Stretnutie s Pascalom II.

1. časť

Predchádzajúcou časťou sme ukončili výučbu štruktúrovaného programovania. Viem, že rozsah seriálu nestačil na vyškolenie profesionálneho programátora. Bolo by vhodné niektoré oblasti vysvetliť podrobnejšie, ale z rozsahových dôvodov to v časopise nie je možné. Keď som však videl programy alebo procedúrky niektorých z vás, čo boli predtým úplní začiatočníci, zistil som, že aj to málo pomohlo. Cieľom bolo zaujať vás, nadchnúť pre Pascal a naviesť vás na jeho cestu. Len neustálym programovaním sa programátor zdokonaľuje, a keď zistí, že už toho vie dosť (zámerne som nenapísal "všetko", lebo nikdy nevie všetko) a zložitejšie programy sú neprehľadné, alebo by chcel skúsiť naprogramovať niečo pod Windows, narazí na obmedzenia štruktúrovaného programovania.
Ale čo ďalej? Preto sa dnes začneme venovať novej oblasti v programovaní, ktoré sa nazýva objektovo orientované programovanie, skrátene OOP. Vlastne to ani nie je také nové, princípy boli položené už v roku 1967 (Zdá sa to neuveriteľné, ale je to tak!), ale mnohým z nás to bude pripadať ako nové, zvláštne, zložité, nepochopiteľné a nepotrebné... Pravda, je to zložité, ale pochopiť sa dá (časom), navyše to budeme potrebovať, ak chceme písať už spomínané veľké a zložité projekty, ale hlavne ak chceme začať programovať vo Windows. Preto si spolu prejdeme základy OOP a potom sa rozdelíme na dva smery:
– tí, čo zostanú verní DOS-u, sa budú venovať tzv. TURBO VISION (TV), čo je mocný nástroj na programovanie veľmi zložitých projektov, ako je účtovníctvo, databázové projekty, skladové hospodárstvo, cestovné poriadky a pod. TV je postavené na objektoch, je súčasťou programovacieho balíka Turbo Pascal 6 a 7. Čo TV dokáže, to vidieť hneď, lebo prostredie TP je napísané v samotnom Turbo Vision. Tento kurz povedie môj priateľ a výborný pascalista - objektista Štefan Stieranka.
– my, čo by sme si chceli vyskúšať, ako chutia Windows a programovanie v nich, sa stretneme s Delphi, novým programovacím produktom, svetovou jednotkou firmy BORLAND, postavenou na Turbo Pascale, v ktorom sa dá aj s minimálnymi základnými znalosťami programovania v TP vytvoriť dobrý program.

No ale aby sme mohli niečo začať, musíme sa trochu potrápiť s teóriou.

História OOP

OOP sa považuje za najmodernejšiu a najprogresívnejšiu programátorskú techniku. Prvé princípy boli použité v jazyku SIMULA 67 (67 je rok vzniku), a teda skutočne nie sú nové, ich masové rozšírenie umožnila až firma BORLAND so svojím Turbo Pascalom verzie 5.5.

Tento dobrý nápad bol rozpracovaný vo verzii 6.0, kde vzniklo aj spomínané Turbo Vision. V súčasnosti je na trhu Turbo Pascal 7.0 s Turbo Vision 2.0 a ich windowsovská implementácia Delphi. Ako už samotný názov prvého objektovo orientovaného jazyka SIMULA naznačuje, bol určený na vytváranie programov simulujúcich a modelujúcich niektorý konkrétny jav. Teda pôvodným zmyslom vzniku metód OOP bola simulácia a modelovanie.

Nejde len o techniku programovania, je to celá programátorská filozofia, myslenie odpozorované z reálneho. Preto ak chceme pochopiť OOP, musíme zabudnúť na štruktúrované princípy (ale nie úplne) a snažiť sa porozumieť objektom.

