Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

ŠTÁTNY PEDAGOGICKÝ ÚSTAV
NATIONAL INSTITUTE FOR EDUCATION

Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Programovanie 9

Predmet: Programovanie

Línia: Vlastný odborový kontext informatiky a informatickej výchovy

Operačný program VZDELÁVANIE

EURÓPSKA ÚNIA
Európsky sociálny fond

Európska únia
Európsky sociálny fond

Moderné vzdelávanie pre vedomostnú spoločnosť/Projekt je spolufinancovaný zo zdrojov ES

Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Programovanie 9

Identifikácia modulu

Aktivita projektu:	1.2 Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ
Línia aktivity:	Informatika
Predmet:	Programovanie

Garant predmetu:

RNDr. Andrej Blaho
KAI FMFI UK, Bratislava
andrej.blaho@gmail.com

Autori:

RNDr. Andrej Blaho
KAI FMFI UK, Bratislava
RNDr. Ľubomír Salanci, PhD.
KZVI FMFI UK, Bratislava

Zaradenie modulu

Moduly Programovanie 1 až Programovanie 9 nadväzujú na modul Programovanie 0. Všetky spolu vytvárajú ucelený kurz programovania, ktorý pokrýva stredoškolský obsah programovania aj s množstvom metodicky vhodných úloh, problémov a projektov. Všetkých 9 modulov je v prvých dvoch semestroch štúdia, nakoľko na nich nadväzujú ďalšie predmety z informatickej línie ale aj z didaktiky programovania.

Druhý semester vzdelávania:

Mat 2	DG 4	DG 5	Did. Inf. 1	Did. Inf. 2	Mod. škola 3
Prog 5	Prog 6	Prog 7	Prog 8	Prog 9	OS 1

Každý programátorský modul obsahuje 2 témy, ktoré môžu, ale aj nemusia spolu súvisieť. Vyplýva to z predpokladanej organizácie štúdia, keď predpokladáme 2 štvorhodinové bloky, pričom každý bude obsahovať nejakú časť prednášky, cvičení a laboratórnej práce pri počítači.

Abstrakt modulu

Modul sa venuje definovaniu vlastných funkcií. Zoznamuje s mechanizmom volania funkcií a s princípom odovzdávania výslednej hodnoty funkcie. Predvádza spôsob zápisu funkcií s najrôznejšími typmi parametrov a tiež s rôznymi typmi výsledku.

Obsah

Programovanie 9 ... 1
- Identifikácia modulu ... 1
- Zaradenie modulu ... 1
- Abstrakt modulu ... 1
- Obsah ... 2
- Úvod ... 3
- Cieľ modulu ... 3
- Vstupné vedomosti ... 3
  - Požadované prerekvizity ... 3
  - Predpokladané vstupné vedomosti, skúsenosti a zručnosti ... 3
  - Preverenie vstupných vedomostí ... 3
- 1. tematická jednotka – funkcie s jednoduchými typmi ... 4
  - 1. Jednoduché číselné funkcie ... 4
  - 2. Funkcia so zložitejším výpočtom ... 9
  - 3. Logické funkcie ... 16
  - 4. Znakové funkcie ... 21
- 2. tematická jednotka – funkcie so zloženými typmi ... 23
  - 1. Funkcie s reťazcami ... 23
  - 2. Funkcie spracovávajúce pole ... 28
  - 3. Pole ako výsledok funkcie ... 31
  - 4. Pole cifier ... 35
- Čo sme sa naučili v tomto module ... 39
  - Preverenie výstupných vedomostí ... 39
- Literatúra a použité zdroje ... 39

Úvod

Modul sa skladá z dvoch tematických jednotiek každá približne rozsahu 3 až 5 vyučovacích hodín. Obe jednotky pokrývajú tieto témy:

1. tematická jednotka – **Funkcie s jednoduchými typmi**
   - Jednoduché číselné funkcie
   - Funkcia so zložitejším výpočtom
   - Logické funkcie
   - Znakové funkcie
2. tematická jednotka – **Funkcie so zloženými typmi**
   - Funkcie s reťazcami
   - Funkcie spracovávajúce pole
   - Pole ako výsledok funkcie
   - Pole cifier

Cieľ modulu

Po absolvovaní tohto modulu sa od účastníka očakáva, že

- pre danú úlohu bude schopný zadefinovať funkcie na riešenie podúloh,
- správne zvolí typy parametrov a výsledku funkcie,
- bude rozumieť mechanizmu volania funkcií a spracovania návratovej hodnoty,
- bude schopný správne pracovať s premennou výsledku,
- bude schopný navrhnúť funkcie, ktoré pracujú aj so zložitejšími dátovými typmi.

Vstupné vedomosti

Požadované prerekvizity

Všetky moduly Programovanie 1 - 8.

Predpokladané vstupné vedomosti, skúsenosti a zručnosti

Predpokladáme, že účastník vzdelávania:

- vie popísať princíp práce s textovým súborom,
- vie správne zvoliť otvorenie súboru buď na čítanie alebo na zápis,
- dokáže analyzovať zadanie a tiež možné chyby v riešení, chyby vie nájsť a opraviť,
- dokáže vytvoriť aplikácie, ktoré naraz pracujú s viacerými textovými súbormi,
- rozumie rozdielom medzi rôznymi dátovými typmi, ktoré sa ukladajú, resp. čítajú z textového súboru.

Preverenie vstupných vedomostí

Účastník vzdelávania vie naprogramovať takúto aplikáciu:

> Textový súbor **subor.txt** obsahuje celé čísla aj nejaké texty. Prekopírujte ho do dvoch súborov takto: do súboru **cisla.txt** zapíšte len všetky čísla a navzájom ich oddeľte medzerami. Do súboru **text.txt** prekopírujte pôvodný text ale už bez čísel, ktoré sme zapísali do prvého výstupného súboru.

1. tematická jednotka – funkcie s jednoduchými typmi

Zopár funkcií s jedným parametrom, ktoré by sme už mohli poznať:

- IntToStr, FloatToStr
- StrToInt, StrToFloat
- RGBToColor
- Red, Green, Blue
- Round, Sqr, Sqrt
- Sin, Cos
- Random, Odd
- Ord, Char, UpCase

Niektoré funkcie môžu mať aj zložitejšie parametre, napr.

- Eof, Eoln
- SeekEof, SeekEoln
- Copy, Pos
- Length, High, Low

Funkcia je taký typ podprogramu, ktorý niečo počíta a nakoniec **vráti** nejakú hodnotu. S touto hodnotou ďalej pracujeme vo výrazoch, v testoch, v priradeniach a pod. Pascal má veľa dopredu zadefinovaných (preddefinovaných) funkcií, ktoré sme už používali od prvých stretnutí s programovaním.

Tieto funkcie majú jeden parameter, ktorý je buď číslo (**Integer** alebo **Real**), alebo znak (**Char**) alebo znakový reťazec (**string**). Výsledný typ funkcie je opäť rôzny: číslo, znak, znakový reťazec alebo logická hodnota. Niektoré funkcie sú zaujímavé tým, že parametrom môže byť nielen jednoduchá hodnota, ale aj premenná textový súbor (napr. funkcie **Eof**, **Eoln**), ale aj ľubovoľné pole (funkcie **Length**, **High**, **Low**). Niektoré funkcie môžu mať aj viac parametrov (napr. **Copy**, **Pos**) a vtedy zrejme záleží na ich poradí.

V tomto module sa naučíme vytvárať vlastné funkcie a budeme rozumieť princípom, ako to funguje. Definovanie funkcií sa veľmi podobá definovaniu procedúr. Procedúry aj funkcie sú podprogramy. Preto, keď si pripomenieme, čo už vieme o procedúrach, veľmi nám to zjednoduší pochopenie funkcií:

- ako definujeme procedúry,
- ako zapisujeme parametre ,
- ako voláme procedúry, t.j. mechanizmus volania procedúr,
- ako fungujú lokálne premenné procedúr,
- aký je rozdiel medzi lokálnymi a globálnymi procedúrami.

Tieto skúsenosti s procedúrami využijeme aj pri funkciách.

1. Jednoduché číselné funkcie

.

Funkcie sú teda také podprogramy, ktoré budú vytvárajú nejakú hodnotu. Typ parametrov a aj typ výsledku treba zadefinovať v hlavičke funkcie:

```
function meno_funkcie(parametre): typ_výsledku;
```

Od hlavičky procedúry sa toto líši len slovom **function** namiesto **procedure** a typom funkcie za dvojbodkou. Aj ďalej sa funkcia podobá na procedúru s jedným veľmi dôležitým rozdielom: musíme určiť, aká bude výsledná hodnota funkcie. V tele funkcie máme k dispozícii špeciálnu premennú **Result** – túto nedeklarujeme, pretože to za nás spravil Pascal. Táto špeciálna premenná sa správa rovnako ako obyčajná lokálna premenná:

- je rovnakého typu ako typ funkcie,
- na začiatku má nedefinovanú hodnotu,
- hodnota, ktorú má **Result** pri skončení podprogramu sa zapamätá a teda vráti ako hodnota funkcie.

Vysvetlime to na tomto jednoduchom príklade:

```
function Mocnina(X: Integer): Integer;
begin
  Result := X * X;
end;
```

Tento zápis znamená:

- definujeme funkciu (podprogram) s menom **Mocnina**,
- podprogram má jeden celočíselný parameter **X**,
- výsledkom funkcie bude tiež celé číslo,
- telo tejto funkcie sa skladá z jediného priraďovacieho príkazu: do špeciálnej premennej **Result** priraďujeme výsledok funkcie, teda druhú mocninu parametra **X**.

Teraz, ak by sme chceli použiť túto funkciu, zapíšeme napr.:

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

  function Mocnina(X: Integer): Integer;
  begin
    Result := X * X;
  end;

var
  I, Sucet: Integer;
begin
  Sucet := 0;
  for I := 1 to 10 do
    Sucet := Sucet + Mocnina(I);
  Memo1.Lines.Append('súčet = ' + IntToStr(Sucet));
end;
```

Po spustení program vypíše:

```
súčet = 385
```

Tento prvý testovací program 10-krát zavolal túto novú funkciu **Mocnina** a pri každom tomto volaní vypočítal príslušnú mocninu čísla a výsledok pripočítal k doterajšiemu súčtu. Každé volanie funkcie **Mocnina** beží podobne ako už poznáme mechanizmus volania procedúr. Tento mechanizmus volania procedúr upravíme tak, aby fungoval pre funkcie. Teda pri **volaní funkcie** sa postupne:

1. **zapamätá sa návratové miesto**, t.j. riadok programu, kam sa bude treba vrátiť po skončení funkcie;
2. **vytvoria sa všetky zadeklarované lokálne premenné** funkcie - tieto majú nedefinovanou hodnotou;
   - aj parametre sú lokálne premenné, preto sa na tomto mieste **vytvoria aj všetky parametre** (premenné) a do nich sa priradia príslušné vstupné hodnoty
   - okrem toho sa automaticky vytvorí špeciálna lokálna premenná **Result**, ktorá je rovnakého typu ako typ funkcie - zatiaľ má nedefinovanú hodnotu;
3. prenesie sa **riadenie** programu **do tela podprogramu** (za príslušný **begin**) a vykonajú sa všetky príkazy podprogramu (až po koncový **end**);
4. **zrušia sa lokálne premenné** - a teda sa zrušia aj parametre;
   - pri tom sa zapamätá hodnota špeciálnej premennej **Result**;
5. **riadenie sa vráti** za miesto v programe, odkiaľ bol podprogram volaný - t.j. **na návratové miesto** - sem sa dosadí zapamätaný výsledok funkcie.

Takto zapísanú funkciu môžeme používať v najrôznejších miestach nášho programu, napr. vo výrazoch, v testoch, ako parametre procedúr, pri výpise a pod. Napr.

