardné funkcie, procedúry: Ord, Pred, Succ, Low, High, Inc, Dec
- môžeme ho použiť ako index prvkov poľa, resp. riadiaca premenná for-cyklu
- fungujú všetky relácie: =, <>, <=, >=, <, >
- aj na typ Boolean by sme sa mohli pozerať ako na vymenovaný typ, t.j. ako keby

  type Boolean = (False, True);

- vymenovaný typ sa nedá vypísať ani prečítať do, resp. zo súboru
- ordinálna hodnota je definovaná tak, že prvá konštanta typu má vnútornú hodnotu 0 a všetky ďalšie postupne o 1 viac, t.j.

```
Ord(pon) = 0; Ord(uto) = 1; Ord(str) = 2; ... Ord(ned) = 6;
```

- meno typu (v našom prípade Tyzden) je automaticky menom konverznej funkcie, ktorá z celého čísla vyrobí príslušnú konštantu vymenovaného typu, napr.

```
Tyzden(2) = str; Tyzden(5) = sob;
```

Príklad s vymenovaným typom:

```
type
  Tyzden = (pon, uto, str, stv, pia, sob, ned);
var
  d: Tyzden;
begin
  d := pon;
  d := Pred(pia);    // d=stv
  d := Succ(sob);    // d=ned
  d := uto;
  Inc(d);            // d=str
  Inc(d, 3);         // d=sob
  i := Ord(pon);     // i=0     prvá konštanta má hodnotu 0
  i := Ord(uto);     // i=1
  d := Tyzden(4);    // d=pia - identifikátor typu ako funkcia
  d := Low(Tyzden);  // d=pon
  d := High(d);      // d=ned
end;
```

Ak potrebujeme hodnoty vymenovaného typu prečítať alebo vypísať, tak buď pri vstupe/výstupe pracujeme iba s ordinálnymi hodnotami, alebo si vytvoríme pomocné pole mien konštánt, napr.

```
const
  a: array [Tyzden] of String =
      ('pon', 'uto', 'str', 'stv', 'pia', 'sob', 'ned');
var
  s: String;
begin
  s := Edit1.Text;
  d := Low(Tyzden);
  while (d < High(Tyzden)) and (a[d] <> s) do
    Inc(d);
  if a[d] <> s then
    Memo1.Lines.Append('... chyba ...')
  else
    Memo1.Lines.Append(a[Succ(d)]);    // spadne, ak d=ned
end;
```

Zistite, v čom je chybné takéto riešenie:

```
const
  a: array [Tyzden] of String =
      ('pon', 'uto', 'str', 'stv', 'pia', 'sob', ' ned');
var
  s: String;
begin
  s := Edit1.Text;
  d := Low(Tyzden);
  while (d <= High(Tyzden)) and (a[d] <> s) do Inc(d);
  if d > High(Tyzden) then
    Memo1.Lines.Append('... chyba ...')
  else
    Memo1.Lines.Append(a[Succ(d)]);
end;
```

Vymenovaný typ sme už používali predtým, napr. nasledujúce typy sú už preddefinované v knižniciach Delphi:

```
type
  TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot,
               psClear, psInsideFrame);
  TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical,
                 bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross);

// a my sme mohli písať:
  g.Pen.Style := psDash;
  g.Brush.Style := bsClear;
```

V nasledujúcom príklade funkcia porovnaj vráti jednu z hodnôt vymenovaného typu (mensi, rovny, vacsi) podľa toho, v akej relácii sú dva reťazce:

```
type
  Porovnanie = (mensi, rovny, vacsi);

function porovnaj(s1, s2: String): Porovnanie;
begin
  if s1 < s2 then
    Result := mensi
  else if s1 = s2 then
    Result := rovny
  else
    Result := vacsi;
end;
```

iné možnosti definovania konštánt vymenovaného typu

Zatiaľ sme sa naučili, že prvá konštanta vymenovaného typu má ordinálnu hodnotu 0 a posledná počet-1. Toto pravidlo môžeme podľa potreby zmeniť: pri definovaní vymenovaného typu môžeme zároveň definovať aj ordinálne hodnoty konštánt (funguje to až od verzie Delphi 6), napr.

```
type
  Porovnanie = (mensi=-1, rovny, vacsi=3);
```

Pritom platí:

- ak je za konštantou (za znakom rovná sa) celé číslo, toto definuje jeho ordinálnu hodnotu
- ak prvá konštanta nemá definovanú žiadnu hodnotu (napr. pon v Tyzden), tak dostáva hodnotu 0
- ak niektorá konštanta (okrem prvej) nemá definovanú svoju ordinálnu hodnotu, automaticky dostáva o 1 vyššiu ako predchádzajúca konštanta
- viac identifikátorov konštánt môže mať rovnaké ordinálne hodnoty
- medzi minimálnou a maximálnou hodnotou nemusia byť všetky ordinálne hodnoty pomenované konštantami, napr. v type Porovnanie má konštanta rovny hodnotu 0 a hodnotu 2 sme nepomenovali - aj tak môže premenná takéhoto vymenovaného typu nadobúdať aj nepomenované hodnoty, napr.

```
var
  p: Porovnanie;
begin
  p := Porovnanie(2);
```

Z tohto vyplýva, že pre vymenovaný typ je dôležitá minimálna a maximálna konštanta - tieto dve definujú interval a všetky ostatné konštanty len pomenúvajú niektoré hodnoty z daného intervalu, napr.

```
type
  Cislo = (minc=1, maxc=100, spec=10);
```

definuje vymenovaný typ, ktorý je intervalom s ordinálnymi hodnotami 1..100, okrem toho je ešte definovaná jedna konštanta tohto typu spec s ordinálnou hodnotou 10. Aj pomocou const môžeme dodatočne zadefinovať ďalšie konštanty do vymenovaného typu, napr.

```
const
  Extra = Cislo(27);
```

k trom konštantám typu Cislo dodefinoval ďalšiu s ordinálnou hodnotou 27.

Delphi poskytujú aj ďalšie knižničné funkcie na prácu s vymenovaným typom: GetEnumName a GetEnumValue umožňujú z ordinálnej hodnoty zistiť reťazec, resp. naopak, z reťazca získať ordinálnu hodnotu - Delphi toto robia "vo svojej réžii", takže nie je potrebné si udržiavať nejaké vlastné tabuľky; popis týchto funkcií nájdete v Helpe

typ množina

je taký údajový typ, ktorý v pascale umožňuje pracovať s nektorými typmi množín podobným spôsobom ako je to obvyklé v matematike. V pascale je povolené vytvárať množiny zložené len z prvkov rovnakého a to niektorého ordinálneho typu - hovoríme mu **bázový** (základný) typ. Napr. môžeme vytvoriť množinu znakov, množinu malých celých čísel, množinu dní v týždni a pod. Nemôžeme ale vytvoriť množinu, ktorá bude obsahovať napr. čísla aj znaky. Definícia množiny sa vždy skladá aj z definície jej bázového typu, napr. zápis

```
type
  Mnozina = set of 1..100;
var
  a, b: Mnozina;
```

definuje nový typ množina, ktorá môže obsahovať len čísla z intervalu (to je ordinálny typ) 1 až 100 - zrejme premenné takéhoto typu môžu byť napr. prázdna množina, alebo jednoprvková množina s prvkom 13 alebo dvojprvková množina s prvkami 2 a 3 a pod. Do premenných a, b môžeme priraďovať (iba množiny), môžeme s nimi manipulovať pomocou množinových operácií a relácií - nemôžeme ich priamo vypísať alebo prečítať do/zo súboru, musíme si to naprogramovať.

Ako pracujeme s typom množina:

- môžeme používať množinové konštanty:
  - prázdna množina [],
  - vymenovanie prvkov a intervalov [2], [1, 3], [1, 2, 3], [5..15, 20, 23..27]
  - ale aj [1, 1, 1], [3, 3, 1], [1..2, 2..3, 1..3]
- množinové operácie musia mať oba operandy navzájom kompatibilné množiny (t.j. majú kompatibilné bázové typy):
  - zjednotenie a := a + b;
    - pridaj prvok: a := a + [5];
  - prienik a := a * [2, 3];
    - vyhoď prvok: a := a - [5];
  - rozdiel a := b - [1, 3];
- príslušnosť prvku (prvý operand je bázového typu, druhý je množina):
  - číslo 3 je prvkom množiny a: if 3 **in** a then ...
  - hodnota premennej i nie je prvkom množiny b: if **not** (i **in** b) then ...
- relácie:
  - rovnosť, nerovnosť: if a = b then if c <> [] then ...
    - if a * [5] <> [] then ...
  - podmnožiny: if (a <= b) or (a >= b) then ...

Príklady:

```
var
  z: Char;
...
  if z in ['A'..'Z', 'a'..'z'] then ...
  if not (z in ['0'..'9']) then ...
```

Testovanie odpovede áno/nie:

```
const
  ano = ['a', 'A', 'y', 'Y'];
  nie = ['n', 'N'];
...
  s := Edit1.Text;
  if (s <> '') and (s[1] in ano) then ...
  else if (s <> '') and (s[1] in nie) then ...
  else
    MessageBox('chybná odpoveď - ano/nie');
```

Generovanie množiny pomocou cyklu:

```
var
  x: set of 1..100;
...
  x := [];
  for i := 1 to 100 do
    x := x + [i];
...
  x := [];
  for i := 1 to 50 do
    x := x + [i, 101-i];
...
  x := [];
  for i := 1 to 100 do
    if i mod 7 = 0 then
      x := x + [i];
```

Ak chceme vypísať množinu do textovej plochy, použijeme pomocný reťazec a cyklus:

```
type
  Baza = 1..100;
  Mnozina = set of Baza;

procedure vypis(const m: Mnozina);
var
  i: Baza;   // 1..100
  s: String;
begin
  s := '[';
  for i := Low(Baza) to High(Baza) do
    if i in m then
      s := s + IntToStr(i) + ', ';
  Memo1.Lines.Append(s + ']');
end;
```

Výpis prvkov množiny do súboru tak, aby sa za posledným číslom nevypisovala čiarka:

```
procedure vypis(const t: TextFile; const m: Mnozina);
var
  i: Baza;
  b: Boolean;
```

```
begin
  Write(t, '[');
  b := False;
  for i := Low(Baza) to High(Baza) do
    if i in m then
    begin
      if b then
        Write(t, ', ');
      Write(t, i);
      b := True;
    end;
  Write(t, ']');
end;
```

alebo pomocou funkcie

```
function MnozinaToStr(m: Mnozina): String;
var
  i: Integer;
begin
  Result := '[';
  for i := 0 to 255 do
    if i in m then
    begin
      if Result <> '[' then
        Result := Result + ', ';
      Result := Result + IntToStr(i);
    end;
  Result := Result + ']';
end;
```

Ďalšie námety:

- vypisovanie množiny aj s intervalmi, napr. [1, 3..5, 10..41, 43, 45]
- čítanie množiny zo vstupu (na vstupe môžu byť aj intervaly)

V štandardných knižniciach Delphi je štýl fontov definovaný ako množina. Ukážme, ako vyzerajú deklarácie a ako sa s tým pracuje:

```
type
  TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);
  TFontStyles = set of TFontStyle;
  // a pre font je definovaný
  // Style: TFontStyles;
...
  Image1.Canvas.Font.Style := [];
  Image1.Canvas.Font.Style := [fsBold];
  Image1.Canvas.Font.Style := Image1.Canvas.Font.Style - [fsItalic];
  Memo1.Font.Style := [fsItalic, fsUnderline];
```

príklad

Program, ktorý z textového súboru stud.txt prečíta mená študentov (študentov nie je viac ako 100), ich priemerné známky a informáciu o tom, či študent dostal alebo nedostal internát (v tvare 0 alebo 1). Program potom vypíše

a. všetkých študentov
b. dobrých študentov (priemer do 1.5), ktorí majú internát
c. dobrých študentov, ktorí nemajú internát

```
type
  Baza = 1..100;
```

```
  Tabulka = array[Baza] of record
    meno: String;
    znamka: Real;
    internat: Boolean
  end;
  Mnozina = set of Baza;

procedure citaj(var t: Tabulka; var n: Integer);
var
  f: TextFile;
  bb: Integer;
  z: Char;
begin
  AssignFile(f, 'stud.txt');
  Reset(f);
  n := 0;
  while not seekeof(f) do
  begin
    Inc(n);
    with t[n] do
    begin
      meno := '';
      repeat
        Read(f, z);
        if z <> ';' then
          meno := meno + z;
      until z = ';';
      Readln(f, znamka, bb);
      internat := bb = 1;
    end;
  end;
  CloseFile(f);
end;

procedure vypis(const t: Tabulka; m: Mnozina);
var
  i: Baza;
begin
  for i := Low(Baza) to High(Baza) do
    if i in m then
      Memo1.Lines.Append(t[i].meno);
end;

var
  t: Tabulka;
  a, b: Mnozina;
     // a - indexy do tabuľky t, pre ktorých znamka <= 1.5
     // b - indexy do tabuľky t - študentov, ktorí majú internát
  n, i: Integer;
begin
  citaj(t, n);      // čítanie tabuľky
  a := [];
  b := [];
  for i := 1 to n do
  begin
    if t[i].znamka <= 1.5 then
      a := a + [i];
    if t[i].internat then
      b := b + [i]
  end;
  Memo1.Lines.Append('*** Všetci');
  vypis(t, [1..n]);
  Memo1.Lines.Append('*** dobrí, majú internát');
  vypis(t, a * b);
  Memo1.Lines.Append('*** dobrí, nemajú internát');
  vypis(t, a - b);
```

```
end;
```

príklad

Napíšeme program, ktorý vytvorí podmnožinu M celých čísel z 1..250 takú, že

- 1 je z množiny M
- ak i je z množiny, tak 2i+1 aj 3i+1 sú z množiny M

Najprv zapíšme riešenie, ktoré vyzerá správne, ale sú v ňom chyby:

```
var
  m: set of 1..250;
  i: Integer;
begin
  m: = [1];
  for i := 1 to 250 do
    if i in m then
      m := m + [2*i+1, 3*i+1]
end;
```

Do množiny nemôžeme pridávať ľubovoľné celočíselné hodnoty, ale len po istú hranicu. Napr. ak i=202, tak by sme do množiny chceli pridávať aj čísla 2*202+1 a 3*202+1, t.j. 405 a 607 - lenže tieto dve hodnoty výrazne prevyšujú maximálnu hodnotu v množine. Preto program upravíme tak, aby sme do množiny pridávali len korektné hodnoty do 250:

```
var
  m: set of 1..250;
  i: Integer;
  s: String;
begin
  m:=[1];
  for i := 1 to 124 do
    if i in m then
    begin
      m := m + [2*i+1];           // 2*i+1 je vždy z 1..250
      if 3*i+1 <= 250 then        // ak je aj 3*i+1 z 1..250,
        m := m + [3*i+1];         // pridáme ho do množiny
    end;
  s := '';
  for i := 1 to 250 do
    if i in m then
      s := s + IntToStr(i) + ' ';
  Memo1.Text := s;
end;
```

Všimnite si, že do textovej plochy sme zapísali riadok nie pomocou známeho Memo1.Lines.Append(...), ale sme priradili Memo1.Text := ... Týmto spôsobom prepíšeme celý pôvodný obsah textovej plochy novým reťazcom. Ten, ak by obsahoval #13#10, vytváral by nové riadky.

realizácia množiny v pamäti počítača

Pascalovské množiny sú v pamäti počítača reprezentované ako postupnosť bitov, t.j. postupnosť 0 a 1. Napr. pre set of 0..99 je vyhradených 100 bitov, t.j. 13 bajtov. V i-tom bite je buď 0, ak i nepatrí do množiny alebo 1, ak i patrí do množiny. Priradenie a := []; vynuluje všetky bity. Priradenie a := [5, 7, 8]; nastaví na 1 iba bity 5, 7, 8, ostatné vynuluje. Množinová operácia a + b postupne ide po bitoch v oboch postupnostiach a aj b a robí operáciu binárne OR, t.j. ak aspoň v jednej z nich je 1, tak aj vo výsledku bude 1. Pre množiny set of baza musí platiť: 0 <= Low(baza) <= High(baza) <= 255. To znamená, že takto nevyrobíme množiny s viac ako 256 prvkami. Najväčšia

možná množina set of Byte zaberá 32 bajtov.

Väčšie množiny môžeme reprezentovať pomocou poľa množín - budeme tomu hovoriť "veľké množiny". Napr. ak by sme potrebovali množinu s 1000 prvkami, t.j. postupnosť 1000 bitov, položíme tesne za seba 4 množiny po 256 prvkoch a budeme predpokladať, že

- v prvej množine budú prvky od 0 do 255
- v druhej množine budú prvky od 256 do 511 - budú ale posunuté tak, že 256 bude mať v druhej množine číslo 0, t.j. o 256 menej
- v tretej množine budú prvky od 512 do 767 - prvky budú posunuté o 512
- ...

Aby sa nám s takýmito veľkými množinami čo najpohodlnejšie pracovalo, pole množín budeme indexovať od 0: potom pre prvok x ľahko vypočítame poradové číslo množiny ako x div 256 a jeho posunuté číslo v tejto množine ako x mod 256. Ak chceme pridať prvok do množiny, alebo zistiť, či nejaké číslo patrí do množiny zapíšeme:

```
var
  m: array[0..max] of set of 0..255;
  x: Integer;
...
// pridávame x do takejto množiny:
  m[x div 256] := m[x div 256] + [x mod 256]
// zistíme, či je x v množine:
  if (x mod 256) in m[x div 256] then ...
```

Operátor in má rovnakú prioritu ako všetky relačné operátory a preto by sme x mod 256 nemuseli písať do zátvoriek - takto je to ale čitateľnejšie.

Teraz môžeme prepísať predchádzajúci príklad s konštruovaním množiny pre veľkú množinu. Vytvoríme ju z 20 obyčajných 256-prvkových množín, t.j. zmestí sa nám 20*256 prvkov, t.j. čísla od 0 do 5119.

```
type
  Mnozina = array[0..19] of set of Byte;
var
  m: Mnozina;
  i, j :Integer;
  s: String;
begin
  m[0] := [1];
  for i := 1 to 19 do
    m[i] := [];
  for i := 1 to 2559 do
    if (i mod 256) in m[i div 256] then
    begin
      j := 2*i+1;
      m[j div 256] := m[j div 256] + [j mod 256];
      j := 3*i+1;
      if j div 256 <= 19 then
        m[j div 256] := m[j div 256] + [j mod 256];
    end;
  s := '';
  for i := 1 to 5119 do
    if (i mod 256) in m[i div 256] then
      s := s + IntToStr(i) + ' ';
  Memo1.Text := s;
end;
```

Takéto riešenie už nie je tak jednoducho čitateľné, ako riešenie s obyčajnými množinami. Vytvorme si preto sadu pomocných podprogramov na prácu s veľkými množinami:

```
type
  Mnozina = record
    m: array[0..100] of set of Byte;
    max: Integer;     // maximálne prípustné číslo v množine
  end;

procedure inicMn(var mn: Mnozina);
var
  i: Integer;
begin
  with mn do
  begin
    max := Length(m)*256-1;    // Length(m) - počet prvkov poľa m
    for i := 0 to High(m) do
      m[i] := [];
  end;
end;

procedure pridajMn(var mn: Mnozina; i: Integer);
begin
  with mn do
    if (i >= 0) and (i <= max) then
      m[i div 256] := m[i div 256] + [i mod 256];
end;

function vMn(const mn: Mnozina; i: Integer): Boolean;
begin
  if (i < 0) or (i > mn.max) then
    Result := False
  else
    Result := (i mod 256) in mn.m[i div 256];
end;

function pocetMn(const mn: Mnozina): Integer;  // počet prvkov
var
  i: Integer;
begin
  Result := 0;
  for i := 0 to mn.max do
    if vMn(mn, i) then
      Inc(Result);
end;

function textMn(const mn: Mnozina): String;
var
  i: Integer;
begin
  Result := '[';
  for i := 0 to mn.max do
    if vMn(mn, i) then
      Result := Result + IntToStr(i) + ', ';
  if Result <> '[' then                   // ak množina nie je prázdna,
    SetLength(Result, Length(Result)-2);   // vyhoď posledné 2 znaky
  Result := Result + ']';
end;

procedure vsetkyMn(var mn: Mnozina);
var
  i: Integer;
begin
  for i := 0 to High(mn.m) do
    mn.m[i] := [0..255];
end;

procedure uberMn(var mn: Mnozina; i: Integer);
begin
  with mn do
```

```
    if (i >= 0) and (i <= max) then
      m[i div 256] := m[i div 256] - [i mod 256];
end;
```

Teraz prepíšeme program na konštruovanie množiny:

```
var
  mn: Mnozina;
  i: Integer;
begin
  inicMn(mn);
  pridajMn(mn, 1);
  for i := 1 to (mn.max-1) div 2 do
    if vMn(mn, i) then
    begin
      pridajMn(mn, 2*i+1);
      if 3*i+1 <= mn.max then
        pridajMn(mn, 3*i+1);
    end;
  Memo1.Text := textMn(mn);
  Memo1.Lines.Append('počet prvkov = ' + IntToStr(pocetMn(mn)));
end;
```

Všimnite si, že procedúra pridajMn kontroluje, či je pridávaný prvok v správnom intervale a preto môžeme tento program ešte zjednodušiť:

```
var
  mn: Mnozina;
  i: Integer;
begin
  inicMn(mn);
  pridajMn(mn, 1);
  for i := 1 to mn.max do
    if vMn(mn, i) then
    begin
      pridajMn(mn, 2*i+1);
      pridajMn(mn, 3*i+1);
    end;
  Memo1.Text := textMn(mn);
  Memo1.Lines.Append('počet prvkov = ' + IntToStr(pocetMn(mn)));
end;
```

príklad - Eratostenovo sito

Grécky matematik Eratostenes už pred viac ako 2000 rokmi popísal algoritmus na hľadanie prvočísel:

- zapíšme si do radu postupnosť celých čísel od 2 po nejaké maximum
- zoberme prvé číslo (teda 2) - označme ho a všetky jeho násobky vyškrtnime
- zoberme ďalšie ešte nevyškrtnuté číslo (teda 3) - a urobme to isté, t.j. označme ho a vyškrtnime všetky jeho násobky
- toto opakujme, kým nie sú všetky čísla buď označené alebo vyškrtnuté

To čo ostane označené, sú prvočísla. Naprogramujme tento algoritmus najprv pomocou obyčajnej množiny:

```
var
  m: set of Byte;
  i, j: Integer;
  s: String;
begin
  m := [2..255];
  for i := 2 to 127 do
```

```
    if i in m then
    begin
      j := 2*i;
      while j <= 255 do
      begin
        m := m - [j];
        Inc(j, i);
      end;
    end;
  s := '';
  for i := 0 to 255 do
    if i in m then
      s := s + IntToStr(i) + ' ';
  Memo1.Text := s;
end;
```

A teraz pomocou pomocných podprogramov s veľkou množinou:

```
var
  mn: Mnozina;
  i, j: Integer;
begin
  inicMn(mn);
  vsetkyMn(mn);
  uberMn(mn, 0);
  uberMn(mn, 1);
  for i := 2 to mn.max div 2 do
    if vMn(mn, i) then
    begin
      j := i + i;
      while j <= mn.max do
      begin
        uberMn(mn, j);
        Inc(j, i);
      end;
    end;
  Memo1.Text := textMn(mn);
  Memo1.Lines.Append('počet prvkov = ' + IntToStr(pocetMn(mn)));
end;
```

10. prednáška: skener, bitmapy

čo už vieme:

- keď spracovávame zložitejší textový súbor (napr. pascalovský program), môžeme čítať buď po znakoch alebo prečítať celý riadok naraz
- súbor s obrázkom (s príponou .BMP) môžeme pomocou LoadFromFile prečítať do grafickej plochy

čo sa na tejto prednáške naučíme:

- textové súbory so zložitejšou štruktúrou je niekedy výhodné predspracovať lexikálnou analýzou
- v programe môžeme manipulovať aj s obrázkami, môžeme ich napríklad "opečiatkovať" do grafickej plochy

Lexikálna analýza - skener (scanner)

Lexikálna analýza je algoritmus, ktorý slúži na predspracovanie textového vstupu: rozloží text na logické časti, tzv. **lexikálne jednotky** - **lexémy**. Najlepšie sa využije pri spracovaní zdrojových textov programov.

My to najčastejšie využijeme na spracovanie pascalovských programov. Lexikálnymi jednotkami v pascale sú, napr.

- identifikátory (vrátane rezervovaných slov) - napr. RobotUnit, Integer, begin, String, setxy, ...
- číselné konštanty - napr. 0, 37255, 3.14159, 2e-10, $7FFF, ...
- konštanty znak a znakový reťazec - napr. '', 'a', #13, '"', 'hello'#13#10'world', ...
- špeciálne symboly - napr. rôzne zátvorky, bodkočiarka, čiarka, priradenie, relačné a aritmetické operátory a pod.

Pomocou lexikálnej analýzy môžeme napr. veľmi jednoducho spracovať identifikátory premenných alebo procedúr, rôznym spôsobom vypisovať pascalovský program, farebne vyznačovať niektoré časti a pod.

Princíp práce je tento:

- hlavná je procedúra **skener**, ktorá vždy, keď je zavolaná, zanalyzuje ďalšiu časť textu
- zrejme si musí pamätať, kde v texte sa nachádza
- táto procedúra vracia informácie o nasledujúcej lexéme v texte - priradí ich do nejakých globálnych premenných: **typ lexémy** a ďalšiu doplnkovú informáciu lexémy (napr. hodnota pre číslo alebo reťazec, reťazec identifikátora, typ relačného operátora a pod.)
- po spracovaní lexémy skener ostane nastavený **na znaku tesne za lexémou** - preto vždy, keď ho zavoláme, už počíta s tým, že je asi nastavený na prvom znaku nejakej ďalšej lexémy (alebo napr. na medzere pred ňou)
- niektoré reťazce sú pri spracovávaní textu nedôležité a preto ich môže skener filtrovať, napr. medzery, tabulátor, konce riadkov a tiež všetky typy komentárov {...}, //..., (*...*) - často slúžia len ako oddeľovače lexém

Upresnime pravidlá pre vytvorenie skenera:

- najprv určíme, aké rôzne typy lexém nás budú zaujímať (ak chceme zisťovať napr. len identifikátory premenných, nepotrebujeme poznať ani čísla, ani znakové reťazce a ani žiadne špeciálne symboly)
- podľa toho zadefinujeme globálnu premennú vymenovaného typu, napr.

```
var
  lexema: (koniec, ident, cislo, retazec, symbol, ...);
```

- kde každá hodnota tejto premennej označuje typ lexémy (lexéma koniec označuje koniec textového súboru)
- vytvoríme pomocnú procedúru znak, ktorá číta textový súbor a do globálnej znakovej premennej **z** priradí načítaný znak:
  - #0, keď je koniec súboru
  - #1, na konci riadka, pričom spracuje tento koniec riadka
  - inak prečíta znak zo súboru
- okrem globálnej znakovej premennej z aj premenná t typu TextFile je globálna
- podľa spracovávaných typov lexém zadefinujeme ešte ďalšie globálne premenné, do ktorých bude skener priraďovať doplnkové informácie, napr. hodnota celočíselných konštánt, hodnota reálnych konštánt, hodnota znakového reťazca, typ číselnej operácie, typ relačnej operácie, typ rezervovaného slova, ... - závisí to od toho, aký komplexný skener sa chystáme naprogramovať

Poďme krok za krokom naprogramovať prvú jednoduchú aplikáciu so skenerom - program zistí, či ku každému begin v pascalovskom programe správne prislúcha end. Začneme procedúrou znak:

```
var
  z: Char;
  t: TextFile;

procedure znak;   // procedúra na čítanie zo súboru
begin
  if Eof(t) then
    z := #0
  else if Eoln(t) then
  begin
    z := #1;
    Readln(t);
  end
  else
    Read(t, z);
end;
```

Ďalej zapíšme najprv všeobecnú schému procedúry skener - spracováva znaky a stará sa o tie postupnosti znakov, ktoré tvoria nejaké lexémy (každé volanie zistí nasledujúcu lexému):

```
procedure skener;
var
  ok: Boolean;
begin
  ok := False;
  repeat
    case z of
      #0:                  // už je koniec súboru
        begin
          lexema := koniec;
          ok := True;
        end;
      '{':                 // začína komentár
        begin
          // nájdeme k nemu druhú zátvorku alebo koniec súboru
        end;
      '''':                // začína reťazec
        ...
      'A'..'Z', 'a'..'z':  // začína identifikátor
        ...
      // ďalšie typy lexém
      else
        ...
    end;   // case
  until ok;
end;
```

Lokálna premenná ok slúži na to, aby sme ukončili repeat-cyklus, keď nájdeme nejakú lexému. Vtedy je v globálnej premennej **lexema** typ lexémy a v iných premenných je ďalšia informácia (napr. reťazec identifikátora premennej). Náš program, ktorý pracuje so skenerom môže vyzerať napr. takto:

```
begin
  AssignFile(t, ...);
  Reset(t);
  znak;
  skener;
  while lexema <> koniec do
  begin
    // spracovanie tých lexém, ktoré nás zaujímajú
    skener;
  end;
  CloseFile(t);
```

```
end;
```

A teraz konkrétne pre program, ktorý bude kontrolovať, či navzájom zodpovedajú begin a end. Nakoľko v pascalovských programoch môžu byť aj iné konštrukcie, ktorých súčasťou je slovo end, musíme počítať aj s nimi. Sú to record, case a class a taktiež na konci každého unitu je jeden end (s bodkou) navyše. Použijeme takýto algoritmus: pri každom begin, record , case a class pripočítame 1 k nejakému počítadlu a pri end odpočítame 1. Ak momentálny súčet bude menší ako 0, znamená, že práve bolo viac end ako "otvorených" begin. Po skončení analýzy musí byť súčet -1, lebo na záver je jedno end s bodkou.

Náš skener nebude potrebovať vracať všetky typy lexém. Bude nám stačiť len ident, t.j. identifikátor a teda aj rezervované slovo. Budeme filtrovať nielen komentáre {...} a //... ale tiež aj konštanty znakové reťazce, lebo tieto by mohli v sebe obsahovať napr. slovo end a takéto by sme nemali započítať.

Zjednodušený skener:

```
var
  lexema: (koniec, ident);
  hodnota: String;

procedure skener;
var
  ok: Boolean;
begin
  ok := False;
  repeat
    case z of
      #0:
        begin
          lexema := koniec;
          ok := True;
        end;
      '{':
        begin
          repeat
            znak;
          until z in ['}', #0];
          znak;
        end;
      '/':
        begin
          znak;
          if z='/' then
            repeat
              znak;
            until z in [#1, #0];
        end;
      '''':
        begin
          repeat
            znak;
          until z in ['''', #0, #1];
          znak;
        end;
      'A'..'Z', 'a'..'z', '_':
        begin
          hodnota := '';
          lexema := ident;
          ok := True;
          while z in ['A'..'Z', 'a'..'z', '0'..'9', '_'] do
          begin
            hodnota := hodnota + z;
            znak;
          end;
```

```
        end;
      else
        znak;
    end;
  until ok;
end;
```

Program, ktorý spracováva text, dostáva od skenera len 2 typy lexém: koniec a ident, preto v tele cyklu je jasné, že lexéma môže byť len ident.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  poc: Integer;
begin
  AssignFile(t, 'unit1.pas');
  Reset(t);
  znak;
  skener;
  poc := 0;
  while lexema <> koniec do
  begin
    hodnota := LowerCase(hodnota);
    if (hodnota = 'begin') or (hodnota = 'case') or
       (hodnota = 'record') or (hodnota = 'class') then
      Memo1.Lines.Append(hodnota + ' ... ' + IntToStr(poc));
      if poc < 0 then
        Memo1.Lines.Append('>>>> zle je');
      Inc(poc);
    end
    else if hodnota = 'end' then
    begin
      Dec(poc);
      Memo1.Lines.Append(hodnota + ' ... ' + IntToStr(poc));
    end;
    skener;
  end;
  CloseFile(t);
  Memo1.Lines.Append('============================');
  if poc <> -1 then
    Memo1.Lines.Append('zlý výsledný počet = ' + IntToStr(poc))
  else
    Memo1.Lines.Append('ok');
end;
```

Príklad

V ďalšom príklade budeme v textovom súbore unit2.pas, ktorý obsahuje pascalovský program, nahrádzať všetky výskyty nejakého identifikátora (v premennej r1) reťazcom v premennej r2. Zrejme, že nahrádzať nebudeme ani v reťazcoch ani v komentároch. Zadefinujeme pomocnú procedúru pis, ktorá vypíše znak **z** do výstupného súboru t1:

```
var
  t1: TextFile;

procedure pis;
begin
  if z <> #0 then
    if z = #1 then
      Writeln(t1)
    else
      Write(t1, z);
end;
```

Všimnite si procedúru **skener** - komentáre aj reťazce sa len kopírujú zo vstupu na výstup (po každom znak je volanie pis). Podobne sa kopírujú aj všetky symboly a aj čísla, ktoré nie sú súčasťou identifikátorov. Skener rozpozná len identifikátory a samozrejme aj koniec vstupu:

```
var
  lexema: (koniec, ident);
  hodnota: String;

procedure skener;
var
  ok: Boolean;
begin
  ok := False;
  repeat
    case z of
      #0:
        begin
          lexema := koniec;
          ok := True;
        end;
      '{':
        begin
          repeat
            pis;
            znak;
          until z in ['}', #0];
          pis;
          znak;
        end;
      '/':
        begin
          pis;
          znak;
          if z = '/' then
            repeat
              pis;
              znak;
            until z in [#1, #0];
        end;
      '''':
        begin
          repeat
            pis;
            znak;
          until z in ['''', #0, #1];
          pis;
          znak;
        end;
      'A'..'Z', 'a'..'z', '_':
        begin
          hodnota := '';
          lexema := ident;
          ok := True;
          while z in ['A'..'Z', 'a'..'z', '0'..'9', '_'] do
          begin
            hodnota := hodnota + z;
            znak;
          end;
        end;
      else
        pis;
        znak;
    end;
  until ok;
```

```
end;
```

Samotný program, prečíta z dvoch komponentov editovací riadok dva reťazce - identifikátor, ktorý nahrádzame a reťazec, ktorým nahrádzame (Edit1 a Edit2). V textovej ploche Memo1 sa vypíše obsah súboru pred aj po nahrádzaní:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('unit2.pas');
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2: String;      // všetky výskyty r1 zmení na r2
  pocet: Integer;
begin
  r1 := LowerCase(Trim(Edit1.Text));
  r2 := Trim(Edit2.Text);
  if (r1 = '') or (r2 = '') then
  begin
    ShowMessage('Najprv zadaj oba reťazce');
    Exit;
  end;
  AssignFile(t, 'unit2.pas');
  Reset(t);
  AssignFile(t1, 'unit2a.pas');
  Rewrite(t1);
  znak;
  skener;
  pocet := 0;
  while lexema <> koniec do
  begin
    if LowerCase(hodnota) = r1 then
    begin
      Inc(pocet);
      Write(t1, r2);
    end
    else
      Write(t1, hodnota);
    skener;
  end;
  CloseFile(t);
  CloseFile(t1);
  DeleteFile('unit2.pas');
  RenameFile('unit2a.pas', 'unit2.pas');
  Memo1.Lines.LoadFromFile('unit2.pas');
  Memo1.Lines.Append('=============');
  Memo1.Lines.Append('počet výmen = ' + IntToStr(pocet));
end;
```

Použili sme tu dve systémové procedúry DeleteFile a RenameFile, aby sme mohli prepísať pôvodný obsah súboru unit2.pas novým opraveným obsahom.

Príklad

Program bude čítať textový súbor unit3.pas, v ktorom sa nachádza nejaký pascalovský program a vytvorí súbor unit3a.pas, ktorý bude maximálne zhustený na nejakú zadanú šírku riadka (napr. 80 znakov). V každom riadku bude maximálny počet znakov, ktoré sa sem zmestia a v súbore bude minimálny počet medzier tak, aby program ešte ostal funkčný. Komentáre z programu vynecháme.

Aby bol program rovnako funkčný ako originál, musíme komentáre, ktoré začínajú znakom $, tiež preniesť do

výstupu - bude to pre nás špeciálny symbol, napr. {$R *.dfm}. Niektoré pascalovské viacznakové lexémy by mohli robiť problémy, keby sa rozdelili do dvoch riadkov, napr. znak priradenia :=, relačné operátory <=, dve bodky pre interval .. a pod. - musíme z nich vyrobiť jedinú lexému typu symbol. Všimnite si aj spracovanie apostrofu v znakových reťazcoch. Najprv vylepšený skener:

```
var
  lexema: (koniec, ident, cislo, retazec, symbol);
  hodnota: String;

procedure skener;
var
  ok: Boolean;
begin
  repeat
    ok := True;
    case z of
      #0:
        lexema := koniec;
      ' ', #1, #9:    // #9 znamená znak tabulátor
        begin
          ok := False;
          znak;
        end;
      '{':
        begin
          hodnota := z;
          znak;
          lexema := symbol;
          ok := z='$';              // napr. pre {$R *.dfm}
          while not (z in ['}', #0]) do
          begin
            hodnota := hodnota + z;
            znak;
          end;
          hodnota := hodnota + z;
          znak;
        end;
      '''':
        begin
          hodnota := '';
          lexema := retazec;
          repeat
            znak;
            while not (z in ['''', #0, #1]) do
            begin
              hodnota := hodnota + z;
              znak;
            end;
            if z = '''' then
              znak;
            if z = '''' then
              hodnota := hodnota + '''';
          until z <> '''';
        end;
      'A'..'Z', 'a'..'z', '_':
        begin
          hodnota := '';
          lexema := ident;
          while z in ['A'..'Z', 'a'..'z', '0'..'9', '_'] do
          begin
            hodnota := hodnota + z;
            znak;
          end;
        end;
      '#', '0'..'9':
```

```
      begin
        hodnota := z;
        znak;
        lexema := cislo;
        while z in ['0'..'9'] do
        begin
          hodnota := hodnota + z;
          znak;
        end;
      end;
    '/':
      begin
        znak;
        if z = '/' then      // komentár
        begin
          repeat
            znak;
          until z in [#1, #0];
          ok := False;
        end
        else
        begin
          hodnota := '/';
          lexema := symbol;
        end;
      end;
    ':':
      begin
        hodnota := z;
        znak;
        lexema := symbol;
        if z = '=' then
        begin
          hodnota := hodnota + z;
          znak;
        end;
      end;
    '.':
      begin
        hodnota := z;
        znak;
        lexema := symbol;
        if z = '.' then
        begin
          hodnota := hodnota + z;
          znak;
        end;
      end;
    '<', '>':
      begin
        hodnota := z;
        znak;
        lexema := symbol;
        if (z = '=') or (hodnota = '<') and (z = '>') then
        begin
          hodnota := hodnota + z;
          znak;
        end;
      end;
    else
      lexema := symbol;
      hodnota := z;
      znak;
  end;
until ok;
```

```
end;
```

Samotný algoritmus zhusťovania textu - medzi niektoré dvojice za sebou idúcich lexém musíme pridávať medzeru:

```
procedure TForm1.Button1Click(Sender: TObject);
const
  maxsir = 60;
var
  t1: TextFile;
  i, sir: Integer;   // sir - doterajšia šírka riadka
  b0, b1: Boolean;   // či vkladať medzery medzi lexémy
  s: String;
begin
  AssignFile(t, 'unit3.pas');
  Reset(t);
  AssignFile(t1, 'unit3a.pas');
  Rewrite(t1);
  sir := 0;
  b0 := False;
  znak;
  skener;
  while lexema <> koniec do
  begin
    b1 := lexema in [cislo, ident];
    s := hodnota;
    if lexema = retazec then
    begin
      s := '';
      for i := 1 to Length(hodnota) do
        if hodnota[i] = '''' then
          s := s + ''''''
        else
          s := s + hodnota[i];
      s := '''' + s + '''';
    end;
    if b0 and b1 then
      s := ' '+s;
    b0 := b1;
    if (sir > 0) and (sir+Length(s) > maxsir) then
    begin
      Writeln(t1);
      sir := 0;
    end;
    if (sir = 0) and (s[1] = ' ') then
      delete(s, 1, 1);
    Write(t1, s);
    Inc(sir, Length(s));
    skener;
  end;
  CloseFile(t);
  CloseFile(t1);
  Memo1.Lines.LoadFromFile('unit3a.pas');
end;
```

Zahustený ale ešte stále funkčný program vyzerá asi takto:

```
unit Unit3;interface uses Windows,Messages,SysUtils,Variants
,Classes,Graphics,Controls,Forms,Dialogs,StdCtrls;type
TForm1=class(TForm)Memo1:TMemo;Button1:TButton;procedure
Button1Click(Sender:TObject);private public end;var Form1:
TForm1;implementation{$R *.dfm}var z:Char;t:TextFile;
procedure znak;begin if Eof(t)then z:=#0 else if Eoln(t)then
begin z:=#1;Readln(t);end else Read(t,z);end;var lexema:(
koniec,ident,cislo,retazec,symbol);hodnota:String;procedure
skener;var ok:Boolean;begin hodnota:='aaa''bbb';repeat ok:=
```

```
True;case z of #0:lexema:=koniec;' ',#1,#9:begin ok:=False;
znak;end;'{':begin hodnota:=z;znak;lexema:=symbol;ok:=z='$';
while not(z in['}',#0])do begin hodnota:=hodnota+z;znak;end;
hodnota:=hodnota+z;znak;end;'''':begin hodnota:='';lexema:=
retazec;repeat znak;while not(z in['''',#0,#1])do begin
hodnota:=hodnota+z;znak;end;if z=''''then znak;if z=''''then
hodnota:=hodnota+'''';until z<>'''';end;'A'..'Z','a'..'z',
'_':begin hodnota:='';lexema:=ident;while z in['A'..'Z','a'
..'z','0'..'9','_']do begin hodnota:=hodnota+z;znak;end;end;
'#','0'..'9':begin hodnota:=z;znak;lexema:=cislo;while z in[
'0'..'9']do begin hodnota:=hodnota+z;znak;end;end;'/':begin
znak;if z='/'then begin repeat znak;until z in[#1,#0];ok:=
False;end else begin hodnota:='/';lexema:=symbol;end;end;':'
:begin hodnota:=z;znak;lexema:=symbol;if z='='then begin
hodnota:=hodnota+z;znak;end;end;'.':begin hodnota:=z;znak;
lexema:=symbol;if z='.'then begin hodnota:=hodnota+z;znak;
end;end;'<','>':begin hodnota:=z;znak;lexema:=symbol;if(z=
'=')or(hodnota='<')and(z='>')then begin hodnota:=hodnota+z;
znak;end;end;else lexema:=symbol;hodnota:=z;znak;end;until
ok;end;procedure TForm1.Button1Click(Sender:TObject);const
maxsir=60;var t1:TextFile;i,sir:Integer;b0,b1:Boolean;s:
String;begin AssignFile(t,'unit3.pas');Reset(t);AssignFile(
t1,'unit3a.pas');Rewrite(t1);sir:=0;b0:=False;znak;skener;
while lexema<>koniec do begin b1:=lexema in[cislo,ident];s:=
hodnota;if lexema=retazec then begin s:='';for i:=1 to
Length(hodnota)do if hodnota[i]=''''then s:=s+''''''else s:=
s+hodnota[i];s:=''''+s+'''';end;if b0 and b1 then s:=' '+s;
b0:=b1;if(sir>0)and(sir+Length(s)>maxsir)then begin Writeln(
t1);sir:=0;end;if(sir=0)and(s[1]=' ')then delete(s,1,1);
Write(t1,s);Inc(sir,Length(s));skener;end;CloseFile(t);
CloseFile(t1);Memo1.Lines.LoadFromFile('unit3a.pas');end;end
.
```

Ďalšie námety:

- dorobte do skenera aj **(*...*)** komentáre
- dorobte skener aj pre reálne čísla, šestnástkové konštanty '$' a znakové konštanty s '#'
- zameňte všetky identifikátory premenných (nie rezervovaných slov pascalu) identifikátormi, ktoré obsahujú len znaky O, 0 a Q (písmeno O, nula a Q), napr.

  ```
  OOOO, OOOO, OOOO, OOOO, OOOO, OOQO, QOQO, ...
  ```

- a pod. tak, aby sa program stal maximálne nečitateľný, ale aby ostal funkčný

Bitmapy

V tejto časti môžete použiť súbory z predchádzajúcich prednášok **obrazky.zip** ale aj nové súbory s obrázkami **obrazky2.zip**. Zatiaľ sme poznali jeden spôsob, ako vložiť obrázok (súbor .BMP) do grafickej plochy a to pomocou LoadFromFile, napr.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('tiger.bmp');
end;
```

Pomocou tohto spôsobu sa niekedy zmení rozmer grafickej plochy a okrem toho môže byť nepoužiteľný, keď potrebujeme nejaký obrázok umiestniť nie do ľavého horného rohu, ale na nejaké konkrétne súradnice grafickej plochy. Naučíme sa pracovať s novým objektom - **bitmapa** - môžeme si všimnúť istú podobnosť s objektom robot:

- deklarovaním premennej bmp: TBitmap objekt (inštancia) ešte nevzniká - nový objekt treba ešte vytvoriť

- bmp := TBitmap.Create - vytvorí sa nová bitmapa (obrázok), ktorý je zatiaľ prázdny - má rozmery 0x0
- do existujúceho objektu bitmapa môžeme prečítať obrázok zo súboru pomocou LoadFromFile (ak už bitmapa mala nejaký obsah, tento sa stráca a nahradí sa prečítaným obrázkom - zmení sa jej veľkosť)
- teraz môžeme "opečiatkovať" náš nový objekt bitmapa do grafickej plochy Image1 pomocou metódy Draw
- na záver si treba zvyknúť na jedno dôležité pravidlo: vždy, keď skončíme pracovať s objektom bitmapa, tak ju musíme uvoľniť zo systému pomocou metódy Free - bitmapa je totiž taký špeciálny objekt, ktorý odčerpáva zdroje z Windows a ak takýchto zdrojov Windows minieme priveľa, systém sa môže zrútiť...

Nakreslíme bitmapu do pozadia grafickej plochy:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tiger.bmp');
  Image1.Canvas.Draw(30, 70, bmp);
  bmp.Free;
end;
```

Metóda Draw opečiatkuje bitmapu do Canvasu na súradnice zadané dvoma prvými parametrami - tieto určujú ľavý horný roh kladeného obrázka. V našom príklade sa obrázok tigra umiestni tak, že jeho ľavý horný roh je na súradniciach (30, 70). Niekedy sa môžu hodiť hoci aj záporné súradnice - vtedy nejaká časť obrázka z plochy vypadne.