Základné princípy OOP

Základným pojmom v OOP je objekt. Môžeme ho chápať ako abstrakciu objektu z reálneho sveta. A aby sme nechodili ďaleko, taký reálny objekt máme na stole. Budeme hovoriť o počítačoch. Počítač je (ne)vďačný objekt, ktorý má určité vlastnosti, napr. šírku zbernice a veľkosť operačnej pamäte, no má aj určité schopnosti, napr. vie počítať alebo pípať.

Objekt v zmysle OOP je akýmsi rozšírením dátového typu záznam (record), ktorý už dávno poznáme. Ten umožnuje vyjadriť obidva uvedené typy charakteristík. Vlastnosťou je šírka zbernice a veľkosť pamäte, ktoré sú prezentované dátovými položkami rovnako ako vo zvyčajnom zázname. Schopnosti sú opísané procedúrami, v OOP nazývanými metódy, ktoré sú súčasťou objektu.

Toto už klasický záznam neumožňoval. V Turbo Pascale môžu byť metódami tak procedúry, ako aj funkcie. Potom môžeme počítač zadefinovať ako objekt takto:

```
Pocitac = object
sirka_zbernice : (rozna, 8, 16, 32, 64);
velkost_pamate : string;
procedure Pipni;
procedure Pocitaj;
```

```
end;
```

Týmto zápisom sme zaviedli triedu (class) objektov, nazvanú Pocitac. V terminológii OOP všetky objekty s rovnakými vlastnosťami patria do jednej triedy. Opis triedy teda zodpovedá definícii typu. Konkrétna realizácia objektu triedy daného typu (vlastne pamäťová premenná) sa v slovenskej (anglickej) terminológii nazýva **inštancia** (instance).

Vidíme, že OOP umožňuje združovať logicky súvisiace dáta a kód (metódy), ktoré s týmito dátami pracujú. Hovoríme, že dáta aj metódy sú vnútri objektu zapuzdrené a táto vlastnosť sa nazýva **zapuzdrenie** alebo **encapsulation**. Táto vlastnosť OOP je prvým z troch najdôležitejších princípov celej filozofie OOP.

Teraz si predstavme, že pred nami stojí ZX Spectrum (ach, zlatý Sinclair...) a átečko. Čo o nich môžeme povedať? Ide o dva počítače, majú teda určitú šírku zbernice a veľkosť pamäte, pípajú a vedia počítať. Zároveň sú však každý iný. Spectrum bol skonštruovaný na hry, ale pracovať s programami typu CAD by sa s ním asi nedalo, zatiaľ čo átečko je určené na serióznu prácu, ale spúšťať na ňom iba hry by bolo zbytočným prepychom. V OOP by sme naznačenú situáciu opísali takto:

```
Sinclair = object (pocitac)
procedure Hraj_Hry;
end;
Atecko = object (pocitac)
procedure Konstruuj;
end;
```

Objekty Sinclair a Atecko teraz majú všetky vlastnosti a vedia všetko to, čo vie objekt pocitac. Každý z nich však vie oproti počítaču navyše niečo, čo nevie ten druhý. Objekt pocitac sa nazýva **predok** (*ancestor*), objekty Sinclair a Atecko sú **potomkovia** (*descendant*). Opísaná črta, spočívajúca v odovzdávaní vlastnosti predka potomkovi sa rovnako ako v živote nazýva **dedičnosť** (*inheritance*). Je to druhý základný princíp OOP.

Predstavme si, že počítanie, teda metódu *Pocitaj* budeme na počítači simulovať tak, že na obrazovku PC budeme vypisovať parametre toho-ktorého počítača. Vypisovanie realizujeme zvláštnou metódou *Urci_Typ_procesora*. Túto metódu bude využívať metóda Pocitaj na simuláciu výpočtu. Definíciu objektu teda rozšírime:

```
pocitac = object
          sirka_zbernice : (rozna, 8,
16, 32, 64);
          procedure Urci_Typ_procesora;
          procedure Pipni;
          procedure Pocitaj;
end;
```

a kód pre metódu *Pocitaj* bude obsahovať volanie metódy *Urci_Typ_procesora*:

```
procedure Pocitaj;
begin
      .
      .
      .
Urci_Typ_procesora;
      .
      .
      .
end;
```

Čo sa teraz stane, ak použijeme metódu *Pocitaj* pre inštanciu objektu *Sinclair* a rovnakú metódu pre objekt *Atecko*?