```
A := Mocnina(15)-
      Mocnina(12);

if Mocnina(X) > 50 then
  X := 0;

Image1.Canvas.
  LineTo(I, Mocnina(I));

Writeln(Subor, X:5,
  Mocnina(X):8);
```

Takže, čo sa naozaj stane v počítači, keď program príde na riadok, kde je volanie funkcie:

```
Sucet := Sucet + Mocnina(I);
```

Z bodov mechanizmu volania funkcie vidíme, že

1. zapamätá sa pozícia tohto riadka programu, aby sa sem výpočet vedel vrátiť (tzv. návratové miesto);
2. vytvoria sa všetky lokálne premenné: funkcia **Mocnina** má jeden parameter **X** - to je prvá lokálna premenná, okrem toho sa automaticky vytvorí aj druhá lokálna premenná **Result**,
   - premenná **X** dostáva hodnotu vstupnej premennej **I**, teda 1;
   - premenná **Result** má zatiaľ nedefinovanú hodnotu;
3. vykoná sa jediný príkaz tela funkcie, teda do premennej **Result** sa priradí hodnota **X * X**, t.j. v **Result** je teraz 1;
4. zapamätá sa výsledok funkcie (obsah premennej **Result**) a všetky lokálne premenné sa zrušia (**X** aj **Result**);
5. pokračuje sa v rozrobenom priraďovacom príkaze, z ktorého sa skočilo do funkcie **Mocnina**. - k premennej **Sucet** sa pripočíta 1 (výsledok volania funkcie **Mocnina(1)**).

My poznáme funkcie aj z matematiky. Napr. matematické zápisy

$$f(x) = x^2 + 3x - 2$$
$$g(x) = 2x^2 - 5x + 3$$

môžeme zapísať aj v Pascale a potom môžeme s takýmito funkciami riešiť rôzne úlohy. Vytvorme teraz textový súbor, ktorý bude obsahovať tabuľku hodnôt pre obe funkcie. Prvý stĺpec tabuľky bude obsahovať premennú **X** z intervalu -1 až 1 s krokom 0.01. V ďalších dvoch stĺpcoch budú hodnoty **f(x)** a **g(x)**. Tie budeme vypisovať s presnosťou 4 desatinné miesta. Posledný stĺpec bude obsahovať rozdiel týchto dvoch funkčných hodnôt.

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

  function F(X: Real): Real;
  begin
    Result := X * X + 3 * X - 2;
  end;

  function G(X: Real): Real;
  begin
    Result := 2 * X * X - 5 * X + 3;
  end;

var
  X: Real;
  T: TextFile;
begin
  AssignFile(T, 'tabulka.txt');
  Rewrite(T);
  X := -1;
  while X <= 1 do
  begin
    WriteLn(T, X:7:2, F(X):10:4, G(X):10:4, F(X) - G(X):10:4);
    X := X + 0.01;
  end;
  CloseFile(T);
  Memo1.Lines.LoadFromFile('tabulka.txt');
end;
```

Prvé riadky tabuľky v súbore vyzerajú takto:

```
   -1.00    -4.0000   10.0000   -14.0000
   -0.99    -3.9899    9.9102   -13.9001
   -0.98    -3.9796    9.8208   -13.8004
   -0.97    -3.9691    9.7318   -13.7009
   -0.96    -3.9584    9.6432   -13.6016
   -0.95    -3.9475    9.5550   -13.5025
...
```

Funkcie sa veľmi výhodne používajú pri generovaní náhodných čísel. Napríklad niekedy sa nám môže zísť generovať náhodné číslo, ktoré ale nebude od 0 ale od 1:

Funkciu **Nahodne** môžeme využiť napr. pre nastavenie náhodnej hrúbky pera:

```
Image1.Canvas.Pen.
  Width := Nahodne(10);
```

```
function Nahodne(N: Integer): Integer;
begin
  Result := Random(N) + 1;
end;
```

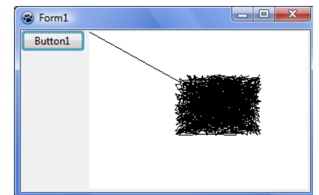
Inokedy sa môže hodiť náhodný generátor pre čísla z intervalu od A do B:

```
function Nahodne(A, B: Integer): Integer;
begin
  Result := Random(B - A + 1) + A;
end;
```

Funkciu môžeme otestovať pri kreslení náhodných čiar:

```
procedure Form1.Timer1Timer(Sender: TObject);
begin
  Image1.Canvas.LineTo(Nahodne(100, 200), Nahodne(50, 120));
end;
```

Po spustení vidíme náhodné intervaly pre x-ovú aj y-ovú súradnicu:



Funkcie, podobne ako aj procedúry, môžu mať ľubovoľný počet parametrov. Hoci procedúry bez parametrov sú dosť bežné, funkcie bez parametrov sú dosť zriedkavé. Funkcia bez parametrov sa môže využiť, napr. pre označenie nejakej významnej konštanty. Zapíšme funkciu, ktorá vráti približnú hodnotu Ludolfovho čísla **π**:

```
function PI: Real;
begin
  Result := 3.1415926;
end;
```

Môžeme zapísať, napr.

```
X := 100 * Sin(U/180*PI);
```

Hoci takáto funkcia je vo FreePascale už definovaná, my na nej môžeme vidieť spôsob zápisu funkcie bez parametrov.

Funkcia bez parametrov sa môže výhodne využiť s generátorom náhodných čísel. Zapíšme funkciu, ktorá pri každom zavolaní vygeneruje náhodné číslo od 1 do 6, ktoré označuje hod hracou kockou:

```
function HodKockou: Integer;
begin
  Result := Random(6) + 1;
end;
```

Môžeme zapísať, napr.

```
if HodKockou = 6 then
  NovaFigurka;
```

Ďalším príkladom je generovanie náhodnej farby:

```
function NahodnaFarba: TColor;
begin
  Result := Random(256 * 256 * 256);
end;
```

Každé zavolanie tejto funkcie náhodne zvolí nejakú farbu. Niektorí programátori obľubujú písať okrúhle zátvorky, aj keď funkcia nemá parametre, napr. hlavičku funkcie

```
function NahodnaFarba(): TColor;
```

Je to len vecou zvyku. Okrúhle zátvorky môžete (nemusíte) písať aj pri volaní tejto funkcie, napr.

```
Image1.Canvas.Pen.Color := NahodnaFarba();
Image1.Canvas.Brush.Color := NahodnaFarba;
Image1.Canvas.Font.Color := NahodnaFarba;
```

Čo sme sa naučili

Ako sa deklaruje funkcia, ako sa zapisujú parametre, aký je spôsob zápisu výsledku funkcie. Funkcie môžu byť bez parametrov, ale môžu mať aj viac parametrov. V tele funkcie musíme zapísať priradenie do premennej **Result**. Týmto priradením definujeme výsledok funkcie.

V tejto časti sme ukázali nasledovné techniky:

- mechanizmus volania funkcií

Úlohy na precvičenie

Zadanie 1	Napíšte funkciu **CS2** s jedným celočíselným parametrom. Predpokladáme, že parameter je dvojciferné číslo. Funkcia spočíta jeho ciferný súčet.
Zadanie 2	Napíšte funkcie na výpočet • pre 2 parametre: obsah a obvod obdĺžnika • pre 1 parameter: obsah a obvod kruhu • pre 3 parametre: objem a povrch kvádra, • pre 1 parameter: objem gule
Zadanie 3	Napíšte funkciu ktorá pre dva body v rovine (X1, Y1) a (X2, Y2) vypočíta ich vzdialenosť.
Zadanie 4	Napíšte funkciu, ktorá vygeneruje náhodné reálne číslo z intervalu od 0 do 1. Náhodné číslo bude na 2 desatinné miesta.
Zadanie 5	Napíšte funkciu, ktorá vygeneruje náhodnú farbu, ale iba v červenom odtieni (náhodne sa nastavuje len červená zložka **RGB**).
Zadanie 6	Napíšte funkciu **Nahodne3** bez parametrov, ktorá náhodne vygeneruje jednu z troch hodnôt -1, 0 a 1.
Zadanie 7	Napíšte funkciu **Nahodne2** bez parametrov, ktorá náhodne vygeneruje jednu z dvoch hodnôt -1 a 1.
Zadanie 8	Napíšte funkciu **HodMincou** bez parametrov, ktorá náhodne vygeneruje jednu z dvoch hodnôt 0 a 1.
Zadanie 9	Napíšte funkciu **HodMincouX** bez parametrov, ktorá náhodne vygeneruje jednu z dvoch hodnôt 0 a 1. Táto funkcia bude podvádzať: 1 padne dvakrát častejšie ako 0. (Môžete využiť funkciu **Nahodne3** a štandardnú funkciu, ktorá počíta absolútnu hodnotu **Abs**).

2. Funkcia so zložitejším výpočtom

Niekedy sa môžeme stretnúť s funkciami, ktoré majú zložitejší predpis a vtedy by sa hodilo, keby sme nejaké pomocné výpočty mohli priradiť do lokálnej premennej. Napíšme funkciu, ktorá v štvorcifernom celom čísle navzájom vymení prvé dve cifry s ďalšími dvoma ciframi:

```
function Vymen(Cislo: Integer): Integer;
var
  Cislo1, Cislo2: Integer;
begin
  Cislo1 := Cislo div 100;
  Cislo2 := Cislo mod 100;
  Result := Cislo2 * 100 + Cislo1;
end;
```

Hoci sme mohli túto funkciu zapísať len jedným priradením, tento zápis je čitateľnejší. Funkcia najprv do premennej **Cislo1** priradí prvé dve cifry zo zadaného čísla, potom do **Cislo2** priradí zvyšné dve cifry. Nakoniec z týchto dvoch pomocných premenných poskladá výsledok.

Ďalší program bude počítať ciferný súčet nejakého štvorciferného čísla. Najprv to zapíšeme bez cyklu:

```
function CS(Cislo: Integer): Integer;
var
  Sucet: Integer;
begin
  Sucet := Cislo mod 10;
  Cislo := Cislo div 10;
  Sucet := Sucet + Cislo mod 10;
  Cislo := Cislo div 10;
  Sucet := Sucet + Cislo mod 10;
  Cislo := Cislo div 10;
  Sucet := Sucet + Cislo mod 10;
  Result := Sucet;
end;
```

Zadefinovali sme tu jednu lokálnu premennú **Sucet**, do ktorej najprv priradíme poslednú cifru čísla (**Cislo mod 10**) a potom postupne pripočítavame ďalšie cifry zadaného čísla. Pritom pôvodné číslo stále o pripočítanú poslednú cifru skrátime (**Cislo := Cislo div 10**). Tu sme využili to, že parameter je tiež len obyčajná lokálna premenná a ak potrebujeme, môžeme jej meniť hodnotu. My sme sa s týmto algoritmom stretli už dávnejšie a riešili sme ho pomocou cyklu. Pripomeňme si ho:

```
function CS(Cislo: Integer): Integer;
var
  Sucet: Integer;
begin
  Sucet := 0;
  while Cislo <> 0 do
  begin
    Sucet := Sucet + Cislo mod 10;
    Cislo := Cislo div 10;
  end;
  Result := Sucet;
end;
```

Takto to teraz funguje nielen pre štvorciferné čísla, ale aj pre ľubovoľne ciferné čísla. V tejto funkcii sme použili pomocnú lokálnu premennú **Sucet**. Do nej postupne pripočítavame jednotlivé cifry a tým dostávame očakávaný výsledok.

Už v predchádzajúcej časti sme uviedli, že premenná **Result** je taká lokálna premenná, ktorá sa automaticky zadeklaruje (my by sme ju deklarovať ako lokálnu premennú nemali) a v tele funkcie by sa do nej mala priradiť výsledná hodnota. Lenže túto premennú môžeme v tele podprogramu využiť ako bežnú premennú a

počítať do nej aj medzivýsledky. Vtedy nebudeme potrebovať pomocnú premennú **Sucet**. Zapíšme to:

```
function CS(Cislo: Integer): Integer;
begin
  Result := 0;
  while Cislo <> 0 do
  begin
    Result := Result + Cislo mod 10;
    Cislo := Cislo div 10;
  end;
end;
```

Takýto spôsob zápisu funkcie je v Pascale veľmi bežný. Treba si uvedomiť, že do premennej **Result** môžeme priraďovať koľkokrát chceme a môžeme túto premennú používať aj vo výrazoch. Dôležité je to, aká hodnota je v tejto premennej, keď funkcia končí.

Túto funkciu môžeme otestovať takýmto programom:

```
function CS(Cislo: Integer): Integer;
begin
  Result := 0;
  while Cislo <> 0 do
  begin
    Result := Result + Cislo mod 10;
    Cislo := Cislo div 10;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  I, J: Integer;
begin
  J := Random(100000) + 100000;
  for I := J to J + 9 do
    Memo1.Lines.Append(IntToStr(I) + ' ' + IntToStr(CS(I)));
end;
```

Po spustení vidíme v prvom stĺpci tabuľky nejaké čísla a v druhom stĺpci je vypočítaný ich ciferný súčet, napr.:

```
178337 29
178338 30
178339 31
178340 23
178341 24
178342 25
178343 26
178344 27
178345 28
178346 29
```

Všimnite si, že tentoraz sme funkciu definovali ako globálnu - nie je vnorená v **Button1Click**. Toto robíme vtedy, keď predpokladáme, že funkciu budeme volať viackrát z rôznych procedúr. Samozrejme, že funkcia musí byť v programovej jednotke definovaná ešte pred všetkými procedúrami, ktoré ju volajú.

Podobným spôsobom môžeme vytvoriť funkcie na súčet, resp. súčin prvých **N** čísel:

```
function Sucet(N: Integer): Integer;
begin
  Result := 0;
  while N > 0 do
  begin
    Inc(Result, N);      // to je to isté ako Result := Result + N;
    Dec(N);
  end;
end;

function Faktorial(N: Integer): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 2 to N do
    Result := Result * I;
end;
```

Vo funkcii **Faktorial** sme použili for-cyklus a preto sme museli zadeklarovať ďalšiu lokálnu premennú. Hoci aj táto funkcia by sa dala zapísať podobne ako funkcia **Sucet** pomocou while-cyklu. Všimnite si, že v oboch prípadoch sme do premennej **Result** na začiatku priradili nejakú štartovú hodnotu: buď 0 alebo 1.