Existuje ešte špeciálny prípad metódy Draw - položenie obrázka do plochy so zmenou veľkosti StretchDraw - obrázok môžeme ľubovoľne zmenšiť alebo zväčšiť, môžeme ho napríklad natiahnuť na celú grafickú plochu. StretchDraw má dva parametre: prvým je obdĺžnik (Rectangle), do ktorého treba umiestniť obrázok a druhým je samotná bitmapa. Na definovanie obdĺžnika často použijeme konštrukciu Rect(x1, y1, x2, y2), v ktorej zadáme súradnice ľavého horného a pravého dolného rohu oblasti, napr.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tiger.bmp');
  Image1.Canvas.StretchDraw(Rect(70, 30, 200, 300), bmp);
  bmp.Free;
end;
```

alebo natiahnutie obrázka na celú plochu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tiger.bmp');
  Image1.Canvas.StretchDraw(Image1.ClientRect, bmp);
  bmp.Free;
end;
```

Použili sme tu Image1.ClientRect, čo je to isté ako zápis Rect(0, 0, Image1.Width, Image1.Height).

Ukážeme typickú prácu s bitmapou - opečiatkujeme obrázok viackrát vedľa seba a pod seba tak, aby presne vyplnil celé pozadie grafickej plochy. Tomuto sa niekedy hovorí vykachličkovanie plochy nejakým obrázkom:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  bmp: TBitmap;
  x, y: Integer;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('pozadie.bmp');
  x := 0;
  while x < Image1.Width do
  begin
    y := 0;
    while y < Image1.Height do
    begin
      Image1.Canvas.Draw(x, y, bmp);
      Inc(y, bmp.Height);
    end;
    Inc(x, bmp.Width);
  end;
  bmp.Free;
end;
```

Ďalšie námety

- vycentrujete bitmapu do grafickej plochy
- vykachličkujte plochu dvomi rôznymi bitmapami rovnakých rozmerov šachovnicovým spôsobom - na striedačku
- vykachličkujte plochu ale dvojnásobne zväčšeným (zmenšeným) obrázkom
- natiahnite obrázok tak, aby pokryl celú plochu, ale pritom, aby ostal pomer strán obrázka zachovaný - pravdepodobne obrázok v jednom rozmere vypadne z plochy

Teraz budeme pracovať s 8 pripravenými súbormi s obrázkami - bmp1.bmp, bmp2.bmp, ... (sú v súbore **obrazky2.zip**). Po zatlačení tlačidla sa na náhodnom mieste plochy položí náhodne vybraný jeden z týchto obrázkov:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('bmp'+IntToStr(Random(8)+1)+'.bmp');
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Náhodnú pozíciu obrázka sme zvolili tak, aby obrázok nevypadol z plochy - použili sme bmp.Width a bmp.Height - rovnako ako pre Image1 to označuje šírku a výšku bitmapy. Ďalej si môžete všimnúť, že niektoré časti obrázkov sú biele, ale bolo by krajšie, keby boli priesvitné. Bitmapám môžeme jednu farbu určiť ako priesvitnú a potom kladenie obrázka do plochy pomocou Draw alebo StretchDraw ponechá priesvitné časti nezmenené. Bitmape musíme nastaviť stavovú premennú Transparent na True:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('bmp'+IntToStr(Random(8)+1)+'.bmp');
  bmp.Transparent := True;
```

```
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Týmto sa Delphi rozhodli, že v bitmape budú všetky farebné body (pixel), ktoré sú rovnaké ako bod v ľavom hornom rohu, priesvitné. Ak potrebujeme vyhlásiť za priesvitnú inú farbu, použijeme stavovú premennú TransparentColor:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('bmp'+IntToStr(Random(8)+1)+'.bmp');
  bmp.TransparentColor := clFuchsia;
  bmp.Transparent := True;
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Pracovať môžeme nielen s bitmapami, ktoré sme prečítali zo súboru, ale bitmapu si môžeme nakresliť aj sami. Bitmapa má rovnaký Canvas ako grafická plocha Image1 a teda do nej môžeme kresliť úplne rovnakým spôsobom. Ak nebudeme do práve vytvorenej bitmapy (TBitmap.Create) čítať súbor pomocou LoadFromFile, ale chystáme sa kresliť do jej Canvasu, musíme jej najprv určiť veľkosť bmp.Width a bmp.Height a farebný formát bmp.PixelFormat:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.Width := 100;
  bmp.Height := 100;
  bmp.PixelFormat := pf24bit;
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Takto sme vytvorili bitmapu - biely štvorec veľkosti 100x100. Nakreslime do tejto novej bitmapy červený kruh a okraje jej spravme priesvitné:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.Width := 100;
  bmp.Height := 100;
  bmp.PixelFormat := pf24bit;
  bmp.Canvas.Brush.Color := clRed;
  bmp.Canvas.Ellipse(0, 0, 100, 100);
  bmp.Transparent := True;
```

```
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Do takejto bitmapy môžeme opečiatkovať aj inú bitmapu, napr.

```
procedure TForm1.Button4Click(Sender: TObject);
var
  bmp, bmp1: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.Width := 100;
  bmp.Height := 100;
  bmp.PixelFormat := pf24bit;
  bmp.Canvas.Brush.Color := clYellow;
  bmp.Canvas.Ellipse(0, 0, 100, 100);
  bmp.Transparent := True;
    bmp1 := TBitmap.Create;
    bmp1.LoadFromFile('bmp'+IntToStr(Random(8)+1)+'.bmp');
    bmp1.Transparent := True;
    bmp.Canvas.Draw(20, 20, bmp1);
    bmp1.Free;
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Niekedy potrebujeme z väčšej bitmapy vystrihnúť menšiu obdĺžnikovú časť a ďalej pracovať len s týmto kúskom. Môžeme to urobiť napr. takto

```
procedure TForm1.Button5Click(Sender: TObject);
var
  bmp, bmp1: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.Width := 130;
  bmp.Height := 152;
  bmp.PixelFormat := pf24bit;
    bmp1 := TBitmap.Create;
    bmp1.LoadFromFile('tiger.bmp');
    bmp.Canvas.Draw(-64, -51, bmp1);
    bmp1.Free;
  Image1.Canvas.Draw(
      Random(Image1.Width-bmp.Width),
      Random(Image1.Height-bmp.Height),
      bmp);
  bmp.Free;
end;
```

Je to rovnaký princíp pečiatkovania jednej bitmapy do druhej ako v predchádzajúcom príklade len obrázok, ktorý pečiatkujem je výrazne väčší ako bitmapa, do ktorej pečiatkujem. Všimnite si, že do príkazu Draw sme zadali záporné súradnice ľavého horného rohu tigra. Niekedy sa namiesto Draw môže hodiť aj príkaz CopyRect - pozrite si ho v helpe.

Častou začiatočníckou chybou býva vzájomné priraďovanie bitmáp, napr.

```
procedure TForm1.Button6Click(Sender: TObject);
```

```
var
  bmp, bmp1: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tiger.bmp');

    bmp1 := TBitmap.Create;
    bmp1 := bmp;                    // toto je chyba !
    Image1.Canvas.Draw(0, 0, bmp1);
    bmp1.Free;

  bmp.Free;
end;
```

Neskôr, keď sa naučíme, ako fungujú objekty a smerníky, pochopíme, akú strašnú chybu sme spravili. Zapamätajme si, že ak chceme do jednej bitmapy priradiť obsah inej, buď to urobíme opečiatkovaním, alebo použijeme špeciálnu metódu Assign:

```
procedure TForm1.Button6Click(Sender: TObject);
var
  bmp, bmp1: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tiger.bmp');

    bmp1 := TBitmap.Create;
    bmp1.Assign(bmp);           // takto je to správne
    Image1.Canvas.Draw(0, 0, bmp1);
    bmp1.Free;

  bmp.Free;
end;
```

Túto ideu môžeme využiť aj na zapamätanie si celej grafickej plochy, napr.

```
procedure TForm1.Button7Click(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.Assign(Image1.Picture);
  Image1.Canvas.Draw(Random(400), Random(400), bmp);
  bmp.Free;
end;
```

Najprv vytvorí prázdnu bitmapu (TBitmap.Create), potom do nej prekopíruje obsah celej grafickej plochy (bmp.Assign(Image1.Picture)) a túto bitmapu ďalej niekam opečiatkuje.

Bitmapa ako globálna premenná

Často sa nám môže hodiť, aby sme nemuseli často používanú bitmapu (ale aj viac bitmáp - možno aj pole bitmáp) tesne pred použitím vytvoriť a prečítať zo súboru a po použití (napr. po Draw) ju pomocou Free uvoľniť. Najjednoduchším spôsobom je vytvorenie globálnych bitmapových premenných pri štarte formulára, t.j. vo FormCreate. Teraz tieto bitmapy môžeme používať. Na záver by sme nemali nezabudnúť uvoľniť tieto bitmapy pri konci aplikácie - v udalosti FormDestroy.

Ukážka jednoduchého programu s bitmapou v globálnej premennej:

```
var
  bmp: TBitmap;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tiger.bmp');
  DoubleBuffered := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  x, y, w, h: Integer;
begin
  w := 20+Random(200);
  h := 20+Random(200);
  x := Random(Image1.Width-w);
  y := Random(Image1.Height-h);
  Image1.Canvas.StretchDraw(Rect(x, y, x+w, y+h), bmp);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  bmp.Free;
end;
```

Zhrnutie triedy TBitmap

TBitmap je preddefinovaná trieda, ktorá slúži na manipuláciu s obrázkami - môžeme si ju predstaviť ako obsah súboru s príponou .BMP (dá sa s nimi pracovať napr. v programe Paint/Skicár). Táto trieda má niekoľko užitočných stavových premenných (vlastnosti - property) a metód.

Niektoré stavové premenné:

- Width, Height - momentálna šírka a výška obrázka (môžeme ju zmeniť priradením nových hodnôt do týchto premenných)
- PixelFormat - farebný vnútorný formát bitmapy, t.j. koľko bitov zaberá každý jeden pixel obrázka - my budeme používať konštantu pf24bit
- Canvas - grafická plocha obrázka
- Transparent - či má nejaké priesvitné časti (inak je to nepriesvitný obdĺžnik)
- TransparentColor - ktorá farba v obrázku je považovaná za priesvitnú

Niektoré metódy:

- konštruktor Create - vytvorí zatiaľ prázdny obrázok
- Free - uvoľní bitmapu z pamäti Windows
- LoadFromFile - načíta obrázok zo súboru vo formáte .BMP
- SaveToFile - uloží obrázok do súboru vo formáte .BMP
- Assign - urobí kópiu obrázka z inej bitmapy, resp. grafickej plochy

Môžeme pracovať s Canvasom bitmapy, t.j. s dvojrozmerným poľom farebných Pixelov (bodov) - do bitmapy môžeme kresliť, môžeme pracovať s jednotlivými pixelmi - úplne rovnako ako v obyčajnej grafickej ploche (TImage), napr.

- Canvas.FillRect
- Canvas.Pixels[riadok, stĺpec]
- Canvas.Rectangle, Ellipse, TextOut, ...
- Canvas.MoveTo, LineTo, Polygon, Polyline, FloodFill, ...

a teda môžeme týmto metódam nastaviť pero, štetec, font, ..., napr. Canvas.Pen.Color :=...

- Canvas.Draw - opečiatkuje inú bitmapu
- Canvas.StretchDraw - opečiatkuje inú bitmapu, pričom ju môže zväčšiť/zmenšiť
- Canvas.CopyRect - opečiatkuje výrez iného Canvasu (bitmapy alebo grafickej plochy)
- Canvas.BrushCopy - podobný CopyRect - jednu z farieb pri kopírovaní vie nahradiť inou

11. prednáška: rekurzia

čo už vieme:

- poznáme mechanizmus volania podprogramov:
  - zapamätá sa návratová adresa
  - vyhradí sa pamäť pre lokálne premenné (aj hodnotové parametre - tie sa aj inicializujú)
  - vykoná sa telo podprogramu
  - uvoľní sa pamäť lokálnych premenných - zabudnú sa ich hodnoty
  - program pokračuje na zapamätanej návratovej adrese

čo sa na tejto prednáške naučíme:

- mechanizmus rekurzie (zásobník), rekurzívne krivky (binárne stromy, fraktály)

Nekonečná rekurzia

Na testovanie rekurzívnych procedúr z tejto prednášky môžete využiť pripravený Projekt - do tohto projektu môžete pridávať ďalšie procedúry a sledovať, ako sa správa zásobník. V tomto testovacom projekte si všimnite, ako sa pomocou dvoch pomocných procedúr push a pop simuluje činnosť zásobníka.

Teraz si už pozrite prvú rekurzívnu procedúru. Procedúra xy volá samu seba:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;

  procedure xy(s: Real);
  begin
    r.fd(s);
    r.lt(60);
    wait(1);
    xy(s+1);
  end;

begin
  r := TRobot.Create;
  xy(10);
  r.Free;
end;
```

Trasujeme tento program (použijeme známy mechanizmus volania podprogramov):

- z hlavného programu je volanie procedúry xy => vytvorí sa lokálna premenná s, ktorej hodnota je 10,
- nakreslí sa čiara dĺžky s a robot sa otočí doľava o 60 stupňov,

- znovu sa volá procedúra xy s parametrom s zmeneným na s+1, t.j. vytvorí sa nová lokálna premenná s s hodnotou 11 (táto premenná sa vytvára na zásobníku),
- toto sa robí donekonečna. Časom sa ale zaplní pamäť počítača, ktorá je určená na mechanizmus volania podprogramov (je to niekoľko 100 Kb) - hovoríme, že pretečie zásobník, t.j. správa **Stack overflow**.

Zásobník (stack) je údajová štruktúra, ktorá má tieto vlastnosti:

- nové prvky pridávame na vrch (napr. kopa tanierov)
- keď potrebujeme zo zásobníka nejaký prvok, vždy ho odoberáme z vrchu (posledne položený tanier)

V ukážke si ešte všimnite volanie r.Free na konci programu: na tento príkaz sa výpočet nikdy nedostane.

Chvostová rekurzia (nepravá)

Aby sme nevytvárali nikdy nekončiace programy, t.j. nekonečnú rekurziu, niekde do tela rekurzívnej procedúry musíme vložiť test, ktorý zabezpečí, že v niektorých prípadoch rekurzia predsa len skončí. Najčastejšie to budeme riešiť tzv. **triviálnym prípadom**: na začiatok podprogramu umiestnime podmienený príkaz if, ktorý otestuje triviálny prípad, t.j. prípad, keď už nebudeme procedúru rekurzívne volať, ale vykonáme len nejaké "nerekurzívne" príkazy. Môžeme si to predstaviť aj takto: rekurzívna procedúra rieši nejaký komplexný problém a pri jeho riešení volá samu seba (rekurzívne volanie) väčšinou s nejakými pozmenenými údajmi. V niektorých prípadoch ale rekurziu na riešenie problému nepotrebujeme, ale vieme to vyriešiť "triviálne" aj bez nej (riešenie takejto úlohy je už "triviálne"). V takto riešených úlohách vidíme, že procedúra sa skladá z dvoch častí:

- pri splnení nejakej podmienky, sa vykonajú príkazy bez rekurzívneho volania (triviálny prípad),
- inak sa vykonajú príkazy, ktoré v sebe obsahujú rekurzívne volanie.

Zrejme má toto šancu fungovať, len keď po nejakom čase naozaj nastane podmienka triviálneho prípadu, t.j. keď sa tak menia parametre rekuzívneho volania, že sa k triviálnemu prípadu nejako blížime. V nasledujúcej ukážke môžete vidieť, že rekurzívna špirála sa kreslí tak, že sa najprv nakreslí úsečka dĺžky s, robot sa otočí o 60 stupňov vľavo a dokreslí sa špirála väčšej veľkosti. Toto celé skončí, keď už budeme chcieť nakresliť špirálu väčšiu ako 100 - takáto špirála sa nenakreslí. Triviálnym prípadom je tu **nič**, t.j. žiadna akcia pre príliš veľké špirály.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRobot;

  procedure spir(s: Real);
  begin
    if s > 100 then
      // nič nerob len skonči
    else
    begin
      r.fd(s);
      r.lt(60);
      wait(1);
      spir(s+3);
    end;
  end;

begin
  r := TRobot.Create;
  spir(10);
  r.Free;
end;
```

Trasujme volanie so simuláciou na zásobníku s počiatočnou hodnotou parametra s, napr. 92:

- na zásobníku vznikne nová lokálna premenná s s hodnotou 92 ... robot nakreslí čiaru, otočí sa a volá xy s

parametrom 95,
- na zásobníku vznikne nová lokálna premenná s s hodnotou 95 ... robot nakreslí čiaru, otočí sa a volá xy s parametrom 98,
- na zásobníku vznikne nová lokálna premenná s s hodnotou 98 ... robot nakreslí čiaru, otočí sa a volá xy s parametrom 101,
- na zásobníku vznikne nová lokálna premenná s s hodnotou 101 ... robot už nič nekreslí ani sa nič nevolá ... procedúra xy končí, t.j.
- zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná s s hodnotou 101 a riadenie sa vráti za posledné volanie procedúry xy - tá ale končí, t.j.
- zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná s s hodnotou 98 a riadenie sa vráti za posledné volanie procedúry xy - tá ale končí, t.j.
- zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná s s hodnotou 95 a riadenie sa vráti za posledné volanie procedúry xy - tá ale končí, t.j.
- zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná s s hodnotou 92 a riadenie sa vráti za posledné volanie procedúry xy - teda do hlavného programu.

Nakoľko rekurzívne volanie procedúry je iba na jednom mieste, za ktorým už nenasledujú ďalšie príkazy procedúry, toto rekurzívne volanie sa dá ľahko prepísať cyklom while. Rekurzia, v ktorej za rekurzívnym volaním nie sú ďalšie príkazy, hovoríme **chvostová rekurzia**. Predchádzajúcu ukážku môžeme prepísať takto:

```
  while s <= 100 do
  begin
    r.fd(s);
    r.lt(60);
    wait(1);
    s := s+3;
  end;
```

Rekurziu môžeme používať nielen pri kreslení pomocou robota, ale napr. aj pri výpise do textovej plochy. V nasledujúcom príklade vypisujeme pod seba čísla N, N-1, N-2, ..., 2, 1:

```
procedure TForm1.Button1Click(Sender: TObject);

  procedure vypis(n: Integer);
  begin
    if n < 1 then
      // nič nerob len skonči
    else
    begin
      Memo1.Lines.Append(IntToStr(n));
      vypis(n-1);
    end;
  end;

begin
  vypis(30);
end;
```

Asi je veľmi jednoduché prepísať to bez použitia rekurzie pomocou while-cyklu. Poexperimentujme, a vymeňme vypisovanie do textovej plochy (Memo1.Lines.Append) s rekurzívnym volaním (vypis). Po spustení vidíte, že aj táto nová rekurzívna procedúra sa dá prepísať len pomocou while-cyklu (resp. for-cyklu) ale jej činnosť už nemusí byť pre každého na prvý pohľad až tak jasná - odtrasujte túto zmenenú verziu:

```
procedure vypis(n: Integer);
begin
  if n < 1 then
    // skonči
```

```
  else
  begin
    vypis(n-1);
    Memo1.Lines.Append(IntToStr(n));
  end;        // už to nie je chvostová rekurzia
end;
```

Pravá rekurzia

Rekuzie, ktoré už nie sú obyčajne chvostové, sú na pochopenie trochu zložitejšie. Pozrime takéto kreslenie špirály:

```
procedure spir(s: Real);
begin
  if s > 100 then
    r.PC := clRed     // a skonči
  else
  begin
    r.fd(s);
    r.lt(60);
    wait(1);
    spir(s+3);
    r.rt(60);
    r.fd(-s);
    wait(1);
  end;
end;
```

Nejaké príkazy sú pred aj za rekurzívnym volaním. Aby sme to lepšie rozlíšili, triviálny prípad nastaví inú farbu pera. Teraz trasujte jej volanie, napr. pre spir(189)...

Aj takéto rekurzívne volanie sa dá prepísať pomocou dvoch cyklov:

```
  pocet := 0;
  while s <= 100 do           // čo sa deje pred rekurzívnym volaním
  begin
    r.fd(s);
    r.lt(60);
    s := s+3;
    Inc(pocet);
    wait(1);
  end;
  r.PC := clRed;              // triviálny prípad
  while pocet > 0 do          // čo sa deje po vynáraní z rekurzie
  begin
    s := s-3;
    r.rt(60);
    r.fd(-s);
    Dec(pocet);
    wait(1);
  end;
```

Aj v ďalších príkladoch môžete vidieť pravú rekurziu. Napr. vylepšená procedúra vypis vypisuje postupnosť čísel:

```
procedure vypis(n: Integer);
begin
  if n < 1 then
    // skonči
  else
  begin
    Memo1.Lines.Append(IntToStr(n));
    vypis(n-1);
    Memo1.Lines.Append(IntToStr(n));
  end;
```

```
end;
```

Keď ako triviálny prípad pridáme riadok s hviezdičkami, tento sa vypíše niekde medzi postupnosť čísel. Viete, kde sa vypíšu tieto hviezdičky?

```
procedure vypis(n: Integer);
begin
  if n < 1 then
    Memo1.Lines.Append('***')
  else
  begin
    Memo1.Lines.Append(IntToStr(n));
    vypis(n-1);
    Memo1.Lines.Append(IntToStr(n));
  end;
end;
```

V ďalších príkladoch s robotom využívame známu procedúru poly. Zistite, čo kreslia procedúry stvorec a stvorec1.

```
procedure poly(n: Integer; s, u: Real);
begin
  while n > 0 do
  begin
    r.fd(s);
    r.lt(u);
    Dec(n);
  end;
  wait(1);
end;

procedure stvorec(a: Integer);
begin
  if a > 100 then
     // nič nerob len skonči
  else
  begin
    poly(4, a, 90);
    stvorec(a+5);
  end;
end;

procedure stvorec1(a: Integer);
begin
  if a > 100 then
    r.lt(180)    // a skonči
  else
  begin
    poly(4, a, 90);
    stvorec1(a+5);
    poly(4, a, 90);
  end;
end;
```

Všetky tieto príklady s pravou rekurziou by ste mali vedieť jednoducho prepísať do niekoľkých cyklov.

V nasledujúcom príklade počítame **faktoriál** prirodzeného čísla N, pričom vieme, že

- 0! = 1 ... triviálny prípad
- n! = (n-1)!*n ... rekurzívne volanie

```
function fakt(n: Integer): Integer;
begin
  if n = 0 then
```

```
    Result := 1
  else
    Result := fakt(n-1)*n;
end;
```

Triviálnym prípadom je tu úloha, ako vyriešiť 0! Toto vieme aj bez rekurzie, lebo je to 1. Ostatné prípady sú už rekurzívne: na to, aby sme vyriešili zložitejší problém (n faktoriál), najprv vypočítame jednoduchší (n-1 faktoriál) - zrejme pomocou rekurzie - a z neho skombinujeme (násobením) požadovaný výsledok. Hoci toto riešenie nie je chvostová rekurzia (po rekurzívnom volaní fakt sa musí ešte násobiť), vieme ho jednoducho prepísať pomocou cyklu.

Ďalšie námety:

- zapíšte poly pomocou rekurzie - bez zápisu cyklu
- zapíšte fibonacciho postupnosť rekurzívnou funkciou - zistite, koľkokrát sa rekurzívne zavolá pre dané n
- riešte rekurziou iné známe matematické funkcie, napr. najväčší spoločný deliteľ pomocou euklidovho algoritmu, umocňovanie čísla na celočíselný exponent pomocou násobenia, výpočet kombinačného čísla a pod.

Binárne stromy

Medzi informatikmi sú veľmi populárne binárne stromy. Najlepšie sa kreslia pomocou robota. Hovoríme, že binárne stromy majú nejakú svoju úroveň n a sú definované takto:

- ak je úroveň stromu n = 0, nakreslí sa len čiara nejakej dĺžky;
- pre n >= 1, sa najprv nakreslí čiara, potom sa na jej konci nakreslí najprv vľavo binárny strom (n-1). úrovne a potom vpravo opäť binárny strom (n-1). úrovne - hovoríme im podstromy;
- po skončení kreslenia stromu ľubovoľnej úrovne sa robot nachádza na mieste, kde začal kresliť;
- ľavé aj pravé podstromy môžu mať buď rovnako veľké konáre ako kmeň stromu, alebo sa môžu v nižších úrovniach zmenšovať.

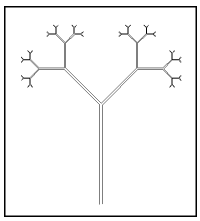
Najprv ukážeme binárny strom, ktorý má vo všetkých úrovniach rovnako veľké podstromy:

```
procedure strom(n: Integer);
begin
  if n = 0 then
  begin              // triviálny prípad
    r.fd(30);
    r.fd(-30);
  end
  else
    with r do
    begin
      fd(30);
      lt(45);        // natoč sa na kreslenie ľavého podstromu
      strom(n-1);    // nakresli ľavý podstrom (n-1). úrovne
      rt(90);        // natoč sa na kreslenie pravého podstromu
      strom(n-1);    // nakresli pravý podstrom (n-1). úrovne
      lt(45);        // natoč sa do pôvodného smeru
      fd(-30);       // vráť sa na pôvodné miesto
    end;
  wait(1);
end;
```

Binárne stromy môžeme rôzne vylepšovať, napr. vetvy stromu sa vo vyšších úrovniach môžu skracovať na polovicu, alebo počas triviálneho prípadu, môžeme okrem úsečky (čiara tam a späť) urobiť nejakú akciu, napr. urobiť "úkrok" vpravo o 1 bodku, alebo aj niečo nakresliť a pod.:

```
procedure strom1(n: Integer; v: Real);
begin
  with r do
    if n = 0 then    // s hrubšími vetvami
    begin
      fd(v);
      rt(90);
      fd(1);
      lt(90);
      fd(-v);
    end
    else
    begin
      fd(v);
      lt(45);
      strom1(n-1, v/2);
      rt(90);
      strom1(n-1, v/2);
      lt(45);
      fd(-v);
    end;
  wait(1);
end;
```

Nakoľko sa robot nevracia presne po tej istej čiare, ale s malou odchýlkou a táto odchýlka sa stále zväčšuje, vzniká tento zaujímavý efekt:



Ďalšie námety na binárne stromy:

- porozmýšľajte, ako nakresliť binárny strom bez rekurzie (budeme to spoločne riešiť neskôr);
- dokresliť stromu lístočky na posledných vetvách;
- náhodne nastaviť veľkosť podstromu, náhodne nastaviť uhol vetiev;
- podľa úrovne vnorenia meniť hrúbku pera.

Nakreslime strom, ktorý sa rozvetvuje nie na dve ale na tri časti - nazýva sa ternárny strom:

```
procedure strom3(n: Integer; a: Real);
var
  i: Integer;
begin
  if n = 1 then
    poly(2, a, 180)
  else
    with r do
    begin
      fd(a);
      lt(45);
      for i := 1 to 3 do
      begin
        strom3(n-1, a/2);
        rt(45);
      end;
      lt(90);
      fd(-a);
    end;
  wait(1);
```

```
end;
```

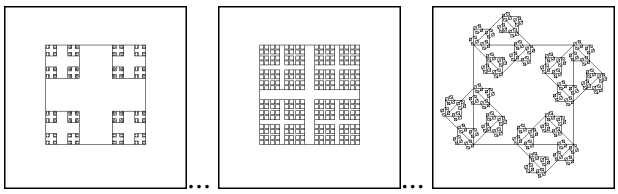
Napíšeme procedúru, ktorá nakreslí obrázok stvorce úrovne N, veľkosti a

- pre n = 0 nerobí nič
- pre n = 1 kreslí štvorec so stranou dĺžky a
- pre n > 1 kreslí štvorec, v ktorom v každom jeho rohu (smerom dnu) je obrázok stvorce úrovne N-1 veľkosti a/3

štvorce v každom rohu štvorca:

```
procedure stvorce(n: Integer; a: Real);
var
  i: Integer;
begin
  if n = 0 then
     // nerob nič len skonči
  else
    for i := 1 to 4 do
    begin
      r.fd(a);
      r.lt(90);
      wait(1);
      stvorce(n-1, a/3);     // skúste: stvorce(n-1, a*0.45);
    end;
end;
```

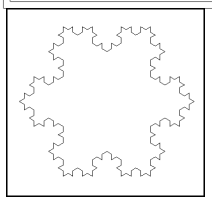
Uvedomte si, že to nie je chvostová rekurzia. Obrázok vyzerá takto (ďalšie sú malé modifikácie programu):



Ďalej ukážeme veľmi známu rekurzívnu krivku - snehovú vločku:

```
var
  r: TRobot;

  procedure vlocka(n: Integer; s: Real);
  begin
    if n = 0 then
      r.fd(s)
    else
    begin
      vlocka(n-1, s/3);
      r.rt(60);
      vlocka(n-1, s/3);
      r.lt(120);
      vlocka(n-1, s/3);
      r.rt(60);
      vlocka(n-1, s/3);
    end;
    wait(1);
  end;

var
  i: Integer;
begin
  r := TRobot.Create(300, 350);
  for i := 1 to 3 do
  begin
```

```
    vlocka(3, 300);
    r.lt(120);
  end;
  r.Free;
end;
```



Ďalšie námety pre snehovú vločku:

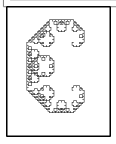
- dorobte do cyklu zmenu farby pera pred každým volaním vlocka(...) - budete lepšie vidieť, čo kreslí rekurzívna procedúra;
- zistite, ako sa zmení obrázok snehovej vločky, keď v cykle nahradíme r.lt(120) príkazom r.rt(120);

Fraktály

Špeciálnou skupinou rekurzívnych kriviek sú fraktály. Ešte pred érou počítačov sa s nimi "hrali" aj významní matematici (niektoré krivky podľa nich dostali aj svoje meno). Zjednodušene by sme mohli povedať, že fraktál je je taká krovka, ktorá sa skladá zo svojich zmenšených kópií. Keby sme sa na nejakú jej časť pozreli lupou, videli by sme opäť skoro tú istú krivku. Napr. aj binárne stromy a aj snehovú vločku by sme mohli považovať za fraktály.

Začneme veľmi jednoduchou, tzv. **C-krivkou**:

```
var
  r: TRobot;

  procedure ckrivka(n: Integer; s: Real);
  begin
    if n = 0 then
      r.fd(s)
    else
    begin
      ckrivka(n-1, s);
      r.lt(90);
      ckrivka(n-1, s);
      r.rt(90);
    end;
    wait(1);
  end;

begin
  r := TRobot.Create(200, 100);
  ckrivka(10, 2);         // skúste: ckrivka(13, 2);
  r.Free;
end;
```



C-krivke sa veľmi podobá **Dračia krivka**, ktorá sa skladá z dvoch "zrkadlových" procedúr: ldrak a pdrak. Problémom u týchto dvoch rekurzívnych procedúr je to, že prvá rekurzívne volá samu seba ale aj druhú a druhá volá seba aj prvú. Tým problémom je poradie, v akom pri ich zadefinujeme: ak bude najprv prvá a až za ňou druhá, prvá nemôže volať druhú. Ak to bude naopak, tak máme ten istý problém. Jednou z možností je použitie

direktívy forward:

```
var
  r: TRobot;

  procedure pdrak(n: Integer; s: Real); forward;

  procedure ldrak(n: Integer; s: Real);
  begin
    if n = 0 then
      r.fd(s)
    else
    begin
      ldrak(n-1, s);
      r.lt(90);
      pdrak(n-1, s);
    end;
    wait(1);
  end;

  procedure pdrak(n: Integer; s: Real);
  begin
    if n = 0 then
      r.fd(s)
    else
    begin
      ldrak(n-1, s);
      r.rt(90);
      pdrak(n-1, s);
    end;
    wait(1);
  end;

begin
  r := TRobot.Create(300, 200);
  ldrak(11, 5);
  r.Free;
end;
```

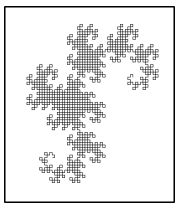
Táto direktíva označuje, že uvedená hlavička procedúry bude definovaná až neskôr, ale kompilátor o tejto procedúre už vie a teda ju môžeme volať (ešte pred jej definíciou).

Iným riešením je vzájomné vnorenie procedúr:

```
var
  r: TRobot;

  procedure ldrak(n: Integer; s: Real);

    procedure pdrak(n: Integer; s: Real);
    begin
      if n = 0 then
        r.fd(s)
      else
      begin
        ldrak(n-1, s);
        r.rt(90);
        pdrak(n-1, s);
      end;
      wait(1);
    end;

  begin
    if n = 0 then
      r.fd(s)
    else
```

```
    begin
      ldrak(n-1, s);
      r.lt(90);
      pdrak(n-1, s);
    end;
    wait(1);
  end;

begin
  r := TRobot.Create(300, 200);
  ldrak(11, 5);
  r.Free;
end;
```

Obe riešenia sú rovnako správne. Všimnite si, že robot pri kreslení krivky, neprejde po žiadnej čiare viackrát:



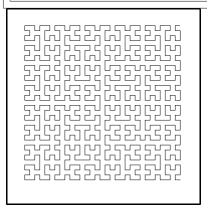
Dračiu krivku môžeme nakresliť aj len jednou procedúrou - táto bude mať o jeden parameter viac a to, či je to ľavá alebo pravá verzia procedúry:

```
var
  r: TRobot;

  procedure drak(n: Integer; s: Real; u: Integer);
  begin
    if n = 0 then
      r.fd(s)
    else
    begin
      drak(n-1, s, 90);
      r.lt(u);
      drak(n-1, s, -90);
    end;
    wait(1);
  end;

begin
  r := TRobot.Create(200, 100);
  drak(13, 3, 90);
  r.Free;
end;
```

V literatúre je veľmi známou **Hilbertova krivka**, ktorá sa tiež skladá z dvoch zrkadlových častí (ako dračia krivka) a preto ich definujeme jednou procedúrou a parametrom uhol (ľavá alebo pravá verzia):

```
var
  r: TRobot;

  procedure hilbert(n: Integer; s: Real; u: Integer);
  begin
    if n > 0 then
    begin
      r.lt(u);
      hilbert(n-1, s, -u);
      r.fd(s);
      r.rt(u);
      hilbert(n-1, s, u);
      r.fd(s);
```

```
      hilbert(n-1, s, u);
      r.rt(u);
      r.fd(s);
      hilbert(n-1, s, -u);
      r.lt(u);
      wait(1);
    end;
  end;

begin
  r := TRobot.Create(400,400);
  hilbert(5, 10, 90);             // skúste: hilbert(7, 2, 90);
  r.Free;
end;
```



12. prednáška: myš, časovač

čo už vieme:

- do grafickej plochy vieme kresliť pomocou grafických príkazov - riadením grafického pera

čo sa na tejto prednáške naučíme:

- kresliť do grafickej plochy pomocou myši
- využívať časovač

Riadenie pomocou myši

Pripomeňme si, že pri práci s Pixels v grafickej ploche sme trochu experimentovali aj s myšou: pri pohyboch myšou nad grafickou plochou Image2 sme zobrazovali príslušné bodky obrázka z Image1. Využili sme pritom udalosť onMouseMove, ktorá do obslužnej procedúry Image2MouseMove posiela aj súradnice myši v parametroch X a Y.

Teraz sa naučíme - už podrobnejšie, ako pracovať s myšou. Grafická plocha má 3 rôzne udalosti:

- onMouseDown - zatlačili sme nad plochou niektoré tlačidlo myši
- onMouseMove - myš sa hýbe nad plochou
- onMouseUp - pustili sme tlačidlo myši

Začneme najjednoduchším prípadom: vo formulári máme grafickú plochu Image1 a priradíme jej udalosť onMouseMove (dvojklikneme na riadok s onMouseMove v záložke Events pre Image1 v Inšpektore objektov):

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
```

```
  Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
end;
```

Teraz tento program pri pohybe myši nad Image1 vykresľuje bodky malé krúžky. Opäť si pripomenieme, že aby plocha pri kreslení neblikala, pridáme udalosť onCreate pre Form1 (dvojklikneme v Inšpektore objektov v záložke Events na riadok onCreate):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := True;
end;
```

Pri pohybe myšou a teda pri kreslení krúžkov môžeme zisťovať, či sme zároveň stlačili aj nejaké tlačidlo myši. Slúži na to parameter Shift, ktorý môže mať napr. tieto hodnoty:

```
Shift = []                    // nezatlačili sme žiadne tlačidlo
Shift = [ssLeft]              // zatlačili sme len ľavé tlačidlo
Shift = [ssRight]             // zatlačili sme len pravé tlačidlo
Shift = [ssLeft, ssRight]     // zatlačili sme naraz obe tlačidlá
```

Totiž Shift je množinového typu TShiftState a je definovaný takto

```
type
  TShiftState = set of (ssShift, ssAlt, ssCtrl,
                        ssLeft, ssRight, ssMiddle, ssDouble);
```

V tomto parametri sa môžeme dozvedieť či sú zatlačené tlačidlá na myši ale aj niektoré špeciálne klávesy na klávesnici (Shift, Alt alebo Ctrl).