Matematici vedia, že súčet čísel od 1 do N môžeme vypočítať aj bez cyklu pomocou vzorca:

```
        súčet = N (N + 1) / 2
```

Zapíšeme ho ako funkciu:

```
function Sucet1(N: Integer): Integer;
begin
  Result := N * (N + 1) div 2;
end;
```

Malým testovacím programom preveríme funkčnosť všetkých troch funkcií:

```
var
  T: TextFile;
  I: Integer;
begin
  AssignFile(T, 'tabulka.txt');
  Rewrite(T);
  for I := 1 to 10 do
    WriteLn(T, I:2, Sucet(I):5, Sucet1(I):5, Faktorial(I):10);
  CloseFile(T);
  Memo1.Lines.LoadFromFile('tabulka.txt');
end;
```

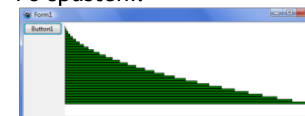
Po spustení vidíme, že obe funkcie **Sucet** aj **Sucet1** v druhom a treťom stĺpci tabuľky dávajú rovnaké výsledky:

```
 1    1    1         1
 2    3    3         2
 3    6    6         6
 4   10   10        24
 5   15   15       120
 6   21   21       720
 7   28   28      5040
 8   36   36     40320
 9   45   45    362880
10   55   55   3628800
```

Funkciu **Sucet** môže znázorniť aj pomocou stĺpcového grafu:

```
var
  N: Integer;
begin
  Image1.Canvas.Brush.Color := clGreen;
  for N := 1 to 30 do
    Image1.Canvas.Rectangle(0, N*5, Sucet(N), N*5 + 5);
end;
```

Po spustení:



Ďalšia skupina funkcií, ktorú budeme vytvárať, bude potrebovať podmienený príkaz. Začneme dvojicou funkcií, ktoré počítajú minimum, resp. maximum dvoch čísel:

```
function Min(A, B: Integer): Integer;
begin
  if A < B then
    Result := A
  else
    Result := B;
end;
```

```
function Max(A, B: Integer): Integer;
begin
  if A > B then
    Result := A
  else
    Result := B;
end;
```

Vidíme, že sa navzájom líšia len v podmienke. Obe tieto funkcie sa veľmi často vyskytujú v najrôznejších algoritmoch. Zaujímavou je funkcia, ktorá zisťuje minimum z troch čísel. Môžeme to zapísať takto komplikovane:

```
function Min3(A, B, C: Integer): Integer;
begin
  if A < B then
    if A < C then
      Result := A
    else
      Result := C
  else
    if B < C then
      Result := B
    else
      Result := C;
end;
```

ale môžeme využiť aj funkciu **Min**, ktorú sme definovali predtým:

```
function Min3(A, B, C: Integer): Integer;
begin
  Result := Min(Min(A, B), C);
end;
```

V tejto novej verzii funkcie sa najprv vypočíta minimum z prvých dvoch čísel a potom sa toto nájdené minimum ešte porovná s tretím číslom.

Pre hľadanie minima dvoch celých čísel existuje ešte jedno zaujímavé riešenie, ktoré nepoužíva podmienený príkaz, ale absolútnu hodnotu:

Táto idea počítania minima by fungovala aj pre verziu funkcie s reálnymi číslami (**div** sa nahradí operáciou reálneho delenia /). Lenže tu okrem samotného pretečenia môžu nastať aj zaokrúhľovacie chyby reálnej aritmetiky. Takže s niektorými funkciami, ktoré si sami naprogramujeme a majú takéto riziko, môžeme pracovať, len ak s tým počítame.

Vedeli by ste na rovnakom princípe vypočítať maximum dvoch čísel (celých alebo reálnych)?

```
function Min2(A, B: Integer): Integer;
begin
  Result := (A + B - Abs(A - B)) div 2;
end;
```

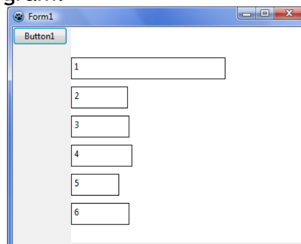
Matematicky by sa dalo ukázať, že naozaj takýto zápis správne počíta minimum dvoch čísel. Problém nastáva vtedy, keď buď súčet alebo rozdiel týchto dvoch čísel presiahne rozsah celočíselnej aritmetiky. Vtedy dá program zlý výsledok, alebo, ak máme zapnutú kontrolu pretečenia (vo voľbách prekladača), program spadne.

V ďalšej úlohe predpokladáme, že máme k dispozícii zdrojový kód programu nejakej hry. Odhalili sme, že v tejto hre nebol úplne korektne naprogramovaný náhodný generátor na hádzanie hracou kockou so šiestimi stranami. Je nám jasné, že pre správnu kocku je rovnaká pravdepodobnosť, že padne približne rovnako často každé zo šiestich čísel. Teda pravdepodobnosť je 1/6. Ak by sme kockou hádzali 600-krát (počítač to vie urobiť veľmi rýchlo), tak každé z čísel by v priemere padlo okolo 100-krát. My sme v programe počítačovej hry našli takúto funkciu:

```
function HodKockou: Integer;
begin
  Result := Random(8) - 1;
  if Result < 1 then
    Result := 1;
end;
```

Aby sme túto funkciu otestovali, vytvorili sme aplikáciu s grafickou plochou. Sem sa vykreslí stĺpcový diagram: dĺžka riadka vyjadruje počet padnutí príslušného čísla na kocke:

Po zatlačení tlačidla **Button1** (môžeme zatlačiť aj viackrát za sebou) dostaneme približne takýto diagram:



Program zobrazuje priebeh simulácie: obdĺžniky sa postupne zväčšujú, podľa toho, aké číslo padlo na kocke.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Pole: array [1..6] of Integer;
  I, J: Integer;
begin
  Image1.Canvas.FillRect(Image1.ClientRect);
  for I := 1 to 6 do
    Pole[I] := 0;
  for I := 1 to 600 do
  begin
    J := HodKockou;
    Inc(Pole[J]);
    Image1.Canvas.Rectangle(0, J*40, Pole[J], J*40 + 30);
    Image1.Canvas.TextOut(5, J*40 + 5, IntToStr(J));
    Image1.Repaint;
    Sleep(1);
  end;
end;
```

V premennej **Pole** si evidujeme počet padnutí jednotlivých čísel na kocke. Na začiatku je toto pole vynulované. Potom sa pri každom hode kocky príslušný prvok poľa zvýši o jedna.

Z diagramu vidíme, že 1 padne na kocke 3-krát častejšie ako iné čísla. Zo samotnej definície funkcie **HodKockou** vidíme, že najprv sa ako výsledok (**Result**) priradí náhodné číslo od -1 do 6 a potom sa ešte upravia hodnoty -1 a 0 na 1. Vďaka tomuto sa 1 generuje 3-krát častejšie oko zvyšné čísla.

Zaujímavou skupinou funkcií, sú celočíselné, funkcie, ktoré zisťujú niečo s deliteľmi parametra. V tomto kurze sme už niekoľko krát zisťovali delitele nejakého čísla. Zapíšme funkciu, ktorá zisťuje počet deliteľov:

```
function PocetDelitelov(N: Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to N do
    if N mod I = 0 then
      Inc(Result);
end;
```

Pomocou tejto funkcie môžeme teraz veľmi jednoducho vypísať niekoľko prvočísel:

```
var
  I: Integer;
begin
  for I := 2 to 100 do
    if PocetDelitelov(I) = 2 then
      Memo1.Lines.Append(IntToStr(I));
end;
```

Ďalšia funkcia sa veľmi podobá na funkciu **PocetDelitelov**:

```
function SucetDelitelov(N: Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to N div 2 do
    if N mod I = 0 then
      Inc(Result, I);
end;
```

Ale počíta súčet všetkých deliteľov nejakého čísla (okrem samého seba). Pomocou tejto funkcie môžeme vypísať niekoľko **dokonalých čísel**:

```
var
  I: Integer;
begin
  for I := 2 to 1000 do
    if SucetDelitelov(I) = I then
      Memo1.Lines.Append(IntToStr(I));
end;
```

Po spustení program vypíše tri dokonalé čísla:

```
6
28
496
```

Hádam jedným z najznámejších a najstarším algoritmov v histórii matematiky je **Euklidov algoritmus** na zisťovanie najväčšieho spoločného deliteľa (**NSD**) dvoch celých čísel (Euklides ho uviedol asi v roku 300 p.n.l.). Dá sa zapísať rôznymi spôsobmi. My si najprv ukážeme veľmi jednoduchú verziu. V tejto sa od väčšieho z čísel odpočítava menšie a toto sa robí dovtedy, kým sa obe čísla už nerovnajú:

```
function NSD(A, B: Integer): Integer;
begin
  while A <> B do
    if A > B then
      A := A - B
    else
      B := B - A;
  Result := A;
end;
```

Odkrokujme ručne tento algoritmus pre nejaké dve čísla, napr.
NSD(84, 216):

A	B
84	216
84	132
84	48
36	48
36	12
24	12
12	12

Samozrejme, že predpokladáme, že sú obe čísla rôzne od 0. Inak by sa algoritmus zacyklil.

Hoci funguje veľmi dobre, pre niektoré čísla je veľmi pomalý. Napríklad, ak chceme vypočítať **NSD(MaxInt, 2)**, tak aj na rýchlych počítačoch to trvá niekoľko sekúnd.

Veď začnime krokovať:

NSD(MaxInt, 2):

A	B
2147483647	2
2147483645	2
2147483643	2
2147483641	2
2147483639	2
2147483637	2
2147483635	2
2147483633	2
2147483631	2
...	2

Tabuľka by ďalej pokračovala vyše miliardou takýchto riadkov ...

Odkrokujme ešte tento nový algoritmus

NSD(84, 216):

A	B
84	216
216	84
84	48
48	36
36	12
12	0

Dáva správny výsledok, ale aj pre

NSD(MaxInt, 2):

A	B
2147483647	2
2	1
1	0

dostávame veľmi rýchlo správny výsledok: **NSD(MaxInt, 2) = 1.**

Predstavme si, že by sme tento algoritmus rozdelili na niekoľko samostatných cyklov, v ktorých sa odpočítava od väčšieho čísla menšie, a to, kým sa dá:

```
while A > B do
  A := A - B;
```

Tento cyklus, ale v konečnom dôsledku vypočíta nám známu vec: zvyšok po delení, a preto by sme ho mohli nahradiť priradením:

```
A := A mod B;
```

Toto využijeme v ďalšej verzii, v ktorej funkciu **NSD** prepíšeme použitím zvyšku po delení:

```
function NSD(A, B: Integer): Integer;
var
  Pom: Integer;
begin
  while B <> 0 do
  begin
    Pom := A mod B;
    A := B;
    B := Pom;
  end;
  Result := A;
end;
```

Všimnite si, že v cykle okrem samotného výpočtu zvyšku po delení, ešte obe tieto čísla navzájom vymeníme. Vďaka tomuto bude vždy v premennej **A** to väčšie z nich. Ešte je tu jedna dôležitá zmena: cyklus nebude končiť vtedy, keď **A** sa rovná **B**, ale vtedy, keď zvyšok po delení **A** a **B** bude 0.

Pre zaujímavosť ešte ukážeme tento istý algoritmus zapísaný bez pomocnej premennej **Pom**, len s využitím premennej **Result**:

```
function NSD(A, B: Integer): Integer;
begin
  while B <> 0 do
  begin
    Result := B;
    B := A mod B;
    A := Result;
  end;
end;
```

Hoci tento algoritmus pracuje presne rovnako ako predchádzajúci, je menej čitateľný a na vysvetľovanie asi horšie zrozumiteľný. Preto asi budeme uprednostňovať predchádzajúcu verziu.

Čo sme sa naučili

Pri výpočte hodnoty funkcie môžeme využívať nielen postupnosť príkazov, ale aj zložitejšie konštrukcie ako cykly a podmienky. Niekedy využívame aj pomocné lokálne premenné.

V tejto časti sme ukázali nasledovné techniky:

- použitie premennej **Result** ako pomocnej premennej vo výpočtoch
- zisťovanie minima a maxima čísel pomocou príkazu **if** ale aj pomocou výpočtu
- výpočet najväčšieho spoločného deliteľa **Euklidovým algoritmom**

Úlohy na precvičenie

Zadanie 1	Zapíšte funkciu bez parametrov **HodMincou**, ktorá bude simulovať hádzanie mincou. Bude vracať náhodné čísla buď 1 alebo 2, ale ich pravdepodobnosť výskytov bude v pomere 3:4. Funkciu otestujte pre veľký počet volaní.
Zadanie 2	Zapíšte funkciu **Znamienko**, ktorá pre celé číslo zistí či je záporné (vráti -1), kladné (vráti 1) alebo 0 (vráti 0).
Zadanie 3	Zapíšte funkciu **Stred** s troma číselnými parametrami. Funkcia vráti tú hodnotu, ktorá je medzi zvyšnými dvoma. Napr. **Stred(5,9,6)** vráti hodnotu **6**.
Zadanie 4	Zapíšte funkciu **Mocnina** s dvoma celočíselnými parametrami **N** a **K**. Funkcia vypočíta mocninu N^K, pričom ak výsledok presiahol **MaxInt**, funkcia vráti 0.

Zadanie 4	Zapíšte celočíselnú funkciu **Odmocnina**, ktorá z celého čísla vypočíta celú časť odmocniny. Hoci by sa to dalo vypočítať s prevodom z reálnych čísel:
	`function Odmocnina(Cislo: Integer): Integer;`
	`begin`
	`  Result := Trunc(Sqrt(Cislo));`
	`end;`
	funkciu zapíšte bez reálnej aritmetiky pomocou cyklu.

Zadanie 5	Zapíšte funkciu tangens **Tg**, ktorá počíta túto goniometrickú funkciu podľa vzťahu: `tg x = sin x / cos x` Zabezpečte, aby funkcia nikdy nespadla (keď **cos x** je 0), ale vtedy vrátila nejakú špeciálnu hodnotu, napr. 1e100 alebo -1e100.
Zadanie 6	Napíšte funkciu **Koren**, ktorá pre tri reálne parametre vráti jeden z koreňov kvadratickej rovnice: $ax^2 + bx + c = 0$ Zabezpečte, aby sa nejako riešila aj situácia, keď rovnica nemá žiadne korene.
Zadanie 7	Napíšte funkciu **PocetKorenov**, ktorá pre tri reálne parametre vráti počet rôznych reálnych koreňov kvadratickej rovnice: $ax^2 + bx + c = 0$
Zadanie 8	Napíšte reálnu funkciu **E** s jedným parametrom, ktorá vypočíta základ prirodzených logaritmov podľa tohto vzorca: $\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\frac{1}{5!}+\cdots$ Parameter funkcie určuje počet členov tohto radu.

3. Logické funkcie

Napr.

- **Odd** - zistí, či je číslo nepárne
- **Eof** - zistí, či sme pri čítaní zo súboru už na konci
- **Eoln** - zistí, či sme pri čítaní zo súboru už na konci riadka

Doteraz sme sa zaoberali len číselnými funkciami, ktoré s niekoľkými číselnými parametrami (alebo bez parametrov) vypočítali nejakú číselnú hodnotu. Ale my už poznáme niekoľko štandardných funkcií, ktoré zisťujú, či niečo platí. Takéto logické funkcie najčastejšie používame ako testy v podmienenom príkaze **if** alebo vo **while**-cykle. Často sú tieto funkcie súčasťou zložitejších logických výrazov. Výsledok logickej funkcie môžeme priradiť do nejakej premennej alebo poslať ako hodnotu parametra do podprogramu.

Zostavenie vlastnej logickej funkcie vysvetlíme na funkcii, ktorá o celom čísle zistí či je nepárne:

```
function JeNeparne(K: Integer): Boolean;
begin
  if K mod 2 = 1 then
    Result := True
  else
    Result := False;
end;
```

Keď túto funkciu zavoláme z hlavného programu, napr.

```
var
  B: Boolean;
begin
  B := JeNeparne(33);
```

do premennej **B** sa má priradiť výsledok volania funkcie **JeNeparne**. Pri volaní funkcie sa do parametra, teda lokálnej premennej **K**, priradí hodnota 33. Potom sa otestuje podmienka **K mod 2 = 1**. Keďže podmienka je splnená (zvyšok po delení 33 číslom 2 je 1), do premennej **Result** sa priradí **True**. Keďže funkcia hneď končí, zapamätá sa výsledná hodnota, všetky lokálne premenné sa zrušia (**K** aj **Result**) a výpočet sa vráti na priradenie B := **JeNeparne(33);** - do premennej **B** sa priradí **True**.

Veľmi často sa podmienený príkaz **if**, ktorý v oboch vetvách obsahuje len priradenie testovanej hodnoty (teda samotnej podmienky) do nejakej premennej, skrátene zapisuje takto:

Takáto funkcia v Pascale existuje aj ako štandardná funkcia **Odd**.

```
function JeNeparne(K: Integer): Boolean;
begin
  Result := K mod 2 = 1;
end;
```

Toto priradenie hovorí presne toto: ak je výraz **K mod 2 = 1** pravdivý, do Result priraď **True**, inak priraď **False**.

Ďalší príklad je funkcia bez parametrov. Simuluje náhodné hádzanie mincou, teda dva rôzne stavy **True** a **False**:

```
function HodMincou: Boolean;
begin
  Result := Random(2) = 1;
end;
```

Pri tejto funkcii môžeme využiť to, že v počítači sú logické hodnoty reprezentované číslom 0 (pre **False**) alebo 1 (pre **True**). Poznáme štandardnú funkciu **Ord**, ktorá napr. z logickej hodnoty **False** alebo **True** vráti 0 alebo 1. Opačná funkcia, ktorá z 0 a 1 vyrobí logické hodnoty **False** a **True**, je funkcia **Boolean**. Nech vás nemýli, že má rovnaké meno ako meno typu. Takže funkciu **HodMincou** môžeme zapísať aj takto:

```
function HodMincou: Boolean;
begin
  Result := Boolean(Random(2));
end;
```

Ďalšia funkcia zisťuje, či je číslo druhou mocninou nejakého celého čísla (napr. pre 25 by mala vrátiť **True** a pre 24 **False**) :

```
function JeMocninou(K: Integer): Boolean;
var
  I: Integer;
begin
  I := 0;
  while I * I < K do
    Inc(I);
  Result := I * I = K;
end;
```

Posledný riadok tela funkcie:

```
Result := I * I = K;
```

môžeme prečítať ako "výsledkom funkcie je pravda, keď **I*I** sa rovná **K**", t.j. našli sme také **I**, ktorého druhá mocnina je **K** a teda odmocnina **K** je **I**. Uvedomte si, že while-cyklus v tele funkcie skončí, keď už je podmienka nesplnená, t.j. keď sa prvýkrát stane **I * I >= K**.

Môžeme vytvoriť aj niekoľko užitočných funkcií, ktoré niečo zisťujú o nejakom znaku. Napr. funkcia, ktorá zistí, či je znak cifra:

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

Môžeme použiť napr. takto:

```
Read(Subor, Znak);
while JeCifra(Znak) do
begin
  R := R + Znak;
  Read(Subor, Znak);
end;
```

Čo už vieme zapísať aj elegantnejšie:

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

Funkcia, ktorá zistí, či je znak písmeno:

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

Zisťuje, či je to veľké alebo malé písmeno.

Identifikátory v Pascale sa skladajú z písmen, číslic a môžu obsahovať aj znak podčiarkovník. Funkcia **JeIdent** zistí, či je znak vhodný pre pascalovský identifikátor:

```
function JeIdent(Znak: Char): Boolean;
begin
  Result := JeCifra(Znak) or JePismeno(Znak) or (Znak = '_');
end;
```

Všimnite si, že sme ti využili už predtým definované funkcie.

Venujme sa chvíľu prvočíslam. Už sme dávnejšie písali algoritmus, ktorý zisťuje, či je nejaké číslo prvočíslom. Tento algoritmus sa snažil v cykle nájsť nejakých deliteľov čísla, a ak sa mu to nepodarilo, dané číslo bolo prvočíslom. V predchádzajúcej časti sme zostavili funkciu **PocetDelitelov**, ktorá zisťovala počet deliteľov zadaného čísla.

Zostavíme funkciu, ktorá o zadanom čísle zisťuje, či je to prvočíslo:

```
function JePrvocislo(Cislo: Integer): Boolean;
begin
  Result := PocetDelitelov(Cislo) = 2;
end;
```

Uvedomte si, že tento spôsob zisťovania, či je číslo prvočíslom, nie je najefektívnejší: zisťuje totiž počet všetkých deliteľov čísla. Nám by stačilo už pri prvom deliteli, ktorý je väčší ako 1, skončiť hľadanie ďalších deliteľov, lebo už vieme, že to určite nebude prvočíslo.

Funkcia o čísle rozhodne, či je prvočíslom, na základe toho, či má iba dvoch deliteľov. Vedeli by ste zistiť, či funkcia pracuje správne aj pre číslo 1? Číslo 1 sa totiž nepovažuje za prvočíslo. A ako to bude so zápornými číslami – dá pre záporné čísla správny výsledok (záporné čísla by nemali byť prvočísla)?

Využime funkciu na zisťovanie prvočísel na nájdenie všetkých prvočíselných dvojičiek: sú to také dve prvočísla, ktorých rozdiel je 2. Také dvojičky sú napr. 5 a 7, 17 a 19 a pod.

Algoritmus na hľadanie dvojičiek prvočísel do 100 začína od čísla 3:

```
var
  I: Integer;
begin
  I := 3;
  while I < 100 do
  begin
    if JePrvocislo(I) and JePrvocislo(I + 2) then
      Memo1.Lines.Append(IntToStr(I) + ' ' + IntToStr(I + 2));
    I := I + 2;
  end;
end;
```

Po spustení programu dostávame výpis všetkých prvočíslených dvojičiek:

```
3 5
5 7
11 13
17 19
29 31
41 43
59 61
71 73
```

Všimnite si test

```
JePrvocislo(I) and
  JePrvocislo(I + 2),
```

ktorý zisťuje, či je súčasne prvočíslom číslo **I** a aj číslo **I+2**. Pre FreePascal platí dôležitá vlastnosť: pri vyhodnocovaní logických operácií and a or sa druhý člen nebude vôbec vyhodnocovať, ak už je z hodnoty prvého operandu jasné, aký bude výsledok. Napr. pre operáciu and sa najprv vyhodnotí prvý operand (zistí sa či platí **JePrvocislo(I)**) a ak je už tento nepravdivý (**I** nie je prvočíslo), druhý operand sa vyhodnocovať vôbec nebude. To znamená, že sa nebude zbytočne volať funkcia **JePrvocislo(I+2)**.

Zapíšme ešte jednu logickú funkciu, ktorá zistí, či sa v nejakom intervale čísel nachádza aspoň jedno prvočíslo. Funkcia dostáva dva parametre: začiatok a koniec intervalu. Keďže všetky prvočísla (okrem 2) sú nepárne, stačí nám kontrolovať len nepárne čísla v tomto intervale. Preto na začiatok sme funkcie pridali test, ktorý zisťuje, či je začiatok intervalu nepárny. Ak nie je, začiatok intervalu sa zvýši o 1:

```
function ExistujePrvocislo(Z, K: Integer): Boolean;
begin
  if not Odd(Z) then
    Inc(Z);
  while (Z <= K) and not JePrvocislo(Z) do
    Z := Z + 2;
  Result := Z <= K;
end;
```

Opäť si všimnite podmienku vo while-cykle: **(Z <= K) and not JePrvocislo(Z)**. Podmienka hovorí, že pokračujeme v hľadaní prvočísla, kým sme ešte neprešli celý interval a zatiaľ ešte neboli prvočísla. Cyklus skončí, keď **Z > K** (prešli sme neúspešne celý interval) alebo, keď platí **JePrvocislo(Z)** (našli sme nejaké prvočíslo). Preto sa ako výsledok funkcie (**Result**) hodnota **Z <= K**. Otestujte túto funkciu pre rôzne intervaly, napr. ani 889..905, ani 31399..31467 neobsahujú žiadne prvočíslo.

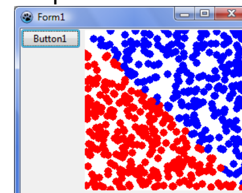
Na záver časti o logických funkciách pozrime funkcie, ktoré niečo zisťujú o bode v rovine, t.j. či daný bod (X,Y) leží v nejakej oblasti. Napr. takáto funkcia:

```
function VOblasti1(X, Y: Integer): Boolean;
begin
  Result := X < Y;
end;
```

definuje nejakú oblasť v grafickej ploche. Na otestovanie takejto funkcie vložíme do formulára časovač (s hodnotou **Interval** = 10). V časovači budeme kresliť malé krúžky buď červené, ak je vygenerovaný bod v oblasti, alebo modré, ak je mimo oblasti:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  X, Y: Integer;
begin
  X := Random(Image1.Width);
  Y := Random(Image1.Height);
  if VOblasti1(X, Y) then
    Image1.Canvas.Brush.Color := clRed
  else
    Image1.Canvas.Brush.Color := clBlue;
  Image1.Canvas.Pen.Color := Image1.Canvas.Brush.Color;
  Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
end;
```

Po spustení dostávame:



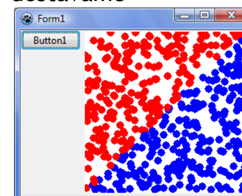
Môžeme experimentovať s rôznymi oblasťami. Napr. :

```
function VOblasti2(X, Y: Integer): Boolean;
begin
  Result := X < 200 - Y;
end;
```

a vtedy po oprave podmieneného príkazu

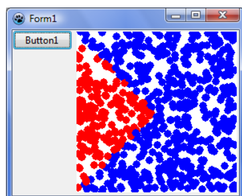
```
if VOblasti2(X, Y) then
```

dostávame

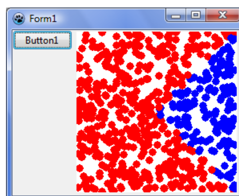


Takto definované oblasti môžeme "skladať" logickými operáciami **and** a **or**. Napr.

if VOblasti1(X, Y) and VOblasti2(X, Y) then



if VOblasti1(X, Y) or VOblasti2(X, Y) then



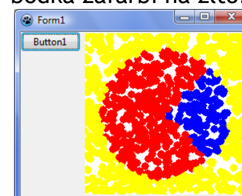
Zadefinujme oblasť pre test vnútra kruhu so stredom (100, 100) a s polomerom 80:

```
function VOblasti3(X, Y: Integer): Boolean;
begin
  Result := Sqr(X - 100) + Sqr(Y - 100) < Sqr(80);
end;
```

Dopíšme do programu aj testovanie tejto tretej oblasti:

```
begin
  X := Random(Image1.Width);
  Y := Random(Image1.Height);
  if VOblasti3(X, Y) then
    if VOblasti1(X, Y) or VOblasti2(X, Y) then
      Image1.Canvas.Brush.Color := clRed
    else
      Image1.Canvas.Brush.Color := clBlue
  else
    Image1.Canvas.Brush.Color := clYellow;
  Image1.Canvas.Pen.Color := Image1.Canvas.Brush.Color;
  Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
end;
```

Ak je vygenerovaný bod v 3. oblasti, tak sa ešte testujú aj oblasti 1. a 2. inak (bod je mimo kruhu) sa bodka zafarbí na žlto:



V tomto programe sme viackrát zapisovali nejaké podmienené príkazy typu:

```
if podmienka then
  Image1.Canvas.Brush.Color := clRed
else
  Image1.Canvas.Brush.Color := clBlue;
```

Ak by sme si pripravili takúto pomocnú funkciu:

```
function Ak(Podmienka: Boolean; Cislo1, Cislo2: Integer): Integer;
begin
  if Podmienka then
    Result := Cislo1
  else
    Result := Cislo2;
end;
```

Teraz by sme mohli prepísať poslednú verziu programu s oblasťami, kde bolo

```
if VOblasti3(X, Y) then
  if VOblasti1(X, Y) or VOblasti2(X, Y) then
    Image1.Canvas.Brush.Color := clRed
  else
    Image1.Canvas.Brush.Color := clBlue
else
  Image1.Canvas.Brush.Color := clYellow;
```