Program teraz kreslí krúžky len pri zatlačenom ľavom tlačidle myši:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if Shift = [ssLeft] then
    Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
end;
```

Na jednom mieste môžeme testovať aj naraz viac podmienok, napr.

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if Shift = [ssLeft] then
    Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5)
  else if Shift = [ssRight] then
    Image1.Canvas.FillRect(Image1.ClientRect)
  else
    Image1.Canvas.Pixels[X, Y] := clBlack;
end;
```

Už sme sa stretli aj s udalosťou onMouseDown, pomocou ktorej vieme rozlíšiť moment, keď klikneme do grafickej plochy a potom ťaháme - t.j. počiatočný bod ťahania a teda kreslenia. Napr. nasledujúci úsek programu zmení farbu pera pre každú ťahanú čiaru:

```
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Shift = [ssLeft] then
```

```
    Image1.Canvas.Pen.Color := Random(256*256*256);
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if Shift = [ssLeft] then
    Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
end;
```

Ak chceme kresliť čiary pomocou príkazov MoveTo a LineTo, môžeme to zapísať napr. takto:

```
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Shift = [ssLeft] then
    Image1.Canvas.MoveTo(X, Y);
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if Shift = [ssLeft] then
    Image1.Canvas.LineTo(X, Y);
end;
```

Hoci je toto teraz už dobré riešenie, niekedy to môže robiť problémy (vyskúšajte počas kreslenia zatlačiť kláves Ctrl), preto môžeme použiť pomocnú logickú premennú kreslim, ktorá označuje, či práve kreslíme (či sme začali kresliť) a ďalej sa už nepozerá na parameter Shift:

```
var
  kreslim: Boolean = False;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  kreslim := Shift = [ssLeft];    // alebo kreslim := ssLeft in Shift;
  if kreslim then
    Image1.Canvas.MoveTo(X, Y);
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if kreslim then
    Image1.Canvas.LineTo(X, Y);
end;

procedure TForm1.Image1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  kreslim := False;
end;
```

Popri kreslení čiar, resp. iných útvarov v udalosti onMouseMove môžeme, napr. aj vypisovať alebo zaznamenávať si nejaké informácie o polohe myši, napr.:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if Shift = [ssLeft] then
    ...
```

```
  Label1.Caption := IntToStr(X) + ' ' + IntToStr(Y);
end;
```

Všimnite si, že sme použili komponent Label (textový popis) - umiestnili sme ho niekam do formulára a zmenou jeho stavovej premennej (property) Caption meníme jeho zobrazenie - v našom príklade sú to súradnice pozície myši.

Skicár - kresliaca aplikácia

Postupne predvedieme konštrukciu jednoduchej kresliacej aplikácie. Najprv do formulára vložíme grafickú plochu (Image1), pod ňu textový informačný riadok (Label1) a nad plochu tlačidlo s textom **zmaž** (Button1). Prvá verzia programu:

```
var
  kreslim: Boolean = False;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := True;
  Button1.Click;                  // zmaže sa grafická plocha
  Label1.Caption := '';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  with Image1, Canvas do   // medzi Image1 a Canvas je čiarka
  begin
    Brush.Color := clWhite;
    Brush.Style := bsSolid;
    FillRect(ClientRect);  // ClientRect patrí Image1 a nie pre Canvas
  end;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  kreslim := ssLeft in Shift;
  if kreslim then
    Image1.Canvas.MoveTo(X, Y);
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y:Integer);
begin
  Label1.Caption := IntToStr(X)+' '+IntToStr(Y);
  if kreslim then
    Image1.Canvas.LineTo(X, Y);
end;

procedure TForm1.Image1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  kreslim := False;
end;
```

Paleta farieb

Druhým krokom bude pridanie farebnej palety do aplikácie. Zrejme v každom grafickom programe by mala byť možnosť zvoliť si farbu kreslenia kliknutím do nejakej farby vo ponuke farieb. Do formulára najprv položíme novú menšiu grafickú plochu Image2 (veľkosti 225x33) - tesne nad Image1 a vložíme do nej obrázok palety (bitmapa

paleta.bmp). Pred Image2 položíme ďalšiu malú plôšku Image3 (veľkosti 25x25) - sem sa bude vykresľovať aktuálna zvolená farba - pri štarte programu bude čierna. Ďalej zabezpečíme, že kliknutím do Image2 sa zvolí farba kresliaceho pera, t.j. nastaví sa farba pera pre Image1 a zafarbí sa Image3:

```
procedure TForm1.Image2MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Image1.Canvas.Pen.Color := Image2.Canvas.Pixels[X, Y];
  with Image3, Canvas do
  begin
    Brush.Color := Image1.Canvas.Pen.Color;
    FillRect(ClientRect);
  end;
end;
```

Ešte treba zafarbiť Image3 už pri štarte programu, t.j. do FormCreate pridáme jeho zafarbenie:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ...
  with Image3, Canvas do
  begin
    Brush.Color := Image1.Canvas.Pen.Color;
    FillRect(ClientRect);
  end;
end;
```

Posúvač - posuvná lišta

Na nastavenie hrúbky pera použijeme posúvač, napr. komponent TrackBar (môžete použiť aj ScrollBar). TrackBar1 položíme naľavo od Image3 a trochu prispôsobíme jeho veľkosť voľnej ploche vo formulári. Zmeníme aj dve jeho stavové premenné (property) Min a Max, ktoré vyjadrujú minimálnu a maximálnu dosiahnuteľnú hodnotu pre posúvač - nastavíme Min na 1 a Max napr. na 20. Pre tento komponent využijeme udalosť onChange, ktorá je zavolaná vždy, keď používateľ posúva bežca na lište: podľa momentálnej pozície bežca budeme nastavovať hrúbku pera grafickej plochy. onChange procedúra sa automaticky pripraví dvojklikom na komponent vo formulári. Spracovanie nastavovania hrúbky pera:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  Image1.Canvas.Pen.Width := TrackBar1.Position;
end;
```

Tlačidlá nástrojov kreslenia

Každý aj najjednoduchší kresliaci program umožňuje kresliť rôznymi nástrojmi, napr. nielen voľné kreslenie perom, ale aj úsečky, obdĺžniky, elipsy ale aj vypĺňanie oblasti nejakou farbou. Zadefinujeme 5 malých tlačidiel - komponentov SpeedButton. Tieto majú na rozdiel od klasického Button viac vylepšení: napr. keď na ne klikneme, môžu ostať zatlačené (potom to vyjadruje zvolený nástroj) a tiež okrem textu môžu obsahovať obrázok (poexperimentujte s vlastnosťou Glyph). Tieto tlačidlá môžeme zoskupiť do skupiny, v ktorej iba jedno z nich bude zatlačené - zatlačenie iného spôsobí automatické vyskočenie predchádzajúceho. Preto do stavovej premennej (property) GroupIndex všetkým nastavíme rovnaké číslo 1. Na tlačidlá napíšeme texty (je vhodné zvoliť nejaký malý font): pero, úsečka, obdĺžnik, elipsa, vyplň. Prvému tlačidlu (s textom pero) nastavíme stavovú premennú Down na True, t.j. toto tlačidlo bude zatlačené už pri štarte programu. Zadefinujeme aj globálnu premennú rezim, ktorá bude obsahovať momentálne zvolený nástroj - táto premenná sa bude nastavovať v udalosti onClick pre všetky SpeedButton1, SpeedButton2, ...

```
var
  rezim: (pero, usecka, obdlznik, elipsa, vypln) = pero;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  rezim := pero;
end;

procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  rezim := usecka;
end;

...
```

Asi by bolo krajšie, keby sme nemuseli vytvárať pre každé tlačidlo zvlášť procedúru (v inej aplikácii by ich mohlo byť oveľa viac) - ale napísali by sme jednu univerzálnu, napr. procedúra SpeedButton1Click by mohla byť spoločná pre všetky. Potom, ako je zadefinovaná pre SpeedButton1, ju nastavíme aj všetkým ostatným tlačidlám nástrojov: v záložke udalosti (Events) inšpektora objektov pre SpeedButton2 sa nastavíme na riadok s udalosťou onClick a z ponuky vyberieme SpeedButton1Click - toto urobíme pre všetky tlačidlá nástrojov (ak sme stihli vytvoriť SpeedButton2Click, ... - zrušíme ich tak, že vyprázdnime telo procedúr a po uložení - Save unit - sa korektne zrušia celé procedúry). SpeedButton1Click teraz na základe parametra Sender zistí, ktorý komponent ju zavolal a na základe toho nastaví premennú rezim. Teraz už univerzálna procedúra SpeedButton1Click môže vyzerať takto:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  if Sender = SpeedButton1 then
    rezim := pero
  else if Sender = SpeedButton2 then
    rezim := usecka
  else if Sender = SpeedButton3 then
    rezim := obdlznik
  else if Sender = SpeedButton4 then
    rezim := elipsa
  else if Sender = SpeedButton5 then
    rezim := vypln;
end;
```

Skôr ako sa pustíme do upravovania procedúr Image1MouseDown a Image1MouseMove, aby zvládali nové nástroje, musíme vysvetliť ako budú pracovať. Úsečka, obdĺžnik a elipsa budú pracovať na veľmi podobnom princípe - pri štarte nástroja (teda zatlačení myšou) sa zapamätá momentálny stav grafickej plochy v pomocnej bitmape bmp (globálna premenná typu TBitmap) a pri každom ťahaní nástroja, napr. úsečky, sa najprv grafická plocha vráti do tohto pôvodného stavu a potom sa nakreslí nová úsečka. Pritom si budeme pamätať počiatočný bod kreslenia v globálnych premenných **x0** a **y0**. Pomocná premenná bmp sa musí inicializovať vo FormCreate.

Režim vypĺňania oblasti je ešte jednoduchší - celý sa zrealizuje pri zatlačení tlačidla myši, t.j. v Image1MouseDown (mohol by byť napr. len v Image1MouseUp - premyslite si, čo by sa tým zmenilo). Na vypĺňanie oblasti použijeme metódu FloodFill.

```
var
  kreslim: Boolean = False;
  rezim: (pero, usecka, obdlznik, elipsa, vypln) = pero;
  bmp: TBitmap;
  x0, y0: Integer;


procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
  DoubleBuffered := True;
  Button1.Click;                  // zmaže sa grafická plocha
  Label1.Caption := '';
  with Image3, Canvas do
  begin
    Brush.Color := Image1.Canvas.Pen.Color;
    FillRect(ClientRect);
  end;
  bmp := TBitmap.Create;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  kreslim := ssLeft in Shift;
  if kreslim then
    with Image1.Canvas do
      case rezim of
        pero:
          MoveTo(X, Y);
        usecka, obdlznik, elipsa:
          begin
            bmp.Assign(Image1.Picture);
            x0 := X;
            y0 := Y;
          end;
        vypln:
          begin
            Brush.Color := Pen.Color;
            FloodFill(X, Y, Pixels[X,Y], fsSurface);
          end;
      end;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  Label1.Caption := IntToStr(X) + ' ' + IntToStr(Y);
  if kreslim then
    with Image1.Canvas do
      case rezim of
        pero:
          LineTo(X, Y);
        usecka:
          begin
            Draw(0, 0, bmp);
            MoveTo(x0, y0);
            LineTo(X, Y);
          end;
        obdlznik:
          begin
            Draw(0, 0, bmp);
            Brush.Style := bsClear;
            Rectangle(x0, y0, X, Y);
          end;
        elipsa:
          begin
            Draw(0, 0, bmp);
            Brush.Style := bsClear;
            Ellipse(x0, y0, X, Y);
          end;
      end;
end;
```

Zaškrtávacie políčko

Ešte urobíme posledné vylepšenie: umožníme používateľovi programu aby si mohol zvoliť, či obdĺžnik, resp. elipsa budú pri vykreslení vyplnené farbou (zatiaľ sme ich vyrobili "deravé" - nastavili sme Brush.Style := bsClear;) Do formulára vložíme ešte jeden komponent: zaškrtávacie políčko CheckBox. Nastavíme mu popis (Caption) na slovo "**plný**" a opravíme v metóde Image1MouseMove kreslenie obdĺžnika a elipsy:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  ...
        obdlznik:
          begin
            Draw(0, 0, bmp);
            Brush.Color := Pen.Color;
            if CheckBox1.Checked then
              Brush.Style :=bsSolid
            else
              Brush.Style :=bsClear;
            Rectangle(x0, y0, X, Y);
          end;
        elipsa:
          begin
            Draw(0, 0, bmp);
            Brush.Color := Pen.Color;
            if CheckBox1.Checked then
              Brush.Style :=bsSolid
            else
              Brush.Style :=bsClear;
            Ellipse(x0, y0, X, Y);
          end;
      end;
end;
```

Uvádzam kompletný projekt s touto kresliacou aplikáciou Skicar.zip.

Zhrnutie

Stavové premenné komponentov (vidíme ich v Inšpektore objektov) sú podobné položkám záznamu (record). Pri ich zmene v inšpektore objektov sa automaticky prejaví aj ich vizuálna zmena vo formulári (napr. šírka, text, bublinkový help a pod.). V Delphi sa takýto typ stavových premenných nazýva Vlastnosť, t.j. Property. Niektoré sú pre nás užitočné už počas tvorby formulára (Caption, Name, Hint, Width, Min, Max, ...), iné využijeme hlavne za behu programu (niektoré fungujú len za behu, a preto v inšpektore objektov nie sú). Meno komponentu (Name) treba zmeniť (ak ho z nejakých dôvodov zmeniť chceme) hneď ako sa položí do formulára, inak niektoré situácie Delphi nemusia správne uhádnuť a opraviť túto zmenu korektne. Vymenujme niektoré užitočné vlastnosti (property) nám známych komponentov:

- formulár TForm
  - Caption - String - titulok okna
  - ClientWidth, ClientHeight - veľkosť vnútra okna (nie obvodu okna, pre ten je veľkosť Width a Height)
  - Left, Top - x-ová a y-ová súradnica okna na obrazovke - môžeme celé okno posúvať
  - Visible - či je okno viditeľné
  - WindowState - okno môžeme zminimalizovať, zmaximalizovať
  - Canvas (funguje len počas behu programu) - šedá plocha - môžeme do nej kresliť (v udalosti OnPaint)
- tlačidlo TButton

        - Caption - text na tlačidle
        - Font - rovnako ako v grafickej ploche
        - Height, Width - veľkosť tlačidla
        - Left, Top - poloha vo formulári
        - Visible - môžeme ho skryť - False znamená, že ho nebude vidieť
        - Hint, ShowHint - String a Boolean - bublinková nápoveď (help) a informácia o tom, či sa má zobrazovať
        - Enabled - tlačidlo môžeme zablokovať - False znamená, že tlačidlo bude zašedené

Všimnite si, že vlastnosti Width, Height, Left, Top, Visible, Hint, ShowHint, Enabled - fungujú skoro rovnako pre skoro všetky typy komponentov.

- grafická plocha TImage
    - Canvas (funguje len počas behu programu) - sprístupnenie kreslenia do grafickej plochy
    - ClientRect (funguje len počas behu programu) - hodnota typu TRect, t.j. obdĺžnik, ktorý popisuje veľkosť plochy
- malé tlačidlo TSpeedButton - veľmi podobné obyčajnému tlačidlu, ale tieto tlačidlá môžu tvoriť skupinu (vzájomne sa vylučujú - maximálne jedno z nich je zatlačené), môže mať aj obrázok
    - Caption - text na tlačidle (môže byť spolu s obrázkom)
    - GroupIndex - poradové číslo skupiny tlačidiel, ktoré sa navzájom vylučujú
    - Down - či je zatlačené
    - Glyph - obrázok na tlačidle
- posuvná lišta TTrackBar
    - Min, Max - minimálna a maximálna hodnota
    - Position - momentálna hodnota
    - Orientation - (trHorizontal, trVertical)
- zaškrtávacie políčko TCheckBox
    - Caption - text za políčkom
    - Font - písmo pre text
    - Checked - či je zaškrtnuté
- text TLabel
    - Caption - ak reťazec obsahuje #13#10, bude viacriadkový
    - Font - písmo pre text
- textová plocha TMemo
    - Color - farba podkladu
    - Font - písmo
    - Lines - obsah textovej plochy - pole reťazcov
    - ReadOnly - či má používateľ zakázané meniť obsah
- editovací riadok TEdit - je veľmi podobný TMemo
    - Color - farba podkladu
    - Font - písmo
    - Text - momentálny obsah riadka
    - ReadOnly - či má používateľ zakázané meniť obsah
    - PasswordChar - ak má hodnotu napr. znak '*', tak sa namiesto zadávaných znakov budú vypisovať hviezdičky - používa sa pri zadávaní hesla

Udalosti (**Events**) môžeme definovať v záložke v inšpektora objektov. Slúžia na definovanie správania pri špecifických situáciách. Využívame to, že inšpektor objektov nám pomáha správne zadeklarovať tieto procedúry., ktorú vyvolá Windows, keď vznikne príslušná udalosť (napr. klikne sa, zmení sa, hýbe sa a pod.) * tieto procedúry

definujeme tak, že v záložke Events (v riadku) pri danej udalosti dvojklikneme v pravom stĺpci (alebo si zvolíme procedúru z už existujúcej ponuky procedúr) * nedopisujte manuálne žiadne nové procedúry na spracovávanie udalostí - vždy použite inšpektor objektov * ak potrebujete nejakú procedúru obsluhujúcu udalosť zrušiť, najlepšie to urobíte tak, že vyčistíte obsah procedúry - necháte v nej len počiatočný begin a koncový end a pri nasledujúcom uložení na disk (napr. Ctrl+S) ju Delphi automaticky vyhodí

Myš a robot

Robot je grafické pero, ktoré dokáže nejaké veci navyše: napr. jednoducho "vie natočiť súradnicovú sústavu", v grafickej ploche ich môžeme definovať ľubovoľný počet, a pod. Prvý príklad ilustruje použitie robota ako obyčajného grafického pera:

```
var
  r: TRobot;

procedure TForm1.FormCreate(Sender: TObject);
begin
  r := TRobot.Create;
  r.pu;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  r.pd;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  r.setxy(X, Y);
end;

procedure TForm1.Image1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  r.pu;
end;
```

V tomto programe je robot "prilepený" k myši - pri každom pohybe myši (onMouseMove) sa pohne na jej miesto. Robot má ale stále pero hore a kreslí len, keď príde udalosť onMouseDown, t.j. keď stlačíme nejaké tlačidlo myši. Do podprogramu pre udalosť onMouseMove môžeme pridať aj iné akcie, napr. sa robot najprv otočí smerom k novej pozícii (použili sme príkaz towards(x, y) - otočí robota k zadanému bodu v rovine) a až potom sa na ňu presunie - pritom robot môže niečo aj kresliť, napr. úsečky:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  r.towards(X, Y);
  r.movexy(X, Y);
  r.fd(100);
  r.fd(-100);
end;
```

môžeme pridať cs:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  cs;
```

```
  r.towards(X, Y);
  r.movexy(X, Y);
  r.fd(100);
  r.fd(-100);
end;
```

alebo

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
var
  i: Integer;
begin
  r.towards(X, Y);
  r.movexy(X, Y);
  for i := 1 to 5 do
  begin
    r.fd(30);
    r.rt(144);
  end;
end;
```

alebo farebné paličky:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if r.ispd then    // či má robot spustené pero
  begin
    r.PW := 5;
    r.PC := Random(256*256*256);
    r.towards(X, Y);
    r.movexy(X, Y);
    r.lt(90);
    r.fd(30);
    r.point(15);
    r.fd(-60);
    r.fd(30);
  end;
end;
```

alebo rovnostranný trojuholník:

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  cs;
  r.towards(X, Y);
  r.movexy(X, Y);
  r.PW := 5;
  r.PC := clGreen;
  r.rt(90);
  r.fd(8);
  r.lt(105);
  r.fd(30.9);
  r.lt(150);
  r.fd(30.9);
  r.lt(105);
  r.fd(8);
end;
```

Ďalší príklad ukáže generovanie robotov pri klikaní myšou: kliknutie na voľné miesto vygeneruje nového robota,

kliknutie na už existujúceho robota ho mierne otočí:

```
var
  r: array[1..100] of TRobot;
  n: Integer = 0;     // aktuálny počet robotov

procedure TForm1.FormCreate(Sender: TObject);
begin
  cs;   // aby sa grafická plocha zobrazila už pri štarte
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  i: Integer;
begin
  i := 1;
  while (i <= n) and not r[i].isnear(X, Y) do
    Inc(i);
  if i <= n then
    with r[i] do
    begin
      rt(15);
      fd(50);
      fd(-50);
    end
  else
  begin
    Inc(n);
    r[n] := TRobot.Create(X, Y);
    with r[n] do
    begin
      PW := 5;
      PC := Random(256*256*256);
      fd(50);
      fd(-50);
    end;
  end;
end;
```

Použili sme preddefinovanú logickú funkciu (metódu) isnear, ktorá pre daný bod v rovine odpovie, či je robot od neho blízko

Ďalšie námety:

- nové roboty by mohli vznikať na kliknutie so zatlačeným pravým tlačidlom myši, na ľavé tlačidlo môže robot urobiť nejakú akciu - nakresliť nejaký obrázok, alebo sa prilepí na myš a do nasledujúceho kliknutia putuje s myšou

Časovač

Predchádzajúcu úlohu, v ktorej sme klikaním vytvárali nové roboty, resp. ich otáčali, trochu pozmeníme: kliknutie na robota naštartuje kreslenie kružnice - a to pomaly - použime známu procedúru wait

kliknutie naštartuje kružnicu:

```
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  i, j: Integer;
begin
  i := 1;
  while (i <= n) and not r[i].isnear(X, Y) do
    Inc(i);
```

```
  if i > n then
  begin
    Inc(n);
    r[n] := TRobot.Create(X, Y);
    with r[n] do
    begin
      PW := 5;
      PC := Random(256*256*256);
      point;
    end;
  end
  else
    with r[i] do
      for j := 1 to 36 do
      begin
        fd(10);
        rt(10);
        wait(50);       // sleep(50); Image1.Repaint;
      end;
end;
```

vzniká zaujímavý efekt: kliknutie na robota ho rozbehne - počas tohto klikania môžeme ďalej klikať do plochy, na iné roboty - asi ľahko odhalíte, že každé nové kliknutie na robota, zastaví predchádzajúci pohyb a naštartuje nový pohyb - po skončení tohto nového pohybu sa dokončí aj ten pozastavený...

ak namiesto wait použijeme systémový podprogram sleep (zakomentovaná časť), tak sa tento efekt stratí: po rozbehnutí robota program nereaguje, až kým robot nedobehne celú kružnicu

- celé je to spôsobené tým, že v procedúre wait je ukryté Application.ProcessMessages - to znamená, že počas čakania 50 milisekúnd dovolíme Windows reagovať na iné udalosti, napr. aj na kliknutie myšou a teda umožníme znovu naštartovať novú kružnicu (po jej skončení sa pokračuje v prerušenej akcii)
- procedúra sleep nemá v sebe zabudované takéto mechanizmy a preto je teraz pre nás veľmi nebezpečná

ukážeme iný spôsob, ako sa rieši problém so spomaľovaním akcií: použijeme nový komponent časovač - Timer (z palety komponentov System) - položíme ho niekam do plochy - je to nevizuálny komponent a teda po spustení aplikácie nebude zobrazený

časovač si môžeme predstaviť ako hodiny, ktoré s nejakou frekvenciou "tikajú" - pri každom tiknutí môžu vykonať nejakú akciu

frekvenciu "tikania" nastavíme v stavovej premennej Interval, pozastavenie / naštartovanie časovača môžeme urobiť logickou stavovou premennou Enabled - True znamená, že hodiny bežia, False znamená, že sme ich pozastavili

časovač má jedinú udalosť onTimer, ktorá sa naštartuje po uplynutí "tikacieho" intervalu, t.j. zodpovedá jej procedúra Timer1Timer

v nasledujúcom príklade predpokladáme časovač so sekundovým tikaním, t.j. Interval je 1000; robot bude kresliť úsečku dĺžky 100 a každú sekundu sa otočí o 6 stupňov

sekundová ručička:

```
var
  r: TRobot;

procedure TForm1.FormCreate(Sender: TObject);
begin
  r := TRobot.Create;
  r.PW := 5;
end;
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  cs;
  r.rt(6);
  r.fd(100);
  r.fd(-100);
end;
```

príklad s robotmi, ktoré sa po kliknutí rozbehnú po kružnici preprogramujeme pomocou časovača: pre každého robota si budeme pamätať v poli kolko, koľko krokov po kružnici ešte musí prebehnúť

roboty krúžia po svojich kružniciach:

```
var
  r: array[1..100] of TRobot;
  kolko: array[1..100] of Integer;
  n: Integer = 0;

procedure TForm1.FormCreate(Sender: TObject);
begin
  cs;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  i: Integer;
begin
  i := 1;
  while (i <= n) and not r[i].isnear(X, Y) do
    Inc(i);
  if i > n then
  begin
    Inc(n);
    r[n] := TRobot.Create(X, Y);
    with r[n] do
    begin
      point(8);
      PW := 3;
    end;
    kolko[n] := 0;
  end
  else
  begin
    r[i].PC := Random(256*256*256);
    kolko[i] := 36;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to n do
    if kolko[i] > 0 then
      with r[i] do
      begin
        fd(10);
        rt(10);
        Dec(kolko[i]);
      end;
end;
```

časovaču Timer1 sme nastavili Interval na 50 milisekúnd

Ďalšie námety:

- umiestnite do plochy väčší počet robotov - napr. pravidelne ich rozmiestnite na priamke alebo kružnici a nechajte sa ich hýbať podľa nejakých pravidiel, napr. rôznou rýchlosťou po kružnici, alebo kmitať hore - dolu, tiež s rôznou rýchlosťou - napr. prvý nech má rýchlosť 1, druhý 1.1, tretí 1.2, ...

Jednoduchá animácia

Na záver ukážeme príklad práce s časovačom a bitmapami: na nejakom pozadí, napr. na fotografii tiger.bmp sa bude pomaly pohybovať nejaký obrázok, napr. opica.bmp - táto sa bude odrážať od bočných stien grafickej plochy ako lopta od stien miestnosti (v súbore opicatiger.zip sú obidve bitmapy). Okrem grafickej plochy (Image1) umiestnime do formulára aj časovač Timer1 s premennou Intreval nastavenou na 100.

```
var
  pozadie, obr: TBitmap;
  x, y, dx, dy, x2, y2: Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := True;
  pozadie := TBitmap.Create;
  pozadie.LoadFromFile('tiger.bmp');
  obr := TBitmap.Create;
  obr.LoadFromFile('opica.bmp');
  obr.Transparent := True;
  x := 100;
  y := 50;
  dx := 3;
  dy := 4;
  x2 := obr.Width div 2;
  y2 := obr.Height div 2;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Image1.Canvas.Draw(0, 0, pozadie);
  // Image1.Canvas.StretchDraw(Image1.ClientRect, pozadie);
  Inc(x, dx);
  if (x < x2) or (x >= Image1.Width-x2) then
    dx := -dx;
  Inc(y, dy);
  if (y < y2) or (y >= Image1.Height-y2) then
    dy := -dy;
  Image1.Canvas.Draw(x-x2, y-y2, obr);
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  pozadie.Free;
  obr.Free;
end;
```

namiesto Draw pri kreslení pozadia môžeme použiť aj iné prostriedky, napr. vykachličkovanie alebo StretchDraw, ale tie môžu výrazne spomaliť priebeh animácie

ukážkový priebežný test

Preverte si svoje momentálne schopnosti v programovaní v objektovom pascale a vyskúšajte vyriešiť tento test. V šk. roku 2004/2005 ho prváci riešili po 12. prednáške. Mali na to 60 minút a nesmeli používať žiadne pomôcky.

1. Súbor **subor1.txt** obsahuje tieto 3 riadky:

```
prvy##riadok#
dru#h#y#ria#dok
#ko#ni#ec#su#bo#ru#
```

Súbor **subor2.txt** obsahuje tento jeden riadok:

```
ABCDEFG
```

Zistite, čo bude v súbore subor3.txt po vykonaní tohto programu:

```
var
  t1, t2, t3: TextFile;
  z: Char;
begin
  AssignFile(t1, 'subor1.txt');
  Reset(t1);
  AssignFile(t2, 'subor2.txt');
  Reset(t2);
  AssignFile(t3, 'subor3.txt');
  Rewrite(t3);
  while not Eof(t1) do
  begin
    Read(t1, z);
    if (z = '#') or (z > 's') then
    begin
      if Eof(t2) then
        Reset(t2);
      Read(t2, z);
    end;
    Write(t3, z);
  end;
  CloseFile(t1);
  CloseFile(t2);
  CloseFile(t3);
end;
```

2. Rekurzívna procedúra **rekurzia** kreslí špirálu a pri návrate z rekurzie kreslí tiež nejaké čiary. Dopíšte do triviálneho prípadu príkazy, vďaka ktorým sa nakreslená špirála úplne vymaže (prekreslí) bielou farbou – zmeňte len farbu pera a natočenie robota.

```
var
  r: TRobot;

  procedure rekurzia(n: Integer);
  begin
    r.fd(n*10);
    r.rt(120);
    if n = 0 then
    begin
      ____________________________________________
    end
    else
```

```
      rekurzia(n-1);
    r.fd(n*10);
    r.lt(120);
  end;

begin
  r := TRobot.Create;
  r.PW := 8;
  rekurzia(10);
end;
```

3. V tomto programe sa postupne generuje 100 robotov, ktoré kreslia nejaké čiary. Po nejakom čase tento program spadne. Zistite, akú celkovú dráhu tieto roboty stihnú prejsť, kým ešte program nespadne. Robot **r** nekreslí žiadne čiary.

```
var
  r: TRobot;
  pole: array[1..100] of TRobot;
  i, n: Integer;
begin
  r := TRobot.Create;
  r.pu;
  n := 1;
  while True do
  begin
    r.fd(n);
    r.rt(30);
    pole[n] := TRobot.Create(r.X, r.Y, r.H);
    for i := 1 to n do
      pole[i].fd(1);
    Inc(n);
  end;
end;
```

4. Nasledujúca funkcia spracováva pomocou poľa množín nejaký znakový reťazec. Zistite čo vráti, ak zadáme
   a. spracuj('abcde')
   b. spracuj('mamamaemuaemamamamu')

```
function spracuj(const s: String): String;
var
  p: array['a'..'z'] of set of 'a'..'z';
  z, z1: Char;
  i: Integer;
begin
  for z := 'a' to 'z' do
    p[z] := [];
  for i := 1 to length(s)-1 do
    p[s[i]] := p[s[i]] + [s[i+1]];
  Result := '';
  for z := 'a' to 'z' do
    for z1 := 'a' to 'z' do
      if z1 in p[z] then
        Result := Result + z + z1 + ' ';
end;
```

5. Zistite, aké prvky bude obsahovať množina **m** na konci programu:

```
type
  Enum = (a, b, c, d, e, f, g);
var
  m: set of Enum;
  i: Integer;
```

```
  j: Enum;
begin
  m := [];
  for i := 100 to 150 do
  begin
    j := Enum(i mod (Ord(High(Enum))+1));
    if j in m then
      m := m - [j]
    else
      m := m + [j];
  end;
  // výpis množiny m
end;
```

6. Nasledujúca procedúra by mala upraviť súbor s obrázkom tak, aby mal veľkosť presne **sir** x **vys**. Preto najprv bitmapu načíta, vytvorí si novú s požadovanými rozmermi a na presnú pozíciu sem nakopíruje pôvodnú bitmapu, tak aby bola vycentrovaná – ak bude napr. **sir** x **vys** menšie ako rozmery bitmapy, vtedy v novej bitampe bude len stredná časť obrázka. Na záver túto bitmapu zapíše naspäť do súboru. Opravte všetky chyby v programe a dopíšte chýbajúce časti:

```
procedure uprav(subor: String; sir, vys: Integer);
var
  bmp1, bmp2: TBitmap;
begin
  bmp1 := TBitmap.Create;
  bmp1.LoadFromFile(subor);
  bmp2 := TBitmap.Create;
  bmp2.Width := bmp1.Width;
  bmp2.Height := bmp1.Height;
  bmp2.PixelFormat := pf24bit;
  bmp1.Canvas.Draw(__________________, __________________, bmp2);
  bmp1.Free;
  bmp2.Free;
  bmp1.SaveToFile(subor);
end;
```

7. Nasledujúca rekurzívna funkcia nejako spracováva číselné vstupy.
    a. Zistite, akú hodnotu vráti pre volanie cislo(24, 2)
    b. Nájdite aspoň 3 rôzne čísla x, ktoré pre volanie cislo(x, 2) vrátia hodnotu **15**. Výsledok môžete vyjadriť aj v tvare nejakého aritmetického výrazu.

```
function cislo(a, b: Integer): Integer;
begin
  Result := 0;
  while a mod b = 0 do
  begin
    Inc(Result, b);
    a := a div b;
  end;
  if a > 1 then
    Result := Result + cislo(a, b+1);
end;
```

8. Funkcia vyrob generuje maticu 10x10. Chceli by sme, aby 1 boli len na oboch uhlopriečkach. Doplňte chýbajúce časti v programe.

```
type
  Matica = array[1..10, 1..10] of 0..1;

function vyrob: Matica;
```

```
var
  i, j, k, k1, k2: Integer;
begin
  for i := 1 to 10 do
  begin
    k1 := 1; k2 := 1;
    for j := 1 to 9 do
      if _________ then k1 := 2*k1
      else k2 := 2*k2;
    k := ___________________________________;
    for j := 1 to 10 do
    begin
      Result[i, j] := k mod 2;
      k := ____________________;
    end;
  end;
end;
```

9. Nasledujúca funkcia nejako spracováva celočíselné pole:

```
type
  Pole = array[1..15] of 1..255;

function prerob(a: Pole): Pole;
var
  c: array[1..255] of Integer;
  i: Integer;
begin
  for i := 1 to 255 do
    c[i] := 0;
  for i := 1 to High(a) do
    c[a[i]] := c[a[i]]+1;
  for i := 2 to 255 do
    c[i] := c[i-1]+c[i];
  for i := High(a) downto 1 do
  begin
    Result[c[a[i]]] := a[i];
    Dec(c[a[i]]);
  end;
end;
```

Zistite, aké pole bude výsledkom pre vstupné pole (1,3,5,7,9,11,9,7,5,3,1,20,7,2,1).

10. Funkcia subor číta otvorený textový súbor a vytvára z neho nejaký znakový reťazec:

```
function subor(var t: TextFile): String;
var
  z1, z2: Char;
begin
  Result := '';
  z1 := ' ';
  while not Eof(t) do
  begin
    Read(t, z2);
    if (z1 = ' ') <> (z2 = ' ') then
      Result := Result + z2;
    z1 := z2;
  end;
end;
```

Zistite, čo vráti, ak jej dáme spracovať súbor so zdrojovým textom tejto funkcie a začíname ho čítať od riadka so slovom **begin**. Stačí zistiť prvých 20 znakov z výsledného reťazca.

Teoreticky by sa za tento test dalo získať 18.8 bodov, ale maximum bolo len 10 bodov. Preto nebolo treba riešiť všetky príklady. Kto vyriešil viac, započítalo sa mu len týchto 10 (takýchto študentov bolo 14 zo 74). Nasleduje tabuľka hodnôt jednotlivých príkladov - študenti ale o tom dopredu nevedeli.

1	2	3	4	5	6	7	8	9	10	súčet
1,4	**2**	**2**	**1,3**	**1,5**	**2**	**2,3**	**2,3**	**2**	**2**	**18,8**

rozpracovaná 13. prednáška: úvod do OOP

čo už vieme:

- už vieme pracovať s niektorými druhmi objektov, napr. s robotmi, bitmapami a tiež s mnohými komponentmi, ktoré sú tiež objektami

čo sa na tejto prednáške naučíme:

- základné pravidlá pri definovaní a jednoduchom používaní objektov
- zoznámime sa s pojmami **trieda**, **inštancia**, **metóda**, **konštruktor**, **dedičnosť**, ...

Objekt robot

Pripomeňme si, aké pojmy sme zaviedli v 2. prednáške pri definovaní robota:

V doterajších príkladoch sme zadefinovali a používali buď celočíselné premenné (Integer) alebo premenné typu TRobot. Tieto robotové premenné sa líšia od "obyčajných" premenných tým, že

- si pamätajú svoj momentálny stav v svojich tzv. **stavových premenných** (napr. pozícia, farba, ...)
- majú svoje *súkromné* príkazy, pomocou ktorých ich nejako riadime, resp. meníme ich stavové premenné - takýmto príkazom - sú to procedúry - hovoríme **metódy** a "rozumejú" im len roboty (premenné typu TRobot)
- musia byť vytvorené (nie deklarované) špeciálnym spôsobom (TRobot.Create(...);) a kým sa takto nevytvoria, nesmú sa vôbec používať
- takýmto premenným hovoríme **OBJEKT** a typom, z ktorých vytvárame objekty (napr. TRobot) hovoríme **TRIEDA** (po anglicky object a class) - niekedy sa objektu hovorí aj **inštancia triedy**
- okrem robotov sme sa už stretli aj s inými objektmi, napr. Form1, Image1, Button1 ale aj Image1.Canvas, ktorá je inštanciou triedy TCanvas
- zatiaľ si o objektoch treba zapamätať, že sú to premenné, ktoré môžu v sebe obsahovať veľa stavových premenných a tiež "v sebe" obsahujú nejaké svoje procedúry (metódy) => tomuto hovoríme **zapuzdrenie** (enkapsulácia), lebo v jednom "puzdre" sú aj údaje (stavové premenné) aj algoritmy (metódy), ktoré vedia s týmito údajmi pracovať.
- Z doterajších skúseností vieme, že s objektom robot sa pracuje veľmi podobne ako s obyčajným záznamom

(record). Pozrime sa na tieto deklarácie (definujeme miesto - bod v grafickej ploche):

porovnanie deklarácií záznamu a triedy:

```
type
  Miesto = record
    x, y: Integer;
  end;

var
  zm: Miesto;
```

```
type
  TMiesto = class
    x, y: Integer;
  end;

var
  om: TMiesto;
```

druhá definícia deklaruje triedu (a teda premenná **om** je objekt) a pre používateľa pritom vznikajú tieto rozdiely:

- premenná **zm** typu **Miesto** už po svojom zadeklarovaní existuje a zatiaľ má nedefinovanú hodnotu, môžeme do nej (do jej položiek) už priradzovať (napr. `zm.x := 10;`)
- premenná **om** typu **TMiesto**, t.j. inštancia triedy **TMiesto**, zatiaľ ešte **nie je vytvorená** a preto s ňou nesmieme pracovať, musíme ju najprv vytvoriť (skonštruovať) a až potom s ňou môžeme pracovať, napr. priradiť hodnoty do jej položiek (stavových premenných)
- objekty sa vytvárajú pomocou takéhoto zápisu:

  inštancia := *trieda*.**Create**;

  alebo

  inštancia := *trieda*.**Create**(*parametre*);

  napr. musíme zapísať

  m := TMiesto.Create;

- s položkami záznamu aj objektu pracujeme úplne rovnako ... (napr. môžeme použiť **with**)
- zvykneme po skončení práce s objektom tento zrušiť pomocou **Free**, napr. **om.Free;** (je to podobné uzatvoreniu súboru pomocou CloseFile)
- Ak zadefinujeme **TMiesto** ako triedu namiesto záznamu získavame silný nástroj - objektovo orientované programovanie. Ukážeme novú vlastnosť, ktorú poskytujú iba objekty.