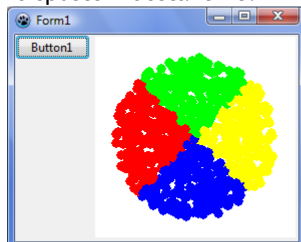
pomocou novej funkcie **Ak**:

```
Image1.Canvas.Brush.Color :=
  Ak(VOblasti3(X, Y),
    Ak(VOblasti1(X, Y) or VOblasti2(X, Y), clRed, clBlue),
    clYellow);
```

Ak by ste to takto prepísali, dostanete úplne rovnaký obrázok ako predtým. Preskúmajte tento iný zápis použitia funkcie **Ak**:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  X, Y: Integer;
begin
  X := Random(Image1.Width);
  Y := Random(Image1.Height);
  if VOblasti3(X, Y) then
  begin
    Image1.Canvas.Brush.Color :=
      Ak(VOblasti1(X, Y), Ak(VOblasti2(X, Y), clRed, clBlue),
                          Ak(VOblasti2(X, Y), clLime, clYellow));
    Image1.Canvas.Pen.Color := Image1.Canvas.Brush.Color;
    Image1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
  end;
end;
```

Po spustení dostaneme:



Čo sme sa naučili

Môžeme definovať aj funkcie, ktoré vracajú hodnotu **True** alebo **False**. Hovoríme im logické funkcie. Využijú sa pri konštruovaní zložitejších podmienok pre príkazy vetvenia **if** a cyklu **while**.

Úlohy na precvičenie

Zadanie 1	Zapíšte funkciu, ktorá zistí, či sú dve reálne čísla blízko seba, t.j. sú od seba vzdialené len veľmi malú hodnotu napr. 1e-4.
Zadanie 2	Zapíšte funkciu, ktorá zistí, či sa nejaké tri čísla rovnajú.
Zadanie 3	Zapíšte funkciu, ktorá zistí, či sa nejaké tri logické hodnoty rovnajú.
Zadanie 4	Zapíšte funkciu, ktorá zistí, či je dané číslo nejakou mocninou 2.
Zadanie 5	Zapíšte funkciu, ktorá zistí, či dané tri reálne čísla môžu byť dĺžkami strán nejakého trojuholníka.
Zadanie 6	Zapíšte funkciu, ktorá zistí, či daný znak môže byť cifrou v šestnástkovej sústave (znaky 0..9, a..f).
Zadanie 7	Zapíšte funkciu, ktorá zistí, či dané číslo je Fibonacciho číslo (jedno z čísel Fibonacciho postupnosti),

4. Znakové funkcie

Malou ale užitočnou skupinou funkcií sú funkcie, ktoré nejako spracovávajú jeden znak.Najprv vymenujeme niekoľko definícií takýchto funkcií a na záver si pozrieme použitie v programe. Hneď prvá funkcia z čísla od 0 do 9 vyrobí zodpovedajúci znak:

Pri práci so znakmi, najmä v algoritmoch so znakovými reťazcami a textovými súbormi, potrebujeme prerábať znaky na čísla, napríklad číslice alebo písmená. Alebo prerábame znaky na nejaké iné, napr. malé písmená na veľké. Väčšina funkcií využíva štandardné konverzné funkcie **Ord** a **Char**, ktoré buď zo znaku vyrobia jeho číselnú reprezentáciu (***Ascii kód***) alebo naopak, z čísla vyrobia znak.

```
function Cislica(Cislo: Integer): Char;
begin
  Result := Char(Cislo + Ord('0'));
end;
```

Namiesto **Ord('0')** môžeme písať konštantu 48. Podobne zapíšeme aj funkciu, ktorá vyrobí číslici v 16-ovej sústave. Tu treba hodnoty 10, 11, 12, 13, 14, 15 prerábať na znaky 'a', 'b', 'c', 'd', 'e', 'f' (alebo príslušné veľké písmená):

```
function Cislica16(Cislo: Integer): Char;
begin
  if Cislo <= 9 then
    Result := Char(Cislo + Ord('0'))
  else
    Result := Char(Cislo - 10 + Ord('a'));
end;
```

V niektorých situáciách sa nám môže hodiť, ak vieme vytvoriť z čísla od 1 do 26 príslušné písmeno malej abecedy (resp. veľkej):

```
function Pismeno(Cislo: Integer): Char;
begin
  Result := Char(Cislo + Ord('a') - 1);
end;
```

Štandardne funkcie **UpCase** a **LowerCase** prerábajú malé písmená na veľké, resp. veľké na malé. Naprogramujme si ich aj my:

```
function UpCase(Znak: Char): Char;
begin
  Result := Znak;
  if (Znak >= 'a') and (Znak <= 'z') then
    Dec(Result, 32);
end;
```

Funkcia **UpCase** najprv do **Result** priradí vstupný znak a potom, ak to bolo malé písmeno, posunie ho v Ascii kódoch o 32 znakov nadol. Totiž platí: **Ord('a')-Ord('A')=32**. Využili sme tu príkaz **Dec**, ktorý vie zmenšovať nielen celé čísla, ale vie zmenšovať Ascii kódy znakov.

Neodporúčame ale takto prepisovať štandardné funkcie. Totiž v Lazaruse sú definované nielen pre znaky, ale aj pre znakové reťazce. Keby sme ich takto definovali v našom programe, neboli by potom prístupné ich reťazcové verzie.

Podobne funguje aj **LowerCase**, ktorý z veľkých znakov spraví malé:

```
function LowerCase(Znak: Char): Char;
begin
  Result := Znak;
  if (Znak >= 'A') and (Znak <= 'Z') then
    Inc(Result, 32);
end;
```

Ďalšia funkcia vyrobí zo znaku cifra číslo. Tu sa musíme rozhodnúť, čo robiť, v prípade, že daný znak nie je cifra. Takýto chybový stav môžeme vyriešiť napr. ak, že funkcie vráti -1:

```
function Cifra(Znak: Char): Integer;
begin
  if (Znak >= '0') and (Znak <= '9') then
    Result := Ord(Znak) - Ord('0')
  else
    Result := -1;
end;
```

Podobne bude vyzerať funkcia, ktorá z písmena vráti jeho poradie v abecede:

```
function VAbecede(Znak: Char): Integer;
begin
  if (Znak >= 'a') and (Znak <= 'z') then
    Result := Ord(Znak) - Ord('a') + 1
  else if (Znak >= 'A') and (Znak <= 'Z') then
    Result := Ord(Znak) - Ord('A') + 1
  else
    Result := -1;
end;
```

Zaujímavou skupinou funkcií sú také, ktoré generujú náhodné znaky. Napr. funkcia vygeneruje náhodnú cifru:

```
function NahodnaCifra: Char;
begin
  Result := Char(Random(10) + 48);   // 48 = Ord('0')
end;
```

Na rovnakom princípe funguje aj generovanie malého písmena:

```
function NahodneMalePismeno: Char;
begin
  Result := Char(Random(26) + 97);   // 97 = Ord('a')
end;
```

Aby sme mohli generovať náhodné texty, budeme s nejakou pravdepodobnosťou dávať aj medzery:

```
function NahodnyZnak: Char;
begin
  if Random(5) = 0 then
    Result := ' '
  else
    Result := Char(Random(26) + 97);   // 97 = Ord('a')
end;
```

Takýto generátor môžeme otestovať:

```
var
  S: string;
  I: Integer;
begin
  S := '';
  for I := 1 to 300 do
    S := S + NahodnyZnak;
  Memo1.Lines.Append(S);
end;
```

Po spustení dostávame približne takýto text:

```
pvwwqjhbhmvmkv cavu wmunr
qdyn metfd q fkxlpxc rr j
ijo qy feqgmgecrdfjv  vczmz
la hdhdikbsognco hav shepa
j  oshltkxqxrlz
dtkrgizprnfkqm hrhqld
cjoazifdczgnl  jmv w  q
wuo a sls jaf jsenbf
mwypp eq szgrwtzoff ytmfg
mwzq abrlnxzfrg  tijpvq ilx
yrfytgfnaf sqfeut nd smv d
dsatcf op fhr   y vzse mz
```

Úlohy na precvičenie

Zadanie 1	Napíšte funkciu, ktorá zistí, či je daný znak nejaká samohláska.
Zadanie 2	Napíšte dve funkcie, z ktorých prvá náhodne vygeneruje nejakú samohlásku a, e, i, o , u, y. Druhá funkcia náhodne generuje len spoluhlásky.
Zadanie 3	Napíšte funkciu, ktorá všetky znaky okrem písmen a číslic zmení na medzery.
Zadanie 4	Napíšte funkciu **Cifra16**, ktorá zo šestnástkovej cifry vráti číslo. Napr. **Cifra16('7')** vráti 7, **Cifra16('b')** vráti 11.

2. tematická jednotka – funkcie so zloženými typmi

V tejto časti sa venujeme funkciám, ktoré spracovávajú znakové reťazce a polia. Stretneme sa s úlohami, v ktorých buď parameter alebo výsledok funkcie je nejaký štruktúrovaný typ. Ako štruktúrované typy poznáme znakové reťazce a jednorozmerné polia. Premenné týchto dvoch typov v sebe obsahujú ďalšie prvky, napr. znakový reťazec obsahuje nejakú postupnosť znakov a pole je zas postupnosť nejakých hodnôt rovnakého typu, napr. čísel, logických hodnôt, reťazcov atď. K týmto prvkom pristupujeme cez ich indexy. V znakových reťazcoch sú tieto indexy vždy od 1 po dĺžku reťazca, pre polia sú indexy definované nejakým intervalom priamo v deklarácii poľa.

Pre funkcie v tejto časti bude charakteristické to, že buď parameter alebo dokonca výsledok budeme nejako indexovať, prípadne reťazce môžeme skladať pomocou operácie zreťazenia.

1. Funkcie s reťazcami

V tejto časti sa venujeme funkciám, ktorých parametre alebo výsledok sú znakové reťazce.

Začneme s funkciou, ktorá vygeneruje reťazec **N** hviezdičiek. Prvá verzia tejto funkcie začne od prázdneho reťazca (pomocná premenná **Retazec**) a potom do neho **N**-krát na koniec prilepí jeden znak hviezdička. Na záver takýto reťazec priradí do výsledku:

```
function Hviezdicky(N: Integer): string;
var
  I: Integer;
  Retazec: string;
begin
  Retazec := '';
  for I := 1 to N do
    Retazec := Retazec + '*';
  Result := Retazec;
end;
```

Zapíšme takúto časť programu:

```
var
  I: Integer;
begin
  for I := 6 to 15 do
    Memo1.Lines.Append(Hviezdicky(I));
end;
```

Do textovej plochy by sa vypísalo:

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

Zdá sa, že funkcia pracuje správne.

Keďže my vieme dopredu dĺžku výsledného reťazca, funkciu môžeme zapísať aj efektívnejšie. Najprv vyrobíme nejaký **N**-znakový reťazec (pomocou príkazu **SetLength**), v ktorom sú všetky znaky zatiaľ ešte nedefinované. Potom v tomto reťazci všetky znaky opravíme na hviezdičky:

```
function Hviezdicky(N: Integer): string;
var
  I: Integer;
  Retazec: string;
begin
  SetLength(Retazec, N);
  for I := 1 to N do
    Retazec[I] := '*';
  Result := Retazec;
end;
```

Skúsme si teraz uvedomiť, že vo funkcii máme k dispozícii premennú **Result**. Je tu už automaticky zadeklarovaná a má rovnaký typ ako funkcia, teda **string**. Vďaka

tomu vlastne vôbec nepotrebujeme pomocnú premennú **Retazec**, a môžeme ju nahradiť priamo premennou **Result**:

```
function Hviezdicky(N: Integer): string;
var
  I: Integer;
begin
  SetLength(Result, N);
  for I := 1 to N do
    Result[I] := '*';
end;
```

Premennú **Result** môžeme používať nielen na to, aby sme do nej priradili už pripravený výsledok funkcie, ale túto premennú môžeme aktívne používať na vytváranie výsledku. Môžeme ju teda aj indexovať, keďže je typu znakový reťazec.

Už dlhšie poznáme dve štandardné reťazcové funkcie:

- **IntToStr** - z celého čísla vytvorí znakový reťazec
- **FloatToStr** - z reálneho čísla vytvorí znakový reťazec

Reťazcové funkcie často využívame pri príprave výstupu, napr. keď chceme vypisovať nejaké čísla. Naprogramujme si jednu z nich ako malé cvičenie:

```
function IntToStr(N: Integer): string;
var
  Zaporne: Boolean;
begin
  Zaporne := N < 0;
  if Zaporne then
    N := -N;
  Result := '';
  repeat
    Result := Char(N mod 10 + 48) + Result;
    N := N div 10;
  until N = 0;
  if Zaporne then
    Result := '-' + Result;
end;
```

Na podobnom princípe by sme mohli zapísať funkcie, ktoré prevádzajú celé číslo do nejakej inej ako 10-ovej sústavy, napr. dvojkovej, osmičkovej a pod.

Ďalší príklad využíva reťazcovú funkciu **IntToStr** (buď štandardnú alebo túto našu) :

```
function Dvojica(A, B: Integer): string;
begin
  Result := '(' + IntToStr(A) + ', ' + IntToStr(B) + ')';
end;
```

Funkcia z dvoch celých čísel vytvorí reťazec, v ktorom sú čísla oddelené čiarkou a celé je to zabalené do zátvoriek. Takúto funkciu využijeme pri najrôznejších výpisoch, napr:

```
var
  I: Integer;
begin
  for I := 87 to 93 do
    Memo1.Lines.Append(Dvojica(I, Sqr(I)));
end;
```

Do textovej plochy vypíše:

```
(87, 7569)
(88, 7744)
(89, 7921)
(90, 8100)
(91, 8281)
(92, 8464)
(93, 8649)
```

Ďalšou skupinou reťazcových funkcií sú také, ktoré ako parameter dostávajú nejaký reťazec, niečo z neho zistia a vrátia o tom informáciu.

Poznáme už štandardné funkcie:

- **StrToInt**, **StrToFloat** - z reťazca vytvorí číslo
- **Length** - zistí dĺžku reťazca
- **Pos** - hľadá výskyt podreťazca - zistí jeho pozíciu

Napríklad nasledovná funkcia zistí počet výskytov nejakého podreťazca v reťazci:

```
function PocetVyskytov(P, S: string): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to Length(s) do
    if P = Copy(S, I, Length(P)) then
      Inc(Result);
end;
```

Napríklad volanie

```
PocetVyskytov('ma', 'mama ma emu a ema ma mamu')  vráti 6
PocetVyskytov('ma ', 'mama ma emu a ema ma mamu') vráti 4
PocetVyskytov('mam', 'mama ma emu a ema ma mamu') vráti 2
```

Nielen programátori obľubujú slovné palindrómy, t.j. reťazce, ktoré sa rovnako čítajú spredu aj zozadu. Napíšme funkciu, ktorá to preverí:

Známy je palindróm napr. **'Jeleňovi pivo nelej'** (samozrejme, že bez medzier a diakritických znamienok)

```
function JePalindrom(S: string): Boolean;
var
  I, J: Integer;
begin
  I := 1;
  J := Length(S);
  while (I < J) and (S[I] = S[J]) do
  begin
    Inc(I); Dec(J);
  end;
  Result := I >= J;
end;
```

Funkcia postupne prechádza znaky spredu aj zozadu a kým sa rovnajú, tak sa posúva. Ak nájde dvojicu znakov, ktoré nie sú rovnaké, while-cyklus skončí a funkcia vráti **False**. Napríklad pre volanie

```
JePalindrom('jelenovipivonelej')   ;vráti True
JePalindrom('jelenovi pivo nelej') vráti False
JePalindrom('kobylamamalybok')     vráti True
JePalindrom('kobyla ma maly bok')  vráti False
```

Ak by sme chceli pri kontrole ignorovať medzery pridáme na začiatok funkcie **JePalindrom** cyklus, v ktorom z reťazca vyhodíme všetky medzery:

```
function JePalindrom(S: string): Boolean;
var
  I, J: Integer;
begin
  repeat
    I := Pos(' ', S);
    if I <> 0 then
      Delete(S, I, 1);
  until I = 0;
  I := 1;    J := Length(S);
  while (I < J) and (S[I] = S[J]) do
  begin
    Inc(I); Dec(J);
  end;
  Result := I >= J;
end;
```

Teraz správne fungujú aj vety s medzerami.

Uvedieme, ako by sa dala naprogramovať štandardná funkcia **Pos**, ktorá hľadá prvý výskyt nejakého podreťazca. Hoci asi nemá zmysel takto predefinovávať štandardné funkcie, môže byť poučné vidieť, ako takéto funkcie vyzerajú:

```
function Pos(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;
```

Skutočná štandardná funkcia **Pos** je naprogramovaná výrazne efektívnejšie, ale pre

študijné účely je táto verzia dobrá. Všimnite si použitie pomocnej logickej premennej **Nasiel**, v ktorej si zaznačujeme, či sme hľadaný podreťazec v cykle už našli (**True**) alebo ešte nie (**False**).

Ďalšia skupina funkcií dostáva parameter reťazec a vráti nejaký iný reťazec. Najznámejšou takouto štandardnou funkciou je **Copy**, ktorá vyberá z reťazca nejaký jeho podreťazec. Ukážme niekoľko takýchto funkcií. Napríklad funkcia **Opakuj** niekoľkokrát za sebou zopakuje nejaký reťazec:

Ak zapíšeme
Opakuj(10, 'bla')
dostaneme
blablablablablablablablablabla

```
function Opakuj(N: Integer; Slovo: string): string;
var
  I: Integer;
begin
  Result := '';
  for I := 1 to N do
    Result := Result + Slovo;
end;
```

Ďalšia funkcia, ktorá vráti prvé slovo z nejakej vety:

Predpokladáme, že slová sú oddelené aspoň jednou medzerou.

```
function PrveSlovo(Veta: string): string;
begin
  Result := Copy(Veta, 1, Pos(' ', Veta + ' ') - 1);
end;
```

Zápis **Pos(' ', Veta + ' ')** vyhľadá vo vete prvý výskyt medzery. Ak v tejto vete nebola žiadna medzera (napr. je to len jedno slovo), tak vďaka pridanej medzere na koniec vety, funkcia **Pos** vráti index znaku za toto jediné slovo. Ďalej pomocou funkcie **Copy** vystrihneme z vety podreťazec od prvého znaku až po výskyt medzery mínus jedna (bez tejto medzery). Toto je dosť častý spôsob zápisu reťazcových funkcií, keď konštruujeme dosť komplexné reťazcové výrazy. Pre začínajúceho programátora by sme to mohli zapísať skôr takto:

```
function PrveSlovo(Veta: string): string;
var
  P: Integer;
begin
  Veta := Veta + ' ';          // pre istotu pridáme medzeru na koniec
  P := Pos(' ', Veta);         // pozícia prvej medzery vo vete
  Result := Copy(Veta, 1, P - 1);    // prvé slovo končí pred medzerou
end;
```

Napr. posunutím písmena **'A'** o **3** dostaneme písmeno **'D'**, alebo posunutím písmena **'b'** o **-3** dostaneme **'y'** (abecedu sme tu zacyklili). Vďaka tomuto budeme vedieť zakódovať nejaký text a ak vieme akým číslom to bolo zakódované, budeme to vedieť aj odkódovať.

Ďalšia funkcia cyklicky v abecede posunie všetky písmená v reťazci o nejakú konštantu:

Napr. pre
Koduj**('Dnes ideme na Pivo.', 23)**
dostaneme
'Akbp fabjb kx Mfsl.'

čo môžeme odkódovať volaním
Koduj**('Akbp fabjb kx Mfsl.', -23)**
dostaneme
'Dnes ideme na Pivo.'

```
function Koduj(Veta: string; Cislo: Integer): string;

  function Koduj1(Znak: Char): Char;
  begin
    if (Znak >= 'a') and (Znak <= 'z') then
      Result := Char(97 + (Ord(Znak) - 97 + Cislo) mod 26)
    else if (Znak >= 'A') and (Znak <= 'Z') then
      Result := Char(65 + (Ord(Znak) - 65 + Cislo) mod 26)
    else
      Result := Znak;
  end;

var
  I: Integer;
begin
  Cislo := Cislo mod 26;
  if Cislo < 0 then
    Cislo := Cislo + 26;
  SetLength(Result, Length(Veta));
  for I := 1 to Length(Veta) do
    Result[I] := Koduj1(Veta[I]);
end;
```

Funkcia v sebe obsahuje ďalšiu pomocnú znakovú funkciu **Koduj1**. Táto funkcia zakóduje len jeden znak: ak je to malé alebo veľké písmeno, cyklicky ho posunie o

hodnotu premennej **Cislo**. Ak to nebolo písmeno, ale iný znak, ten sa neprekóduje, ale ostáva. Telo funkcie **Koduj** najprv upraví premennú **Cislo**, aby nebolo záporné: cyklicky posunúť o -10 je to isté ako posunúť o +16. Potom funkcia prekóduje každý znak v reťazci.

Na základe tejto funkcie môžete navrhnúť aj dômyselnejšie kódovanie.

Čo sme sa naučili

V reťazcových funkciách môžeme prechádzať reťazce, ktoré sú parametrami alebo výsledkom aj po znakoch. Môžeme používať štandardné funkcie a procedúry, napr. **Length**, **Copy**, **Pos**, **Delete** a **Insert**.

V tejto časti sme ukázali nasledovné techniky:

- zisťovanie, či reťazec je palindróm
- zakódovanie a odkódovanie reťazca cyklickým posunom písmen

Úlohy na precvičenie

Zadanie 1	Napíšte funkciu **IntToStr** s druhým parametrom, ktorý označuje dĺžku výsledného znakového reťazca. Ak je reťazec, ktorý vznikne z čísla, kratší, doplní sa medzerami, ak je dlhší ureže sa.
Zadanie 2	Zapíšte funkciu **IntToHex**, ktorá prevedie celé číslo do 16-ovej sústavy.
Zadanie 3	Zapíšte funkciu **HexToInt**, ktorá zo znakového reťazca, ktorý reprezentuje číslo v 16-ovej sústave, vráti celé číslo.
Zadanie 4	Napíšte funkciu **Nahrad** s troma parametrami **R1**, **R2**, **R3**, ktorá vráti upravený reťazec **R1**. V tomto reťazci sa všetky výskyty reťazca **R2** nahradia reťazcom **R3**.
Zadanie 5	Napíšte funkciu **BezDiakritiky**, ktorá v znakovom reťazci s diakritikou nahradí znaky, tak, aby bol text bez diakritiky. Napr. **BezDiakritiky('Môj Počítač')** vráti '**Moj Pocitac**'.
Zadanie 6	Napíšte funkciu **Skrat**, ktorá v prípade, že vstupný reťazec je dlhší ako 100 znakov, ho skráti na 100 a pridá k nemu '...'.
Zadanie 7	Napíšte funkciu, ktorá generuje náhodné slovo. Slovo je náhodnej dĺžky od 1 do 10, pričom sa v ňom striedajú náhodné samohlásky a náhodné spoluhlásky.
Zadanie 8	Napíšte funkciu s jedným parametrom – menom textového súboru. Funkcia prečíta celý súbor a jeho riadky uloží do jediného reťazca - výsledku funkcie. Riadky pritom ukončuje dvojicou znakov #13#10.
Zadanie 9	Napíšte funkciu, ktorá v znakovom reťazci zistí počet výskytov dvojíc znakov #13#10. Pre reťazec vyrobený predchádzajúcou funkciou, tento počet označuje počet riadkov v súbore.

2. Funkcie spracovávajúce pole

Už v predchádzajúcich moduloch sme sa stretli s definovaním nového typu pomocou zápisu

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

Takýmto zápisom sme vytvorili nový typ pre premenné, ale aj pre parametre a neskôr uvidíme, že aj pre výsledok funkcie.

Teraz môžeme definovať, napr. premenné

```
 var
   Pole, A, B: TPole;
```

Ale môžeme takto popísať aj parameter funkcie, napr.

```
function SucetPrvkov
  (Pole: TPole): Integer;
function JeNulove
  (Pole: TPole): Boolean;
function SuRovnake
  (A, B: TPole): Boolean;
```

Takto definované nové typy zvykneme v Pascale pomenovávať tak, že začínajú veľkým písmenom **T**, za ktoré píšeme upresňujúci text, napr.

```
 type
   TPoleZnakov =
    array [1..100] of Char;
   TTabulka =
    array [1..200] of Real;
   TPoctyVyskytov =
array [1..256] of Integer;
```

Je dobrou programátorskou praxou najprv pre pole definovať nový typ a potom tento typ používať aj pri deklarovaní premenných aj pri popise parametrov podprogramov (procedúr aj funkcií). Parametre funkcií aj procedúr musia mať v hlavičke uvedené len identifikátory už predtým definovaných typov a preto sa v týchto prípadoch definovaniu typov nevyhneme.

Napíšme funkciu, ktorá počíta súčet prvkov celočíselného poľa:

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

function SucetPrvkov(P: TPole): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to 100 do
    Result := Result + P[I];
end;
```

Táto funkcia funguje len pre 100-prvkové pole typu **TPole**.

Keď sme sa prvýkrát zoznámili s poľami, uviedli sme, že existujú dve užitočné štandardné funkcie **Low** a **High**, pomocou ktorých vieme zistiť dolný a horný rozsah indexu ľubovoľného poľa. Okrem toho známa funkcia **Length** nám pre pole vráti počet prvkov poľa. Využijeme to najmä vtedy, keď predpokladáme, že sa možno v budúcnosti bude meniť rozsah poľa.

Nasledujúca funkcia vypočíta priemer zo všetkých prvkov poľa. Správne by fungovala aj po zmene hornej hranice indexu:

```
type
  TPole = array [1..150] of Integer;

function Priemer(P: TPole): Real;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to High(P) do
    Result := Result + P[I];
  Result := Result / Length(P);
end;
```

Vo for-cykle sme nastavili hornú hranicu ako **High(P)**, teda momentálne je to 150. Vo for-cykle sa najprv vypočítal súčet prvkov a tento sa na záver vydelil počtom prvkov poľa **Length(P)**, čo je tiež 150.

Uvedomte si, že nie vždy sa rovná počet prvkov poľa (**Length**) hornému rozsahu indexu (**High**). Toto platí len vtedy, keď dolný rozsah indexu je 1, ale napr. pre pole

```
type
 TPole2 =
  array [100..200] of
   Integer;
```

má horný rozsah indexu (**High**) hodnotu 200, ale počet prvkov (**Length**) je len 101.

Teraz uveďme funkciu, ktorá zistí, či sú všetky prvky nulové:

```
function JeNulove(P: TPole): Boolean;
var
  I: Integer;
begin
  I := Low(P);
  while (I <= High(P)) and (P[I] = 0) do
    Inc(I);
  Result := I > High(P);
end;
```

Funkcia postupne prechádza všetky prvky poľa a kým sú nulové (je to ešte v poriadku), presunie sa nasledovný prvok (zvýši index o 1). While-cyklus skončí buď vtedy, keď **I > High(P)** (prekontrolovali sme už všetky prvky) alebo keď **P[I] <> 0** (našli sme nevyhovujúci prvok). Na základe tohto sa nastaví výsledok funkcie buď na

True alebo **False**. Túto funkciu môžeme zapísať aj trochu inak:

```
function JeNulove(P: TPole): Boolean;
var
  I: Integer;
begin
  Result := True;
  for I := Low(P) to High(P) do
    if P[I] <> 0 Then
      Result := False;
end;
```

Funkcia aj takto pracuje správne, ale tento algoritmus je veľmi neefektívny: aj keď nájde nevyhovujúci prvok (možno hneď prvý prvok poľa), pokračuje v porovnávaní aj ďalších prvkov, či sú tiež nenulové. To-to ale robí úplne zbytočne.

Zápis funkcie hovorí, že postupne prejdeme všetky prvky poľa a vždy, keď nájdeme nevyhovujúci prvok (**P[I] <> 0**) zapamätáme si, že výsledok funkcie bude **False**.

Toto neefektívne riešenie môžeme ale trochu vylepšiť pomocou nového príkazu **Exit**. Tento príkaz, ak sa použije v tele podprogramu (vo funkcii ale aj v procedúre), spôsobí okamžité ukončenie podprogramu. Samozrejme, že vtedy sa vykonajú všetky záverečné úkony ***mechanizmu volania podprogramu***: teda sa zrušia všetky lokálne premenné a vykonávanie programu sa presunie na návratové miesto volania. Prepíšme funkciu s použitím príkazu **Exit**:

```
function JeNulove(P: TPole): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low(P) to High(P) do
    if P[I] <> 0 Then
      Exit;
  Result := True;
end;
```

Ak si toto riešenie porovnáte s predchádzajúcou neefektívnou verziou, vidíte, že sme na začiatku zmenili hodnotu premennej **Result** na **False**. Teraz totiž predpokladáme, že asi bude v poli nenulový prvok a teda výsledkom bude **False**. Keď takýto prvok nájdeme, stačí urobiť **Exit** a s touto hodnotou funkcia aj skončí. Po ukončení cyklu, teda úspešne sme prekontrolovali všetky prvky poľa, nastavíme výsledok funkcie (**Result**) na **True**.

Parameter **Pole** je lokálna premenná - pole môžeme modifikovať - nemá to vplyv na premennú, ktorú sme ako hodnotu do funkcie poslali. Môžeme to otestovať takouto funkciou:

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

function PocetNulovych(A: TPole): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    if A[I] = 0 then
      Inc(Result)
    else
      A[I] := 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Pole: TPole;
  I: Integer;
begin
  for I := 1 to 100 do
    Pole[I] := I mod 10;
  Memo1.Lines.Append(IntToStr(PocetNulovych(Pole)));
  Pole[1] := 0;
  Memo1.Lines.Append(IntToStr(PocetNulovych(Pole)));
end;
```

Zapísali sme funkciu **PocetNulovych**, ktorá počíta počet nulových prvkov poľa. Všimnite si, že okrem toho všetky nenulové prvky v poli vynuluje. My ale vieme, že parameter **A** je len lokálnou premennou a funkcia s ňou môže "*robiť čo len chce*", nemá to žiaden vplyv na hodnotu premennej **Pole**, ktorú sme sem poslali. Samotný program (pri zatlačení tlačidla **Button1**), najprv do poľa priradí nejaké hodnoty

(každá desiata hodnota je 0) a potom prvýkrát zistí počet nulových prvkov. Podľa očakávania vypíše hodnotu 10. Potom prvý prvok vynuluje a znovu zistí počet nulových prvkov. Teraz je ich už o jedna viac a program preto vypíše 11. Je jasné, že žiadne nulovanie nenastalo.

Čo sme sa naučili

Parametrom funkcie môže byť aj pole. Vtedy musí byť typ tohto poľa deklarovaný ako novy typ pomocou **type**. Ak parameter typu pole je len lokálna premenná funkcie a preto zmena jeho prvkov vo vnútri funkcie, nemá vplyv na premennú, ktorej hodnotu sme poslali do funkcie.

Ukázali sme použitie nového príkazu **Exit** na vyskočenie z podprogramu.

V tejto časti sme ukázali nasledovné techniky:

- efektívne a neefektívne prezeranie prvkov poľa.

Úlohy na precvičenie

Zadanie 1	Zapíšte logickú funkciu **JeRastuca**, ktorá o prvkoch poľa zistí, či tvoria rastúcu postupnosť.
Zadanie 2	Zapíšte funkciu **TextPola**, ktorá hodnoty prvkov celočíselného poľa zapíše ako znakový reťazec. Hodnoty sú oddelené znakom čiarka.
Zadanie 3	Zapíšte funkciu **IndexZaporneho**, ktorá vyhľadá v poli prvý výskyt prvku so zápornou hodnotou. Ak sú všetky prvky v poli nezáporné, funkcia vráti hodnotu **Low(Pole)-1**.
Zadanie 4	Zapíšte funkciu **SuRovnake**, ktorá má dva parametre typu pole. Funkcia zistí, či sú obe polia rovnaké.
Zadanie 5	Zapíšte funkcie **Min** a **Max**, ktoré nájdu minimálnu a maximálnu hodnotu v poli.
Zadanie 6	Zapíšte funkciu **PocetNadpriemerom**, ktorá pre vstupný parameter reálne pole zistí koľko prvkov v poli má nadpriemernú hodnotu. Napr. pre pole nameraných teplôt by sme zistili, koľko dní bola nadpriemerná teplota.

3. Pole ako výsledok funkcie

Doteraz sme sa stretli len s funkciami, ktorých typ výsledku boli buď jednoduché typy (celé čísla, znaky a logické hodnoty) alebo znakové reťazce. Typom výsledku funkcie môže byť ale aj pole. Podobne ako parameter typu pole, musí byť aj typ funkcie zadefinovaný ako nový typ pomocou popisu **type**. Totiž zápis

```
function NahodnePole(N: Integer): array [1..100] of Integer;
```

nie je v Pascale prípustný. Treba to zapísať napr. takto

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

function NahodnePole(N: Integer): TPole;
```

Funkcie, ktorých výsledkom je pole, vytvárame úplne rovnako, ako reťazcové funkcie. Vtedy bola premenná **Result** znakovým reťazcom a my sme do nej mohli priraďovať hodnoty, a tiež indexovať prvky reťazca.

Podobne je to s poľami. Vo funkcii, ktorá je typu napr. **TPole**, je automaticky zadeklarovaná lokálna premenná **Result** tiež typu **TPole**. Tak, ako aj iné lokálne premenné, tak aj **Result** má na začiatku nedefinované všetkých hodnoty prvkov.

Začnime s jednoduchou funkciou, ktorá vráti inicializované pole nejakou hodnotou:

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

function InicalizujPole(N: Integer): TPole;
var
  I: Integer;
begin
  for I := 1 to High(Result) do
    Result[I] := N;
end;
```

alebo veľmi podobná funkcia, ktorá priradí prvkom poľa nejaké náhodné hodnoty:

```
function NahodnePole(N: Integer): TPole;
var
  I: Integer;
begin
  for I := 1 to High(Result) do
    Result[I] := Random(N);
end;
```

V programe by sme potom mohli zapísať:

```
var
  A, B: TPole;
begin
  A := InicalizujPole(1);
  B := NahodnePole(21);
```

Najprv sa 10-prvkové pole **A** inicializuje hodnotou 1. V skutočnosti sa vo funkcii **InicializujPole** vytvorí 10-prvkové pole, do ktorého sa priradia jednotky a potom sa toto pole priraďovacím príkazom prekopíruje do premennej **A**. Podobne to bude s poľom **B**, v ktorom budú náhodné čísla od 0 do 20.

Predpokladajme, že z predchádzajúcich častí sme vytvorili funkciu, ktorá z prvkov poľa vytvorí znakový reťazec - prvky sú tu oddelené čiarkami, napr. :

```
function TextPola(P: TPole): string;
var
  I: Integer;
begin
  Result := '';
  for I := 1 to High(P) do
    Result := Result + IntToStr(P[I]) + ',';
end;
```

Teraz môžeme obe polia **A** aj **B** vypísať:

```
var
  A, B: TPole;
begin
  A := InicalizujPole(1);
  B := NahodnePole(21);
  Memo1.Lines.Append('A = ' + TextPola(A));
  Memo1.Lines.Append('B = ' + TextPola(B));
end;
```

do textovej plochy vypíše:

```
A = 1,1,1,1,1,1,1,1,1,1,
B = 13,8,9,6,18,1,20,5,8,10,
```

Teraz zapíšeme opačnú funkciu k **TextPola**: funkcia dostáva reťazec, v ktorom sú čísla oddelené čiarkami. Týmito hodnotami zaplní prvky poľa. Ak by sa ale pritom stalo, že je v reťazci menej takýchto hodnôt ako je prvkov poľa, program by mal zvyšné prvky vynulovať. V tomto podprograme využijeme ďalšiu štandardnú funkciu **StrToIntDef**, ktorá väčšinou funguje rovnako ako **StrToInt** a teda z reťazca prečíta číslo. Lenže funkcia **StrToInt** v prípade chyby spadne a program vyhlási nejakú nepríjemnú správu. **StrToIntDef** v prípade chyby v reťazci nepadá, ale vtedy použije druhý parameter funkcie, čo je náhradná hodnota výsledku. Napr.

```
StrToIntDef(' 512', -1)     vráti hodnotu 512
StrToIntDef(' 512x', -1)    vráti náhradnú hodnotu -1
StrToIntDef('', 0)          vráti náhradnú hodnotu 0
```

Funkcia, ktorá zo znakového reťazca poskladá prvky poľa, teraz vyzerá takto:

```
function ZTextu(S: String): TPole;
var
  I, J: Integer;
begin
  for I := 1 to High(Result) do
  begin
    J := Pos(',', S + ',');
    Result[I] := StrToIntDef(Copy(S, 1, J-1), 0);
    Delete(S, 1, J);
  end;
end;
```

Ak túto funkciu otestujeme napr.

```
A := ZTextu('10,20,30,,,7,8');
B := ZTextu(TextPola(A));
Memo1.Lines.Append('A = ' + TextPola(A));
Memo1.Lines.Append('B = ' + TextPola(B));
```

dostaneme:

```
A = 10,20,30,0,0,7,8,0,0,0,
B = 10,20,30,0,0,7,8,0,0,0,
```

Všimnite si, že kvôli otestovaniu funkčnosti tejto novej funkcie, sme pole **B** vytvorili ako kópiu poľa **A**, ale komplikovaným spôsobom: z poľa A sme vyrobili reťazec, v ktorom sú čísla oddelené čiarkami (**TextPola(A)**) a potom sme tieto znaky spätne prerobili na čísla (pomocou **ZTextu**). Vidíme, že sme dostali rovnaké obsahy oboch polí.

Ďalšie príklady ukážu funkcie, ktoré dostávajú parameter pole a výsledkom je tiež pole. Prvá funkcia **Pricitaj1** vráti pole, ktoré je rovnaké ako parameter, ale prvý prvok je zväčšený o 1:

```
function Pricitaj1(P: TPole): TPole;
begin
  Result := P;
  Result[1] := Result[1] + 1;
end;
```

Funkcia najprv celý obsah poľa **P** prekopíruje do premennej **Result** a potom tu opraví prvý prvok. Ďalšia funkcia vráti otočené pole, t.j. prvky poľa budú v opačnom poradí:

```
function Otoc(P: TPole): TPole;
var
  I: Integer;
begin
  for I := Low(Result) to High(Result) do
    Result[I] := P[High(P) - I + 1];
end;
```

Vidíme, že tu sme vytvárali pole **Result** postupne po jednom prvku, podobne ako napr. vo funkcii **InicializujPole**. Ešte jedna funkcia predvedie funkciu s poľom:

```
function Nezaporne(P: TPole): TPole;
var
  I: Integer;
begin
  for I := Low(Result) to High(Result) do
    if P[I] >= 0 then
      Result[I] := P[I]
    else
      Result[I] := 0;
end;
```

Táto funkcia necháva v poli len nezáporné prvky, všetky záporné vynuluje. Na podobnom princípe môžeme vytvárať najrôznejšie algoritmy, ktoré spracovávajú prvky poľa, napr. náhodne ich zamiešať, presunúť nulové prvky na koniec, cyklicky posunúť prvky o jednu pozíciu vľavo alebo vpravo.

Na záver funkcia, ktorá má dva parametre - dve rovnako veľké polia. Funkcia vytvorí pole, ktorého prvky vzniknú ako súčet zodpovedajúcich prvkov v daných dvoch poliach (niečo ako vektorový súčet):

```
function Scitaj(P, Q: TPole): TPole;
var
  I: Integer;
begin
  for I := Low(Result) to High(Result) do
    Result[I] := P[I] + Q[I];
end;
```

Ešte otestujeme niektoré z uvedených funkcií:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  A, B: TPole;
begin
  A := ZTextu('-1,1,-2,2,-3,3,-4,4,-5,5,');
  B := Otoc(Nezaporne(A));
  Memo1.Lines.Append('A = ' + TextPola(A));
  Memo1.Lines.Append('B = ' + TextPola(B));
  Memo1.Lines.Append('A + B = ' + TextPola(Scitaj(A, B)));
end;
```

do textovej plochy vypíše:

```
A = -1,1,-2,2,-3,3,-4,4,-5,5,
B = 5,0,4,0,3,0,2,0,1,0,
A + B = 4,1,2,2,0,3,-2,4,-4,5,
```

Vidíme, že pole B je vytvorené z poľa A tak, že sa z neho vyhodili záporné prvky a pole sa ešte otočilo. Okrem toho vidíme, ako vznikol súčet dvoch polí.

Čo sme sa naučili

Na príkladoch sme videli, ako vytvárame funkcie, ktorých výsledkom je pole. Pole môžeme buď generovať na základe nejakého predpisu, alebo nejako spracovať vstupné parametre, ktoré sú tiež typu pole.