Dedenie

Z hotovej triedy môžeme vytvoriť ďalšie odvodené triedy tak, že jazyku pascal oznámime, že chceme ponechať všetko, čo už bolo zadefinované doteraz a pridáme nejaké nové položky (stavové premenné). Napr.

```
type
  TMiesto = class
    x, y: Integer;
  end;

  TBod = class(TMiesto)
    vid: Boolean;
  end;

var
  m: TMiesto;
  b: TBod;
```

Trieda **TBod** je vytvorená tak, že má všetky vlastnosti triedy **TMiesto** a pridala si novú stavovú premennú **vid**, t.j.

táto trieda má 3 stavové premenné; nové pojmy:

- trieda **TBod** vznikla ako **potomok** triedy **TMiesto** (descendent type [di'sendent])
- trieda **TMiesto** je **predok** triedy **TBod** (ancestor type [aenseste])
- trieda **TBod zdedila** všetky vlastnosti triedy **TMiesto** (inherit)
- každá trieda môže mať ľubovoľný počet potomkov, ale len jediného predka
- potomkovia môžu dodefinovať nové stavové premenné, ale môžu aj predefinovať už zdedené

príklad demonštruje použitie inštancií týchto tried:

```
type
  TMiesto = class
    x, y: Integer;
  end;
  TBod = class(TMiesto)
    vid: Boolean;
  end;

var
  m: TMiesto;
  b: TBod
begin
  m := TMiesto.Create;
  b := TBod.Create;
  m.x := 100;
  m.y := 150;
  with b do
  begin
     x := 200;
     y := m.y;
  end;
  m.Free;
  b.Free;
end;
```

Metódy objektu

sú to procedúry alebo funkcie, ktoré sa deklarujú vo vnútri objektu a sú s ním pevne spojené

slúžia napr. na modifikovanie stavových premenných, zabezpečujú "správanie" objektu, ...

zadefinujeme procedúru na nastavenie hodnôt stavových premenných:

```
procedure TMiesto.ZmenXY(NoveX, NoveY: Integer);
begin
  x := NoveX;
  y := NoveY;
end;
```

aby túto definíciu pascal správne pochopil, t.j. že patrí k triede TMiesto, musí byť jej hlavička zadefinovaná aj v definícii triedy:

deklarovanie metódy:

```
type
  TMiesto = class
    x, y: Integer;
    procedure ZmenXY(NoveX, NoveY: Integer);
  end;
```

tieto dve definície metódy musia byť identické

vo vnútri metódy sa pracuje so stavovými premennými momentálnej triedy ako keby bola obklopená neviditeľným príkazom with

nakoľko trieda TBod je potomkom triedy TMiesto, tak zdedila aj metódy triedy TMiesto a preto aj inštancie triedy TBod môžu používať tieto metódy:

dedenie metódy:

```
var
  b: TBod;
begin
  b := TBod.Create;
  b.ZmenXY(200, 150);   // volanie metódy
  ...
```

potomok môže **prekryť** metódu svojho (hociktorého) predka (môže hoci aj zmeniť typ a počet parametrov), napr. prekrytá metóda:

```
type
  TBod = class(TMiesto)
    vid: Boolean;
    procedure ZmenXY(NoveX, NoveY: Integer; NovyVid: Boolean);
  end;

procedure TBod.ZmenXY(NoveX, NoveY: Integer; NovyVid: Boolean);
begin
  x := NoveX;
  y := NoveY;
  vid := NovyVid;
end;
```

alebo môžeme využiť pri definovaní prekrývajúcej metódy aj zdedenú metódu, napr. zdedenie metódy pri prekrytí:

```
procedure TBod.ZmenXY(NoveX, NoveY: Integer; NovyVid: Boolean);
begin
  inherited ZmenXY(NoveX, NoveY);
  vid := NovyVid;
end;
```

Nasledujúci príklad ukáže použitie objektov - kružníc:

do formulára umiestnime grafickú plochu (Image1) a 5 tlačidiel (Button1 až Button5), ktorým budeme postupne priraďovať nejaké akcie

zadefinujeme triedu **TKruh**, ktorá popisuje kružnicu v grafickej ploche (x, y, r, farba, vid) s metódami na vytvorenie kruhu, vykreslenie, ukrytie, zmenu farby a posun obrázku

ďalej zadefinujeme 3 globálne premenné typu **TKruh**

na prvé tlačidlo sa niekde v ploche vytvorí kružnica nejakej veľkosti a farby

podobne aj na 2. a 3. tlačidlo ďalšie kružnice príklad s kružnicami:

```
type
  TKruh = class           // potomok TObject
    x, y, r: Integer;
    f: TColor;
    vid: Boolean;
    constructor Create(xx, yy, rr: Integer);
    procedure ukaz;
    procedure skry;
```

```
    procedure ZmenFarbu(ff: TColor);
    procedure posun(dx, dy: Integer);
  end;

constructor TKruh.Create(xx, yy, rr: Integer);
begin
  // inherited Create;
  x := xx;
  y := yy;
  r := rr;
  vid := False;
  f := clBlack;
end;

procedure TKruh.ukaz;
begin
  with Form1.Image1.Canvas do
  begin
    Pen.Color := f;
    Brush.Style := bsClear;
    Ellipse(x-r, y-r, x+r, y+r);
  end;
  vid := True;
end;

procedure TKruh.skry;
begin
  with Form1.Image1.Canvas do
  begin
    Pen.Color := clWhite;
    Brush.Style := bsClear;
    Ellipse(x-r, y-r, x+r, y+r);
  end;
  vid := False;
end;

procedure TKruh.ZmenFarbu(ff: TColor);
begin
  f := ff;
  if vid then
    ukaz;
end;

procedure TKruh.posun(dx, dy: Integer);
var
  b: Boolean;
begin
  b := vid;
  if vid then
    skry;
  Inc(x, dx);
  Inc(y, dy);
  if b then
    ukaz;
end;

////////////////////////////

var
  a, b, c: TKruh;

procedure TForm1.Button1Click(Sender: TObject);
begin
  a := TKruh.Create(100, 100, 50);
  with a do
  begin
    ZmenFarbu(clGreen);
```

```
    ukaz;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  b := TKruh.Create(300, 200, 100);
  b.ZmenFarbu(clRed);
  b.ukaz;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  c := TKruh.Create(200, 250, 70);
  c.ukaz;
end;
```

v tomto príklade sme predefinovali aj "konštruovanie" objektu (Create)

Konštruktor objektu

je to špeciálna metóda, ktorá sa používa **iba** pre vytvorenie novej inštancie

jej hlavnou úlohou je vyhradiť v počítači miesto pre samotný objekt a do premennej (inštancie) priradiť referenciu na toto miesto

okrem toho môže táto procedúra (konštruktor), napr. aj inicializovať obsahy stavových premenných, prípadne robiť iné akcie

namiesto **procedure** musíme deklarovať slovom **constructor**

všimnite si, že konštruktor sme používali aj v predchádzajúcom príklade o triede **TMiesto** (`m := TMiesto.Create;`) a pritom sme ho nikde nedeklarovali ani nedefinovali - platí jedna dôležitá vec: každá trieda, ktorej neuvedieme predka, je odvodená (je potomkom) základnej triedy **TObject** - tento objekt má definovaný konštruktor **Create**, ktorý má prázdne telo, t.j. platí:

```
constructor TObject.Create;
begin
end;
```

a preto nemá zmysel v nami definovanom konštruktore triedy, ktorá nemá predka, písať

```
inherited Create;
```

namiesto **TKruh = class** by sme mohli deklarovať aj **Tkruh = class(TObject)** - bolo by to to isté

konštruktorov môže byť definovaných aj viac - musia sa líšiť menom, napr. druhý konštruktor:

```
type
  TKruh = class
    ...
    constructor Create(xx, yy, rr: Integer);
    constructor Create1(xx, yy, rr: Integer);
    ...
  end;

constructor TKruh.Create(xx, yy, rr: Integer);
begin
  x := xx;
  y := yy;
  r := rr;
```

```
  vid := False;
  f := clBlack;
end;

constructor TKruh.Create1(xx, yy, rr: Integer);
begin
  Create(xx, yy, rr);
  Ukaz;
end;
```

zadefinovali sme druhý konštruktor **Create1**, ktorý sa líši od prvého tým, že objekt po skonštruovaní hneď zobrazí

zrejme, keď je definovaných viac konštruktorov, pri vytváraní novej inštancie sa musíme rozhodnúť, ktorý z konštruktorov použijeme (vždy môžeme použiť len jeden!)

Pokračujeme v rozrobenom projekte:

štvrté tlačidlo rozhýbe kruhy tak, že každý sa paralelne pohne nejakým smerom 20 krokov

použijeme na to časovač (Timer1- nastavíme mu v inšpektore **Interval** na 100 a **Enabled** na False), napr. posúvanie kruhov:

```
var
  poc: Integer;

procedure TForm1.Button4Click(Sender: TObject);
begin
  poc := 20;
  Timer1.Enabled := True;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  a.posun(3, 2);
  b.posun(-5, -1);
  c.posun(2, -3);
  Dec(poc);
  if poc <= 0 then
    Timer1.Enabled := False;
end;
```

lenže ak by sme stlačili toto tlačidlo predtým, ako sme vyrobili všetky kruhy, program nám spadne na už známej chybe "Acces violation at address...", preto musíme zabezpečiť, že posúvať budeme len ten kruh, ktorý už bol naozaj vyrobený. Ako zistíme, či bola inštancia už vytvorená (Create)?

v skutočnosti v premennej **a** nie je priamo obsah objektu (stavové premenné, tak ako by to bolo v zázname), ale **referencia** na nejaké miesto v pamäti počítača, kde sa naozaj tento objekt nachádza

kým nie je objekt vytvorený (skonštruovaný), tak v premennej **a** nie je **žiadna referencia**, čo sa v pascale označuje slovom **nil** a teda testovanie na nil:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if a <> nil then
    a.posun(3, 2);
  if b <> nil then
    b.posun(-5, -1);
  if c <> nil then
    c.posun(2, -3);
  Dec(poc);
  if poc <= 0 then
    Timer1.Enabled := False;
```

```
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  a := nil;
  b := nil;
  c := nil;
end;
```

je dobré si zvyknúť hneď na začiatok programu (napr. do FormCreate) dať priradenia

```
a := nil; b := nil; ...
```

dohodli sme sa, že každý objekt musíme po skončení uvoľniť pomocou **Free** - v našom príklade to urobíme v udalosti **OnFormDestroy**: záverečné zrušenie inštancií:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  a.Free;    // if a <> nil then a.Destroy;
  b.Free;
  c.Free;
end;
```

piate tlačidlo uvoľní všetky inštancie (a, b, c): korektné zrušenie inštancie:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  if a <> nil then
    with a do
    begin
      if vid then
        skry;
      Free;
      a := nil;      // aby a nereferencovalo na už nedefinovaný objekt
    end;
  if b <> nil then
    with b do
    begin
      if vid then
        skry;
      Free;
      b := nil;
    end;
  if c <> nil then
    with c do
    begin
      if vid then
        skry;
      Free;
      c := nil;
    end;
end;
```

dvojicu príkazov **a.Free; a := nil;** môžeme nahradiť volaním štandardnej procedúry **FreeAndNil(a);**

Vlastnosti objektov:

enkapsulácia (encapsulation - zapuzdrenie)

- nový dátový typ **trieda** (class)
- spojenie typu record a procedúry/funkcie pre manipuláciu so **stavovými premennými**

dedičnosť (inheritance)

- od definovaného objektu môžeme odvodiť celú **hierarchiu** objektov
- t.j. potomkov, ktorí dedia prístup k dátovým aj programovým zložkám

polymorfizmus

- zdieľanie akcií v hierarchii objektov
- budeme sa učiť až neskôr

príklad - trieda veľké čísla:

```
const
  max = 100000;

type
  TVelkeCislo = class
    c: array[1..max] of Byte;
    p: 0..max;                  // počet platných cifier
    constructor Create;
    procedure prirad(x: Integer);
    procedure pricitaj(x: Integer);
    procedure nasob(x: Integer);
    procedure vypis;
  end;
```

aby sa dal, napr. vypočítať veľký faktoriál, napr. použitie veľkých čísel:

```
var
  f: TVelkeCislo;
begin
  f := TVelkeCislo.Create;
  f.prirad(1);
  for i := 2 to N do
    f.nasob(i);
  f.vypis;
end;
```

14. prednáška: programové jednotky, dynamické polia

čo už vieme:

- keď definujeme nejaký formulár, Delphi na to pripraví súbor **Unit1.pas**, v ktorom je samotný program (v súbore **Unit1.dfm** je informácia o komponentoch vo formulári)
- keď potrebujeme pracovať s robotmi, musíme za implementation napísať uses RobotUnit, aby Delphi vedelo, kde sú roboty definované

čo sa na tejto prednáške naučíme:

- vytvárať vlastné programové jednotky, štruktúru a vlastnosti takýchto jednotiek
- realizáciu vlastného typu zásobník pomocou triedy
- čo sú to dynamické polia a ako sa s nimi pracuje
- čo sú to parametre typu otvorené pole a ako ich môžeme používať

Programové jednotky - unity

Programy v pascale sa väčšinou rozdeľujú do viacerých súborov. Každý z týchto súborov potom môže obsahovať nejaké podprogramy, definície typov a konštánt a aj deklarácie premenné. Takémuto balíku podprogramov, premenných, typov a konštánt hovoríme programová jednotka (unit). Ak sa to dobre navrhne, tak rôzne programové jednotky môžu programovať nezávislí programátori na rôznych miestach a v rôznom čase.

Každá programová jednotka zrejme vie riešiť nejaké úlohy (podprogramy) a pritom využíva deklarácie typov, konštánt a premenných buď zo svojej jednotky, alebo z nejakých iných (možno cudzích) jednotiek. A naopak, iné jednotky (možno úplne cudzie) môžu využívať niektoré deklarácie alebo podprogramy z našich jednotiek. Hovoríme, že každá jednotka môže využívať niektoré časti iných jednotiek (import) a tiež môže niektoré svoje časti poskytnúť pre iné jednotky (označí ich na export). Keď je nejaká jednotka naprogramovaná, môže sa skompilovať a ďalším programátorom poskytnúť už len v takomto skompilovanom stave (súbor s príponou .DCU) - ďalší programátori takúto jednotku môžu používať, ale nevidia jej zdrojový tvar a nemôžu ju ani študovať ani modifikovať.

Najčastejšie použitie programovej jednotky je

- ako knižnice podprogramov, ktoré sa dajú pripojiť k rôznym programom (bez sprístupnenia zdrojového kódu) - napr. štandardná jednotka od Borlandu s menom System obsahuje štandardné pascalovské procedúry a funkcie, napr. AssignFile, Reset, Write, Copy, Insert, Cos, Ord, Odd, Pi, ... ale aj definície niektorých typov, napr. triedu TObject,
- ako programový kód, ktorý popisuje správanie formulára (najčastejšie má každý formulár v aplikácii svoju jednotku - napr. Unit1.pas) - vytvorený je automaticky prostredím Delphi pri vytváraní nového formulára (popis komponentov vo formulári je v súbore s príponou .DFM),
- na rozčlenenie rozsiahleho programu do logicky súvisiacich menších modulov, resp. na popis nejakej časti hierarchie objektov.

Programové jednotky sú v súboroch s príponou .PAS, po prekompilovaní je ich preložený kód v súbore s príponou .DCU (aby sme mohli nejaký unit použiť v našom programe, nepotrebujeme jeho zdrojový tvar .PAS, ale stačí nám súbor .DCU).

Programová jednotka má svoju presnú štruktúru. Skladá sa z niekoľkých častí, ktoré sú označené pascalovskými rezervovanými slovami:

- unit - označenie mena jednotky,

- interface - začiatok popisu tých identifikátorov, ktoré plánujeme poskytnúť ďalším jednotkám - tzv. export,
- implemetation - začiatok časti, v ktorej definujeme podprogramy (aj metódy) z časti interface a tiež popisujeme ďalšie súkromné identifikátory (premenné, typy, procedúry) - tieto už iné jednotky nevidia,
- initialization - nepovinná časť - tu sa môžu nachádzať akcie, ktoré by sa mali vykonať počas inicializácie jednotky
- finalization - tiež nepovinná časť - tu sa môžu nachádzať akcie, ktoré by sa mali vykonať počas ukončovania práce celej programovej jednotky,
- end. - ukončenie programovej jednotky.

V zdrojovom tvare programová jednotka vyzerá takto:

```
unit menoUnitu;     // meno jednotky sa musí zhodovať s menom súboru

interface

uses ...

{ verejné deklarácie a definície konštánt, typov (aj tried),
  premenných, procedúr a funkcií (len ich hlavičky), každý, kto
  jednotku používa, k nim môže pristupovať ako k svojim vlastným }

const ...

type ...

procedure ...

function ...

var ...

implementation

uses ...

{ definície verejných procedúr a funkcií (ich definície je možné
  písať bez ohľadu na poradie a vzájomné odkazy, aj bez formálnych
  parametrov, t.j. napísaním len názvu podprogramu), súkromné
  deklarácie a definície konštánt, typov, premenných, procedúr
  a funkcií }

const ...

type ...

var ...

procedure ...

function ...

initialization    // tieto ďalšie časti môžu chýbať

// inicializácia jednotky – spustí sa tesne pred spustením
// hlavného programu

finalization

// finalizácia jednotky – spustí sa po skončení hlavného programu

end.
```

Jednotky môžeme použiť v iných programoch, resp. jednotkách, tak, že do časti deklarácií uses napíšeme mená

týchto jednotiek - ak sa jednotka nenachádza v tom istom priečinku disku ako samotný projekt, musíme zapísať meno súboru aj s cestou (meno súboru sa aj tak musí zhodovať s názvom jednotky), napr.

```
uses Messages, Vektory in '..\vektory.pas', PomocnyUnit;
```

Poradie mien jednotiek v popise uses určuje poradie spúšťania ich inicializácií. Nasledujúci príklad demonštruje jednoduchú programovú jednotku s inicializáciou aj finalizáciou:

```
unit MojUnit;

interface

procedure zapis(s: String);

implementation

var
  t: TextFile;

procedure zapis(s: String);
begin
  Writeln(t, s);
end;

initialization

  AssignFile(t, 'd:\test.txt'); Rewrite(t);

finalization

  CloseFile(t);

end.
```

Všimnite si

- jednotka nemá žiadne uses a teda nepoužíva žiadne iné jednotky,
- v časti interface definuje jediný podprogram: procedúru zapis,
- v časti implementation okrem realizácie procedúry zápis vidíme deklaráciu súkromnej premennej t - typu textový súbor, túto premennú nebude vidieť žiadna iná jednotka,
- časť initialization sa automaticky spustí pri štarte programu: otvorí sa textový súbor na zápis,
- časť finalization sa vykoná pri ukončovaní programu: otvorený súbor sa zatvorí.

Programová jednotka StackUnit - zásobník

Zásobník je údajová štruktúra, ktorá sa podobá postupnosti prvkov. S touto štruktúrou pracujeme pomocou dvoch operácií: pridaj prvok a vyber prvok. Narozdiel od klasického radu, napr. pred obchodom, tejto štruktúre hovoríme nespravodlivý rad: noví zákazníci sa zaraďujú na koniec radu a obslúžený bude ten, kto je v rade na konci - prišiel ako posledný. Tomuto v informatike hovoríme aj LIFO: last in - first out. Takže

- prvky do štruktúry môžeme pridávať len na koniec - operáciou push,
- prvky zo štruktúry môžeme odoberať len z konca - operácia pop,
- pomocná operácia empty slúži na otestovanie, či je štruktúra prázdna.

Túto údajovú štruktúru teraz naprogramujeme ako triedu s metódami push, pop a empty. Zásobník sa po anglicky povie **stack** a preto triedu nazveme TStack a celú ju zadefinujeme v samostatnej programovej jednotke StackUnit. Novú programovú jednotku najlepšie vytvárame takto:

- v menu **File** zvolíme položku **New** a v nej vyberieme **Unit**,

- v editovacom okne Delphi sa vytvorí schéma pre novú programovú jednotku:

```
unit Unit2;

interface

implementation

end.
```

- nie je vhodné premenovávať túto jednotku zmenou identifikátora za slovom unit, ale najlepšie bude tento súbor uložiť na disk a pritom mu zmeniť meno: v menu **File** zvolíme položku **Save As...** a teraz určíme meno StakUnit. Automaticky sa toto meno zapíše aj v hlavičke súboru za slovo unit.
- teraz môžeme zapísať samotný obsah programovej jednotky.

Kompletná jednotka StackUnit môže vyzerať takto:

```
unit StackUnit;

interface

type
  Prvok = String;
  TStack = class
    st: array[1..100] of Prvok;
    vrch: Integer;
    constructor Create;
    procedure push(p: Prvok);
    procedure pop(var p: Prvok);
    function top: Prvok;
    function full: Boolean;
    function empty: Boolean;
  end;

implementation

uses
  Dialogs;    // obsahuje ShowMessage

procedure chyba(s: String);
begin
  ShowMessage(s);          // okno so správou
  Halt;
end;

constructor TStack.Create;
begin
  vrch := 0;
end;

procedure TStack.push(p: Prvok);
begin
  if full then
    chyba('Plný zásobník pri príkaze push');
  Inc(vrch);
  st[vrch] := p;
end;

procedure TStack.pop(var p:Prvok);
begin
  if empty then
    chyba('Prázdny zásobník pri príkaze pop');
  p := st[vrch];
  Dec(vrch);
```

```
end;

function TStack.top: Prvok;
begin
  if empty then
    chyba('Prázdny zásobník pri príkaze top');
  Result := st[vrch];
end;

function TStack.full: Boolean;
begin
  Result := vrch = High(st);
end;

function TStack.empty: Boolean;
begin
  Result := vrch = 0;
end;

end.
```

V tomto programe si všimnite:

- zásobník realizujeme ako 100-prvkové pole znakových reťazcov,
- prvky tejto štruktúry sú definované typom Prvok - ak zmeníme deklaráciu tohto typu, zmení sa aj štruktúra prvkov zásobníka,
- typ Prvok je tiež deklarovaný v časti interface a preto ho budú môcť používať aj tie časti programu, ktoré zapíšu uses StackUnit,
- stavová premenná vrch slúži na zapamätanie pozície posledného prvku v zásobníku (posledne pridávaného) - ďalší pridávaný prvok sa bude dávať za tento vrchný, resp. odoberať zo zásobníka sa bude práve tento vrchný,
- okrem push, pop a empty sme dodefinovali aj tieto ďalšie metódy:
    - full - logická funkcia vráti True, ak je zásobník už plný a teda ďalšie push by spôsobilo spadnutie programu,
    - top - funkcia typu Prvok vráti prvok z vrchu zásobníka, pričom, na rozdiel od pop, tento prvok na zásobníku ponechá - toto môže byť v niektorých aplikáciách dosť užitočná metóda.
- zadefinovali sme pomocnú procedúru chyba, ktorá má za úlohu oznámiť správu o chybe a násilne ukončiť celú aplikáciu:
    - preddefinovaná procedúra ShowMessage sa nachádza v knižničnej programovej jednotke Dialogs a preto sme museli na začiatok implementačnej časti zapísať uses Dialogs,
    - použili sme pascalovský príkaz Halt (štandardnú procedúru), ktorá okamžite ukončí bežiaci program - tento príkaz používajte veľmi rozvážne, lebo pri komplexnejších projektoch môžete pri takomto prerušení programu prísť o dôležité ukončovacie akcie rozbehnutých procesov.

Nasledujúca ukážka demonštruje jednoduché použitie zásobníka: najprv do prvkov zásobníka postupne naukladáme riadky textového súboru a potom tieto riadky vyberáme a ukladáme do druhého textového súboru. Zrejme do druhého súboru sa dostanú v opačnom poradí:

```
uses
  StackUnit;

...

procedure TForm1.Button1Click(...);
var
  t1, t2: TextFile;
  z: TStack;
```

```
  s: String;
begin
  AssignFile(t1, 'unit1.pas');
  Reset(t1);
  AssignFile(t2, 'x.txt');
  Rewrite(t2);
  z := TStack.Create;
  while not Eof(t1) do
  begin
    Readln(t1, s);
    z.push(s);
  end;
  while not z.empty do
  begin
    z.pop(s);
    Writeln(t2, s);
  end;
  z.Free;
  CloseFile(t1);
  CloseFile(t2);
end;
```

Všimnite si, že na záver programu sme použili z.Free - podobne ako pri bitmapách je dobre na záver nezabudnúť každý nami vytvorený objekt aj zrušiť, t.j. uvoľniť ho z pamäti počítača.

Nakoľko hovoríme o programových jednotkách, pozrime sa ako vyzerá tzv. hlavný program. Je to súbor s príponou **.dpr**, ktorý sa vytvára automaticky a zapisovať do neho by sme mali len veľmi opatrne. Tento hlavný program obsahuje vymenované naše programové jednotky (v časti uses) a telo programu: spustenie nejakých troch metód globálneho objektu Application. Táto inštancia je zadeklarovaná v jednotke Forms a automaticky sa vytvára pri štarte programu (v inicializačnej časti tejto jednotky). Tento súbor **Project1.dpr** môže vyzerať približne takto:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  StackUnit in 'StackUnit.pas';

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

O tomto súbore treba vedieť, že je pre celú aplikáciu veľmi dôležitý (on to celé štartuje) a preto pri prenášaní zdrojového tvaru nášho programu do iného počítača nesmieme zabudnúť ani na tento súbor.

Dynamické polia

Doteraz sme sa stretli iba so statickými poľami: poľu sme v deklaráciách určili hranice jeho indexu (jeho veľkosť) a počas behu programu sme už nemali možnosť túto veľkosť zmeniť. Príklad zásobníka nám ukázal, že z daného poľa sme potrebovali vždy len nejaký počet prvkov a počas behu sa tento počet prvkov rôzne menil - použili sme na to premennú vrch, v ktorej sme si pamätali pozíciu posledne pridávaného prvku a teda veľkosť potrebnej časti poľa. Je jasné, že tak ako sme to naprogramovali, tento zásobník zvládne maximálne 100 pridávaných prvkov (operácia push) a potom program zahlási chybu. Môžeme zadeklarovať (a teda vyhradiť) hoci aj milión-prvkové ale toto nie je pekné riešenie. Preto sa naučíme nový spôsob definovania štruktúry pole, pri ktorom nebudeme pri deklarácii definovať hranice indexu ale pole najprv vytvoríme úplne prázdne (0-prvkové) a až počas behu ho budeme

pomocou špeciálnych príkazov zväčšovať alebo zmenšovať.

Takéto polia nazývame dynamické (dynamicky menia svoju veľkosť) a deklarujeme ich takto:

```
var
  a: array of typ_prvku;
```

Premenná a je dynamické pole, ktorá po zadeklarovaní zatiaľ nemá určenú veľkosť. Preto zatiaľ nemôžeme pracovať ani s prvkami tohto poľa. Najprv musíme nastaviť počet prvkov poľa pomocou príkazu (štandardnej procedúry)

SetLength(a, *počet_prvkov*);

Táto procedúra vyhradí pamäť pre požadovaný počet prvkov poľa. Štandardná funkcia **Length**(a) vráti tento nastavený počet prvkov. Prvky sú vždy indexované od 0 do počet-1. Funkcia **Low**(a) má teda vždy hodnotu 0 a **High**(a) sa vždy rovná **Length**(a)-1. Niekedy sa môžete stretnúť aj so zápisom

a := nil;

ktorý označuje priradenie nulového počtu prvkov, čo je to isté ako SetLength(a, 0). Uvedomte si, že keď dynamickému poľu zmenšíme počet jeho prvkov, tak sa hodnoty prvkov, o ktoré sa pole skrátilo, strácajú - pole je skracované od posledných prvkov. Keď dynamickému poľu pridáme nové prvky, tak tieto majú nazačiatku nedefinovanú hodnotu. Často sa do poľa pridáva (na koniec) jeden prvok a hneď sa do neho priradí nejaká hodnota, napr. takto

```
SetLength(a, Length(a)+1);
a[High(a)] := 123;
```

Skrátenie poľa o posledný prvok (zrejme pole musí byť neprázdne) urobíme napr. takto

```
SetLength(a, Length(a)-1);
```

Pre prácu s dynamickými poľami môžeme ešte použiť rovnakú štandardnú funkciu Copy ako pre prácu so znakovými reťazcami. Z ľubovoľného poľa môžeme vybrať súvislý úsek a priradiť ho do iného (ale aj toho istého) dynamického poľa:

Copy(*dynamické_pole*, *od_prvku*, *počet*);

Pracuje rovnako ako pri znakových reťazcoch. Len nezabudnite, že index prvku, od ktorého vyberáme časť poľa, je od 0. Napr.

```
var
  a, b: array of Integer;
 ...
   b := Copy(a, 2, 3);         // b[0]:=a[2]; b[1]:=a[3]; b[2]:=a[4];
   a := Copy(a, 1, MaxInt);    // vyhodí prvý prvok poľa
```

Triedu zásobník zrealizujeme pomocou dynamického poľa:

```
unit StackUnit;

interface

type
  Prvok = String;
  TStack = class
    st: array of Prvok;
    constructor Create;
    procedure push(const p: Prvok);
```

```
    procedure pop(var p: Prvok);
    function top: Prvok;
    function empty: Boolean;
  end;

implementation

uses
  Dialogs;

procedure chyba(const s: String);
begin
  ShowMessage(s);
  Halt;
end;

constructor TStack.Create;
begin
  st := nil;
end;

procedure TStack.push(const p: Prvok);
begin
  SetLength(st, Length(st)+1);
  st[High(st)] := p;
end;

procedure TStack.pop(var p: Prvok);
begin
  if empty then
    chyba('Prázdny zásobník pri príkaze pop');
  p := st[High(st)];
  SetLength(st, Length(st)-1);
end;

function TStack.top: Prvok;
begin
  if empty then chyba('Prázdny zásobník pri príkaze top');
  Result := st[High(st)];
end;

function TStack.empty: Boolean;
begin
  Result := Length(st)=0;   // alebo Result := st=nil;
end;

end.
```

Všimnite si, že už nepotrebujeme metódu full, keďže teraz už nemáme horný index poľa.

Ukážeme ešte jednu verziu triedy zásobník pomocou dynamického poľa. Nakoľko operácia "nafukovania" dynamického poľa je pomerne "drahá", t.j. časovo náročná, môžeme pole zväčšovať po väčších úsekoch ako 1, napr. po 100:

```
unit StackUnit;

interface

type
  TPrvok = String;
  TStack = class
    st: array of Prvok;
    vrch: Integer;
    constructor Create;
    procedure push(const p: Prvok);
```

```
    procedure pop(var p: Prvok);
    function top: Prvok;
    function empty: Boolean;
  end;

implementation

uses
  Dialogs;

procedure chyba(const s: String);
begin
  ShowMessage(s);
  Halt;
end;

constructor TStack.Create;
begin
  st := nil;
  vrch := -1;
end;

procedure TStack.push(const p: Prvok);
begin
  Inc(vrch);
  if vrch > High(st) then
    SetLength(st, Length(st)+100);
  st[vrch] := p;
end;

procedure TStack.pop(var p: Prvok);
begin
  if empty then
    chyba('Prázdny zásobník pri príkaze pop');
  p := st[vrch];
  Dec(vrch);
  if vrch < High(st)-150 then
    SetLength(st, Length(st)-100);
end;

function TStack.top: Prvok;
begin
  if empty then
    chyba('Prázdny zásobník pri príkaze top');
  Result := st[vrch];
end;

function TStack.empty: Boolean;
begin
  Result := vrch < 0;
end;

end.
```

V nasledujúcom príklade použijeme zásobník spolu s robotom. Budeme predpokladať, že prvky zásobníka nie sú reťazce ale reálne čísla, t.j. platí

```
type
  Prvok = Real;
```

Asi v tomto príklade rozpoznáte nakreslenie špirály, pričom sa zapamätávajú dĺžky kreslených čiar. Potom sa kreslí táto špirála v opačnom poradí čiar.

```
uses
  StackUnit, RobotUnit;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  z: TStack;
  r: TRobot;
  i: Integer;
  cislo: Real;          // Prvok
begin
  r := TRobot.Create;
  z := TStack.Create;
  for i := 1 to 20 do
  begin
    r.fd(i*10);
    r.rt(90);
    z.push(i*10);
    wait(50);
  end;
  wait(1000);
  r.PC := clRed;
  r.rt(1);
  while not z.empty do
  begin
    z.pop(cislo);
    r.lt(90);
    r.fd(-cislo);
    wait(50);
  end;
  z.Free;
end;
```

Parameter otvorené pole

Potrebujeme procedúru, ktorá dostane "postupnosť" čísel a na jej základe nakreslí krivku tak, že postupne z nej bude brať dĺžky pre príkaz fd a zakaždým sa otočí o 90 stupňov vpravo. Nakoľko nepoznáme dĺžku vstupnej postupnosti (t.j. poľa) použijeme tzv. parameter otvorené pole, t.j. skutočným parametrom môže byť ľubovoľné jednorozmerné pole, ktoré sa zhoduje s typom prvkov. Zapisujeme, napr.

```
procedure kresli(const d: array of Real);
```

Pozor, takýto formálny parameter nie je dynamické pole. Skutočným parametrom do tejto procedúry môže byť ľubovoľné jednorozmerné pole reálnych čísel (aj dynamické).

Vo vnútri procedúry pristupujeme k prvkom poľa s indexmi od Low(d) až po High(d), pričom Low(d) je **vždy 0** a High(d) je vždy Length(d)-1. Keďže to nie je dynamické pole, nesmieme použiť SetLength ani Copy.

Príklad:

```
var
  r: TRobot;

procedure kresli(d: array of Real);
var
  i: Integer;
begin
  for i := Low(d) to High(d) do        // Low(d) je vždy 0
  begin
    r.fd(d[i]);
    r.rt(90);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
const
```

```
  dd: array[5..10] of Real = (50, 50, 100, 100, 50, 50);
begin
  r := TRobot.Create;
  kresli(dd);
end;
```

Delphi umožňujú poslať ako skutočný parameter typu otvorené pole aj **konštantu otvorené pole**, t.j. "postupnosť" hodnôt rovnakého typu ako prvky otvoreného poľa uzavretú v hranatých zátvorkách, napr. takto

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  k := TRobot.Create;
  kresli([50, 50, 100, 100, 50, 50]);
end;
```

Ak by sme chceli zadeklarovať procedúru, ktorej parameter je dynamické pole, museli by sme to urobiť napr. takto:

```
type
  pole = array of Real;

procedure kresli(d: pole);
...
```

Do tejto procedúry môžeme poslať ako skutočný parameter len dynamické pole reálnych čísel.

Už sme sa stretli s grafickou procedúrou Polygon, ktorá funguje v plátne (Canvas) grafickej plochy. Má jeden parameter otvorené pole typu TPoint. Štandardná funkcia Point vyrába záznam typu TPoint, t.j. platí takéto niečo

```
type
  TPoint = record x, y: Integer end;

procedure TCanvas.Polygon(const Points: array of TPoint);
...

function Point(AX, AY: Integer): TPoint;
begin
  Result.x := AX;
  Result.y := AY;
end;
```

Túto procedúru môžeme zavolať veľmi jednoducho aj s parametrom konštanta otvorené pole, napr. takto

```
  with Image1.Canvas do
  begin
    Brush.Color := clBlue;
    Polygon([Point(100, 100), Point(200, 150), Point(50, 200)]);
  end;
```

15. prednáška: programová jednotka RobotUnit

čo už vieme:

- trieda TRobot je definovaná v programovej jednotke RobotUnit
- definuje niekoľko stavových premenných (X, Y, H, PC, ...) a niekoľko metód (Create, fd, rt, ...)

čo sa na tejto prednáške naučíme:

- ako je naozaj definovaná trieda TRobot - obsahuje súkromné stavové premenné a verejné stavové premenné typu "vlastnosť"
- pri niektorých metódach je pripojené rezervované slovo override, overload alebo virtual
- okrem konštruktora objektu môže existovať aj deštruktor

RobotUnit

Podrobne sa teraz pozrieme na programovú jednotku RobotUnit. Najprv, ako vyzerá samotná deklarácia triedy TRobot:

```
type
  TRobot = class
  private
    FX, FY, FH: Real;
    Fdown: Boolean;
    FPC: TColor;
    FPW: Integer;
  public
    value: Integer;
    myImage: TImage;
    constructor Create; overload;
    constructor Create(nx, ny: Real; a: Real = 0); overload;
    constructor Create(im: TImage; nx, ny: Real; a: Real = 0); overload;

    procedure fd(d: Real); virtual;
    procedure rt(a: Real); virtual;
    procedure lt(a: Real); virtual;
    procedure seth(a: Real); virtual;
    procedure setxy(nx, ny: Real); virtual;
    procedure setx(nx: Real); virtual;
    procedure sety(ny: Real); virtual;
    procedure movexy(nx, ny: Real); virtual;
    procedure pu; virtual;
    procedure pd; virtual;
    procedure setpen(ndown: Boolean); virtual;
    procedure setpc(color: TColor); virtual;
    procedure setpw(width: Integer); virtual;

    procedure point(r: Real = 0); virtual;
    procedure text(text: String); virtual;
    procedure towards(nx, ny: Real); virtual;
    procedure fill(color: TColor); virtual;
    function  dist(nx, ny: Real): Real; virtual;
    function  isnear(nx, ny: Real): Boolean; virtual;
    procedure draw; virtual;
    procedure action; virtual;

    property  X: Real read FX write setx;
    property  Y: Real read FY write sety;
    property  H: Real read FH write seth;
    property  ispd: Boolean read Fdown write setpen;
```

```
    property  PC: TColor read FPC write setpc;
    property  PW: Integer read FPW write setpw;
  end;
```

Väčšinu vecí buď už poznáme, alebo sme s nimi už pracovali predtým. Ale môžeme tu vidieť aj niekoľko noviniek:

- všetky stavové premenné a metódy sú rozdelené do dvoch veľkých skupín: private a public
- skoro za všetkými definíciami metód je kľúčové slovo virtual, ktorého význam sa budeme učiť neskôr
- konštruktor Create má predvolené parametre (default hodnoty)
- niektoré stavové premenné (napr. X, Y, H a pod.) sú uvedené kľúčovým slovom property - tzv. vlastnosť

V tejto prednáške sa s týmito novinkami zoznámime postupne.

privátne definície

Už vieme, že v programovej jednotke môžeme deklarácie uviesť nielen v interface časti, ale aj za implementation, a teda vytvoriť buď verejné definície alebo súkromné (len pre potreby unitu). Podobne je to pri definovaní triedy: definície stavových premenných a metód môžeme rozdeliť do dvoch skupín (neskôr uvidíme aj ďalšie) private a public: na súkromné, ktoré sú určené len pre potreby samotného objektu a zvonku sú neprístupné, a na verejné, ktoré zverejňujeme pre používateľov (programátorov) tejto triedy. To, že private stavové premenné a metódy sú zvonku neprístupné znamená, že ak nejakú triedu zadeklarujeme v inteface časti nejakého unitu, tak ostatné unity, ktoré túto deklaráciu vidia, môžu pracovať len s public deklaráciami. V implementation časti samotného unitu sú všetky deklarácie rovnocenné. Pri práci s formulárom (napr. v triede TForm1) môžeme vidieť, že niektoré prvky (stavové premenné a metódy) nie sú v ani jednej z týchto skupín - okrem týchto dvoch skupín je aj ďalšia: published, ktorá je skoro rovnaká ako public, ale zatiaľ pre jednoduchosť predpokladajme, že ju Delphi potrebujú pre komponenty a metódy, ktoré sú zviazané s nejakým formulárom.

predvolené parametre

Pri volaní procedúry (aj metódy) môžem uviesť menší počet skutočných parametrov ako je počet formálnych. Ak Delphi poznajú predvolené (náhradné) hodnoty týchto parametrov, tak nehlásia chybu, ale doplnia ich týmito hodnotami za nás - predvolenými hodnotami môžu byť len konštanty alebo konštantné výrazy (niečo, čo vie kompilátor vypočítať už počas kompilácie). Predvolené (default) parametre môžu byť len hodnotové alebo const parametre - nie var-parametre. Za predvoleným parametrom (v zozname formálnych parametrov) môžu nasledovať už len predvolené. Robota môžeme skonštruovať napr. takto:

```
r := TRobot.Create(0, 0, 0);
r := TRobot.Create(x, y);
```

Predvolený parameter je aj v definícii procedúry cs:

```
procedure cs(bgcolor: TColor = clWhite);
```

t.j. keď je volaná bez parametrov, farbou pozadia bgcolor bude biela, inak sa použije zadaný parameter.

viac variantov jednej procedúry

Je to spôsob, pomocou ktorého môžeme nazvať rôzne procedúry jedným menom, resp. keď chceme, aby mala jedna procedúra viac rôznych variantov. Aby pascal pri volaní takejto procedúry správne rozhodol, ktorý variant má použiť, musia sa tieto varianty líšiť buď počtom parametrom alebo typmi parametrov. V deklaráciách triedy - v deklarácii metódy - za hlavičku procedúry zapíšeme rezervované slovo **overload** (funguje to aj pre obyčajné podprogramy, ktoré nie sú metódami nejakej triedy). Toto slovo overload zapíšeme za všetky varianty tejto procedúry. Môžeme vidieť tri varianty konštruktora Create - varianty sa líšia počtom parametrov, ale aj dva

varianty globálnej procedúry cs.

vlastnosti (property)

Stavové premenné s kľúčovým slovom property sú špeciálne atribúty triedy (nie sú to pamäťové položky triedy ako v zázname). "Skutočným" stavovým premenným je vyhradené nejaké pamäťové miesto (môžeme zistiť jej hodnotu, resp. ju meniť) - to je analógia položkám záznamov. "Virtuálna" stavová premenná - vlastnosť musí mať priradené nejaké akcie na čítanie, resp. modifikovanie takejto virtuálnej stavovej premennej. Pomocou tohto mechanizmu môžeme kontrolovať prístup ku "skutočným" stavovým premenným alebo môžeme pre takéto stavové premenné vypočítať ich hodnotu, až keď budú požadované (čítané). Syntax je

```
property meno: typ read položka|metóda write položka|metóda;
```

položka|metóda je buď obyčajná položka (skutočná stavová premenná) objektu (hoci aj privátna) alebo nejaká (hoci aj privátna) metóda. Typ položky sa musí zhodovať s typom vlastnosti. Pre read - metóda musí byť bezparametrová funkcia rovnakého typu ako typ vlastnosti. Pre write - metóda musí byť jednoparametrová procedúra s hodnotovým alebo const parametrom rovnakého typu ako typ vlastnosti. Jedna z častí read alebo write môže chýbať => potom tento prístup nie je povolený (napr. nedá sa meniť hodnota stavovej premennej, teda dá sa iba čítať). Vždy, keď sa v programe bude čítať, resp. meniť hodnota takejto "virtuálnej" stavovej premennej - vlastnosti, tak sa buď prečíta hodnota, resp. priradí do položky, alebo sa zavolá príslušná metóda. Napr. robot má tieto vlastnosti:

```
property  X: Real read FX write setx;
property  Y: Real read FY write sety;
property  H: Real read FH write seth;
property  PC: TColor read FPC write setpc;
...
```

Všetky tieto "virtuálne" stavové premenné môžeme čítať (zistiť ich hodnotu) lebo reprezentujú "skutočné" (hoci privátne) stavové premenné. Môžeme im dokonca meniť ich hodnoty, ale treba si uvedomiť, že vtedy sa automaticky zavolá príslušná metóda na zmenu hodnoty. Napr.

```
r.X := r.X + 5;
```

v skutočnosti znamená:

```
r.setx(r.X + 5);
```

keď sa pozriete do kódu metódy setx, tak zistíte, že je to vlastne volanie:

```
r.setxy(r.X + 5, r.Y);
```

Na rovnakom princípe fungujú aj ostatné virtuálne stavové premenné, napr.

```
r.H := 2 * r.H;     // znamená r.seth(2*r.H);
r.PC := clRed;      // znamená r.setpc(clRed);
```

Virtuálna stavová premenná (property) nemusí reprezentovať nejakú skutočnú stavovú premennú, ale jej hodnota môže byť vypočítaná na základe nejakej funkcie, napr.

```
type
  TMojTyp = class
  private
    fprem: Real;
    poc, fyyy: Integer;
    function dajxxx: Integer;
    procedure priradxxx(novexxx: Integer);
    procedure priradyyy(noveyyy: Integer);
```

```
    function hodnota: Real;
  public
    property xxx: Integer read dajxxx write priradxxx;
    property yyy: Integer write priradyyy;
    property prem: Real read hodnota write fprem;
  end;

function TMojTyp.dajxxx: Integer;
begin
  Result := Random(100);
end;

procedure TMojTyp.priradxxx(novexxx: Integer);
begin

end;

procedure TMojTyp.priradyyy(noveyyy: Integer);
begin
  fyyy := noveyyy;
end;

function TMojTyp.hodnota: Real;
             // funkcia počíta počet prístupov ku stavovej premennej
begin
  Result := fprem;
  Inc(poc);
end;
```

V tomto príklade máme máme 3 virtuálne stavové premenné:

- hodnota premennej xxx sa generuje náhodne a hoci môžeme do nej aj priraďovať, toto nemá žiadny efekt;
- do premennej yyy môžeme len priraďovať - túto hodnotu už z nej nemôžeme spätne prečítať (mimo unitu, kde je definovaná trieda TMojTyp);
- s reálnou premennou prem môžeme normálne pracovať - môžeme do nej priraďovať a aj túto hodnotu z nej prečítať, ale v ďalšej súkromnej stavovej premennej poc sa eviduje počet, koľkokrát sme prečítali hodnotu tejto premennej.

Neskôr uvidíme aj ďalšie možnosti stavových premenných - vlastností.

konštruktory robotov

Trieda TRobot má zadefinované tri rôzne konštruktory a všetky majú rovnaké meno Create. Už vieme, že toto je možné vďaka tomu, že sme v deklaráciách uviedli špeciálne slovo overload - Delphi umožňujú definovať viac procedúr s rovnakým menom, ale tieto sa musia líšiť počtom alebo typmi parametrov, tak aby kompilátor vedel vždy pri volaní jednoznačne rozhodnúť, ktorú verziu má použiť. Pozrite sa, ako konštruktor Create inicializuje stavové premenné:

```
constructor TRobot.Create(im: TImage; nx, ny, a: Real);
begin
  myImage := init(im);
  FX := nx;
  FY := ny;
  FH := a;
  FPC := clBlack;
  FPW := 1;
  Fdown := True;
end;
```

Stavová premenná myImage slúži na zapamätanie grafickej plochy, v ktorej je robot definovaný. Funkcia init slúži

na vyhľadanie grafickej plochy, ale len ak im je nil. Inak sa použije toto im ako grafická plocha robota. Všimnite si ako druhý variant konštruktora bez parametrov volá svoj prvý variant s parametrami:

```
constructor TRobot.Create;
begin
  myImage := init(nil);
  Create(myImage, myImage.Width/2, myImage.Height/2);
end;
```

deštruktor Destroy

Pri definovaní triedy môžeme zadefinovať aj vlastný deštruktor. Ten má opačnú úlohu ako konštruktor: automaticky sa zavolá pri metóde Free, t.j. vtedy, keď objekt rušíme. Vtedy na záver môžeme vykonať nejaké upratovacie akcie (napr. uvoľniť bitmapu, zmazať plochu, zatvoriť otvorený súbor a pod.). Volanie metódy obj.Free nejakého objektu obj si môžeme zjednodušene predstaviť ako

```
if obj <> nil then obj.Destroy;
```

Ak by sme triede TRobot definovali deštruktor, vyzeral by takto:

```
destructor TRobot.Destroy;
begin
  ...
  inherited;
end;
```

Pričom inherited na záver metódy je ukážkou toho, že netreba zabudnúť spustiť "upratovacie" akcie aj pre triedu predka - v našom prípade je to zbytočné, lebo predok triedy TRobot má prázdny deštruktor (trieda TRobot je potomkom triedy TObject). V jednotke RobotUnit sú okrem triedy TRobot v interface časti definované aj nejaké konštanty, premenné a procedúry a preto sú viditeľné aj pre programy, ktoré použijú uses RobotUnit:

- cs - má dva varianty a jeden predvolený parameter - farbu pozadia grafickej plochy - bez parametra farby zmaže grafickú plochu na bielo,
- wait - pozdrží vykonávanie programu o zadaný počet milisekúnd - počas tohto čakania sa vykonáva Application.ProcessMessages,
- konštanty rad a deg slúžia na prepočty radiánov a stupňov,
- procedúra setpage slúži na explicitné zadefinovanie grafickej plochy, v ktorej sa budú vytvárať nové roboty.

Pozrime si inicializáciu grafickej plochy, ktorá sa spustí pri prvom volaní TRobot.Create - jej najdôležitejšou úlohou je nájsť v našej aplikácii nejakú grafickú plochu (TImage) a zapamätať si ju. Nemá zmysel sa snažiť do detailov pochopiť, ako to naozaj funguje:

```
function init(im: TImage): TImage;
var
  i, n: Integer;
  f: TForm;
begin
  if im <> nil then
  begin
    Result := im;
    if defimage = nil then
      setpage(im);
    Exit;
  end;

  if defimage <> nil then
  begin
    Result := defimage;
```

```
    Exit;
  end;

  f := Application.MainForm;
  if f = nil then
  begin
    n := Application.ComponentCount;
    i := 0;
    while (i < n) and not (Application.Components[i] is TForm) do
      Inc(i);
    if i = n then
    begin
      ShowMessage('vadná aplikácia - Robot nenašiel formulár');
      Halt;
    end;
    f := TForm(Application.Components[i]);
  end;
  n := f.ControlCount;
  i := 0;
  while (i < n) and not (f.Controls[i] is TImage) do
    Inc(i);
  if i >= n then
  begin
    ShowMessage('vadná aplikácia - Robot nenašiel grafickú plochu');
    Halt;
  end;
  setpage(TImage(f.Controls[i]));
  Result := defimage;
end;
```

Pomocná procedúra setpage nastaví grafickej ploche DoubleBuffered na True (aby plocha pri intenzívnejšom kreslení neblikala) a zapamätá si ju v pomocnej premennej defimage (default image):

```
procedure setpage(im: TImage);
begin
  if im.Owner is TForm then
    TForm(im.Owner).DoubleBuffered := True;
  defimage := im;
end;
```

ako využijeme to, čo sme videli v definícii TRobot

Všimli sme si, že trieda TRobot obsahuje tri rôzne konštruktory. Prvé dva, t.j. bez parametrov a potom s dvoma alebo troma parametrami sme už viackrát použili. Tretí variant konštruktora, kde prvý parameter je typu TImage, sme ešte nepoužívali. Konštruktory, ktoré nemajú prvý parameter grafickú plochu (TImage), sa snažia automaticky vo formulári nejakú grafickú plochu vyhľadať a tú potom danému robotovi priradia - robot sa narodí a bude kresliť v tejto grafickej ploche. Ak ako prvý parameter uvedieme grafickú plochu, tak táto sa stáva plochou pre robota - ak máme viac robotov, každý by mohol existovať v inej grafickej ploche. Vytvorme formulár s dvoma grafickými plochami (rozmerov viac ako 250x250).

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2: TRobot;
begin
  r1 := TRobot.Create(Image1, 50, 150);
  r2 := TRobot.Create(Image2, 100, 100, 30);
  r1.fd(100);
  r2.fd(100);
  r1.Free;
  r2.Free;
```

```
end;
```

Zatlačením tlačidla v každej ploche vytvoríme jedného robota, ktoré potom nakreslia úsečky dĺžky 100. Zo základného robota TRobot môžeme odvodiť vlastnú triedu napr. TMojRobot a tejto môžeme zadefinovať nejakú novú triedu:

```
type
  TMojRobot = class(TRobot)
    procedure poly(n: Integer; s, u: Real);
  end;

procedure TMojRobot.poly(n: Integer; s, u: Real);
begin
  while n > 0 do
  begin
    fd(s);
    rt(u);
    Dec(n);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2: TMojRobot;
begin
  r1 := TMojRobot.Create(Image1, 50, 150);
  r2 := TMojRobot.Create(Image2, 100, 100, 30);
  r1.poly(3, 100, 120);
  r2.poly(5, 100, 144);
  r1.Free;
  r2.Free;
end;
```

Každý robot teraz nakreslí iný mnohouholník. V ďalšom kroku pridáme vlastný konštruktor a prenesieme celú túto deklaráciu triedy do samostatnej programovej jednotky MojRobotUnit.pas:

```
unit MojRobotUnit;

interface

uses
  Graphics, ExtCtrls, RobotUnit;

type
  TMojRobot = class(TRobot)
    constructor Create(im: TImage; nx, ny: Real; a: Real = 0);
    procedure poly(n: Integer; s, u: Real);
  end;

implementation

{ TMojRobot }

constructor TMojRobot.Create(im: TImage; nx, ny, a: Real);
begin
  inherited;
  PW := 5;
  PC := clGreen;
end;

procedure TMojRobot.poly(n: Integer; s, u: Real);
begin
  while n > 0 do
```

```
  begin
    fd(s);
    rt(u);
    Dec(n);
  end;
end;

end.
```

a v programovej jednotke s formulárom (Unit1.pas) zostane iba:

```
...

implementation

{$R *.dfm}

uses
  MojRobotUnit;

procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2: TMojRobot;
begin
  r1 := TMojRobot.Create(Image1, 50, 150);
  r2 := TMojRobot.Create(Image2, 100, 100, 30);
  r1.poly(3, 100, 120);
  r2.PC := clRed;
  r2.poly(5, 100, 144);
  r1.Free;
  r2.Free;
end;
```

Všimnite si, že trojuholník v prvej ploche je zelený a hviezda v druhej je červená. Oba útvary majú veľkosť strany 100.

Do našej novej triedy TMojRobot pridáme aj stavovú premennú k, ktorá bude označovať koeficient zmenšenia/zväčšenia pri kreslení pomocou poly:

```
unit MojRobotUnit;

interface

uses
  Graphics, ExtCtrls, RobotUnit;

type
  TMojRobot = class(TRobot)
    k: Real;
    constructor Create(im: TImage; nx, ny: Real; a: Real = 0);
    procedure poly(n: Integer; s, u: Real);
  end;

implementation

{ TMojRobot }

constructor TMojRobot.Create(im: TImage; nx, ny, a: Real);
begin
  inherited;
  PW := 5;
  PC := clGreen;
  k := 1;
end;
```

```
procedure TMojRobot.poly(n: Integer; s, u: Real);
begin
  while n > 0 do
  begin
    fd(k * s);
    rt(u);
    Dec(n);
  end;
end;

end.
```

Ak nebudeme stavovú premennú meniť, budú sa kresliť rovnako veľké útvary. Ak ale nejakému robotovi zmeníme túto hodnotu, bude kresliť rôzne veľké útvary:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r1, r2: TMojRobot;
  i: Integer;
begin
  r1 := TMojRobot.Create(Image1, 50, 150);
  r2 := TMojRobot.Create(Image2, 100, 100, 30);
  for i := 10 downto 1 do
  begin
    r1.poly(3, 100, 120);
    r1.rt(20);
    r1.k := i / 10;
  end;
  r2.PC := clRed;
  r2.poly(5, 100, 144);
  r1.Free;
  r2.Free;
end;
```

Prvý robot nakreslí 10 zmenšujúcich sa trojuholníkov. Na záver do triedy pridáme metódu na rekurzívne kreslenie binárneho stromu:

```
...

type
  TMojRobot = class(TRobot)
    k: Real;
    constructor Create(im: TImage; nx, ny: Real; a: Real = 0);
    procedure poly(n: Integer; s, u: Real);
    procedure strom(n: Integer; s: Real);
  end;

implementation

...

procedure TMojRobot.strom(n: Integer; s: Real);
begin
  fd(s);
  if n > 0 then
  begin
    lt(45);
    strom(n-1, s*0.6);
    rt(90);
    strom(n-1, s*0.7);
    lt(45);
  end;
  fd(-s);
```

```
end;

end.
```

a strom nakreslíme napr. takto:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  r: TMojRobot;
begin
  r := TMojRobot.Create(Image1, 130, 250, 0);
  r.strom(5, 80);
end;
```

16. prednáška: použitie zásobníka, rad

čo už vieme:

- poznáme údajovú štruktúru zásobník, ktorá má dve základné operácie: push a pop - vloženie a výber hodnoty
- pri volaniach procedúr a funkcii počítač využíva mechanizmus zásobníka: pri každom volaní podprogramu sa do zásobníka zapamätá návratová adresa a tiež sa na ňom vytvoria všetky lokálne premenné (aj hodnotové formálne parametre)

čo sa na tejto prednáške naučíme:

- ukážeme, ako sa dá aj veľmi zložitá rekurzívna procedúra prepísať do nerekurzívneho tvaru - využijeme na to zásobník
- zásobník použijeme aj na spracovanie rôznych typov aritmetických výrazov - uvidíme, čo sú výrazy v **infix**-ovom, **prefix**-ovom a **postfix**-ovom tvare
- zadefinujeme ešte jednu údajovú štruktúru: **rad** (front) - podobá sa na zásobník s tým rozdielom, že hoci hodnoty vkladáme na koniec radu, vyberáme ich zo začiatku radu

prepis rekurzie na nerekurziu

Máme rekurzívnu metódu pre robota, ktorá kreslí špirálu:

```
type
  TMojRobot = class(TRobot)
    procedure spir(n: Integer; s: Real);
  end;

procedure TMojRobot.spir(n: Integer; s: Real);
begin
  if n = 0 then
    lt(177)
  else
  begin
    fd(s);
    lt(90);
    spir(n-1, s+1);
    rt(90);
    fd(-s);
```

```
  end;
end;
```

Na nerekurzívnu metódu využijeme trochu upravený zásobník. Najprv zadefinujeme triedu zásobník pre našu konkrétnu úlohu takto: prvok zásobníka bude záznam, ktorý obsahuje položky adr - "skoková adresa", n a s - zapamätávané hodnoty formálnych parametrov procedúry. Tiež upravíme metódy push a pop tak, aby sa nám so zásobníkom pracovalo pohodlnejšie: nebudeme pracovať naraz s celým prvkom zásobníka (na to by sme potrebovali nejakú premennú typu záznam, t.j. Prvok), ale push a pop budú pracovať priamo so zložkami tohto záznamu. Prispôsobená trieda TStack môže vyzerať takto:

```
type
  Prvok = record
    adr: Integer;     // adresa skoku
    n: Integer;       // zapamätaná hodnota formálneho parametra
    s: Real;          // zapamätaná hodnota formálneho parametra
  end;

  TStack = class
    st: array of Prvok;
    ...
    procedure push(aadr, nn: Integer; ss: Real);
    procedure pop(var aadr, nn: Integer; var ss: Real);
    ...
  end;

procedure TStack.push(aadr, nn: Integer; ss: Real);
begin
  SetLength(st, Length(st)+1);
  with st[High(st)] do
  begin
    adr := aadr;
    n := nn;
    s := ss;
  end;
end;

procedure TStack.pop(var aadr, nn: Integer; var ss: Real);
begin
  if empty then
    chyba('Prázdny zásobník');
  with st[High(st)] do
  begin
    aadr := adr;
    nn := n;
    ss := s;
  end;
  SetLength(st, Length(st)-1);
end;
```

Rekurzívnu procedúru najprv logicky rozdelíme na časti, z ktorých potom budeme skladať nerekurzívne riešenie:

- prvá časť začína začiatkom procedúry - sem sa "skáče" pri prvom zavolaní procedúry ale aj pri každom ďalšom rekurzívnom volaní; prvá časť končí prvým rekurzívnym volaním - začiatok prvej časti označíme "adresa 1"
- za prvým rekurzívnym volaním začína druhá - sem sa skáče po návrate z rekurzívneho volania, t.j. túto adresu si treba zapamätať pri rekurzívnom volaní, t.j. "adresa 2"

```
procedure TMojRobot.spir(n: Integer; s: Real);
begin
// 1:
  if n = 0 then
```

```
    lt(177)
  else
  begin
    fd(s);
    lt(90);
    spir(n-1, s+1);
// 2:
    rt(90);
    fd(-s);
  end;
end;
```

Rekurzívnu procedúru teraz prepíšeme na cyklus, v ktorom sa budú vykonávať buď prvá časť alebo druhá časť, pričom cyklus skončí a teda aj celá procedúra, keď už sa vykonalo príslušný počet krát návrat z rekurzie. Zásobník použijeme takto:

- každý záznam v zásobníku reprezentuje vykonanie jednej z dvoch častí nášho rozdeleného programu, pričom si budeme pamätať aj lokálne premenné (t.j. formálne parametre), ktoré tejto časti prislúchajú,
- na začiatku do zásobníka vložíme jedinú informáciu, že chceme začať vykonávať prvú časť (adr=1) a pričom hodnoty premenných n a s sú presne tie, ktoré prišli ako hodnoty formálnych parametrov - vykonáme push(1, n, s);
- potom prichádza cyklus, ktorý zo zásobníka vyberie vrchnú hodnotu (teda trojicu adr, n, s) a podľa adr skočí buď na prvú alebo druhú časť,
- zrejme, ak je zásobník prázdny, už sme vykonali všetko, čo bolo treba a procedúru treba skončiť (preto je cyklus while not z.empty do),
- najdôležitejšie je správne zrealizovať rekurzívne volanie: vtedy treba do zásobníka uložiť dve informácie - zapamätať si návratovú adresu (push(2, n, s);) a zabezpečiť, aby sa znovu začala vykonávať rekurzívna procedúra od začiatku, teda push(1, n-1, s+1);
- keďže zásobník je objektová premenná, treba ju na začiatku programu vytvoriť (z := TStack.Create) a na konci zase uvoľniť (z.Free).

Nerekurzívna verzia kreslenia špirály môže vyzerať napr. takto:

```
type
  TMojRobot = class(TRobot)
    procedure spir(n: Integer; s: Real);
    procedure spirN(n: Integer; s: Real);
  end;

procedure TMojRobot.spirN(n: Integer; s: Real);
var
  z: TStack;
  adr: Integer;             // adresa skoku do príslušnej časti
begin
  z := TStack.Create;
  z.push(1, n, s);
  while not z.empty do
  begin
    z.pop(adr, n, s);
    case adr of
      1:
        if n = 0 then
          lt(177)
        else
        begin
          fd(s);
          lt(90);
          z.push(2, n, s);        // návrat po rekurzívnom volaní
          z.push(1, n-1, s+1);    // rekurzívne volanie
        end;
```

```
      2:
        begin
          rt(90);
          fd(-s);
        end;
    end;
  end;
  z.Free;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  k: TMojRobot;
begin
  k := TMojRobot.Create;
  k.spirN(100, 1);
  k.Free;
end;
```

procedúra sa dá upraviť tak, aby sa nerobila operácia push, ak to nie je nevyhnutné:

```
procedure TMojRobot.spirN(n: Integer; s: Real);
var
  z: TStack;
  adr: Integer;
begin
  z := TStack.Create;
  adr := 1;
  repeat
    case adr of
      1:
        if n = 0 then
        begin
          lt(177);
          adr := 0;                    // bude treba robiť pop
        end
        else
        begin
          fd(s);
          lt(90);
          z.push(2, n, s);
          adr := 1;
          n := n-1;
          s := s+1;
        end;
      2:
        begin
          rt(90);
          fd(-s);
          adr := 0;                  // bude treba robiť pop
        end;
    end;
    if (adr = 0) and not z.empty then
      z.pop(adr, n, s);
  until adr = 0;
  z.Free;
end;
```

binárny strom

Ilustrujme tento postup aj rekurzívnu procedúru na kreslenie stromu. Najprv rekurzívna verzia:

```
procedure TMojRobot.strom(n: Integer; s: Real);
begin
```

```
// časť 1:
  if n = 0 then
  begin
    fd(s);
    fd(-s);
  end
  else
  begin
    fd(s);
    lt(45);
    strom(n-1, s*0.6);
// časť 2:
    rt(90);
    strom(n-1, s*0.7);
// časť 3:
    lt(45);
    fd(-s);
  end;
end;
```

Nerekurzívna verzia môže vyzerať takto:

```
procedure TMojRobot.stromN(n: Integer; s: Real);
var
  z: TStack;
  adr: Integer;
begin
  z := TStack.Create;
  z.push(1, n, s);
  while not z.empty do
  begin
    z.pop(adr, n, s);
    case adr of
      1:
        if n = 0 then
        begin
          fd(s);
          fd(-s);
        end
        else
        begin
          fd(s);
          lt(45);
          z.push(2, n, s);       // návrat bude na začiatok 2. časti
          z.push(1, n-1, s*0.6);
        end;
      2:
        begin
          rt(90);
          z.push(3, n, s);       // návrat bude na začiatok 3. časti
          z.push(1, n-1, s*0.7);
        end;
      3:
        begin
          lt(45);
          fd(-s);
        end;
    end;
  end;
  z.Free;
end;
```

rekurzívna funkcia

Fibonacciho postupnosť je definovaná rekurentne, napr. takto:

F(0)=0; F(1)=1; F(N)=F(N-1)+F(N-2); pre N > 1

A rekurzívna funkcia:

```
function fib(n: Integer): Integer;
begin
  if n < 2 then
    Result := n
  else
    Result := fib(n-1) + fib(n-2);
end;
```

Postup, pomocou ktorého vieme prepisovať rekurzívne procedúry na nerekurzívne, takto jednoducho nefunguje ani na funkcie ani na var-parametre. Preto najprv túto elegantnú rekurzívnu funkciu prepíšeme na nie veľmi peknú procedúru s jednou globálnou premennou, v ktorej budeme uchovávať výsledok funkcie:

```
var
  fibResult: Integer;    // globálna premenná - vypočítaná hodnota

procedure fib(n: Integer);
var
  f: Integer;           // pomocná premenná pre uchovanie medzivýpočtu
begin
//časť 1:
  if n < 2 then
    fibResult := n
  else
  begin
    fib(n-1);
//časť 2:
    f := fibResult;
    fib(n-2);
//časť 3:
    fibResult := fibResult + f;
  end;
end;
```

Nerekurzívny tvar predchádzajúcej procedúry (Prvok v TStack musí byť teraz typu záznam s tromi Integer):

```
procedure fibN(n: Integer);
var
  f, adr: Integer;
  z: TStack;
begin
  z := TStack.Create;
  z.push(1, n, f);
  while not z.empty do
  begin
    z.pop(adr, n, f);
    case adr of
      1:
        if n < 2 then
          fibResult := n
        else
        begin
          z.push(2, n, f);              // návrat
          z.push(1, n-1, f);            // rekurzívne volanie
        end;
      2:
        begin
```

```
          f := fibResult;
          z.push(3, n, f);              // návrat
          z.push(1, n-2, f);            // fib(n-2)
        end;
      3:
        fibResult := fibResult + f;
    end;
  end;
  z.Free;
end;
```

snehová vločka

Na záver napíšeme nerekurzívny variant procedúry na kreslenie snehovej vločky:

```
procedure TMojRobot.vlocka(n: Integer; s: Real);
begin
  if n <= 0 then
    fd(s)
  else
  begin
    vlocka(n-1, s/3);
    rt(60);
    vlocka(n-1, s/3);
    lt(120);
    vlocka(n-1, s/3);
    rt(60);
    vlocka(n-1, s/3);
  end;
end;
```

A jej nerekurzívna verzia:

```
procedure TMojRobot.vlockaN(n: Integer; s: Real);
var
  z: TStack;
  adr: Integer;
begin
  z := TStack.Create;
  z.push(1, n, s);
  while not z.empty do
  begin
    z.pop(adr, n, s);
    case adr of
      1:
        if n <= 0 then
          fd(s)
        else
        begin
          z.push(2, n, s);
          z.push(1, n-1, s/3);
        end;
      2:
        begin
          rt(60);
          z.push(3, n, s);
          z.push(1, n-1, s/3);
        end;
      3:
        begin
          lt(120);
          z.push(4, n, s);
          z.push(1, n-1, s/3);
        end;
```

```
      4:
        begin
          rt(60);
          // tu už nemusíme uložiť adresu návratu
          // po návrate sa totiž nemusí robiť nič
          z.push(1, n-1, s/3);
        end;
    end;
  end;
  z.Free;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  k: TMojRobot;
  i: Integer;
begin
  k := TMojRobot.Create;
  for i := 1 to 3 do
  begin
    k.vlockaN(4, 100);
    k.lt(120);
  end;
  k.Free;
end;
```

aritmetické výrazy

infix

Z matematiky a aj z programovania poznáme takýto spôsob zápisu aritmetických výrazov:

```
2 + 5
2 + 5 * 6
4 + (2 + 5) * 6
```

Znamienko operácie je **medzi** svojimi operandami. Niektoré operácie (napr. *, /) majú vyššiu prioritu ako iné (napr. +, -) a preto vieme, že 2 + 5 * 6 = 32 a nie 60 (ak by + malo vyššiu prioritu ako *). Poradie vyhodnocovania operácií vo výraze môžeme ešte zmeniť pomocou okrúhlych zátvoriek. Ak by sme chceli tieto výrazy zapísať trochu formálnejšie, mohlo by to vyzerať napr. takto:

výraz == *operand* ***operácia*** *operand*
operácia == +, -, *, /
operand == *číslo*, *výraz*, (*výraz*)

Takémuto zápisu, keď operácia je medzi operandami hovoríme **in**-fixový zápis. V niektorých (programátorských) aplikáciách sa využíva **pre**-fixový (tzv. poľský) zápis alebo **post**-fixový (tzv. prevrátený poľský) zápis.

prefix

Prefixový zápis sa líši od infixového umiestnením znaku operácie vzhľadom k operandom: znamienko nie je medzi ale pred operandami. Formálny zápis by mohol byť:

výraz == ***operácia*** *operand operand*
operácia == +, -, *, /
operand == *číslo*, *výraz*

Pre tento zápis nepotrebujeme rozlišovať prioritu operácií (všetky sú rovnocenné) a ani nepotrebujeme zátvorky.

postfix

Postfixový zápis je veľmi podobný prefixovému, len znak operácie je za operandami. Formálne to vyzerá takto:

výraz == *operand operand* ***operácia***
operácia == +, -, *, /
operand == *číslo*, *výraz*

Aj v tomto zápise nie sú ani priority operácií ani zátvorky. Ukážme zápis niekoľkých aritmetických výrazov v rôznych zápisoch:

infix	*prefix*	*postfix*
4 + 5 * 7	+ 4 * 5 7	4 5 7 * +
(4 + 5) * 7	* + 4 5 7	4 5 + 7 *
1 * 2 * 3 * 4 * 5	* * * * 1 2 3 4 5	1 2 3 4 5 * * * *
9 - 3 / 1 + 2	+ - 9 / 3 1 2	9 3 1 / - 2 +
(9 - 3) / (1 + 2)	/ - 9 3 + 1 2	9 3 - 1 2 + /

Oba tieto zápisy (prefix aj postfix) sú zaujímavé aj tým, že algoritmy, ktoré ich vyhodnocujú sú výrazne jednoduchšie ako pre infixový zápis. Ukážme vyhodnocovanie postfixového zápisu:

- postupne čítame výraz zľava doprava - na vstupe je buď operand alebo znak operácie,
- keď je na vstupe operand (pre jednoduchosť predpokladajme, že je to číslo), tak ho vložíme do zásobníka (teda **push**(číslo)),
- keď je na vstupe znak operácie, tak zrejme sme už predtým museli do zásobníka vložiť minimálne 2 čísla, tieto dve čísla zo zásobníka vyberieme, vykonáme operáciu (napr. +) a opäť vložíme do zásobníka (teda **push**(**pop** + **pop**)),
- po skončení vyhodnocovania celého výrazu na zásobníku ostane výsledná hodnota.

Program na vyhodnotenie postfixu, ktorý prečítame z editovacieho riadka Edit1:

```
uses
  StackUnit;     // type Prvok = Integer;

...

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  z: TStack;
  x, y, i: Integer;
  s: String;
begin
  if Key <> #13 then
    Exit;
  z := TStack.Create;
  s := Edit1.Text;
  Memo1.Lines.Append(s);
  Edit1.Text := '';
  i := 1;
  while i <= Length(s) do
  begin
    case s[i] of
      '0'..'9':
        begin
          x := 0;
          while s[i] in ['0'..'9'] do
          begin
            x := x*10+Ord(s[i])-Ord('0');
            Inc(i);
```

```
          end;
          z.push(x);
          Dec(i);
        end;
      '+':
        begin
          z.pop(x);
          z.pop(y);
          z.push(y + x);
        end;
      '-':-
        begin
          z.pop(x);
          z.pop(y);
          z.push(y - x);
        end;
      '*':
        begin
          z.pop(x);
          z.pop(y);
          z.push(y * x);
        end;
      '/':
        begin
          z.pop(x);
          z.pop(y);
          z.push(y div x);
        end;
    end;
    Inc(i)
  end;
  z.pop(x);
  z.Free;
  Memo1.Lines.Append('výsledok = ' + IntToStr(x));
end;
```

Dorobte program tak, aby vypísal chybovú správu v prípade, keď zadaný výraz nie je správny postfixový výraz.

Ďalšie námety:

- upravte program na postfixový zápis s reálnou aritmetikou
- navrhnite program, ktorý bude vyhodnocovať výrazy v prefixovom zápise
- navrhnite algoritmus na prepis infixového zápisu do postfixového, resp. prefixu (nie je to triviálne)
- úplne uzátvorkovaný výraz: je infix, v ktorom je každá operácia uzátvorkovaná, napr.
  `3*4+5*6-7*8 -> (((3*4)+(5*6))-(7*8))`
  vďaka tomu nepotrebujeme prioritu operátorov
  - urobte prevody z/do úplne uzátvorkovaného do/z pre/post-fixu
  - vyhodnoťte úplne uzátvorkovaný výraz
- navrhnite algoritmus na prepis prefixu do postfixu a naopak
- vyhodnocujte výrazy s premennými, príp. s unárnymi operáciami a funkciami (napr. unárne mínus, funkcia abs, sqr, sqrt a pod.)
- porozmýšľajte nad analógiou s logickými výrazmi a operáciami and, or, xor a not, resp. s relačnými operátromi =, <>, ´<=, ...

rad - queue, front, FIFO

Táto údajová štruktúra sa veľmi podobá zásobníku. Tiež má jednu operáciu na pridávanie do štruktúry a vyberanie zo štruktúry. Táto štruktúra sa ale podobá na ozajstný rad zákazníkov napr. v obchode: ten, kto príde do obchodu prvý, bude prvý obslúžený - teda FIFO: **f**irst **i**n **f**irst **o**ut.

Do radu pridávame nové prvky na koniec - hovoríme tomu rear, tail, chvost - operáciou **append** (po anglicky: pridaj na koniec). Z radu vyberáme prvý (a nie posledný) prvok - hovoríme tomu front, head, hlavička - operáciou **serve** (po anglicky: obslúž). Vytvoríme triedu TQueue. Veľmi sa podobá definícii zásobníka:

```
unit QueueUnit;

interface

type
  QPrvok = String;
  TQueue = class
    q: array of QPrvok;
    constructor Create;
    procedure serve(var p: QPrvok);
    procedure append(p: QPrvok);
    function empty: Boolean;
  end;

implementation

uses
  Dialogs;

procedure chyba(const s: String);
begin
  ShowMessage(s);
  halt;
end;

constructor TQueue.Create;
begin
  q := nil;
end;

procedure TQueue.serve(var p: QPrvok);
begin
  if empty then
    chyba('Rad je prázdny pre príkaz serve');
  p := q[0];
  q := copy(q, 1, maxint);      // vyhoď prvý prvok poľa
end;

procedure TQueue.append(p: QPrvok);
begin
  SetLength(q, Length(q)+1);
  q[High(q)] := p;
end;

function TQueue.empty: Boolean;
begin
  Result := q = nil;
end;

end.
```

Štruktúru rad ukážeme na programe, ktorý číta textový súbor iba raz a vytvára nový zdvojený súbor:

```
...
uses
  QueueUnit;

...

procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  q: TQueue;
  t1, t2: TextFile;
  t: QPrvok;          // QPrvok = String
begin
  q := TQueue.Create;
  AssignFile(t1, 'Unit1.pas');
  Reset(t1);
  AssignFile(t2, 'test.txt');
  Rewrite(t2);
  while not Eof(f) do
  begin
    Readln(f, t);
    q.append(t);
    Writeln(g, t);
  end;
  CloseFile(f);
  while not q.empty do
  begin
    q.serve(t);
    Writeln(g, t);
  end;
  CloseFile(g);
  q.Free;
end;
```

Ďalšie námety:

- prerobte triedu TQueue tak, aby sa nie pri každej operácii append alebo serve musela meniť veľkosť dynamického poľa q, ale aby sa podobne ako pri zásobníku pamätal aktuálny počet a zmena veľkosti poľa sa teda robila menej často
- reprezentujte rad v statickom poli veľkosti N:
    - tzv. **cyklické pole** - s poľom pracujeme tak, ako keby bol zacyklený: za posledným prvkom nasleduje prvý; pamätáme si index na prvý prvok radu (odtiaľ budeme odoberať) a a buď na posledný prvok (sem budeme pridávať) alebo počet prvkov radu;
    - treba správne rozpoznať situáciu, keď je rad plný (metóda full)
- viac zásobníkov a radov môžeme využiť v rôznych situáciách, napr. vzájomná simulácia (napr. jedného zásobníka pomocou jedného alebo viacerých radov a naopak), triedenie nejakých čísel pomocou dvoch zásobníkov (bez ďalšieho poľa), využitie zarážky vo fronte, ...

17. prednáška: polymorfizmus

čo už vieme:

- objektové programovanie pre nás znamená, že môžeme definovať nové triedy a tiež vytvárať inštancie týchto tried
- triedy majú dve vlastnosti: zapuzdrenie (v jednom celku sú stavové premenné a metódy) a dedičnosť

čo sa na tejto prednáške naučíme:

- bez ďalšej vlastnosti - polymorfizmus - by bola práca s objektmi veľmi obmedzená
- polymorfizmus je silný mechanizmus zdieľania metód
- vďaka kompatibilite inštancií môžeme pracovať s polymorfnými objektmi aj polymorfnými parametrami

trieda TKruh a TStvorec

Na 13. prednáške - úvod do objektového programovania - sme vytvorili projekt, v ktorom sme zadefinovali triedu TKruh:

```
type
  TKruh = class
  private
    x, y, r: Integer;
    f: TColor;
    vid: Boolean;
    g: TCanvas;
  public
    constructor Create(gg: TCanvas; xx, yy, rr: Integer);
    procedure ukaz;
    procedure skry;
    procedure zmenFarbu(ff: TColor);
    procedure posun(dx, dy: Integer);
  end;

constructor TKruh.Create(gg: TCanvas; xx, yy, rr: Integer);
begin
  g := gg;    // grafická plocha
  x := xx;
  y := yy;
  r := rr;
  vid := False;
  f := clBlack;
end;

procedure TKruh.ukaz;
begin
  g.Pen.Color := f;
  g.Brush.Style := bsClear;
  g.Ellipse(x-r, y-r, x+r, y+r);
  vid := True;
end;

procedure TKruh.skry;
begin
  if vid then
  begin
    g.Pen.Color := clWhite;
    g.Brush.Style := bsClear;
    g.Ellipse(x-r, y-r, x+r, y+r);
    vid := False;
  end;
end;

procedure TKruh.zmenFarbu(ff: TColor);
begin
  f := ff;
  if vid then
    ukaz;
end;

procedure TKruh.posun(dx, dy: Integer);
var
  staryvid: Boolean;
begin
  staryvid := vid;
  skry;
  Inc(x, dx);
  Inc(y, dy);
  if staryvid then
    ukaz;
```

```
end;
```

Všimnite si, že grafickú plochu (presnejšie plátno grafickej plochy, t.j. TCanvas) posielame objektu pri jeho vytváraní, objekt si ju zapamätá vo svojej súkromnej stavovej premennej a ďalej do plochy kreslí prostredníctvom tejto premennej.

Do formulára sme na tri tlačidlá priradili vytvorenie troch inštancií triedy TKruh. Štvrté tlačidlo tieto kruhy pohybovalo (štartovalo a zastavovalo časovač) a piate tlačidlo rušilo všetky vytvorené inštancie TKruh. Keďže na objekty sa odvolávame v rôznych procedúrach (Button1Click, Button2Click, ...), pamätáme si ich v globálnych premenných a, b, c (tieto sú automaticky inicializované na nil):

```
var
  a, b, c: TKruh;

procedure TForm1.Button1Click(Sender: TObject);
begin
  a := TKruh.Create(Image1.Canvas, 100, 100, 50);
  with a do
  begin
    zmenFarbu(clGreen);
    ukaz;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  b := TKruh.Create(Image1.Canvas, 300, 200, 100);
  b.zmenFarbu(clRed);
  b.ukaz;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  c := TKruh.Create(Image1.Canvas, 200, 250, 70);
  c.ukaz;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  Timer1.Enabled := not Timer1.Enabled;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if a <> nil then
    a.posun(3, 2);
  if b <> nil then
    b.posun(-5, -1);
  if c <> nil then
    c.posun(2, -3);
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  if a <> nil then
    with a do
    begin
      skry;
      Free;
      a := nil;
    end;
  if b <> nil then
  begin
    b.skry;
```

```
    b.Free;
    b := nil;
  end;
  if c <> nil then
  begin
    c.skry;
    c.Free;
    c := nil;
  end;
end;
```

Keďže v objektovom programovaní sme sa už zoznámili aj s dedičnosťou, dodefinujeme novú triedu TStvorec ako podtriedu TKruh. Všimnite si, že v tejto prvej verzii triedy TStvorec sme sa spoľahli len na zdedenie stavových premenných a konštruktora a všetky metódy sme zadefinovali znovu (skopírovali sme ich definície):

```
type
  TStvorec = class(TKruh)
  public
    procedure ukaz;
    procedure skry;
    procedure zmenFarbu(ff: TColor);
    procedure posun(dx, dy: Integer);
  end;

procedure TStvorec.ukaz;
begin
  g.Pen.Color := f;
  g.Brush.Style := bsClear;
  g.Rectangle(x-r, y-r, x+r, y+r);
  vid := True;
end;

procedure TStvorec.skry;
begin
  if vid then
  begin
    g.Pen.Color := clWhite;
    g.Brush.Style := bsClear;
    g.Rectangle(x-r, y-r, x+r, y+r);
    vid := False;
  end;
end;

procedure TStvorec.zmenFarbu(ff: TColor);
begin
  f := ff;
  if vid then
    ukaz;
end;

procedure TStvorec.posun(dx, dy: Integer);
var
  staryvid: Boolean;
begin
  staryvid := vid;
  skry;
  Inc(x, dx);
  Inc(y, dy);
  if staryvid then
    ukaz;
end;
```

Jednu z premenných - inštancií kruhu prerobíme teraz na štvorec:

```
var
  a: TKruh;
  b: TStvorec;
  c: TKruh;
...