Úlohy na precvičenie

Zadanie 1	Zostavte funkciu **Prvocinitele**, ktorá zadané číslo rozloží na prvočinitele (prvočíselné delitele) a uloží ich do výsledného poľa. Ak je prvočiniteľov menej, ako veľkosť poľa, pole doplní nulami.
Zadanie 2	Zostavte funkciu **Pomiesaj**, ktorá náhodne pomieša prvky vstupného poľa.
Zadanie 3	Zostavte funkciu **LenMocniny2**, ktorá ako parameter dostáva nejaké celočíselné pole. Výsledkom je logické pole rovnakej veľkosti ako vstupné pole. **I**-ty prvok výsledku bude **True** vtedy, keď i-ty prvok poľa obsahuje číslo, ktoré je mocninou 2.

Zadanie 4	Predpokladajte, že v dostatočne veľkých poliach X a Y máme uložené súradnice nejakej kresby. Vytvorili sme ich napr. pri ťahaní myšou v grafickej ploche. Vytvorte funkcie, ktoré budú manipulovať s touto kresbou, napr. posunú o konštantu niektorú súradnicu, vynásobia všetky prvky poľa nejakou reálnou konštantou (výsledok zaokrúhlia). Napíšte procedúry, ktorá potom túto kresbu nakreslí. Napr.
	
	`X := Posun(X, -100); Y := Nasob(Y, 0.3);`
	`Kresli(X, Y);`
	
	Ak je v poliach menej súradníc, ako je dĺžka pola, pole je doplnené hodnotou -1. S touto hodnotou by už funkcie pracovať nemali.

Zadanie 5	Zostavte funkciu **Fibonacci**, ktorá do výsledného poľa zapíše prvky Fibonacciho postupnosti. Funkcia je bez parametrov a zaplní celé pole. Funkcia by mala sledovať, či vytvárané číslo nepretečie celočíselnú aritmetiku a vtedy do poľa do týchto prvkov zapíše hodnoty 0.

4. Pole cifier

Záver tohto modulu sa venuje simulácii veľkej aritmetiky v poli cifier. Niečo z tohto sme trénovali vtedy, keď sme sa zoznamovali s poľami. V tejto časti vybudujeme niekoľko užitočných funkcií, ktoré budú vedieť pracovať s takýmito veľkými číslami.

Veľké čísla budeme ukladať do poľa cifier tak, že posledná cifra v desiatkovom zápise bude v prvom prvku poľa, ďalšia cifra v ďalšom prvku poľa, atď. Ak zadeklarujeme

```
type
  TVelkeCislo = array [1..10] of Integer;
var
  V: TVelkeCislo;
```

tak napr. číslo **1346269** bude uložené v poli **C** takto

C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]	C[10]
9	6	2	6	4	3	1	0	0	0

Začnime najjednoduchšou funkciou, ktorá vynuluje celé pole a teda vlastne vytvorí veľké číslo 0:

```
function Vynuluj: TVelkeCislo;
var
  I: Integer;
begin
  for I := 1 to High(Result) do
    Result[I] := 0;
end;
```

Zaujímavejšia bude funkcia, ktorá do veľkého čísla priradí nejakú "malú" celočíselnú konštantu. Samozrejme, že sa táto konštanta musí rozložiť na cifry a tie správne priradiť do poľa:

```
function Konstanta(X: Integer): TVelkeCislo;
var
  I: Integer;
begin
  for I := 1 to High(Result) do
  begin
    Result[I] := X mod 10;
    X := X div 10;
  end;
end;
```

Aby sme mohli vidieť veľké čísla aj vypísané, ďalšia funkcia ho prevedie na znakový reťazec:

```
function DoTextu(Cislo: TVelkeCislo): string;
var
  I: Integer;
begin
  Result := '';
  for I := 1 to High(Cislo) do
    Result := Char(Cislo[I] + 48) + Result;
end;
```

Otestujme tieto funkcie na malom príklade:

```
var
  C: TVelkeCislo;
  I: Integer;
begin
  C := Konstanta(1346269);
  Memo1.Lines.Append('C = ' + DoTextu(C));
end;
```

do textovej plochy vypíše:

```
C = 0001346269
```

Čo je hoci správne, ale nie je zvykom písať na začiatku čísel nuly. Chystáme sa pracovať s číslami, ktoré budú mať stovky alebo tisíce číslic, a tu by takéto veľké množstvo zbytočných núl vadilo. Musíme opraviť funkciu **DoTextu**. Skôr ako budeme vytvárať znakový reťazec, zistíme, kde začína prvá cifra, ktorú už chceme

vypisovať. Asi najlepšie je použiť while-cyklus:

```
function DoTextu(Cislo: TVelkeCislo): string;
var
  I, N: Integer;
begin
  N := High(Cislo);
  while (N > 1) and (Cislo[N] = 0) do
    Dec(N);
  Result := '';
  for I := 1 to N do
    Result := Char(Cislo[I] + 48) + Result;
end;
```

Všimnite si, že v poli prechádzame od posledného prvku, teda od najvyššej cifry, a postupne, kým sú tu ešte nuly, znižujeme tento index.

Prvú úlohu, ktorú budeme riešiť pomocou veľkých čísel bude faktoriál. O ňom vieme, že celočíselná aritmetika má problém spočítať už aj 13! Aby sme mohli prepísať algoritmus na výpočet faktoriálu pre veľké čísla, potrebujeme vedieť násobenie. Tu nám bude stačiť násobiť veľké číslo nejakým malým (typu **Integer**):

```
function Vynasob(Cislo: TVelkeCislo; X: Integer): TVelkeCislo;
var
  I, Prenos, Sucin: Integer;
begin
  Prenos := 0;
  for I := 1 to High(Result) do
  begin
    Sucin := X * Cislo[I] + Prenos;
    Result[I] := Sucin mod 10;
    Prenos := Sucin div 10;
  end;
end;
```

Funkcia na násobenie sa trochu podobá na vytváranie veľkej konštanty: postupne daným číslom násobíme všetky cifry a ak tento súčin presiahne 10, tak zo súčinu zoberieme len poslednú cifru (**mod** 10) a zapamätáme si, že k ďalšej cifre pripočítame prenos (**div** 10).

Aby sme tieto funkcie otestovať aj na väčších faktoriáloch, nesmieme zabudnúť zväčšiť aj definíciu typu **TVelkeCislo**. Zapíšme výpočet prvých 19 faktoriálov:

```
type
  TVelkeCislo = array [1..10000] of Integer;

procedure TForm1.Button1Click(Sender: TObject);
var
  Velke: TVelkeCislo;
  I: Integer;
begin
  Velke := Konstanta(1);
  for I := 1 to 19 do
  begin
    Velke := Vynasob(Velke, I);
    Memo1.Lines.Append(IntToStr(I) + '! = ' + DoTextu(Velke));
  end;
end;
```

Veríme, že samotný program je teraz veľmi dobre čitateľný. Samozrejme, že definícia typu **TVelkeCislo** musí byť v programe zapísaná ešte pred všetkými pomocnými funkciami.

Po spustení sa do textovej plochy vypíše:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

Čo sú naozaj správne výsledky.

Tento program bez problémov zvládne hoci aj 1000! Môžete to vyskúšať:

```
var
  Velke: TVelkeCislo;
  I: Integer;
begin
  Velke := Konstanta(1);
  for I := 1 to 1000 do
    Velke := Vynasob(Velke, I);
  Memo1.Lines.Append('1000! = ' + DoTextu(Velke));
end;
```

Tu v materiáli to vypisovať nebudeme, keďže toto číslo má vyše 2500 znakov.

Ešte ukážme výpočet členov Fibonacciho postupnosti. Aby sme mohli počítať túto postupnosť, potrebujeme funkciu na súčet dvoch veľkých čísel. Niečo podobné sme už robili, takže funkcia by nemusela byť náročná na pochopenie:

```
function Scitaj(Cislo1, Cislo2: TVelkeCislo): TVelkeCislo;
var
  I, Prenos, Sucet: Integer;
begin
  Prenos := 0;
  for I := 1 to High(Result) do
  begin
    Sucet := Cislo1[I] + Cislo2[I] + Prenos;
    Result[I] := Sucet mod 10;
    Prenos := Sucet div 10;
  end;
end;
```

Počítanie členov Fibonacciho postupnosti ste pravdepodobne robili pre obyčajnú celočíselnú aritmetiku. Už 47. číslo v postupnosti pretečie celočíselnú aritmetiku (je väčšie ako **MaxInt**). Preto vyskúšame vypočítať prvých 100 čísel postupnosti:

```
var
  F1, F2, F3: TVelkeCislo;
  I: Integer;
begin
  F1 := Konstanta(1);
  F2 := F1;
  Memo1.Lines.Append('F(1) = ' + DoTextu(F1));
  Memo1.Lines.Append('F(2) = ' + DoTextu(F2));
  for I := 3 to 100 do
  begin
    F3 := Scitaj(F1, F2);
    Memo1.Lines.Append('F(' + IntToStr(I) + ') = ' + DoTextu(F3));
    F1 := F2;
    F2 := F3;
  end;
end;
```

Po spustení sa vypíše prvých 100 čísel (z výpisu tu zobrazujeme len niekoľko častí):

```
F(1)  = 1
F(2)  = 1
F(3)  = 2
F(4)  = 3
F(5)  = 5
F(6)  = 8
F(7)  = 13
F(8)  = 21
F(9)  = 34
F(10)  = 55
...
F(46)  = 1836311903
F(47)  = 2971215073
F(48)  = 4807526976
...
F(98)  = 135301852344706746049
F(99)  = 218922995834555169026
F(100)  = 354224848179261915075
```

Čo sme sa naučili

Ako reprezentovať veľké čísla v poli cifier. Ako by mohli vyzerať niektoré základné aritmetické operácie s takýmito číslami.

V tejto časti sme ukázali nasledovné techniky:

- výpis veľkého čísla
- násobenie veľkého čísla konštantou
- súčet dvoch veľkých čísel.

Úlohy na precvičenie

Zadanie 1	Vytvorte funkcie na výpočet všetkých cifier $\mathbf{2^n}$, $\mathbf{3^n}$, resp. s iným základom.
Zadanie 2	Podobnú aritmetiku môžeme vytvoriť aj pre desatinnú časť čísla. Algoritmy sú ale náročnejšie. Vypočítajte s presnosťou napr. na 100 miest • **e** (základ prirodzených logaritmov) • $\mathbf{2^{-n}}$ (t.j. $\mathbf{1/2^n}$) • **π** (Ludolfovo číslo)

Čo sme sa naučili v tomto module

Zhrnutie

Tento modul zoznámil s týmito základnými stavebnými kameňmi programovacieho jazyka:

- funkcia
- zadefinovanie vlastného typu najmä pre popis parametrov a výsledku funkcie

Okrem toho boli uvedené tieto dôležité koncepty:

- práca s premennou Result v tele funkcie
- vyskočenie z podprogramu pomocou príkazu Exit

Preverenie výstupných vedomostí

Účastník vzdelávania vie naprogramovať takúto aplikáciu:

> Program najprv umožní kresliť myšou do grafickej plochy a pritom ukladá do dvoch celočíselných poli **X** a **Y** súradnice spájaných bodov kresby. Po skončení kreslenia doplní pole konštantou -1 a pomocou funkcie **Dlzka** s dvoma parametrami **X** a **Y** vypočíta dĺžku krivky. Zabezpečte, aby funkcia do výslednej dĺžky nezapočítala doplnené hodnoty -1.

Literatúra a použité zdroje

Základné materiály:

[1] Blaho A., "Informatika pre stredné školy. Programovanie v Delphi", SPN Bratislava, 2006
[2] Cieľové požiadavky na vedomosti a zručnosti maturantov z informatiky, Štátny pedagogický ústav, http://www.statpedu.sk/

Internetové zdroje pre prostredie Delphi

[3] prijímacie pohovory z informatiky na FMFI UK, http://www.prijimacky.input.sk/
[4] Delphi Programming, http://delphi.about.com/
[5] Marco Cantu, http://www.marcocantu.com/
[6] Torry's Delphi Pages, http://www.torry.net/

Internetové zdroje pre prostredie Lazarus

[7] Lazarus - vysokoškolské prednášky na FMFI UK, http://www.pascal.input.sk/
[8] Lazarus wiki, http://wiki.lazarus.freepascal.org/Main_Page/sk
[9] Lazarus Documentation/sk, http://wiki.lazarus.freepascal.org/Lazarus_Documentation/sk
[10] Free pascal, http://www.freepascal.org/

Tento študijný materiál vznikol ako súčasť národného projektu Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika v rámci Aktivity „Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ“.

Autori © RNDr. Andrej Blaho
RNDr. Ľubomír Salanci, PhD.

Názov Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Podnázov Programovanie 9

Študijný materiál prešiel recenzným pokračovaním.

Recenzenti doc. RNDr. Gabriela Andrejková, CSc.
doc. RNDr. Stanislav Krajči, PhD.

Počet strán 40

Náklad 300 ks

Prvé vydanie, Bratislava 2009

Všetky práva vyhradené.

Toto dielo ani žiadnu jeho časť nemožno reprodukovať bez súhlasu majiteľa práv.

Vydal Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, v súčinnosti s Univerzitou Pavla Jozefa Šafárika v Košiciach, Univerzitou Komenského v Bratislave, Univerzitou Konštantína Filozofa v Nitre, Univerzitou Mateja Bela v Banskej Bystrici a Žilinskou univerzitou v Žiline

Vytlačil BRATIA SABOVCI, s r.o., Zvolen

ISBN 978-80-8118-025-5