procedure TForm1.Button2Click(Sender: TObject);
begin
  b := TStvorec.Create(Image1.Canvas, 300, 200, 60);
  b.zmenFarbu(clRed);
  b.ukaz;
end;
```

Takto pozmenený projekt funguje správne: druhé tlačidlo vytvorí štvorec a štvrté tlačidlo rozhýbe všetky útvary, teda hýbe sa aj štvorec.

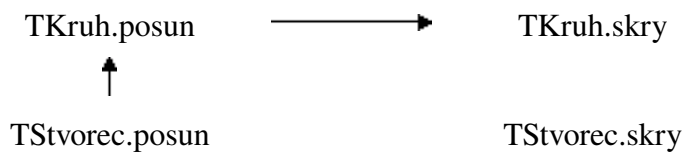
Ak porovnáme metódy zmenFarbu a posun v oboch triedach, zistíme, že sú úplne identické, preto sa ich pokúsime v triede TStvorec vyhodiť. Pre objektový pascal to znamená, že ich definície sa zdedia z triedy TKruh. Lenže dedenie neznamená, že sa v novej triede automaticky vytvorí ich kópia, ale volanie TStvorec.posun spôsobí volanie TKruh.posun. Po spustení programu zistíme, že projekt teraz prestal správne fungovať: zatlačenie druhého tlačidla správne vytvorí červený štvorec, ale hýbanie tohto štvorca v časovači pomocou metódy posun už spraví nezmysel. Štvorec by sa mal posúvať tak, že sa najprv zmaže (prekreslí bielou farbou), potom sa zmenia súradnice a znovu sa nakreslí, teraz už červenou farbou. Lenže štvorec sa teraz maže prekreslením bielej kružnice a po zmene súradníc sa nakreslí ako červená kružnica.

Aby sme pochopili, čo sa udialo a ako tento problém správne vyriešiť, musíme vysvetliť mechanizmus statických a virtuálnych metód.

statické metódy

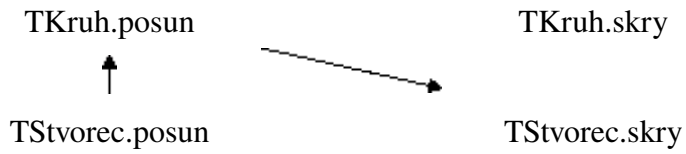
To, čo sa nám teraz stalo, vzniklo z toho dôvodu, že všetky doterajšie metódy sú volané mechanizmom pre statické metódy. Ak zavoláme TStvorec.posun, (t.j. inštancii triedy TStvorec zavoláme metódu posun), vďaka dedičnosti sa zavolá metóda TKruh.posun. Táto v sebe obsahuje dve volania: TKruh.skry a TKruh.ukaz. A toto je ten problém: v prípade že metódu TKruh.posun zavolá štvorec a nie kruh, potrebujeme aby sa nevolali metódy TKruh.skry a TKruh.ukaz, ale TStvorec.skry a TStvorec.ukaz.

Schematicky to môžeme zapísať takto: volanie TStvorec.posun sa prepne na TKruh.posun a toto zavolá TKruh.skry, hoci my by sme očakávali volanie TStvorec.skry:



virtuálne metódy

Objektové programovanie poskytuje nový mechanizmus volania metód, tzv. mechanizmus virtuálnych metód. Tento mechanizmus zabezpečí, že niektoré metódy budú mať významné postavenie a kompilátor pascalu sa pri nich bude správne rozhodovať, ktorú verziu metódy bude v skutočnosti volať. Napr. ak nejaký kruh zavolá svoju metódu posun, tak táto bude volať TKruh.skry, pričom ak túto istú metódu zavolá (vďaka dedičnosti) nejaký štvorec, tak počas nej sa už bude volať TStvorec.skry. Tou špeciálnou metódou je v tomto prípade metóda skry: každý potomok TKruh ju môže mať vo svojej verzii a mechanizmus bude podľa konkrétnej inštancie volať vždy tú správnu verziu. Takýmto špeciálnym metódam hovoríme **virtuálne** metódy, napr. v našom príklade musia byť virtuálnymi metódy skry a ukaz. Schematicky to zakreslíme takto:



Toto rozpoznanie správnej verzie metódy sa zrejme nedá urobiť počas kompilácie - trieda TKruh sa predsa môže nachádzať v samostatnej programovej jednotke a v čase, keď sa bude prekladať, kompilátor ešte nevie o ďalších potenciálnych potomkoch (odvodených triedach).

polymorfizmus

Polymorfizmus teda znamená, že metóda sa bude zdieľať v rôznych stupňoch objektovej hierarchie. Napr. vtedy, ak rôzne triedy zdedia "spoločnú" metódu (napr. metódu posun), ale detaily pre rôzne objekty zodpovedajú ich zvláštnostiam (napr. metóda skry).

včasná a neskorá väzba

Pri preklade projektu sa pri každom volaní podprogramu (metódy) musí kompilátor vedieť rozhodnúť, aký kód bude generovať. Pri obyčajných metódach (statických), vie určiť presnú adresu podprogramu, hovoríme okamžité rozhodnutie.

Pri preklade volaní virtuálnych metód kompilátor zatiaľ nevie, ktorú metódu (ktorého objektu) tam bude treba naozaj volať. Preto tam vygeneruje taký kód, ktorý sa až počas samotného volania "opýta inštancie" (premennej typu objekt), kde sa nachádza jeho verzia tejto virtuálnej metódy. Hovoríme, že pri virtuálnych metódach je odložené rozhodnutie.

Pri statických metódach kompilátor hľadá zodpovedajúce meno metódy v hierarchii smerom k predkom od danej úrovne (kde je definovaná metóda), t.j. ak metóda nebola definovaná na danej úrovni, hľadá sa najbližší predok, ktorý ju má zadefinovanú. Potom sa vygeneruje adresa tejto nájdenej metódy - preto tomu hovoríme **včasná väzba** (early binding).

Pri virtuálnych metódach kompilátor nehľadá nič, ale až počas behu sa hľadá zodpovedajúce meno v hierarchii smerom k predkom. T.j. pri volaní virtuálnej metódy sa v skutočnosti zavolá tá metóda, ktorá prislúcha inštancii a nie triede, kde je toto volanie (ak v TKruh.posun je volanie skry, nehľadá sa táto metóda v TKruh, ale pozrie sa, kto túto TKruh.posun volal a u neho sa hľadá skry). Tomuto mechanizmu hovoríme **neskorá väzba** (late binding).

virtuálne metódy

Metódu označíme, že je virtuálna tak, že v deklarácii triedy za deklaráciu metódy (procedure alebo function) zapíšeme slovo **virtual**. Potom, ak je v nejakej triede definovaná virtuálna metóda, tak aj všetci potomkovia by ju mali mať ako virtuálnu. Buď ju zdedia (znovu nedefinujú) alebo ju chcú prekryť svojou vlastnou verziou a vtedy zapíšu namiesto **virtual** slovo **override**. Samozrejme, že musia mať rovnako definované formálne parametre. Ak by sme virtuálnu metódu prekryli statickou, t.j. použili by sme rovnaké meno ale bez slova **override**, táto definícia sa bude chápať ako "predefinovanie" virtuálnej na statickú a mechanizmus polymorfizmu tu fungovať nebude.

Ako to celé funguje: každá trieda má vytvorenú svoju vlastnú tabuľku všetkých virtuálnych metód, tzv. **VMT** (**Virtual Method Table**) - v tejto tabuľke sú adresy všetkých virtuálnych metód, napr. pre TKruh aj pre TStvorec sú tam adresy vlastných verzií metód skry a ukaz. Keď sa vytvára nová inštancia pomocou konštruktora, tak vtedy sa na túto inštanciu napojí príslušná tabuľka - napr. inštancia typu TKruh dostane referenciu na **VMT** triedy TKruh, každá inštancia typu TStvorec takto dostane referenciu na **VMT** pre TStvorec. Táto referencia (adresa) na **VMT** je vždy prvá položka v zázname, teda všetky stavové premenné objektu nasledujú až za ňou. Hovoríme, že konštruktor môže byť použitý aj na inicializovanie stavových premenných, ale jeho hlavná funkcia je vyhradiť

pamäť pre objekt (stavové premenné) a inicializovať referenciu na **VMT**.

konštruktory a deštruktory

Vidíme, že konštruktory (a teda aj deštruktory) objektu majú špeciálne postavenie. Keďže sú zadefinované v triede TObject, ktorá je automatickým predkom všetkých tried (teda aj pre TKruh), ak niekde tieto definície neprekryjeme novými, tak zrejme zdedíme tieto pôvodné. Konštruktor Create zvyčajne nie je virtuálny a preto k nemu nepíšeme ani virtual ani override - už sme videli, že pre Create môžeme zvoliť rôzne formálne parametre a toto by nebolo možné, keby to bola virtuálna metóda (tá by musela mať presne tak isto definované parametre ako u predka). Tiež už vieme, že konštruktorov môžeme mať definovaných aj viac s rôznymi menami - nemusíme použiť slovo Create.

Na rozdiel od toho, deštruktor Destroy je virtuálna metóda bez parametrov a preto, ak ju definujeme, musíme uviesť slovo override. Deštruktor slúži na to, aby uvoľnil pamäť, ktorú zaberala daná inštancia, t.j. všetky stavové premenné. Objektová premenná je po zrušení pomocou deštruktora, napr. objekt.Destroy alebo objekt.Free, ďalej nepoužiteľná - nemali by sme volať jej žiadne metódy a ani pracovať s jej stavovými premennými. Väčšinou to spôsobí spadnutie programu.

kompatibilita tried (objektových typov)

Do objektovej premennej môžeme priradiť aj inštanciu inej triedy, ako je naozaj zadeklarovaná. Pritom ale musí byť splnená podmienka kompatibility: odvodený typ dedí kompatibilitu so svojimi predkami, t.j. kompatibilita je s predkami a nie potomkami. Hovoríme, že kompatibilita je jednosmerná. T.j. do objektovej premennej môžeme priradiť nielen inštanciu tej istej triedy ale aj ľubovoľnej odvodenej triedy. Vo všeobecnosti platí, že každá objektová premenná môže obsahovať nielen premennú zadeklarovaného typu, ale aj ľubovoľný objekt, ktorý je potomkom triedy tejto triedy. Táto kompatibilita je nielen medzi inštanciami, ale aj medzi formálnymi a skutočnými parametrami.

Zapamätajte si, že do objektovej premennej sa nedá priradiť inštancia triedy, ktorá je našim predkom.

príklady:

máme deklarácie:

```
var
  a, c: TKruh;
  b: TStvorec;
```

potom je povolené:

```
    a := TKruh.Create(...);
    b := TStvorec.Create(...);
    c := b;      // c referencuje tú istú inštanciu ako b
    c := TStvorec.Create(...);

procedure zmen(k: TKruh);
begin
  k.zmenFarbu(clBlue);
  k.posun(1, 1);
end;
```

a tiež môže byť volané:

```
  zmen(b);
```

Takej situácii, keď napr. formálny parameter môže byť nielen typu TKruh (ako je zadeklarovaný), ale aj ľubovoľného potomka TKruh, hovoríme, že formálny parameter k je polymorfný objekt (t.j. rôznorodý). Všimnite si, že potomok môže kdekoľvek nahradiť svojho predka (predok môže zastupovať potomka).

polymorfný objekt

Vidíme, že ako parameter procedúry môže prísť ľubovoľný potomok tohto objektu (zatiaľ nevieme zistiť ktorý). Toto umožňuje spracovať objekty, ktorých typ nie je známy. Tiež môžeme využiť metódy predkov so špecifickými zmenami - samozrejme, že to musíme riadne premyslieť. Ak dobre navrhneme základnú triedu, od ktorej budeme ďalej tvoriť celú hierarchiu tried, umožni nám to začať rozmýšľať nad algoritmami úplne inak - hovoríme tomu objektovo orientované programovanie (OOP).

Preto je dôležitý spôsob (metóda) návrhu hierarchie objektov. Najprv vytypujeme spoločné vlastnosti a metódy a z toho navrhneme bázovú triedu (typ). Rozhodneme, ktoré metódy budú virtuálne (virtual), t.j. budúcemu používateľovi umožníme rozširovanie metód. Môžeme vyrobiť aj "prázdne" virtuálne metódy - pre základnú triedu ešte nevieme zadefinovať funkčnosť, ale myslíme perspektívne rozširovanie odvodených tried.

Ak teda dobre zadefinujeme základné metódy a na ne naviazané virtuálne metódy (ako posun, ktorý využíva skry a ukaz), dovolí nám to potom veľmi elegantné vytváranie nových tried s relatívne malým programovaním (nemusíme definovať zložité metódy posun a zmenFarbu, stačí len nové skry a ukaz). Hovoríme tomu rozšíriteľnosť (extensibility). Uvedomte si, že teraz vlastne meníme funkčnosť dávnejšie naprogramovaných metód (posun a zmenFarbu) bez toho, aby sme museli mať k dispozícii pôvodné zdrojové texty. Ak vieme ako, môžeme takto zmeniť aj funkčnosť niektorých podprogramov, ktoré sú v štandardných knižniciach Delphi, bez toho aby sme zasahovali do ich kódu.

Príklad s polymorfným štvorcom by mal vyzerať takto:

```
type
  TKruh = class
  ...
    procedure ukaz; virtual;
    procedure skry; virtual;
  ...
  end;

  TStvorec = class(TKruh)
  public
    procedure ukaz; override;
    procedure skry; override;
  end;
```

použijeme:

```
var
  a: TKruh;
  b: TKruh;
  c: TKruh;

...

procedure TForm1.Button2Click(Sender: TObject);
begin
  b := TStvorec.Create(Image1.Canvas, 300, 200, 60);
  b.zmenFarbu(clRed);
  b.ukaz;
end;
```

Hoci sme objektovú premennú b deklarovali ako TKruh, priradili sme do nej inštanciu triedy TStvorec. Táto

inštancia o sebe "vie", že nie je kruh, ale štvorec a preto všetky metódy, ktoré volá, napr. zmenFarbu, vedia že ich volal štvorec a nie kruh.

príklad na polymorfné pole

Týmto mechanizmom môžeme zadefinovať aj pole objektov triedy TKruh, ktoré bude v skutočnosti obsahovať nielen kruhy, ale aj štvorce a trojuholníky. Takémuto poľu, o prvkoch ktorého nevieme akého sú naozaj typu, hovoríme polymorfné pole.

Najprv zadefinujeme novú triedu TTroj pre objekt trojuholník, tiež to bude potomok TKruh:

```
type
  TTroj = class(TKruh)
  public
    procedure ukaz; override;
    procedure skry; override;
  end;

procedure TTroj.ukaz;
begin
  g.Pen.Color := f;
  g.Brush.Style := bsClear;
  g.PolyLine(
    [Point(x-r, y), Point(x+r, y), Point(x, y-r), Point(x-r, y)]);
  vid := True;
end;

procedure TTroj.skry;
begin
  g.Pen.Color := clWhite;
  g.Brush.Style := bsClear;
  g.PolyLine(
    [Point(x-r, y), Point(x+r, y), Point(x, y-r), Point(x-r, y)]);
  vid := False;
end;
```

Všimnite si, že sme menili len ukaz a skry a pritom sa spoliehame na to, že bude fungovať aj posun a zmenFarbu. Pole p bude polymorfné pole:

```
const
  n = 20;

var
  p: array[1..n] of TKruh;

procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to n do
    p[i] := nil;
  Randomize;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i, x, y, r: Integer;
begin
  for i := 1 to n do
  begin
    x := Random(Image1.Width);
    y := Random(Image1.Height);
    r := Random(80)+10;
    case Random(3) of
```

```
      0: p[i] := TKruh.Create(Image1.Canvas, x, y, r);
      1: p[i] := TStvorec.Create(Image1.Canvas, x, y, r);
      2: p[i] := TTroj.Create(Image1.Canvas, x, y, r);
    end;
  end;
  for i := 1 to n do
    p[i].ukaz;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  poc := 100;
  Timer1.Enabled := True;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to n do
    if p[i] <> nil then
      with p[i] do
      begin
        if vid then
          skry;
        Free;
        p[i] := nil;
      end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: Integer;
begin
  Timer1.Enabled := False;
  for i := 1 to n do
    if p[i] <> nil then
      p[i].posun(i mod 5-2, i mod 3-1);
  Dec(poc);
  if poc > 0 then
    Timer1.Enabled := True;
end;

procedure TForm1.FormDestroy(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to n do
    p[i].Free;
end;
```

Poznámky:

- zatiaľ o prvkoch polymorfného poľa p nevieme zistiť, akého sú naozaj typu
- vieme, že buď sú to inštancie triedy kruh alebo jeho potomkovia
- každý prvok "sám seba vie" správne ukázať, skryť, zafarbiť a aj posunúť

rozpracovaná 18. prednáška: polymorfné roboty

čo už vieme:

- ako funguje polymorfizmus, čo sú to virtuálne metódy

čo sa na tejto prednáške naučíme:

- ako funguje polymorfizmus pre roboty, ako ho môžeme využiť, ako pracujeme s polymorfným poľom robotov (trieda TRobotGroup)
- indexované vlastnosti (property)
- pretypovanie, resp. zisťovanie, či je daná inštancia nejakého typu

Virtuálne metódy triedy TRobot

Definícia triedy TRobot má skoro všetky metódy virtuálne. To znamená, že budeme môcť v našich programoch využívať polymorfizmus. Okrem toho niektoré metódy triedy TRobot využívajú iné metódy robota a teda ich predefinovanie má za následok zmenu správania aj týchto metód. Napr. ak opravíme setxy, potom to bude mať vplyv aj na metódu fd.

malý príklad:

```
type
  TRobot1 = class(TRobot)
    procedure setxy(nx, ny: Real); override;
  end;

  TRobot2 = class(TRobot)
    procedure setxy(nx, ny: Real); override;
  end;

procedure TRobot1.setxy(nx, ny: Real);
begin
  inherited setxy(X, ny);
end;

procedure TRobot2.setxy(nx, ny: Real);
begin
  inherited setxy(nx, Y);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  r, r1, r2: TRobot;
  u: Integer;
begin
  r := TRobot.Create;
  r.PW := 5;
  r.PC := clRed;
  r1 := TRobot1.Create;
  r1.PW := 5;
  r1.PC := clBlue;
  r2 := TRobot2.Create;
  r2.PW := 5;
  r2.PC := clGreen;
  repeat
    // cs;
    r.fd(5);
```

```
    r1.fd(5);
    r2.fd(5);
    wait(10);
    u := Random(10);
    r.rt(u);
    r1.rt(u);
    r2.rt(u);
  until False;
end;
```

v tomto prvom príklade sa 3 roboty pohybujú úplne rovnakým spôsobom - ak by boli všetky tri rovnakej tiedy TRobot, tak by sme videli kresbu len jedného z nich (poslednej - zelenej)

r1 a r2 majú zmenené správanie tak, že r1 mení iba Y-ovú súradnicu a r2 mení len X-ovú

ak odkomentujete príkaz cs na priebežné zmazávanie plochy, tak roboty nebudú kresliť čiary, ale budú sa zobrazovať krátkymi "paličkami"

v nasledujúcom príklade zatiaľ nevyužívame žiadne predefinovanie metód - pomocou dvoch robotov kreslíme myšou do plochy

"obyčajné" kreslenie myšou:

```
var
  r1, r2: TRobot;

procedure TForm1.FormCreate(Sender: TObject);
begin
  r1 := TRobot.Create;
  r1.pu;
  r1.PC := clLtGray;
  r1.PW := 7;
  r2 := TRobot.Create;
  r2.pu;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  r1.pu;
  r1.setxy(X, Y);
  r1.pd;
  r2.pu;
  r2.setxy(X, Y);
  r2.pd;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var
 v: Real;
begin
  if Shift = [ssLeft] then
  begin
    v := r1.dist(X, Y);
    r1.towards(X, Y);
    r1.fd(v);
    r2.H := r1.H;
    r2.fd(v);
  end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  r1.Free;
```

```
  r2.Free;
end;
```

dodefinujeme novú triedu, ktorej poopravíme metódu seth, aby nastavovala uhol na opačný:

trieda TMojRobot má zmenený seth:

```
type
  TMojRobot = class(TRobot)
    procedure seth(uhol: Real); override;
  end;

procedure TMojRobot.seth(uhol: Real);
begin
  inherited seth(-uhol);
end;

...

procedure TForm1.FormCreate(Sender: TObject);
begin
  r1 := TRobot.Create;
  r1.pu;
  r1.PC := clLtGray;
  r1.PW := 7;
  r2 := TMojRobot.Create;
  r2.pu;
end;
```

teraz sa bude 2. robot správať veľmi čudne

zaujímavý efekt vznikne aj vtedy, keď v Image1MouseDown vyhodíme príkaz r1.pd;

v ďalšej sérii príkladov demonštrujeme rôzne pozmenené fd

najprv základná verzia bez pozmeneného robota - na kliknutie do plochy sa na tom mieste nakreslí domček:

domčeky:

```
procedure domcek(r: TRobot; d: Real);
var
  i: Integer;
begin
  for i := 1 to 4 do
  begin
    r.rt(90);
    r.fd(d);
  end;
  r.rt(30);
  r.fd(d);
  r.rt(120);
  r.fd(d);
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  r: TRobot;
begin
  Randomize;
  r := TRobot.Create(X, Y, 90);
  r.PC := Random(256*256*256);
  domcek(r, 70);
  r.Free;
```

```
end;
```

ak robot nedokáže robiť úsečky presnej dĺžky, ale s nejakou pravdepodobnosťou sa "mýli" o -10% ..10% , môžu vznikať zaujímavé kresby

nepresná dĺžka:

```
type
  TRobot1 = class(TRobot)
    procedure fd(dlzka: Real); override;
  end;

procedure TRobot1.fd(dlzka: Real);
begin
  inherited fd(dlzka*(0.9+Random(20)/100));
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  r: TRobot;
begin
  Randomize;
  r := TRobot1.Create(X, Y);
  r.PC := Random(256*256*256);
  domcek(r, 70);
  r.Free;
end;
```

trieda TRobot2 má metódu fd v poriadku (je odvodená z TRobot), ale otáčanie rt nechodí presne, ale robot sa mýli o -10: ..10: - otestujte to na príklade s domčekom

nepresné otáčanie vpravo:

```
type
  TRobot2 = class(TRobot)
    procedure rt(u: Real); override;
  end;

procedure TRobot2.rt(u: Real);
begin
  inherited rt(u*(0.9+Random(20)/100));
end;

...

  r := TRobot2.Create(X, Y, 90);
```

ak by sme odvodili triedu TRobot2 z TRobot1, dostali by sme robota, ktorý nevie robiť ani presné fd ani presné rt

kombinovaná trieda - preberá vlastnosti z TRobot1:

```
type
  TRobot2 = class(TRobot1)
    ...
```

ďalšie dve triedy ilustrujú iné varianty zmenenej metódy fd:

iné verzie fd:

```
type
  TRobot3 = class(TRobot)
    procedure fd(d: Real); override;
```

```
  end;

  TRobot4 = class(TRobot)
    procedure fd(d: Real); override;
  end;

procedure TRobot3.fd(d: Real);
begin
  lt(Random(41)/10-2);
  inherited;               // to je to isté, ako inherited fd(d);
  lt(180+Random(41)/10-2);
  inherited;
  lt(180+Random(41)/10-2);
  inherited;
end;

procedure TRobot4.fd(d: Real);
begin
  lt(60);
  while d >= 5 do
  begin
    inherited fd(5);
    rt(120);
    inherited fd(5);
    lt(120);
    d := d-5;
  end;
  rt(60);
  inherited fd(d);
end;
```

- otestujte kreslenie domčeka robotom, ktorý je inštanciou týchto tried
- poexperimentujte s tým, ak trieda TRobot3, resp. TRobot4 bude odvodená nie z TRobot ale z TRobot1 alebo TRobot2
- zamyslite sa nad tým, ako bude fungovať TRobot4.fd pre záporné d; prípadne opravte túto metódu tak, aby pracovala pre záporné hodnoty správne

nasledujúci jednoduchý príklad ilustruje pole 10 robotov, ktoré sa pohybujú po rôzne veľkých kružniciach

10 robotov:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  r: array [1..10] of TRobot;
begin
  for i := 1 to 10 do
  begin
    r[i] := TRobot.Create;
    r[i].PW := 7;
  end;
  repeat
    cs;
    for i := 1 to 10 do
    begin
      r[i].fd(i);
      r[i].rt(3);
    end;
    wait(100);
  until False;
end;
```

ak týmto robotom opravíme metódu setxy, tak budeme vidieť tento pohyb po kružniciach "zboku"

predefinujeme setxy:

```
type
  TRobot0 = class(TRobot)
    procedure setxy(xx, yy: Real); override;
  end;

procedure TRobot0.setxy(xx, yy: Real);
begin
  Inherited setxy(X, yy);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  r: array [1..10] of TRobot;
begin
  for i := 1 to 10 do
  begin
    r[i] := TRobot0.Create;
    r[i].PW := 7;
  end;
  ...
```

iných 10 robotov bude kresliť 10 rovnako veľkých kružníc, ktoré ale napriek tomu budú rôzne veľké:

10 rovnako veľkých kružníc:

```
type
  TMojRobot = class(TRobot)
    procedure fd(d: Real); override;
  end;

procedure TMojRobot.fd(d: Real);
begin
  inherited fd(d*1.5);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  r: array [1..10] of TRobot;
  i, j: Integer;
begin
  for i := 1 to 10 do
    if Random(5) = 0 then
      r[i] := TMojRobot.Create(50+i*10, 200)
    else
      r[i] := TRobot.Create(50+i*10, 200);

  for i := 1 to 10 do
    r[i].PW := 3;
  for i := 1 to 10 do
    if r[i] is TMojRobot then
      r[i].PC := clRed;

  for j := 1 to 120 do             // tu sa kreslia kružnice
  begin
    for i := 1 to 10 do
      r[i].fd(3);
    for i := 1 to 10 do
      r[i].rt(3);
    wait(10);
  end;

  for i := 1 to 10 do
    r[i].Free;
```

```
end;
```

všimnite si riadok, v ktorom sme zafarbili tie roboty, ktoré sú odvodené z TMojRobot - použili sme na to operátor is, ktorý otestuje, či je inštancia danej triedy (alebo triedy, ktorá je potomkom tejto triedy)

- častou začiatočníckou chybou je test if r[i] is TRobot then - totiž všetky prvky r[i] sú odvodené z TRobot - aj keď sú odvodené z nejakých tried, ktoré sú potomkami TRobot - teda tento test je splnený vždy
- v tomto príklade sa niektoré metódy často volajú pre všetky existujúce roboty (napr. najprv všetci fd a potom všetci rt) - asi by sa nám tu hodila nejaká trieda, ktorá "zapúzdri" pole robotov a hromadné metódy pre toto pole

Trieda na obsluhu viac robotov - TRobotGroup

Do programovej jednotky RobotGroupUnit zadefinujeme novú triedu TRobotGroup. Táto trieda sa bude starať o dynamické (polymorfné) pole robotov: umožní pridávať a vyhadzovať roboty a tiež paralelne vykonávať všetky grafické príkazy (metódy). Definícia triedy obsahuje:

- súkromné dynamické pole Fr robotov a ich počet Fnum (pomocou vlastností - property umožníme čítať tieto stavové premenné - ale nie modifikovať);
- metódy na pridávanie do tohto poľa:
  - metóda newRobot je definovaná v troch rôznych verziách: verzia bez parametrov, verzia s troma číselnými parametrami x, y, a u; a verzia s polymorfným robotom - podľa typu volaných parametrov sa Delphi rozhodnú, ktorú z týchto verzií naozaj zavolajú (použijeme špecifikátor overload)
- stavové premenné - vlastnosti, pomocou ktorých povolíme čítanie súkromných (a teda chránených) stavových premenných
  - vlastnosť r, ktorá sprístupní pole Fr je "indexovaná" stavová premenná, to znamená, že metóda getRobot musí mať jeden parameter typu index a musí to byť funkcia, ktorá vracia príslušnú hodnotu stavovej premennej

deklarácie triedy:

```
type
  TRobotGroup = class
  private
    Fr: array of TRobot;
    Fnum: Integer;
    function getRobot(i: Integer): TRobot;
  public
    constructor Create;
    destructor Destroy; override;
    procedure newRobot; overload;
    procedure newRobot(x, y: Real; a: Real = 0); overload;
    procedure newRobot(nr: TRobot); overload;
    procedure eraseRobot(i: Integer); overload;
    procedure eraseRobot(nr: TRobot); overload;
    procedure compact;

    procedure fd(d: Real);
    procedure rt(a: Real);
    ...
    procedure fill(color: TColor);
    function  isnear(x, y: Real): TRobot;
    procedure draw;
    procedure action;

    property  r[i: Integer]: TRobot read getRobot; default;
    property  num: Integer read Fnum;
```

```
    property  PW: Integer write setpw;
    property  PC: TColor write setpc;
  end;
```

realizácia kľúčových metód:

```
constructor TRobotGroup.Create;
begin
  Fnum := 0;
end;

destructor TRobotGroup.Destroy;
var
  i: Integer;
begin
  for i := 0 to Fnum-1 do
    Fr[i].Free;
end;

function TRobotGroup.getRobot(i: Integer): TRobot;
begin
  if (i < 0) or (i >= Fnum) then
    Result := nil
  else
    Result := Fr[i];
end;

procedure TRobotGroup.newRobot;
begin
  newRobot(TRobot.Create);              // toto nie je rekurzia
end;

procedure TRobotGroup.newRobot(x, y, a: Real);
begin
  newRobot(TRobot.Create(x, y, a));     // toto nie je rekurzia
end;

procedure TRobotGroup.newRobot(nr: TRobot);
begin
  if Fnum>high(Fr) then
    SetLength(Fr, Length(Fr)+10);
  Fr[Fnum] := nr;
  Inc(Fnum);
end;

procedure TRobotGroup.eraseRobot(i: Integer);
begin
  if (i >= 0) or (i < Fnum) then
  begin
    Fr[i].Free;
    Fr[i] := nil;
  end;
end;

procedure TRobotGroup.eraseRobot(nr: TRobot);
var
  i: Integer;
begin
  i := 0;
  while (i < Fnum) and (Fr[i] <> nr) do Inc(i);
  eraseRobot(i);                    // ani toto nie je rekurzia
end;

procedure TRobotGroup.compact;      // vyhodí nil robotov z poľa
var
  i, j: Integer;
```

```
begin
  j := 0;
  for i := 0 to Fnum-1 do
    if Fr[i] <> nil then
    begin
      Fr[j] := Fr[i];
      Inc(j);
    end;
  Fnum := j;
end;
```

metóda fd posunie všetky roboty

```
procedure TRobotGroup.fd(d: Real);
var
  i: Integer;
begin
  for i := 0 to Fnum-1 do
    if Fr[i] <> nil then
      Fr[i].fd(d);
end;
```

na rovnakom princípe ako metóda fd pracujú skoro všetky ostatné metódy

metóda isnear - vráti robota, ktorá je dostatočne blízko od bodu (x, y):

```
function  TRobotGroup.isnear(x, y: Real): TRobot;
var
  i: Integer;
begin
  i := Fnum-1;
  while (i >= 0) and ((Fr[i] = nil) or not Fr[i].isnear(x, y)) do
    Dec(i);
  if i < 0 then
    Result := nil
  else
    Result := Fr[i];
end;
```

Príklad

- teraz môžeme prepísať príklad s 10 kružnicami s použitím triedy TRobotGroup

použitie triedy TRobotGroup:

```
uses
  RobotUnit, RobotGroupUnit;

...

procedure TForm1.Button1Click(Sender: TObject);
var
  g: TRobotGroup;
  i, j: Integer;
begin
  g := TRobotGroup.Create;
  for i := 1 to 10 do
    if Random(5) = 0 then
      g.newRobot(TMojRobot.Create(50+i*10, 200))
    else
      g.newRobot(50+i*10, 200);
  g.PW := 3;
  for i := 0 to g.num-1 do
    if g.r[i] is TMojRobot then
```

```
      g.r[i].PC := clRed;
  for j := 1 to 120 do
  begin
    g.fd(3);
    g.rt(3);
    wait(10);
  end;
  g.Free;
end;
```

zápis pomocou príkazu with:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j: Integer;
begin
  with TRobotGroup.Create do
  begin
    for i := 1 to 100 do
      if Random(5) = 0 then
        newRobot(TMojRobot.Create(50+i*10, 200))
      else
        newRobot(50+i*10, 200);
    PW := 3;
    for i := 0 to num-1 do
      if not (r[i] is TMojRobot) then
        r[i].PC := clRed;
    for j := 1 to 120 do
    begin
      //cs; fd(23); fd(-20);
      fd(3);
      rt(3);
      wait(10);
    end;
    Free;
  end;
end;
```

- v tomto riešení sme nepotrebovali premennú g typu TRobotGroup - tento objekt existoval len počas platnosti with
- príkaz if not (r[i] is TMojRobot) then ... testuje tie roboty, ktoré nie sú triedy TMojRobot a teda sú to všetky ostatné

Príklad

- nasledujúci príklad demonštruje nutnosť **pretypovať** inštanciu, ak potrebujeme vyvolať metódu (alebo aj použiť stavovú premennú), ktorú nepozná predok (základná trieda) ale iba potomok

roboty nakreslia sínus:

```
type
  TMojRobot = class(TRobot)
  private
    k: Real;
  public
    constructor Create(x, y: Real; u: Real = 0);
    procedure Koef(kk: Real);
    procedure fd(d: Real); override;
  end;

constructor TMojRobot.Create(x, y, u: Real);
begin
```

```
  inherited;
  k := 1;
end;

procedure TMojRobot.Koef(kk: Real);
begin
  k := kk;
end;

procedure TMojRobot.fd(d: Real);
begin
  inherited fd(d*k);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  with TRobotGroup.Create do
  begin
    for i := 1 to Image1.Width div 2 do
    begin
      newRobot(TMojRobot.Create(2*i, 250));
      TMojRobot(r[num-1]).koef(sin(2*rad*i)); // posledne vyrobený robot
    end;
    fd(200);
    Free;         // uvoľní inštanciu TRobotGroup - pole robotov
  end;
end;
```

"sinusové" roboty kmitajú:

```
...

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  with TRobotGroup.Create do
  begin
    for i := 1 to Image1.Width div 2 do
    begin
      newRobot(TMojRobot.Create(2*i, 250));
      (r[num-1] as TMojRobot).koef(sin(2*rad*i));
    end;
    PW := 3;
    fd(200);
    repeat
      rt(180);
      for i := 1 to 100 do
      begin
        cs;
        fd(4);
        wait(1);
      end;
    until False;
  end;
end;
```

v druhom príklade môžeme vidieť iný variant pretypovania - použili sme operátor as, ktorým sa "pozrieme" na nejakú inštanciu (robota) akoby bola inej triedy (svoj potomok)

Projekt, ktorý bol na skúške v 1996:

zadanie:

Napíšte program, ktorý bude riešiť nasledovnú úlohu: textový súbor **subor.txt** obsahuje postupnosť príkazov pre robota, ktorá popisuje nejaký obrázok. Najprv zadefinujete triedu TRobot0 **potomka triedy** TRobot, ktorá bude obsahovať jedinú metódu

```
procedure kresli(var f: TextFile);
```

Táto metóda sa nastaví na začiatok súboru **f**, robotovi zdvihne pero a zinterpretuje príkazy v súbore. Nakoľko takto kreslený obrázok môže byť buď príliš veľký alebo malý alebo môže presahovať okraje grafickej obrazovky, bude ho treba zmenšiť alebo zväčšiť, resp. pri inicializovaní robota nezačať kresliť v strede plochy, ale zvoliť si domovskú pozíciu vhodnejšie a to tak, aby na šírku alebo výšku bol maximálne možne veľký.

Na riešenie tejto úlohy zadefinujete ešte dve podtriedy triedy TRobot0: trieda TRobot1 bude slúžiť na zistenie rozmerov obrázka pomocou zdedenej metódy kresli pričom na obrazovke sa bude pohybovať so zdvihnutým perom. Trieda TRobot2 (tiež podtrieda triedy TRobot0) bude vedieť nakresliť daný obrázok (zdedenou metódou kresli) pričom prekryje virtuálnu metódu fd tak, že obrázok bude požadovanej veľkosti.

Textový súbor **subor.txt** sa skladá z takýchto "viet": prvé číslo každej vety je typu ordinálna hodnota z

```
type TPrik = (pfd, plt, prt, ppc, ppu, ppd);
```

a reprezentuje postupne príkazy (fd, lt, rt, setpc, pu, pd). Pre hodnoty pfd, plt, prt a ppc za týmto číslom nasleduje parameter príslušného príkazu - jedno celé číslo (Integer). Pre hodnoty ppu a ppd veta už neobsahuje ďalšie čísla. Predpokladajte, že vstupný súbor je zadaný korektne (obrázok nemá nulovú ani výšku ani šírku).

Váš program teda najprv popíše tri triedy TRobot0, TRobot1 a TRobot2. Potom pomocou inštancie triedy TRobot1 (s domovskou pozíciou v (0, 0)) zistí rozmery obrázka (použijeme na to stavové premenné minX, minY, maxX a maxY) - zrejme predefinujete virtuálne metódy fd, pu a pd. Nakoniec, pomocou inštancie triedy TRobot2, ktorú zinicializuje v nejakej novej domovskej pozícii, vykreslí daný obrázok správnej veľkosti. Oba roboty majú rovnaký počiatočný smer zadaný konštantou programu, napr.

```
const U = 30;
```

Na testovanie programu môžete použiť súbory subor.txt alebo subor1.txt.

Riešenie:

najprv len vykreslíme súbor pomocou inštancie Robot0, aby sme mohli vidieť, momentálny obsah súboru:

```
type
  TRobot0 = class(TRobot)
    procedure kresli(var t: TextFile);
  end;
```

```
procedure TRobot0.kresli(var t: TextFile);
const
  farba: array[0..7] of TColor =
    (clBlack, clBlue, clGreen, clRed, clYellow, clGray, clMagenta, clWhite);
type
  TPrik = (pfd, plt, prt, ppc, ppu, ppd);
var
  p: TPrik;
  i: Integer;
begin
  pu;
  Reset(t);
  while not SeekEof(t) do
  begin
    Read(t, i);
    p := TPrik(i);
    if p <= ppc then
      Read(t, i);
    case p of
      pfd: fd(i);
      plt: lt(i);
      prt: rt(i);
      ppc: PC := farba[i];
      ppu: pu;
      ppd: pd;
    end;
  end;
end;

const
  u = -45;
  subor = 'subor.txt';

// toto slúži len na otestovanie metódy kresli

procedure TForm1.FormCreate(Sender: TObject);
var
  t: TextFile;
begin
  AssignFile(t, subor);
  with TRobot0.Create(200, 200, u) do
                             // nepotrebujeme premennú typu TRobot0
  begin
    kresli(t);
    Free;
  end;
  CloseFile(t);
end;
```

teraz dodefinujeme TRobot1 a TRobot2:

```
type
  TRobot1 = class(TRobot0)
    b, p: Boolean;
    minx, miny, maxx, maxy: Real;
    procedure fd(d: Real); override;
    procedure pu; override;
    procedure pd; override;
  end;

procedure TRobot1.pu;
begin
  inherited;
  p := False;
end;
```

```
procedure TRobot1.pd;
begin
  p := True;
end;

procedure TRobot1.fd(d: Real);

  procedure xy;
  begin
    if not p then          // ak je pero hore
      Exit;
    if b or (X < minx) then
      minx := X;
    if b or (X > maxx) then
      maxx := X;
    if b or (Y < miny) then
      miny := Y;
    if b or (Y > maxy) then
      maxy := Y;
    b := False;
  end;

begin
  xy;
  inherited;
  xy;
end;
```

trieda TRobot2:

```
type
  TRobot2 = class(TRobot0)
    r: Real;
    procedure fd(d: Real); override;
  end;

procedure TRobot2.fd(d: Real);
begin
  inherited fd(d*r);
end;
```

tlačidlo Button1 naštartuje celý algoritmus:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  t: TextFile;
  rr, r1, r2: Real;
begin
  AssignFile(t, subor);
  with TRobot1.Create(0, 0, u) do
  begin
    b := True;        // bude treba inicializovať minx, maxx, miny, maxy
    kresli(t);        // TRobot1 naozaj nekreslí - len zisťuje oblasť
    r1 := Image1.Width/(maxx-minx+1);
    r2 := Image1.Height/(maxy-miny+1);
    if r1 < r2 then
      rr := r1
    else
      rr := r2;
    with TRobot2.Create(-rr*minx, -rr*miny, u) do
    begin
      r := rr;        // nastaví mierku pre fd
      kresli(t);
      Free;
    end;
```

```
    Free;
  end;
  CloseFile(t);
end;
```

rozpracovaná 19. prednáška: klávesnica, ďalší formulár

čo už vieme:

- pri práci s komponentom TEdit sme využili udalosť onKeyPress

čo sa na tejto prednáške naučíme:

- ovládanie programu pomocou klávesnice, napr. klávesov šípok
- otvorenie ďalšieho formulára

Klávesnica

s klávesnicou pracujeme pomocou troch udalostí:

- onKeyDown - práve bol zatlačený nejaký kláves
- onKeyUp - práve bol pustený nejaký kláves
- onKeyPress - stlačili sme "obyčajný" kláves (ktorý má svoj ASCII kód)

prvé dve udalosti oznámia nie ASCII kód klávesu ale "virtuálny kód" (v Helpe si pozrite identifikátory konštánt) a tiež v parametri Shift zistíme, či bol pri tom zatlačený napr. Shift alebo Ctrl (podobne ako pri udalostiach pri práci s myšou)

nasledujúci program ukáže použitie klávesnice - do formulára umiestnime komponenty Label1 a Timer1 (nastavíme mu nejaký krátky Interval) - teraz môžeme na klávesnici súčasne tlačiť aj viac šípok a tým riadiť pohyb textu po ploche formulára:

pohyb textu vo formulári:

```
var
  dx, dy: Integer;  // môžeme predpokladať, že sú inicializované na 0

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  with Label1 do
  begin
    Left := Left+dx;
          // nedá sa urobiť Inc(Left, dx); lebo Left nie je premenná
    Top := Top+dy;
  end;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
          Shift: TShiftState);
begin
  case Key of
```

```
    vk_left:   dx := -1;
    vk_right:  dx := 1;
    vk_up:     dy := -1;
    vk_down:   dy := 1;
    vk_escape: Close;
  end;
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
          Shift: TShiftState);
begin
  case Key of
    vk_left,
    vk_right: dx := 0;
    vk_up,
    vk_down:  dy := 0;
  end;
end;
```

pri práci s klávesnicou si treba dávať pozor na iné komponenty, ktoré "kradnú" informácie z klávesnice (napr. Memo, Button, Edit a pod.) - najjednoduchšie riešenie v prípade, že chceme pracovať s klávesnicou, je ich nepoužívať, alebo spracovávať vstup z klávesnice na týchto komponentoch...

Príklad

text zadávaný z klávesnice vypisujeme do grafickej plochy Image1, Enter posunie kurzor o riadok nižšie - príklad ilustruje použitie onKeyPress

prvá jednoduchá verzia:

```
var
  x, y: Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  x := 20;
  y := 20;
  with Image1.Canvas.Font do
  begin
    Name := 'Arial';
    Height := 50;
    Style := [fsBold];
  end;
end;

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key >= ' ' then
  begin
    Image1.Canvas.TextOut(x, y, Key);
    Inc(x, 20);
  end
  else if Key = #13 then
  begin
    x := 20;
    Inc(y, 50);
  end;
end;
```

druhá verzia ukazuje, ako môžeme zobrazovať textový kurzor, zároveň používa metódu TextExtend, pomocou ktorej zistí šírku a výšku vypisovaného znaku

druhá verzia - proporčné písmo a kurzor:

```
var
  x, y: Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := True;
  x := 20;
  y := 20;
  with Image1.Canvas, Font do
  begin
    Name := 'Arial';
    Height := 50;
    Style := [fsBold];
    MoveTo(x, y);
    LineTo(x, y+Height);
  end;
end;

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
var
  s: TSize;
begin
  with Image1.Canvas do          // zmaže kurzor
  begin
    Pen.Color := clWhite;
    MoveTo(x, y);
    LineTo(x, y+Font.Height);
  end;
  if Key >= ' ' then
    with Image1.Canvas do
    begin
      TextOut(x, y, Key);
      s := TextExtent(Key);
      Inc(x, s.cx);
      if x > Image1.Width-20 then
        Key := #13;
    end;
  if Key = #13 then
  begin
    s := Image1.Canvas.TextExtent('M');
    x := 20;
    Inc(y, s.cy);
  end;
  with Image1.Canvas do         // nakreslí kurzor
  begin
    Pen.Color := clBlack;
    MoveTo(x, y);
    LineTo(x, y+Font.Height);
  end;
end;
```

Ďalšie námety

- zrealizujte kláves Backspace - treba si pamätať každý zapísaný znak (v niečom podobnom ako zásobník) - aby sme ho mohli zmazať, treba vedieť jeho pozíciu a buď jeho hodnotu alebo veľkosť

Príklad

v ďalšom príklade riadime šípkami viac robotov - len jeden z nich je aktívny, Enter prepína aktívnosť na nasledujúceho robota

viac robotov:

```
uses
```

```
  RobotGroupUnit, RobotUnit;

var
  group: TRobotGroup;
  akt: Integer = 0;

procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Randomize;
  cs;
  group := TRobotGroup.Create;
  for i := 1 to Random(10)+10 do
    group.newRobot(Random(Image1.Width-100)+50,
                   Random(Image1.Height-100)+50);
  group.PW := 7;
  group.r[akt].PC := clRed;
  group.point(5);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  group.Free;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
    vk_left:
      group.r[akt].lt(90);
    vk_right:
      group.r[akt].rt(90);
    vk_up:
      with group.r[akt] do
        begin
          PC := clBlack;
          fd(10);
          PC := clRed;
          point(5);
        end;
    vk_return:
      begin
        with group.r[akt] do
        begin
          PC := clBlack;
          point(5);
        end;
        akt := (akt+1) mod group.num;
        with group.r[akt] do
        begin
          PC := clRed;
          point(5);
        end;
      end;
  end;
end;
```

druhá verzia "vizualizuje" aktívneho robota a jej momentálny smer

využívame tu pomocnú bitmapu na zapamätanie si stavu plochy tesne pred vykreslením červenej šípky

zároveň s ovládaním klávesnice beží časovač, ktorý hýbe aktívneho robota po malých krokoch

viac robotov:

```
uses
  RobotGroupUnit, RobotUnit;

var
  group: TRobotGroup;
  akt: Integer = 0;
  bmp: TBitmap;

procedure kresliAktivneho;
begin
  bmp.Assign(Form1.Image1.Picture);
  with group.r[akt] do
  begin
    PC := clRed;
    fd(20);
    point(8);
    fd(-20);
    PC := clBlack;
  end;
end;


procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Randomize;
  cs;
  group := TRobotGroup.Create;
  for i := 1 to Random(10)+10 do
    group.newRobot(Random(Image1.Width-100)+50,
                   Random(Image1.Height-100)+50);
  group.PW := 3;
  group.point;
  bmp := TBitmap.Create;
  kresliAktivneho;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  group.Free;
  bmp.Free;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  Image1.Canvas.Draw(0, 0, bmp);
  case Key of
    vk_left:
      group.r[akt].lt(90);
    vk_right:
      group.r[akt].rt(90);
    vk_up:
      group.r[akt].fd(10);
    vk_return:
      akt := (akt+1) mod group.num;
    vk_escape:
      Close;
  end;
  kresliAktivneho;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Image1.Canvas.Draw(0, 0, bmp);
  with group.r[akt] do
```

```
  begin
    fd(1);
    lt(1);
  end;
  kresliAktivneho;
end;
```

Príklad

v príklade zadefinujeme jednoduchý objekt s bitmapou, ktorý sa potom hýbe niektorým smerom a odráža sa od okrajov plochy - smer sa určuje klávesmi šípok

definovanie jednoduchého objektu s bitmapou (môžeme umiestniť do inej programovej jednotky, napr. ObrazokUnit.pas):

```
type
  TObrazok = class
    bmp: TBitmap;
    x, y, dx, dy: Integer;
    constructor Create(meno: string; xx, yy: Integer);
    destructor Destroy; override;
    procedure posun(sir, vys: Integer);
    procedure kresli(c: TCanvas);
  end;

constructor TObrazok.Create(meno: string; xx, yy: Integer);
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile(meno);
  bmp.Transparent := True;
  x := xx;
  y := yy;
  dx := 0;
  dy := 0;
end;

destructor TObrazok.Destroy;
begin
  bmp.Free;
end;

procedure TObrazok.posun(sir, vys: Integer);
begin
  Inc(x, dx);
  Inc(y, dy);
  if (x < 0) or (x > sir) then
    dx := -dx;
  if (y < 0) or (y > vys) then
    dy := -dy;
end;

procedure TObrazok.kresli(c: TCanvas);
begin
  c.Draw(x-bmp.Width div 2, y-bmp.Height div 2, bmp);
end;
```

práca s obrázkovým objektom a s klávesnicou:

```
var
  a: TObrazok;

procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
  Randomize;
  a := TObrazok.Create('opica.bmp',
         Random(Image1.Width), Random(Image1.Height));
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Image1.Canvas.FillRect(Image1.ClientRect);
  a.posun(Image1.Width, Image1.Height);
  a.kresli(Image1.Canvas);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  a.Free;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
          Shift: TShiftState);
begin
  with a do
    case Key of
      vk_left:  Dec(dx);
      vk_right: Inc(dx);
      vk_up:    Dec(dy);
      vk_down:  Inc(dy);
      vk_home:  begin
                  dx := 0;
                  dy := 0;
                end;
    end;
end;
```

môžeme experimentovať s odrážaním objektu na hranici plochy:

```
procedure TObrazok.posun1(sir, vys: Integer);
begin
  Inc(x, dx);
  Inc(y, dy);
  if x < 0 then
  begin
    dx := Abs(dx);
    x := 0;
  end;
  if x > sir then
  begin
    dx := -Abs(dx);
    x := sir-1;
  end;
  if y < 0 then
  begin
    dy := Abs(dy);
    y := 0;
  end;
  if y > vys then
  begin
    dy := -Abs(dy);
    y := vys-1;
  end;
end;
```

alebo aj so spomaľovaním:

```
procedure TObrazok.posun2(sir, vys: Integer);
begin
```

```
  Inc(x, dx);
  Inc(y, dy);
  if x < 0 then
  begin
    dx := Abs(dx) div 2;
    x := 0;
  end;
  if x > sir then
  begin
    dx := -Abs(dx) div 2;
    x := sir-1;
  end;
  if y < 0 then
  begin
    dy := Abs(dy) div 2;
    y := 0;
  end;
  if y > vys then
  begin
    dy := -Abs(dy) div 2;
    y := vys-1;
  end;
end;
```

... namiesto zmazania plochy v procedúre od časovača (Toemr1Timer) môžeme do pozadia vykresľovať nejakú bitmapu, napr. pomocou

```
Image1.Picture.LoadFromFile('pozadie.bmp');
```

Otvorenie ďalšieho formulára

v našej aplikácii môžeme pracovať aj s viacerými formulármi:

- niekedy sa nám môže hodiť pre ďalšie výpisy, grafiku, ako dialógové okno pre zadávanie parametrov a pod. (takýto formulár môže fungovať - byť otvorený paralelne s hlavným formulárom),
- inokedy potrebujeme otvoriť nový formulár tak, aby bol hlavný formulár zatiaľ blokovaný a čakalo sa, kým sa tento nový nezavrie, napr. pre informácie o programe

v menu File zvolíme New a Form - automaticky sa vytvorí nový unit (Unit2.pas) - do tohoto nového formulára môžeme poukladať ľubovoľné komponenty, napr. text pomocou Label1, alebo tlačidlo Button1 s textom Ok

do Button1Click môžeme priradiť napríklad Close - uzavretie formulára:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Close;
end;
```

aby sme tento nový formulár mohli otvoriť z hlavného okna, musíme za Implementation vložiť riadok **uses Unit2;** a napr. na nejaké tlačidlo priradiť otvorenie formulára: napr.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;    // alebo Form2.Show;
end;
```

- **ShowModal** znamená, že sa nové okno otvorí tak, že sa čaká na jeho uzavretie - až potom sa pokračuje v hlavnom formulári
- ak by sme namiesto toho použili **Show**, tak by sa druhé okno otvorilo a boli by aktívne obe: hlavné aj toto

nové

skonštruovanie nového formulára

- nový formulár (napr. okno s informáciami o programe) môžeme vytvoriť aj priamo v programe, napr.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f: TForm;
begin
  if Shift = [ssRight] then
  begin
    f := TForm.Create(self);
    with f do
    begin
      Name := 'mojformular';
      Caption := 'O programe';
      Width := 350;
      Height := 253;
      BorderStyle := bsSingle;
      Position := poScreenCenter;
      with TLabel.Create(f) do
      begin
        Name := 'Label';
        Parent := f;
        Font.Height := 24;
        Font.Name := 'Arial';
        Caption := 'Niečo o mojom programe: '#13#13+
                   'je to ukážkový program'#13+
                   'na prácu s rôznymi komponentmi';
        Left := 20;
        Top := 16;
        AutoSize := True;
      end;
      with TButton.Create(f) do
      begin
        Name := 'Ok';
        Parent := f;
        Caption := 'Ok';
        Left := 72;
        Top := 192;
        Width := 75;
        Height := 25;
        ModalResult := mrOK;
      end;
      ShowModal;    // zobrazí nový formulár a čaká na jeho zatvorenie
      Free;         // zruší nový formulár
    end;
  end;
end;
```

Poznámka

- ak v prostredí Delphi pri vytváraní formulára zvolíme cez pravoklikové menu "View as Text", zobrazí sa textová forma definícií komponentov na formulári - toto môžeme využiť pri konštruovaní formulára, resp. komponentov v našom programe

rozpracovaná 20. prednáška: dynamické premenné

čo už vieme:

- poznáme globálne a lokálne premenné
- niektoré premenné sú referenciami iných premenných, napr. inštancie objektov

čo sa na tejto prednáške naučíme:

- ako sa pracuje s dynamickými premennými, ako sa vytvárajú a ako sa rušia
- ukážeme použite netypového smerníka a netypových formálnych parametrov

dynamické premenné

Doteraz sme sa väčšinou stretali s premennými, ktorým hovoríme **statické** - ich veľkosť a adresa v pamäti boli určené už počas kompilácie. Statickými premennými sú buď globálne premenné alebo lokálne:

- globálne premenné (premenné hlavného programu, resp. všetkých programových jednotiek) sú kompilátorom vyhradené v jednom dátovom segmente, tieto premenné automaticky vznikajú pri štarte programu a sú inicializované hodnotou 0;
- lokálne premenné (aj hodnotové parametre) podprogramov sa automaticky vyhradia pri volaní podprogramu (automaticky sa zrušia pri ukončení podprogramov) ale ich veľkosť a pozícia v pamäti (t.j. v systémovom zásobníku) sa určí počas kompilácie - už vieme, že majú neinicializovanú hodnotu.

Statické premenné nemôžu počas behu zmeniť svoju veľkosť a ani adresu (počas tzv. Run Time) - ich vytvorením a rušením sa stará systém (spustení programu, resp. podprogramu).

Na rozdiel od týchto statických premenných má pascal aj mechanizmus na vytváranie **dynamických** premenných:

- vytváranie a rušenie takýchto premenných je v rukách programátora - do programu musí zapísať príkazy na vytváranie a rušenie premenných
- kým takúto premennú programátor nevytvorí, premenná neexistuje a zrejme sa s ňou nedá pracovať
- počas behu programu (Run Time) sa špeciálne na to určenými príkazmi môžu vytvárať a rušiť, tzv. dynamické premenné
- takéto premenné sa nevytvárajú ani v dátovom segmente programu ani v zásobníku ale v špeciálne na to určenom segmente, v ktorom budú všetky dynamické premenné - tento dátový segment nazývame **heap** (zriedkavejšie aj po slovensky ako halda), jeho veľkosť závisí od OS - často je to niekoľko 100 MB.

S niektorými dynamickými premennými sme už pracovali aj doteraz, aj tu bol v rukách programátora mechanizmus na vytváranie a rušenie takýchto premenných, ale zatiaľ sme na to takto nepozerali:

- dynamické pole (napr. var a: array of Integer;) - vytvoríme napr. pomocou SetLength(a, 10); a zrušíme napr. príkazom a := nil;
- znakový reťazec (napr. var s: String;) - vytvoríme napr. pomocou s := 'abcd'; a zrušíme napr. s := '';
- objekt (napr. var r: TRobot;) - vytvoríme pomocou r := TRobot.Create; a zrušíme pomocou r.Free;

Pri dynamických poliach a znakových reťazcoch má Delphi mechanizmus, pri ktorom vie pracovať aj s prázdnou premennou, t.j. hoci sme premennú zrušili, môžeme ju používať, napr. s := ''; s := s + '*';

Aby sme v našich programoch mohli pracovať s dynamickými premennými, ktoré budú vznikať a rušiť sa počas behu, programovací jazyk na to zavádza špeciálny mechanizmus, tzv. odkaz na premennú. Hovoríme tomu **smerník** (po anglicky **pointer**) a slúži na to, aby sme sa mohli odkazovať na nejakú inú premennú - nemusí to byť dynamická premenná. Smerníku sa okrem slova odkaz niekedy hovorí aj referencia alebo adresa.

smerníkové premenné

Smerník je špeciálny typ, ktorý slúži na odkazovanie sa na iné premenné - najčastejšie na dynamické premenné. Premenná typu smerník, t.j. premenná, pomocou ktorej sa odkazujeme na inú premennú, musí byť zadeklarovaná rovnako ako každá iná premenná. Často budú smerníkové premenné statické, t.j. deklarované ako globálne alebo lokálne premenné. V pamäti zaberajú 4 bajty rovnako ako typ Integer (lebo adresy v počítači sú 4-bajtové) a rovnako ako iné lokálne premenné aj tieto majú pri štarte podprogramu nedefinovanú hodnotu.

Každá smerníková premenná sa môže v jednom momente odkazovať len na jednu premennú a to niektorého konkrétneho typu. Pri deklarovaní smerníkovej premennej určíme odkazovaný typ, ale to ešte neznamená, že už je v tejto premennej nastavený odkaz na nejakú hodnotu. Smerníkový typ definujeme znakom ^ (strieška), za ktorým nasleduje identifikátor nejakého existujúceho typu - to bude ten typ, na ktorého premennú sa bude môcť tento odvolávať. Napr.

```
type
  PInteger = ^Integer;
```

označuje smerníkový typ, pomocou ktorého sa budeme môcť odvolávať na nejaké celočíselné premenné. Za znakom môžeme zapísať len identifikátor typu, napr.

```
type
  PReal = ^Real;
  PChar = ^Char;
  PPoint = ^TPoint;
```

Nie je dovolené, za smerníkový znak ^ písať nepriamu definíciu nejakého typu, napr. nasledujúce zápisy nie sú správne

```
type
  Pzaznam = ^record
    x, y: Integer;
  end;
  Pset = ^set of Byte;
  Pinterval = ^1..100;
  Parr = ^array[1..5] of Byte;
```

Skôr ako ukážeme vytváranie a prácu s dynamickými premennými, pozrime prácu so smerníkmi na statické premenné. Použijeme operátor @, ktorý vráti adresu (odkaz na) premennú. Tento odkaz je smerníkového typu na typ premennej:

```
var
  i: Integer;
  s: ^Integer;
begin
  i := 13;
  s := @i;
  s^ := 37;
  s^ := s^ * 2;
```

Premenná s je typu smerník a môže odkazovať iba na celočíselný typ Integer. Príkazom s := @i; sme do premennej s priradili odkaz na premennú i, čo znamená, že keď teraz budeme nepriamo pracovať s premennou, na ktorú odkazuje s, budeme vlastne pracovať s premennou i. Práca s premennou, na ktorú odkazuje smerník sa zapisuje znakom ^, ktorý nasleduje za smerníkovou premennou, t.j. s^ je momentálne teraz práca s premennou i. Preto príkazom s^ := 37; priradíme do premennej i 37 a s^ := s^ * 2; zdvojnásobí obsah i, t.j. v i je teraz už hodnota 74.

Oveľa častejšie sa smerníková premenná využíva na prístup k dynamickým premenným. Slúži na to štandardný príkaz (procedúra) New, ktorý vytvorí novú dynamickú premennú a jej odkaz priradí do smerníkovej premennej, napr.

```
var
  s: ^Integer;
begin
  New(s);
  s^ := 37;
  s^ := s^ * 2;
```

V príklade sa vytvorila nová celočíselná premenná (zatiaľ má nedefinovaný obsah) a jej odkaz sa priradil do premennej s. Potom sa do tejto novej premennej prostredníctvom smerníka s najprv priradila hodnota 37 a ďalej sa jej obsah zdvojnásobil. Túto novú premennú zadeklarovanú nevidíme, vidíme len smerník, pomocou ktorého k nej máme prístup.

štandardné procedúry New a Dispose

Obe tieto procedúry majú jeden formálny parameter (typu var), ktorým musí byť smerníková premenná.

New(*smerníková_premenná***);**

Procedúra podľa typu smerníka vytvorí novú premennú a jej odkaz priradí do smerníkovej premennej. Ak už predtým bola v tejto premennej priradená nejaká hodnota (odkaz na nejakú premennú), tak táto hodnota sa príkazom New zabúda. Aj smerníková premenná (tak ako aj každá iná) môže obsahovať jedinú hodnotu a každé ďalšie priradenie do tejto premennej starú hodnotu zabúda. So smerníkmi je to o to horšie, že ak v ňom máme uchovaný odkaz na nejakú dynamickú premennú, tak priradením inej hodnoty do tohto smerníka strácame možnosť pracovať s touto dynamickou premennou a už nikdy sa k nej nedostaneme. Týmto môžeme stratiť nielen nejaké údaje, ale aj časť pamäte, ktorá nám neskôr môže niekde chýbať.

Činnosť príkazu New(smernik), pre var smernik: ^typ; môžeme zhrnúť takto:

1. zabudne pôvodnú hodnotu premennej smernik,
2. vo voľnej časti pamäti (**heap**) vyhradí úsek veľkosti SizeOf(typ), t.j. toľko bajtov, koľko bude zaberať táto nová dynamická premenná
   - väčšinou je to trochu viac (závisí od organizácie správy pamäti, ktorá sa stará o **heap**),
3. do smernik priradí adresu tejto novej dynamickej premennej.

Štandardná procedúra Dispose má opačnú funkciu ako procedúra New: zruší dynamickú premennú, na ktorá bol odkaz prostredníctvom nejakého smerníka.

Činnosť príkazu Dispose(smernik), pre var smernik: ^typ; môžeme zhrnúť takto:

Dispose(*smerníková_premenná***);**

1. zaradí do voľnej pamäte (**heap**) premennú smernik^
2. smernik má teraz už nedefinovanú hodnotu a preto by sme pomocou tohto odkazu už nemali ďalej pracovať s dynamickou premennou
   - všetky smerníky, ktoré odkazovali na túto istú dynamickú premennú, majú tiež nedefinovanú hodnotu

konštantný smerník nil

Je to špeciálna hodnota, ktorú môžeme priradiť do ľubovoľnej smerníkovej premennej. Používa sa vtedy, keď chceme rozlíšiť stav, že niečo je v smerníku priradené (odkazuje na nejakú premennú) a stav, že tam priradené nič nie je. Táto univerzálna smerníková konštanta sa teda môže priradiť do ľubovoľného smerníka a tiež môžeme ľubovoľný smerník otestovať, či obsahuje práve túto hodnotu. Okrem toho všetky globálne smerníkové premenné majú automaticky túto hodnotu ako svoju inicializačnú. Konštantu nil priraďujeme napr. takto

```
smernik := nil;
```

a preto môžeme túto premennú ďalej testovať, napr. takto

```
if smernik = nil then
  New(smernik);
```

alebo

```
if smernik <> nil then
  smernik^ := 157;
```

Pomocou relačných operátorov (=, <>) môžeme porovnávať aj smerníky navzájom, ale len za predpokladu, že oba odkazujú na ten istý typ. Napr.

```
var
  s1, s2: ^Integer;
begin
  New(s1);
  s1^ := 37;
  s2 := s1;           // môžeme, lebo rovnaký typ smerníkov
  Inc(s1^, 2);
  if (s1 <> s2) and (s2 <> nil) then
    s2^ := s2^ + 2;
```

V nasledujúcom príklade bude kompilátor hlásiť chybu:

```
var
  s1: ^Integer;
  s2: ^Real;
begin
  New(s1);
  s1^ := 37;

  // s2 := s1;      // nemôžeme
  New(s2);
  s2^ := 37;
  if s1^ = s2^ then
    Memo1.Lines.Append('rovnaké hodnoty');
  if s1 = s2 then                          // toto je chyba
    Memo1.Lines.Append('rovnaké smerníky');
```

niektoré zásady slušného programovania

snažíme sa, aby všetky smerníkové premenné mali buď nil alebo skutočne ukazovali na nejaké premenné (t.j. neboli nedefinované)

ak p = nil = > odkaz p^ hlási známu chybu *Access Violation* a preto sa v programe často vyskytuje otázka
if p = nil, prípadne if p <> nil …

nikdy neodkazujeme smerníkovou premennou, o ktorej nie sme 100% presvedčení, že je definovaná a rôzna od nil (radšej buďme pesimisti a všetko kontrolujme)

príklady práce so smerníkmi

Dynamická premenná celé číslo:

```
var
  s: ^Integer;
begin
```

```
  New(s);
  s^ := 0;
  for i := 1 to 10 do
    s^ := s^+i;
  Writeln(t, s^);
  Dispose(s);
  s := nil;
  ...
```

Dynamický záznam:

```
type
  Zaznam = record
    x, y: Integer;
  end;
var
  z: ^Zaznam;
begin
  New(z);
  z^.x := 100;
  z^.y := 200;
  with z^ do
    Image1.Canvas.MoveTo(x, y);
  Inc(z^.x, 100);
  Dec(z^.y, 50);
  Image1.Canvas.LineTo(z^.x, z^.y);
  ...
```

Dynamické jednorozmerné pole:

```
type
  Pole = array[1..10] of Real;
var
  p: ^Pole;
begin
  New(p);
  for i := 1 to 10 do
    Read(t, p^[i]);
  for i := 9 downto 1 do
    p^[i] := p^[i] + p^[i+1];
  ...
```

Problém s veľkým poľom ako lokálna premenná:

```
type
  Pole = array[1..1000000] of Integer;

procedure test;
var
  p: Pole;
begin
  p[1] := 1;
end;
```

Takéto pole chce vzniknúť na zásobníku počas volania tejto procedúry - systém má ale problém s tak veľkým poľom

Veľké pole ako lokálna premenná pomocou smerníka:

```
type
  Pole = array[1..1000000] of Integer;

var
```

```
  n: Integer;

procedure test;
var
  p: ^Pole;
begin
  New(p);
  Inc(n);
  p^[1] := n;
  Form1.Memo1.Lines.Add(IntToStr(p^[1]));
  // Dispose(p);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  while True do
    test;
end;
```

ak sa toto veľké pole pred koncom procedúry neuvoľňuje (Dispose), program po niekoľkých prechodoch spadne na správe "Out of memory." - môžete takto otestovať veľkosť heapu ...

funkcia môže vrátiť aj smerník

vygeneruje náhodnú hodnotu:

```
type
  PInteger = ^Integer;

function daj: PInteger;
begin
  if Random(6) = 0 then
    Result := nil
  else
  begin
    New(Result);
    Result^ := Random(100);
  end;
end;
```

smerník na objekt:

```
type
  Tt = class
    a: Integer;
    constructor Create(aa: Integer);
  end;

constructor Tt.Create(aa: Integer);
begin
  a := aa;
end;

var
  s: ^Tt;
begin
  New(s);
  s^ := Tt.Create(8);
  Writeln(t, s^.a);
  s^.Free;
  Dispose(s);
end;
```

jednorozmerné pole smerníkov:

```
type
  Mnozina = set of Byte;     // 32 bajtov
  Pole = array [1..1000] of ^Mnozina;
                // 4000 bajtov -- inak bez ^ by bolo 32000 bajtov
var
  data: Pole;
  i: Integer;
begin
  for i := 1 to 1000 do
    New(data[i]);
  for i := 1 to 1000 do
    data[i]^ := [Random(256)];
...
  for i := 1 to 1000 do
    Dispose(data[i]);
  // uvoľnený heap -- môže sa ďalej v tomto projekte používať
end;
```

príklady jedno- a dvoj-rozmerných polí smerníkov:

```
type
  Pole = array[1..10] of Integer;
  PPole = ^Pole;
  Pole2 = array[1..20] of Pole;      // obyčajné 2-rozmerné pole
  PPole2 = ^Pole2;
  Pole2PPole = array[1..20] of PPole;
var
  a: Pole2PPole;
  b: PPole2;
  i, j: Integer;
begin
  for i := 1 to 20 do
    New(a[i]);
  for i := 1 to 20 do
    for j := 1 to 10 do
      a[i]^[j] := i + j;
  New(b);
  for i := 1 to 20 do
    for j := 1 to 10 do
      b^[i][j] := i + j;
...
end;
```

smerník na pole:

```
type
  Spole = array[1..10] of ^Integer;  // pole smerníkov na Integer
  PSpole = ^Spole;
  SPpole = array[1..20] of PSpole;
  PSPpole = ^Sppole;
var
  c: SPpole;
  d: PSPpole;
  i, j: Integer;
begin
  for i := 1 to 20 do             // SizeOf(c) = 20*4
     New(c[i]);
  for i := 1 to 20 do             // SizeOf(c[i]) = 4
    for j := 1 to 10 do           // SizeOf(c[i]^) = 10*4
    begin
      New(c[i]^[j]);
      c[i]^[j]^ := i + j;
    end;
  New(d);
  for i := 1 to 20 do       // SizeOf(d) = 4; SizeOf(d^) = 20*4
```

```
    New(d^[i]);
  for i := 1 to 20 do             // SizeOf(d^[i]) = 4
    for j := 1 to 10 do           // SizeOf(d^[i]^) = 10*4
    begin
      New(d^[i]^[j]);
      d^[i]^[j]^ := i + j;
    end;
  ...
end;
```

záznamy, polia a smerníky:

```
type
  Pole = array[1..10] of Integer;
  PPole = ^pole;
  Zazn = record
    x: Pole;
    y: PPole;
    z: array[1..10] of ^Integer;
  end;
  PZazn = ^Zazn;
  PolePZazn = array[1..20] of PZazn;
  PPolePZazn = ^PolePZazn;
var
  a: PZazn;
  b: PolePZazn;
  c: PPolePZazn;
  i, j: Integer;
begin
  New(a);
  for i := 1 to 20 do
    New(b[i]);
  New(c);
  for i := 1 to 20 do
    New(c^[i]);
  for i := 1 to 20 do
  begin
    for j := 1 to 10 do
      c^[i]^.x[j] := i+j;
    New(c^[i]^.y);
    for j := 1 to 10 do
      c^[i]^.y^[j] := i+j;
    for j := 1 to 10 do
      New(c^[i]^.z[j]);
    for j := 1 to 10 do
      c^[i]^.z[j]^ := i+j;
  end;
...
end;
```

postupné prečítanie premennej c^[i]^.y^[j]

c	- je typu PpolePzazn = smerník
c^	- je typu polePzazn = array
c^[i]	- je typu Pzazn = smerník
c^[i]^	- je typu zazn = record
c^[i]^.y	- je typu Ppole = smerník
c^[i]^.y^	- je typu pole = array
c^[i]^.y^[j]	- je typu Integer

zápis "smerníkovanej" premennej môžeme skrátiť: znak ^ vynecháme, ak si ho vedia Delphi jednoznačne

domyslieť, t.j. ak za ^ nasleduje bodka "." alebo hranatá zátvorka "["

smerník na smerník

nasledujúci príklad len ilustruje nezvyčajné použitie smerníkov:

```
type
  PInt = ^Integer;
  PPInt = ^PInt;
  PPPInt = ^PPInt;
var
  i: PInt;
  j: PPInt;
  k: PPPInt;
begin
  New(i);
  i^ := 123;

  New(j);
  New(j^);
  j^^ := 345;

  New(k);
  New(k^);
  New(k^^);
  k^^^ := 567;
...
end;
```

správa pamäti (Memory management)

táto správa sa stará o udržiavanie obsadených a uvoľnených častí dynamickej pamäti (heap)

štandardné procedúry: New, Dispose, GetMem, ReallocMem a FreeMem - využívajú správu pamäti

každý vyhradený pamäťový blok (napr. pomocou New) má dĺžku zaokrúhlenú na najbližší násobok 4 a obsahuje ešte 4-bajtovú hlavičku - dĺžku bloku a iné stavové informácie

správa udržiava tieto dve premenné:

- AllocMemCount - počet pamäťových blokov
- AllocMemSize - dĺžka vyhradených pamäťových blokov

funkcia GetHeapStatus vráti ďalšie užitočné informácie o správe pamäti

rezervované slovo nil je špeciálna smerníková konštanta - vnútorne je reprezentovaná 4 bajtami s hodnotou 0

smerníkový operátor @premenná vráti smerník (referenciu - adresu) na danú premennú (neskôr uvidíme aj smerník na procedúru) - výsledkom je smerník typu ^typ, ak je typ typom premennej

smerníková aritmetika: pomocou štandardných procedúr Inc a Dec môžeme posúvať hodnotu smerníka o dĺžku typu, na ktorý odkazuje

malá ukážka smerníkovej aritmetiky:

```
var
  p: ^Integer;
  a: array[1..10] of Integer;
  i: Integer;
begin
  p := @a[10];
  for i := 1 to 10 do
  begin
```

```
    p^ := i;
    Dec(p);
  end;
  for i := 1 to 10 do
    Memo1.Lines.Add(IntToStr(a[i]));
end;
```

druhý príklad smerníkovej aritmetiky:

```
var
  p1, p2: ^Integer;
  a: array[1..10] of Integer;
  i: Integer;
begin
  for i := 1 to 10 do
    a[i] := i;
  p1 := @a[1];
  p2 := @a[10];
  for i := 1 to 5 do
  begin
    p1^ := p1^ + p2^;
    p2^ := p1^ - p2^;
    p1^ := p1^ - p2^;
    Inc(p1);
    Dec(p2);
  end;
  for i := 1 to 10 do
    Memo1.Lines.Add(IntToStr(a[i]));
end;
```

všetky doterajšie smerníky boli presne zadaného typu (^typ) - priraďovanie a referencovanie smerníkových premenných bolo prísne kontrolované prostredím Delphi

netypový smerník - typ Pointer

univerzálny smerník (podobne ako nil)

kompatibilný so všetkými smerníkmi: môžeme ho priradiť do smerníkovej premennej ľubovoľného typu a naopak (môže to byť veľmi nebezpečné - ľahko môžeme stratiť kontrolu nad smerníkmi)

nemôžeme pomocou neho pracovať s dynamickou premennou, na ktorú odkazuje (buď ho priradíme do typového smerníka, alebo ho pretypujeme)

keď s ním chceme pracovať, tak buď najprv do neho priradíme už nejaký "hotový" smerník na dynamickú premennú alebo vyhradíme novú dynamickú pamäť pomocou štandardnej procedúry GetMem(*premenná_typu_Pointer*, *dĺžka*) - je to podobné New:

```
New(p) =  =  = GetMem(p, SizeOf(p^));
```

takto vyhradenú pamäť uvoľníme štandardnou procedúrou FreeMem(*premenná_typu_Pointer*) - je to podobné Dispose:

```
Dispose(p) =  =  = FreeMem(p);
```

použitie uvidíme pri netypových formálnych parametroch:

netypový formálny parameter

= formálny parameter, ktorý nemá uvedený typ

v tele procedúry ho môžeme použiť nasledujúcimi spôsobmi:

pretypovaním na konkrétny typ

alebo pomocou direktívy absolute (je to pretypovanie počas deklarácií)

alebo poslať ako netypový parameter do inej procedúry

- napr. štandardná procedúra Move(odkiaľ, kam, koľko_bajtov)
- alebo zápis, resp. načítanie do/z netypového súboru – budeme vidieť neskôr

napr. procedúra na výmenu obsahov dvoch ľubovoľných (rovnako veľkých) premenných:

```
procedure vymen(var a, b; dlzka: Integer);
var
  t: Pointer;
begin
  GetMem(t, dlzka);
  Move(a, t^, dlzka);
  Move(b, a, dlzka);
  Move(t^, b, dlzka);
  FreeMem(t);
end;
```

použitie operátora @ a smerníkovej aritmetiky Inc:

```
function dump(const a; dlzka: Integer): String;
var
  p: ^Byte;
begin
  Result := '';
  p := @a;
  while dlzka > 0 do
  begin
    Result := Result + IntToHex(p^, 2) + ' ';
    Inc(p);
    Dec(dlzka);
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  s: array[0..10] of Char;
  i: Integer;
begin
  s := 'Ahoj Delphi';
  Label1.Caption := dump(s, SizeOf(s));
  i := 12345;
  Label2.Caption := dump(i, SizeOf(i));
end;
```

pretypovanie

pomocou pretypovania:

```
function dump(const a; dlzka: Integer): String;
type
  pole = array[1..maxint] of Byte;
var
  i: Integer;
begin
  Result := '';
  for i := 1 to dlzka do
    Result := Result + IntToHex(pole(a)[i], 2) + ' ';
end;
```

je jedno, od akej dolnej hranice poľa začíname - ak by bolo **pole** deklarované napr. array [5..maxint] of Byte, zmenili by sme for-cyklus:

```
for i := 5 to dlzka+4 do ...
```

prekrytie premennej iným menom premennej (aj iného typu) - nejaká časť pamäti dostane ďalšie nové meno - je to veľmi nebezpečné

iná verzia šestnástkového výpisu kusu pamäti pomocou absolute --- nebezpečné, zastaralé:

```
function dump(var a; dlzka: Integer): String;
var
  p: array[1..maxint] of Byte absolute a;
  i: Integer;
begin
  Result := '';
  for i := 1 to dlzka do
    Result := Result + IntToHex(p[i], 2) + ' ';
end;
```

dynamické polia

Sú reprezentované smerníkom na jednorozmerné pole (v dynamickej pamäti **heap**). Deklarácia poľa nealokuje žiadnu pamäť - pole má nedefinovanú dĺžku(mali by sme napr. priradiť nil). SetLength vyhradí pamäť (niečo ako GetMem) - ak už premenná mala vyhradené nejaké pole, tak toto sa automaticky uvoľní (niečo ako FreeMem). Ak X a Y sú premenné rovnakého typu dynamické pole, potom X := Y spôsobí, že X referencuje na to isté pole ako Y (netreba alokovať pamäť pre X) - Delphi si teraz pamätá, že na toto pole sa odkazuje dvomi premennými a pamäť uvoľní, až keď sa zmenia referencie oboch polí. Pre dynamické polia nepoužívajte procedúry New, GetMem a pod. a ani operátor ^. Pozrite nasledujúci príklad:

```
var
  A, B: array of Integer;
begin
  SetLength(A, 4);
  A[0] := 1;
  B := A;           // teraz sú obe polia na tom istom mieste v pamäti
  B[0] := 2;        // aj hodnotou A[0] je 2
  SetLength(B, 3);  // teraz sú obe polia inde v pamäti
end;
```

Všimnite si posledný príkaz SetLength(B, 3), ktorý z poľa B uberie jeden prvok. Vďaka tomuto sa pre B vyhradí nová pamäť, pričom prvé dva prvky budú mať obe tieto polia rovnaké.

Pri porovnávaní premenných typu dynamické pole sa porovnávajú ich referencie a nie nie hodnoty polí (ako pri statických poliach). Napr.

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
  if A = B then ...
end;
```

Porovnanie A = B vráti False ale A[0] = B[0] vráti True. Na skrátenie dynamického poľa sa môže použiť aj funkcia Copy, napr. A := Copy(A, 5, 10); - funguje rovnako ak so znakovými reťazcami.

Premenná typu dynamické pole zaberá 4 bajty - je to smerník na dynamicky alokované pole v **heap**. Buď je to nil alebo smerník na blok pamäti, ktorý je o 8 bajtov dlhší ako vyhradená veľkosť poľa:

- 4 bajty použité ako počítadlo referencií
- 4 bajty na počet prvkov poľa (Length)
- za tým nasledujú prvky poľa

Viacrozmerné dynamické pole je reprezentované úplne rovnako -- je to dynamické pole smerníkov na dynamické polia.

znakové reťazce - String

Sú podobné dynamickým poliam - tiež sú to smerníky na polia znakov. Podobne sa pamätá aj počet referencií a aktuálna dĺžka reťazca (Length). Za posledným znakom v poli je vždy #0 (hoci tento sa nedá indexovať) - vďaka tomu je použiteľný aj ako #0 ukončený reťazec. Prázdny reťazec je uchovaný ako nil (ale napriek tomu do stringovej premennej nemôžeme priradiť nil). Nemôžeme používať ani New ani Dispose a ani iné procedúry správy pamäti. Teraz by ste už mohli správne rozumieť tomuto príkladu:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  s: String;
begin
  s := 'ahoj Delphi';
  label1.Caption := dump(s[1], Length(s));
        // nefunguje iba dump(s, Length(s)); - s je smerník
end;
```

Ak upravíte funkciu dump tak, aby začala vypisovať už o 8 bajtov skôr, môžete vidieť aj jeho počítadlo referencii a aj aktuálnu dĺžku reťazca. Takto môžete "vidieť" nielen reťazce ale aj dynamické polia.

znakové reťazce ukončené #0 (null-terminated strings)

Sú veľmi podobné znakovým reťazcom v C a C++. Niekedy ich treba poznať pri práci so systémom Windows, keď sa vyžadujú takéto parametre niektorých procedúr (väčšinou procedúr nižšej úrovne).

Tieto reťazce sú postupnosti znakov vždy ukončené znakom #0. Pre tieto postupnosti sa nepamätá momentálna dĺžka (ako u String), ale ak ju budeme potrebovať zistiť, musíme postupne pozrieť reťazec znak za znakom a hľadať prvý výskyt #0. Tieto reťazce najčastejšie uchovávame v jednorozmernom poli s dolnou hranicou 0, napr. TFileName = array[0..259] of Char; alebo často aj v dynamickej pamäti. Delphi pre to poskytujú preddefinovaný typ PChar (smerník na postupnosť znakov, t.j. ^Char).

Pracovať s takýmito reťazcami môžeme v mnohých štandardných procedúrach a funkciách podobne ako s typom String. Napr. Read, Readln, Str, Val, Write, Writeln, AssignFile, Rename. Samozrejme, že s týmito reťazcami môžeme pracovať aj úplne rovnako ako s inými poliami znakov. Okrem toho Delphi poskytujú aj množinu špeciálnych funkcií pre takéto reťazce, napr.

- StrCat zreťazenie
- StrComp porovnanie
- StrCopy skopírovanie
- StrLen dĺžka reťazca

A často sa pre PChar využíva smerníková aritmetika, napr.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  p, q: PChar;
```

```
begin
  GetMem(p, 100);      // mohli by sme použiť aj StrNew alebo StrAlloc
  StrCopy(p, 'milujem delphi');
  q := p;
  while q^ <> #0 do
  begin
    q^ := Upcase(q^);
    Inc(q);          // posun na nasledujúci znak reťazca
  end;
  Label1.Caption := p;
  Label2.Caption := dump(p, StrLen(p)+1);
  FreeMem(p);
end;
```

inštancie tried

Každá objektová premenná, t.j. inštancia nejakej triedy, je v skutočnosti smerník na dynamicky alokovaný blok pamäti. Treba na to myslieť pri porovnávaní aj priraďovaní, napr.

```
g := Image1.Canvas;                   // zapamätám si smerník na objekt
if g.Pen = Form1.Canvas.Pen then ...  // tu netestujem, či majú rovnaké pero
```

Všetky **stavové premenné** objektu sú uchované veľmi podobne ako položky v type **záznam**. Už vieme, že informácie o **metódach** sa ukladajú do tabuľky **VMT** (virtual method table) - táto je jediná pre všetky inštancie danej triedy. V pamäti pre objekt je len smerník na túto **VMT**. **VMT** okrem metód obsahuje aj iné informácie o inštancii, napr. informácie o dĺžke, triede a pod.

Asi by vám malo byť jasné, prečo je nezmysel namiesto r := TRobot.Create použiť r.Create. Premenná r je na začiatku nedefinované alebo má možno hodnotu nil a teda ňou sa nemôžeme odkazovať (referencovať) na metódu Create neexistujúcej inštancie.

rozpracovaná 21. prednáška: práca s bitmapou

čo už vieme:

- bitmapy vieme zo súboru prečítať, vieme ich vykresliť do grafickej plochy a tiež ich vieme zapísať do súboru
- práca s farebnými bodmi bitmapy pomocou Pixels je často nepoužiteľne pomalá

čo sa na tejto prednáške naučíme:

- ak budeme pracovať s farebnými bodmi bitmapy pomocou ScanLine, práca sa veľmi zrýchli a konečne môžeme vytvárať veľmi zaujímavé efekty s obrázkami
- vykonanie niektorých dôležitých častí programu môžeme chrániť konštrukciou try ... finally ... end

Práca s bitmapou

Doteraz sme vedeli pracovať s farebnými bodmi bitmapy (s pixelmi) len pomocou vlastnostiPixels[x, y] - o tomto ale vieme, že je to veľmi pomalý mechanizmus a v praxi často nepoužiteľný. Naučíme sa pracovať s oveľa rýchlejším mechanizmom **ScanLine[y]**, pomocou ktorého dostávame "priamy" prístup k jednému celému riadku

bitmapy a veľmi rýchlo môžeme farebné zložky nielen čítať ale aj modifikovať. Na rozdiel od Pixels, ktorý bol vlastnosťou (property) TCanvas, ScanLine je vlastnosťou TBitmap a teda priamo s Image sa takto pracuje komplikovanejšie. Výsledkom ScanLine[y] (túto vlastnosť môžeme len čítať) je netypový smerník na postupnosť, t.j. pole RGB zložiek - trojíc bajtov:

```
type
  TRGB = record
    B, G, R: Byte;
  end;
```

K postupnosti môžeme pristupovať napr. tak, ako s "null-terminated string":

```
type
  PRGB = ^TRGB;
var
  d: PRGB;
...
  d := bmp.ScanLine[y];
  for i := 0 to bmp.Width-1 do
  begin
    d^.R := 0;
    Inc(d);                      // nasledujúci pixel
  end;
```

alebo ako smerník na pole:

```
type
  TRGBArray = array[Word] of TRGB;       // dostatočne veľká hranica
  PRGBArray = ^TRGBArray;
```

V štandardných knižniciach Delphi existujú podobné deklarácie:

- TRGBTriple je podobné nášmu TRGB (v unite Windows)
- PRGBTripleArray je podobné nášmu PRGBArray (v unite Graphics)

Prvý príklad ukazuje, ako môžeme pomocou ScanLine vytvoriť vlastnú bitmapu - predpokladáme, že vo formulári máme Image1 veľkosti 256x256:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bmp: TBitmap;
  i, j: Integer;
begin
  bmp := TBitmap.Create;
  bmp.PixelFormat := pf24bit;
  bmp.Width := 256;
  bmp.Height := 256;

  for i := 0 to 255 do
    for j := 0 to 255 do
      PRGBArray(bmp.ScanLine[i])^[j].B := 0;

  Image1.Canvas.Draw(0, 0, bmp);
  bmp.Free;
end;
```

Keď vytvoríme novú bitmapu (TBitmap.Create), každý jej pixel je biely, t.j. všetky tri zložku RGB majú hodnotu 255. V tejto ukážke sme všetkým pixelom vynulovali modrú zložku RGB, t.j. vytvorili sme žltú. Nasledujúci príklad je veľmi podobný, len každý pixel novej bitmapy bude trochu iný:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  bmp: TBitmap;
  i, j: Integer;
  c: TRGB;
begin
  bmp := TBitmap.Create;
  bmp.PixelFormat := pf24bit;
  bmp.Width := 256;
  bmp.Height := 256;

  for i := 0 to 255 do
    for j := 0 to 255 do
    begin
      c.R := i;
      c.B := 255-i;
      c.G := j;
      PRGBArray(bmp.ScanLine[i])[j] := c;
    end;

  Image1.Canvas.Draw(0, 0, bmp);
  bmp.Free;
end;
```

V nasledujúcich príkladoch predpokladáme, že máme k dispozícii globálnu premennú bmp1, do ktorej vo FormCreate prečítame nejakú bitmapu (napr. parrots.bmp), vo FormDestroy ju uvoľníme a vo formulári je položený Image1 veľkosti 256x256:

```
var
  bmp1: TBitmap;

procedure TForm1.FormCreate(Sender: TObject);
begin
  bmp1 := TBitmap.Create;
  bmp1.LoadFromFile('parrots.bmp');
  Image1.Canvas.Draw(0, 0, bmp1);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  bmp1.Free;
end;
```

Ďalej sa na formulári nachádza Image2 tiež veľkosti 256x256 a niekoľko tlačidiel. Zapíšeme vytvorenie kópie bitmapy - 1. verzia:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bmp2: TBitmap;
  i, j: Integer;
begin
  bmp2 := TBitmap.Create;
  bmp2.PixelFormat := pf24bit;
  bmp2.Width := bmp1.Width;
  bmp2.Height := bmp1.Height;

  for i := 0 to bmp1.Height-1 do
    for j := 0 to bmp1.Width-1 do
      PRGBArray(bmp2.ScanLine[i])^[j] :=
                                  PRGBArray(bmp1.ScanLine[i])^[j];

  Image2.Canvas.Draw(0, 0, bmp2);
  bmp2.Free;
end;
```

samozrejme, že kópiu bitmapy vieme aj jednoduchšie pomocou bmp2.Assign(bmp1) - tu sa ale učíme pracovať so ScanLine

Ukážme 2. verziu kópie bitmapy:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bmp2: TBitmap;
  i, j: Integer;
  r1, r2: PRGBArray;
begin
  bmp2 := TBitmap.Create;
  bmp2.PixelFormat := pf24bit;
  bmp2.Width := bmp1.Width;
  bmp2.Height := bmp1.Height;

  for i := 0 to bmp1.Height-1 do
  begin
    r1 := bmp1.ScanLine[i];
    r2 := bmp2.ScanLine[i];
    for j := 0 to bmp1.Width-1 do
      r2^[j] := r1^[j];       // alebo r2[j] := r1[j];
  end;

  Image2.Canvas.Draw(0, 0, bmp2);
  bmp2.Free;
end;
```

veľmi podobný je zrkadlový obraz:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  bmp2: TBitmap;
  w, h, i, j: Integer;
  r1, r2: PRGBArray;
begin
  w := bmp1.Width;
  h := bmp1.Height;
  bmp2.PixelFormat := pf24bit;
  bmp2 := TBitmap.Create;
  bmp2.Width := w;
  bmp2.Height := h;

  for i := 0 to h-1 do
  begin
    r1 := bmp1.ScanLine[i];
    r2 := bmp2.ScanLine[i];
    for j := 0 to w-1 do
      r2[j] := r1[w-1-j];
  end;

  Image2.Canvas.Draw(0, 0, bmp2);
  bmp2.Free;
end;
```

Chránené príkazy

Pri programovaní sa často stretávame so situáciou, že niektoré príkazy potrebujeme bezpodmienečne vykonať, aj keď sa z nejakých dôvodov preruší výpočet, napr. pre výskyt chyby alebo ukončenie procedúry pomocou Exit. Napr. vždy, keď v procedúre vytvoríme pomocnú bitmapu, pred ukončením procedúry ju musíme zrušiť. Podobne, vždy, keď otvoríme súbor, na záver ho musíme zatvoriť, ... Môžeme použiť konštrukciu:

```
try
```

```
  postupnosťPríkazov1
finally
  postupnosťPríkazov2
end;
```

Ak počas vykonávania prvej postupnosti príkazov vznikne nejaké prerušenie (Exit, Break, spadnutie na chybe), tak sa pokračuje v druhej postupnosti príkazov. Ak prvá časť skončí bez chyby, tak sa normálne pokračuje v druhej časti. try ... end si môžete predstaviť ako begin ... end, pričom, ak v ňom nastane nejaké prerušenie, vykonajú sa ešte upratovacie akcie za finally. Zapamätajte si, že príkazy medzi finally a end sa vykonajú vždy. Konštrukcie try ... end môžu byť navzájom vnorené rovnako ako begin ... end.

V objektovom pascale existuje ešte veľmi podobná konštrukcia try ... except ... end - môžete si ju naštudovať v Helpe.

Uvedieme teraz niekoľko jednoduchých príkladov na try ... finally ... end. Pozrite, ako slušne by sme mohli, resp. mali pracovať so súborom:

```
...
AssignFile(t, 'súbor.txt');
Rewrite(t);                        // alebo Reset(t);
try
  ...
  // práca so súborom
  ...
finally
  CloseFile(t);
end;
...
```

Takto slušne by sme mali pracovať s bitmapami:

```
...
bmp := TBitmap.Create;
try
  ...
  // práca s bitmapou
  ...
finally
  bmp.Free;
end;
...
```

A konkrétne zrkadlovo otočíme bitmapu aj s try a finally:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  bmp2: TBitmap;
  w, h, i, j: Integer;
  r1, r2: PRGBArray;
begin
  w := bmp1.Width;
  h := bmp1.Height;
  bmp2 := TBitmap.Create;
  try
    bmp2.PixelFormat := pf24bit;
    bmp2.Width := w;
    bmp2.Height := h;

    for i := 0 to h-1 do
    begin
      r1 := bmp1.ScanLine[i];
      r2 := bmp2.ScanLine[i];
```

```
      for j := 0 to w-1 do
        r2[j] := r1[w-1-j];
    end;

    Image2.Canvas.Draw(0, 0, bmp2);
  finally
    bmp2.Free;
  end;
end;
```

Otočíme bitmapu symetricky podľa hlavnej uhlopriečky:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  bmp2: TBitmap;
  w, h, i, j: Integer;
  r1, r2: PRGBArray;
begin
  w := bmp1.Width;
  h := bmp1.Height;
  bmp2 := TBitmap.Create;
  try
    bmp2.PixelFormat := pf24bit;
    bmp2.Width := w;
    bmp2.Height := h;

    for i := 0 to h-1 do
    begin
      r1 := bmp1.ScanLine[i];
      for j := 0 to w-1 do
      begin
        r2 := bmp2.ScanLine[j];
        r2[i] := r1[j];
      end;
    end;

    Image2.Canvas.Draw(0, 0, bmp2);
  finally
    bmp2.Free;
  end;
end;
```

Budeme cyklicky posúvať celé riadky o päť (resp. viac) smerom hore:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  bmp2: TBitmap;
  w, h, i: Integer;
  r1, r2: PRGBArray;
begin
  w := bmp1.Width;
  h := bmp1.Height;
  bmp2 := TBitmap.Create;
  try
    bmp2.PixelFormat := pf24bit;
    bmp2.Width := w;
    bmp2.Height := h;

    for i := 0 to h-1 do
    begin
      r1 := bmp1.ScanLine[(i+5) mod h];   // alebo (i+h div 2) mod h
      r2 := bmp2.ScanLine[i];
      Move(r1^, r2^, SizeOf(TRGB)*w);     // SizeOf(TRGB) je 3
    end;
```

```
    Image2.Canvas.Draw(0, 0, bmp2);
  finally
    bmp2.Free;
  end;
end;
```

Ak by sme chceli tú istú bitmapu posúvať viackrát, napr. pomocou časovača, budeme ju kopírovať do pomocnej bitmapy priamo z Image2:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  bmp1 := TBitmap.Create;
  bmp1.LoadFromFile('parrots.bmp');
  Image1.Canvas.Draw(0, 0, bmp1);
  Image2.Canvas.Draw(0, 0, bmp1);
  DoubleBuffered := True;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  bmp1, bmp2: TBitmap;
  w, h, i: Integer;
  r1, r2: PRGBArray;
begin
  bmp1 := TBitmap.Create;
  try
    w := Image2.Width;
    h := Image2.Height;
    bmp1.PixelFormat := pf24bit;
    bmp1.Width := w;
    bmp1.Height := h;
    bmp1.Canvas.Draw(0, 0, Image2.Picture.Graphic);

    bmp2 := TBitmap.Create;
    try
      bmp2.PixelFormat := pf24bit;
      bmp2.Width := w;
      bmp2.Height := h;

      for i := 0 to h-1 do
      begin
        r1 := bmp1.ScanLine[(i+5) mod h];
        r2 := bmp2.ScanLine[i];
        Move(r1^, r2^, 3*w);
      end;

      Image2.Canvas.Draw(0, 0, bmp2);
    finally
      bmp2.Free;
    end;
  finally
      bmp1.Free;
  end;
end;
```

Do bitmapy bmp1 by sme obsah grafickej plochy Image2 mohli dostať aj inak - pomocou Assign, napr. takto

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  bmp1, bmp2: TBitmap;
  w, h, i: Integer;
  r1, r2: PRGBArray;
begin
  bmp1 := TBitmap.Create;
  try
```

```
    w := Image2.Width;
    h := Image2.Height;
    bmp1.Assign(Image2.Picture.Graphic);
    bmp1.PixelFormat := pf24bit;

    bmp2 := TBitmap.Create;
    ...
```

Nakoľko ďalej ideme s bmp1 pracovať pomocou ScanLine, nesmieme zabudnúť po Assign nastaviť PixelFormat na 24-bitovú grafiku.

Pozrite tento veľmi zaujímavý rozmazávací efekt:

```
procedure TForm1.Button5Click(Sender: TObject);
var
  bmp2: TBitmap;
  X, Y, sX, sY: Integer;
  d1, d2: PRGBArray;
begin
  bmp2 := TBitmap.Create;
  try
    bmp2.PixelFormat := pf24bit;
    bmp2.Width := 256;
    bmp2.Height := 256;
    for Y := 0 to 255 do
    begin
      d2 := bmp2.ScanLine[Y];
      for X := 0 to 255 do
      begin
        sX := X+Random(5)-2;
        if sX < 0 then
          sX := -sX;
        if sX > 255 then
          sX := 510-sX;

        sY := Y+Random(5)-2;
        if sY < 0 then
          sY := -sY;
        if sY > 255 then
          sY := 510-sY;

        d1 := bmp1.ScanLine[sY];
        d2[X] := d1[sX];
      end;
    end;
    Image1.Canvas.Draw(0, 0, bmp2);
  finally
    bmp2.Free;
  end;
end;
```

prípadne efekt rozťahovania od stredu smerom ku okrajom:

```
procedure TForm1.Button6Click(Sender: TObject);
var
  bmp2: TBitmap;
  X, Y, sX, sY: Integer;
  d1, d2: PRGBArray;
begin
  bmp2 := TBitmap.Create;
  try
    bmp2.PixelFormat := pf24bit;
    bmp2.Width := 256;
    bmp2.Height := 256;
    for Y := 0 to 255 do
```

```
    begin
      d2 := bmp2.ScanLine[Y];
      for X := 0 to 255 do
      begin
        sX := 128+(X-128)*127 div 130;
        sY := 128+(Y-128)*127 div 130;
        d1 := bmp1.ScanLine[sY];
        d2[x] := d1[sX];
      end;
    end;
    Image1.Canvas.Draw(0, 0, bmp2);
  finally
    bmp2.Free;
  end;
end;
```

Tento posledný efekt vylepšíme takto: pridáme časovač, pre ktorý nastavíme Enabled = False a Interval = 50:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  bmp2: TBitmap;
  X, Y, sX, sY: Integer;
  d1, d2: PRGBArray;
begin
  bmp2 := TBitmap.Create;
  try
    bmp2.PixelFormat := pf24bit;
    bmp2.Width := 256;
    bmp2.Height := 256;
    for Y := 0 to 255 do
    begin
      d2 := bmp2.ScanLine[Y];
      for X := 0 to 255 do
      begin
        sX := 128+(X-128)*127 div (127+Random(5));
        sY := 128+(Y-128)*127 div (127+Random(5));
        d1 := bmp1.ScanLine[sY];
        d2[x] := d1[sX];
      end;
    end;
    d2 := bmp2.ScanLine[128];
    d2[128].R := (Random(256)+d2[128].R) div 2;
    d2[128].G := (Random(256)+d2[128].G) div 2;
    d2[128].B := (Random(256)+d2[128].B) div 2;
    bmp1.Free;
    bmp1 := bmp2;
    bmp2 := nil;
    Image1.Canvas.Draw(0, 0, bmp1);
  finally
    bmp2.Free;
  end;
end;

procedure TForm1.Button7Click(Sender: TObject);
begin
  Timer1.Enabled := not Timer1.Enabled;
end;
```

Od konštanty v Random(5) závisí, ako rýchlo sa bude útvar rozmazávať - vyskúšajte napr. Random(14). Všimnite si, že na záver do stredu bitmapy (bod [128, 128]) dávame "skoro náhodne" zafarbenú bodku.

ďalšie námety:

- premyslite, ako dorobiť všetky dnešné programy tak, aby fungovali pre ľubovoľne veľké bitmapy

- podobne ako rozmazávanie s časovačom zrealizujte aj cyklické rolovanie s časovačom
- do formulára položte aj TrackBar, pomocou ktorého budete meniť počet, o koľko riadkov sa bude rolovanie posúvať - t.j. rýchlosť posúvania
- zrealizujte čo najkrajšie otáčanie bitmapy o ľubovoľný uhol
- rozmazávanie bitmapy: každý pixel sa vypočíta ako priemer so svojimi susedmi (stačí susedov v riadku)

rozpracovaná 22. prednáška: jednoduchá animácia

čo už vieme:

- vieme v grafickej ploche kresliť a hýbať bitmapy

čo sa na tejto prednáške naučíme:

- ako zabezpečiť striedanie fáz nejakej animácie
- čo je to plánovací kalendár a ako sa dá použiť na animácie

malá animačná aplikácia

Postupne budeme vytvárať takúto aplikáciu s animovanými objektmi: v grafickej ploche je ako podklad nejaká veľká bitmapa - Image1 zaberá celú plochu formuláru (nastavili sme mu Align na alClient). Každým kliknutím myšou do plochy sa na tom mieste vytvorí animovaný obrázok, tieto budú postupne striedať svoje fázy. Každý obrázok sa bude pomaly pohybovať nejakým smerom, pričom, ak na jednej strane "vypadne" z plochy, tak sa objaví na opačnom konci. Neskôr zabezpečíme, aby sa každý obrázok mohol animovať s rôznou frekvenciou - niektoré obrázky budú striedať fázy častejšie ako iné.

Všetky obrázky z projektu si môžete stiahnuť zo súboru bitmapy.zip.

trieda animovaný obrázok

Animáciu budeme zabezpečovať cyklickým striedaním fáz - bitmáp. V prvej verzii budeme predpokladať, že každú fázu animácie máme uloženú v jednom súbore, napr.

Nakoľko chceme zabezpečiť, aby niektoré časti bitmapy boli priesvitné, pre bitmapy v našej aplikácii sa dohodneme, že "priesvitná farba" je farba bodu v ľavom hornom rohu každej bitmapy aby správne fungovalo nastavenie Transparent. Zadefinujeme triedu:

```
type
  TObrazok = class
    bm: array of TBitmap;
    x, y, f: Integer;
    c: TCanvas;
    constructor Create(cc: TCanvas; meno: String;
      pocet, xx, yy: Integer);
    destructor Destroy; override;
```

```
    procedure kresli; virtual;
    procedure krok; virtual;
  end;
```

a metódy:

```
constructor TObrazok.Create(cc: TCanvas; meno: String;
  pocet, xx, yy: Integer);
var
  i: Integer;
begin
  SetLength(bm, pocet);
  for i := 0 to High(bm) do
  begin
    bm[i] := TBitmap.Create;
    bm[i].LoadFromFile(meno+IntToStr(i)+'.bmp');
    bm[i].Transparent := True;
  end;
  x := xx;
  y := yy;
  f := 0;         // prvá fáza
  c := cc;        // zapamätáme si plochu, kde bude objekt existovať
end;

destructor TObrazok.Destroy;
var
  i: Integer;
begin
  for i := 0 to High(bm) do
    bm[i].Free;
end;

procedure TObrazok.kresli;
begin
  c.Draw(x - bm[f].Width div 2, y - bm[f].Height div 2, bm[f]);
end;

procedure TObrazok.krok;
begin
  f := (f+1) mod Length(bm);
end;
```

Vo formulári je Image1 a Timer1, ktorý má nastavený Interval na hodnotu 50. Teraz spracovanie udalostí vyzerá takto:

```
var
  a: array of TObrazok;   // zoznam všetkých animovaných objektov
  pozadie: TBitmap;       // pozadie grafickej plochy

procedure TForm1.FormCreate(Sender: TObject);
begin
  pozadie := TBitmap.Create;
  pozadie.LoadFromFile('jazero.bmp');
  Image1.Canvas.Draw(0, 0, pozadie);
  DoubleBuffered := True;
end;

procedure TForm1.FormDestroy(Sender: TObject);
var
  i: Integer;
begin
  pozadie.Free;
  for i := 0 to high(a) do
    a[i].Free;
end;
```

```
procedure TForm1.Image1MouseDown(...);
begin
  SetLength(a, Length(a)+1);
  a[High(a)] := TObrazok.Create(Image1.Canvas, 'vtak', 8, x, y);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: Integer;
begin
  if a = nil then
    Exit;  // kým je prázdny zoznam objektov, netreba nič prekresľovať
  Image1.Canvas.Draw(0, 0, pozadie);
  for i := 0 to high(a) do
  begin
    a[i].krok;
    a[i].kresli;
  end;
end;
```

Vďaka nastaveniu DoubleBuffered zabezpečíme, že animovanie obrázkov nebude blikať. V praxi sa často namiesto viac súborov s bitmapami pre fázy animácie používa jedna bitmapa, ktorá vedľa seba obsahuje všetky fázy a program si túto veľkú bitmapu "rozstrihá", napr. pre

zajo.bmp

vytvoríme takýto nový konštruktor:

```
constructor TObrazok.Create1(cc: TCanvas; meno: String;
  pocet, xx, yy: Integer);
var
  i, w, h: Integer;
  bmp: TBitmap;
begin
  SetLength(bm, pocet);
  bmp := TBitmap.Create;
  try
    bmp.LoadFromFile(meno+'.bmp');
    w := bmp.Width div pocet;
    h := bmp.Height;
    for i := 0 to High(bm) do
    begin
      bm[i] := TBitmap.Create;
      with bm[i] do
      begin
        Width := w;
        Height := h;
        Canvas.Draw(-i*w, 0, bmp);
        Transparent := True;
      end;
    end;
  finally
    bmp.Free;
  end;
  x := xx;
  y := yy;
  f := 0;
  c := cc;
end
```

a kliknutie myšou do plochy napr.

```
procedure TForm1.Image1MouseDown(...);
var
  aa: TObrazok;
begin
  case Random(2) of
    0: aa := TObrazok.Create(Image1.Canvas, 'vtak', 8, x, y);
    1: aa := TObrazok.Create1(Image1.Canvas, 'zajo', 8, x, y);
  end;
  SetLength(a, Length(a)+1);
  a[High(a)] := aa;
end;
```

uloženie bitmáp do projektu

Už viackrát by sa nám hodilo, keby sme s našim projektom, napr. **Project1.exe**, nemuseli nosiť niekedy aj väčšie množstvo bitmáp (a aj iných súborov). Možností je niekoľko, napr. do formulára hocikam položíme niekoľko komponentov Image a nastavíme im vlastnosť Visible na False (počas behu ich nebude vidieť) - nastavíme im vlastnosť Picture (dvojklikom do vnútra) na nejakú konkrétnu bitmapu a počas behu túto bitmapu môžeme z tohto Image jednoducho kopírovať.

Druhou možnosťou je vytvoriť .RES súbor, ktorý bude obsahovať všetky súbory, ktoré chceme uložiť do projektu .EXE. Takéto súbory nebudeme čítať pomocou LoadFromFile ale pomocou LoadFromResourceName. Ukážeme ako budeme postupovať pri tejto druhej možnosti: najprv pripravíme textový súbor Bitmapy.rc s týmto obsahom (bude to obyčajný textový súbor - môžeme ho vytvoriť aj v Delphi editore):

```
jazero BITMAP jazero.bmp
vtak0 BITMAP vtak0.bmp
vtak1 BITMAP vtak1.bmp
vtak2 BITMAP vtak2.bmp
vtak3 BITMAP vtak3.bmp
vtak4 BITMAP vtak4.bmp
vtak5 BITMAP vtak5.bmp
vtak6 BITMAP vtak6.bmp
vtak7 BITMAP vtak7.bmp
zajo BITMAP zajo.bmp
```

V každom riadku je trojica reťazcov: meno (pomocou neho sa budeme na toto odvolávať), BITMAP znamená typ súboru a meno súboru. Pomocou identifikátora pred slovom BITMAP sa budeme v našom programe odvolávať na danú bitmapu. Najprv do **Project1.dpr** pridáme jeden riadok:

```
program Project1;

{$R 'Bitmapy.res' 'Bitmapy.rc'}

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

a všetky LoadFromFile pozmeníme takto:

```
constructor TObrazok.Create(cc: TCanvas; meno: String;
  pocet, xx, yy: Integer);
...
    bm[i] := TBitmap.Create;
    bm[i].LoadFromResourceName(HInstance, meno+IntToStr(i));
...

constructor TObrazok.Create1(cc: TCanvas; const meno: String;
  pocet, xx, yy: Integer);
...
  try
    b.LoadFromResourceName(HInstance, meno);
...

procedure TForm1.FormCreate(Sender: TObject);
begin
  pozadie := TBitmap.Create;
  pozadie.LoadFromResourceName(HInstance, 'jazero');
...
```

všimnite si, že po prvej kompilácii sa automaticky vytvoril dosť veľký súbor Bitmapy.res - tento už obsahuje všetky bitmapy - výsledná aplikácia (Project1.exe) už nepotrebuje na spustenie veľké množstvo bitmáp, už ich má uložené v sebe

súbor Bitmapy.rc aj riadok v projekte *{$R 'Bitmapy.res' 'Bitmapy.rc'}* môžeme teraz vyhodiť a namiesto toho niekam nižšie buď v Project1.dpr alebo do Unit1.pas dáme riadok *{$R Bitmapy.res}*

pohyb animovaných obrázkov

Každému objektu obrázok pridáme tri nové stavové premenné - informácie o smere a rýchlosti pohybu (dx, dy: Integer) a tiež o obdĺžniku (obd: TRect), v ktorom sa bude tento obrázok pohybovať. Ďalej urobíme taký efekt, že keď objekt vypadne z obdĺžnika, tak sa objaví na opačnej strane (tzv. wrap - efekt). TRect už poznáme - špecifikuje obdĺžnik = record Left, Top, Right, Bottom: Integer; end. Pridáme metódu zmenDXY a vylepšíme metódu krok:

```
procedure TObrazok.zmenDXY(dxx, dyy: Integer; r: TRect);
begin
  dx := dxx;
  dy := dyy;
  obd := r;
end;

procedure TObrazok.krok;
begin
  f := (f+1) mod Length(bm);
  if (dx = 0) and (dy = 0) then
    Exit;
  Inc(x, dx);
  Inc(y, dy);
  with obd do
  begin
    if x < Left then
      Inc(x, Right-Left);
    if x >= Right then
      Dec(x, Right-Left);
    if y < Top then
      Inc(y, Bottom-Top);
    if y >= Bottom then
      Dec(y, Bottom-Top);
  end;
end;
```

Zrejme konštruktor bude dx a dy inicializovať na 0.

Pri vytvorení nového obrázka mu nastavíme pohyb aj jeho obdĺžnik, napr.

```
procedure TForm1.Image1MouseDown(...);
var
  aa: TObrazok;
begin
  case Random(2) of
    0:
      begin
        aa := TObrazok.Create(Image1.Canvas, 'vtak', 8, x, y);
        aa.zmenDXY(Random(5), Random(5)-2, Image1.ClientRect);
      end;
    1:
      begin
        aa := TObrazok.Create1(Image1.Canvas, 'zajo', 8, x, y);
        aa.zmenDXY(-Random(5)-1, Random(3)-1,
             Rect(0, 400, Image1.ClientWidth, Image1.ClientHeight));
      end;
  end;
  SetLength(a, Length(a)+1);
  a[High(a)] := aa;
end;
```

Animované obrázky sa budú teraz hýbať.

animovaný obrázok s iným pohybom

Do projektu pridáme nový obrázok - zemeguľu a zmeníme jej správanie tak, že na okraji plochy sa bude odrážať:

```
type
  TObrazok1 = class(TObrazok)
    procedure krok; override;
  end;

procedure TObrazok1.krok;
begin
  f := (f+1) mod Length(bm);
  if (dx = 0) and (dy = 0) then
    Exit;
  Inc(x, dx);
  Inc(y, dy);
  with obd do
  begin
    if (x < Left) or (x >= Right) then
      dx := -dx;
    if (y < Top) or (y >= Bottom) then
      dy := -dy;;
  end;
end;

...

procedure TForm1.Image1MouseDown(...);
var
  aa: TObrazok;
  obdl: TRect;
begin
  case Random(3) of
    0:
      begin
        aa := TObrazok.Create(Image1.Canvas, 'vtak', 8, x, y);
        aa.zmenDXY(Random(5), Random(5)-2, Image1.ClientRect);
      end;
```

```
    1:
      begin
        aa := TObrazok.Create1(Image1.Canvas, 'zajo', 8, x, y);
        aa.zmenDXY(-Random(5)-1, 0,
          Rect(0, 400, Image1.ClientWidth, Image1.ClientHeight));
      end;
    2:
      begin
        aa := TObrazok1.Create1(Image1.Canvas, 'zemegula', 21, x, y);
        obdl := Rect(50, 50,
                  Image1.ClientWidth-50, Image1.ClientHeight-50);
        aa.zmenDXY(Random(5)-2, Random(5)-2, obdl);
      end;
  end;
  SetLength(a, Length(a)+1);
  a[High(a)] := aa;
end;
```

nezabudneme aj zemegula.bmp pridať do Bitmapy.rc

plánovač

Plánovačom bude špeciálny **front** (rad), do ktorého sa pridáva nie na koniec, ale na správne miesto podľa času. Preto metóda insert musí najprv vyhľadať v rade správne miesto, kam treba zaradiť prichádzajúcu požiadavku, potom na tomto mieste celý rad roztiahne a až na toto nové miesto zaradí túto novú položku. Plánovač teraz vyzerá takto:

```
unit QueueUnit;

interface

type
  TQueue = class
    q: array of record
      tim: TDateTime;
      ob: TObject;
    end;
    procedure insert(tik: Integer; ob: TObject);
    procedure serve(var ob: TObject);
    function first: TObject;
    function toptime: TDateTime;
    function empty: Boolean;
  end;

var
  q: TQueue;

implementation

uses
  SysUtils;

procedure TQueue.insert(tik: Integer; ob: TObject);
var
  i: Integer;
  tim: TDateTime;
begin
  tim := Now + tik/MSecsPerDay;
  i := High(q);
  SetLength(q, Length(q)+1);
  while (i >= 0) and (q[i].tim > tim) do
  begin
    q[i+1] := q[i];
    Dec(i);
```

```
  end;
  q[i+1].tim := tim;
  q[i+1].ob := ob;
end;

procedure TQueue.serve(var ob: TObject);
begin
  if empty then
    ob := nil
  else
  begin
    ob := q[0].ob;
    q := Copy(q, 1, MaxInt);
  end;
end;

  // first je ako serve, len hodnotu vráti ako výsledok funkcie
function TQueue.first: TObject;
begin
  if empty then
    Result := nil
  else
  begin
    Result := q[0].ob;
    q := Copy(q, 1, MaxInt);
  end;
end;

function TQueue.toptime: TDateTime;
begin
  if empty then       // čas niekedy v budúcnosti
    Result := Now+1
  else
    Result := q[0].tim;
end;

function TQueue.empty: Boolean;
begin
  Result := q=nil;
end;

end.
```

Všimnite si, že sme nepotrebovali vytvoriť konštruktor Create - spoľahli sme sa na to, že Delphi automaticky inicializujú stavové premenné, ktoré sú dynamické polia, reťazce a objekty - tieto majú hodnotu nil, resp. prázdny reťazec. Metódu insert by sme mohli prepísať aj s použitím štandardnej procedúry Move, ktorá tu rozsúvanie prvkov v poli urobila rýchlejšie, napr. takto:

```
procedure TQueue.insert(tik: Integer; ob: TObject);
var
  i: Integer;
  tim: TDateTime;
begin
  tim := Now + tik/MSecsPerDay;
  i := 0;
  while (i <= High(q)) and (q[i].tim <= tim) do
    Inc(i);
  SetLength(q, Length(q)+1);
  Move(q[i], q[i+1], (Length(q)-i-1)*SizeOf(q[0]));
  q[i].tim := tim;
  q[i].ob := ob;
end;
```

Pri definovaní takéhoto plánovača sme nešpecifikovali aké objekty budeme do neho ukladať a z neho vyberať -

použili sme univerzálne TObject.

Všetkým objektom animovaný obrázok pridáme novú stavovú premennú tik, ktorá bude obsahovať čas v ms na zmenu ďalšej fázy:

```
type
  TObrazok = class
    bm: array of TBitmap;
    x, y, f, dx, dy: Integer;
    obd: TRect;
    tik: Integer;
    constructor Create(cc: TCanvas; meno: String;
      pocet, xx, yy: Integer);
    constructor Create1(cc: TCanvas; meno: String;
      pocet, xx, yy: Integer);
    destructor Destroy; override;
    procedure zmenDXY(dxx, dyy: Integer; r: TRect); virtual;
    procedure kresli(c: TCanvas); virtual;
    procedure krok; virtual;
  end;

constructor TObrazok.Create(cc: TCanvas; meno: String;
  pocet, xx, yy: Integer);
...
  tik := 0;
end;

constructor TObrazok.Create1(cc: TCanvas; meno: String;
  pocet, xx, yy: Integer);
...
  tik := 0;
end;
```

Premennú tik v konštruktoroch nemusíme inicializovať na 0, lebo aj tak sa automaticky vynuluje.

Do metódy krok doplníme "naplánovanie" ďalšieho volania krok:

```
procedure TObrazok.krok;
begin
  f := (f+1) mod Length(bm);
  if tik > 0 then
    q.insert(tik, self);
  if (dx <> 0) or (dy <> 0) then
  begin
    Inc(x, dx);
    Inc(y, dy);
    with obd do
    begin
      if x < Left then
        Inc(x, Right-Left);
      if x >= Right then
        Dec(x, Right-Left);
      if y < Top then
        Inc(y, Bottom-Top);
      if y >= Bottom then
        Dec(y, Bottom-Top);
    end;
  end;
end;

procedure TObrazok1.krok;
begin
  f := (f+1) mod Length(bm);
  if tik > 0 then
    q.insert(tik, self);
  ...
```

```
end;
```

Nezabudneme do FormCreate pridať q := TQueue.Create. Pri vytváraní objektov im hneď vygenerujeme ich časový interval (rýchlosť animácie):

```
procedure TForm1.Image1MouseDown(...);
var
  aa: TObrazok;
  obdl: TRect;
begin
  case Random(3) of
    0:
      begin
        aa := TObrazok.Create(Image1.Canvas, 'vtak', 8, x, y);
        aa.zmenDXY(Random(5), Random(5)-2, Image1.ClientRect);
        aa.tik := 50+50*Random(6);
      end;
    1:
      begin
        aa := TObrazok.Create1(Image1.Canvas, 'zajo', 8, x, y);
        aa.zmenDXY(-Random(5)-1, Random(5)-2, Image1.ClientRect);
        aa.tik := 100+10*Random(6);
      end;
    2:
      begin
        aa := TObrazok1.Create1(Image1.Canvas, 'zemegula', 21, x, y);
        obdl := Rect(50, 50,
                  Image1.ClientWidth-50, Image1.ClientHeight-50);
        aa.zmenDXY(Random(5)-2, Random(5)-2, obdl);
        aa.tik := 10;
      end;
  end;
  SetLength(a, Length(a)+1);
  a[High(a)] := aa;
  if aa.tik > 0 then     // aby sa naštartovala animácia
    q.insert(0, aa);
end;
```

ešte časovač:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: Integer;
  zmena: Boolean;
// p: TObject;
begin
  zmena := False;
  while not q.empty and (q.toptime <= Now) do
  begin
    // q.serve(p); TObrazok(p).krok;
    TObrazok(q.first).krok;
    zmena := True;
  end;
  if zmena then
  begin
    Image1.Canvas.Draw(0, 0, pozadie);
    for i := 0 to high(a) do
      a[i].kresli;
  end;
end;
```

Plochu prekresľujeme, len ak sa na nej niečo zmenilo (premenná zmena). Všimnite si spôsob, ako manipulujeme s objektom, ktorý sme vybrali z frontu. Buď použijeme pomocnú premennú p typu TObject:

```
q.serve(p);
TObrazok(p).krok;
```

alebo novú metódu funkciu first triedy TQueue, ktorá pracuje podobne ako serve, len vybratý prvok nevráti ako parameter ale ako hodnotu funkcie:

```
TObrazok(q.first).krok;
```

ďalšie námety:

- objektu môžeme naplánovať viac rôznych akcií v rôznych časoch - vymyslite, ako plánovaču (teda pre časovač) povedať, že má pre rôzne objekty spúšťať rôzne akcie (nielen krok)
- stavová premenná tik, ktorá sa používa ako čas na prechod do nasledujúcej fázy, by mohla byť buď dynamickým poľom (pre každú fázu iný čas) alebo funkciou, ktorá závisí aj od iných okolností - premyslite túto ideu

Ešte jeden príklad na použitie radu

Ukážeme riešenie takejto úlohy: do grafickej plochy môžeme kresliť čierne čiary pomocou ľavého tlačidla myši. Keď niekam klikneme pravým tlačidlom myši, tak sa od tohto bodu začne na všetky strany šíriť nejaká farba a toto vypĺňanie sa zastaví až na čiernej čiare alebo nejako zafarbenej oblasti. Pravým tlačidlom myši môžeme naraz naštartovať viac takýchto vypĺňacích akcií. Využijeme obyčajnú dátovú štruktúru rad. Ukážme najprv klasický rad (pamätáme si súradnice bodu a farbu):

```
type
  TQueue = class
    q: array of record
      x, y: Integer;
      col: TColor;
    end;
    procedure append(x, y: Integer; col: TColor);
    procedure serve(var x, y: Integer; var col: TColor);
    function empty: Boolean;
  end;

procedure TQueue.append(x, y: Integer; col: TColor);
begin
  SetLength(q, Length(q)+1);
  q[High(q)].x := x;
  q[High(q)].y := y;
  q[High(q)].col := col;
end;

procedure TQueue.serve(var x, y: Integer; var col: TColor);
begin
  if empty then
  begin
    showmessage('empty');
    Exit;
  end;
  x := q[0].x;
  y := q[0].y;
  col := q[0].col;
  Move(q[1], q[0], (Length(q)-1)*Sizeof(q[0]));
  SetLength(q, Length(q)-1);
end;

function TQueue.empty: Boolean;
begin
  Result := q=nil;
```

```
end;
```

Využijeme komponent TTimer s nastaveným Interval napr. na 10. Metóda Timer1Timer teraz vyzerá takto:

```
var
  q: TQueue;
  kreslim: Boolean;   // či sa kreslí ťahaním myši

procedure TForm1.FormCreate(Sender: TObject);
begin
  q := TQueue.Create;
  Randomize;
  DoubleBuffered := True;
end;

procedure TForm1.Image1MouseDown(...);
const
  col: array [0..5] of TColor =
        (clRed, clBlue, clGreen, clYellow, clAqua, clLime);
begin
  kreslim := Shift = [ssLeft];
  if kreslim then
    Image1.Canvas.MoveTo(x, y)
  else if Shift = [ssRight] then
    q.append(x, y, col[Random(Length(col))]);
end;

procedure TForm1.Image1MouseMove(...);
begin
  if kreslim then
    Image1.Canvas.LineTo(x, y);
end;

procedure TForm1.Image1MouseUp(...);
begin
  kreslim := False;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  x, y: Integer;
  col: TColor;
begin
  if q.empty then
    Exit;
  q.serve(x, y, col);
  if Image1.Canvas.Pixels[x, y] <> clWhite then
    Exit;
  Image1.Canvas.Pixels[x, y] := col;
  q.append(x-1, y, col);
  q.append(x+1, y, col);
  q.append(x, y-1, col);
  q.append(x, y+1, col);
end;
```

Takéto vyfarbovanie plochy nejakými farbami je veľmi pomalé, ale rôznymi vylepšeniami sa môže urýchliť.

ApplicationEvents onIdle

Túto istú úlohu môžeme riešiť aj tak, že namiesto TTimer použijeme nový komponent TApplicationEvents. Timer1 z formulára vyhodíme a nahradíme ho TApplicationEvents z palety **Additional**. Rovnako ako Timer je to nevizuálny komponent, teda ho za behu programu nebude vidieť. Využijeme jeho udalosť onIdle, ktorá bude zavolaná podobne ako onTimer pre časovač, ale vždy vtedy, keď "Windows nemajú čo robiť" - t.j. keď momentálne

nie sú zaťažované inými úlohami. Hneď ako príde nejaká iná udalosť (napr. pohneme alebo klikneme myšou, alebo prišiel tik od časovača a pod.), ďalšie volanie onIdle sa už nezrealizuje. Teda do onIdle dáme spracovanie jedného bodu z frontu čakajúcich bodov a oznámime systému, že nás má volať stále, kým systém nemá čo robiť (nastavíme premennú Done na False) - Done s hodnotou True znamená, že onIdle sa najbližšie zavolá až po nejakej inej udalosti, keď už systém opäť nebude mať čo robiť. Namiesto metódy Timer1Timer teda napíšeme ApplicationEvents1Idle:

```
procedure TForm1.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
var
  x, y: Integer;
  col: TColor;
begin
  Done := q.empty;
  if Done then
    Exit;
  q.serve(x, y, col);
  if Image1.Canvas.Pixels[x, y] <> clWhite then
    Exit;
  Image1.Canvas.Pixels[x, y] := col;
  q.append(x-1, y, col);
  q.append(x+1, y, col);
  q.append(x, y-1, col);
  q.append(x, y+1, col);
end;
```

ďalšie námety:

- urýchlite prácu s radom - pole zväčšujte napr. s krokom 100, pamätajte si začiatok a počet prvkov a pole posúvajte len v nutných prípadoch
- ďalšie urýchlenie: v metóde Timer1Timer, resp. ApplicationEvents1Idle spracujte nielen jeden bod, ale napr. v cykle naraz 20
- vymyslite iné pravidlá farbenia: nielen keď je biela farba, ale napr. nie čierna a rôzna od aktuálnej a pod.