Predpokladáme, že triedy *Sinclair* a *Atecko* sú potomkovia novo nadefinovanej triedy objektov *Pocitac*. Odpoveď znie: na obrazovke uvidíme výpis - typ procesora, šírku zbernice a veľkosť pamäte, ale nebude to ani Sinclair, ani átečko.

Lenže čo vtedy, ak chceme, aby sa výpisy líšili a zodpovedali typu počítača? Postačí, ak nadefinujeme v objektoch *Sinclair* a *Atecko* vlastnú metódu *Urci_Typ_procesora*? Nie, situácia sa ani týmto krokom veľmi nezmení. Ak vyvoláme teraz metódu *Urci_Typ_procesora* z inštancie objektu Sinclair, vypíše sa na obrazovku typ Z80. Ak vyvoláme z tej istej inštancie metódu Pocitaj, bude sa opisovať všeobecný počítač.

Je to preto, lebo všetky metódy boli definované ako statické, a tak došlo k tzv. včasnej väzbe (*early binding*). To znamená asi toľko, že prekladač predpokladá, že pozná všetky okolnosti už v dobe prekladu a vygeneruje kód, kde je v metóde *Pocitaj* natvrdo volaná metóda *Urci_Typ_procesora* z triedy *Pocitac*. Riešením je použitie mechanizmu **polymorfizmu** (tretí princíp). Navonok sa to prejaví tak, že metóda *Urci_Typ_procesora* bude vo všetkých triedach definovaná ako **virtuálna**:

```
pocitac = objekt
          .
          .
          .
procedure Urci_Typ_procesora;
virtual;
procedure Pocitaj;
          .
          .
          .
end;

Sinclair = object (pocitac);
          .
          .
          .
procedure Urci_Typ_procesora;
virtual;
          .
end;

Atecko = object (pocitac);
          .
          .
procedure Urci_Typ_procesora;
virtual;
          .
end;
```

Teraz bude všetko pracovať podľa predpokladov. Prekladač totiž po tom, čo zistil, že sa použila virtuálna metóda, vytvoril pre každú z nadefinovaných tried objektov tzv. tabuľku virtuálnych metód (TVM). V kóde metódy Pocitaj teraz nie je priamy skok na kód metódy Urci_Typ_procesora, ale volanie sa uskutoční odkazom do tejto tabuľky. **Až pri behu programu sa zistí, do tabuľky ktorej triedy sa má siahnuť po adresu metódy** *Urci_Typ_procesora*. Toto je stručný opis mechanizmu neskorej väzby (late binding).

Z neho je možné odvodiť, že ak je metóda v predkovi definovaná ako virtuálna, musí byť virtuálna aj vo všetkých potomkoch. Toto možno chápať ako súčasť syntaktických pravidiel jazyka, ktoré si ešte preberieme podrobnejšie.

Na listingu č. 1 je výpis zdrojového textu programu **OOP1.PAS**. Môžete si ho skúšať a ladiť, zmeňte virtuálne metódy na statické jednoduchým zmazaním slova virtual, preložte a sledujte jeho správanie. Mnoho vecí sa vám objasní. Zásady pri písaní takýchto programov, ako aj tento konkrétny program si podrobnejšie vysvetlíme nabudúce.

Listing 1:

```
program OOP;
uses crt;
type tbity = (rozna, osem,
sestnast, tridsatdva, sestde-
siatstyri) ;
```

```
(*********)
pocitac = object
sirka__zbernice : tbity;
velkost_pamate : string;
constructor
Init(abity:tbity;pamat :
string);
procedure Pipni;
procedure Urci_Typ_procesora;
virtual;
procedure Pocitaj;
end;
Sinclair = object (pocitac)
procedure Urci_Typ_procesora;
virtual;
end;
atecko = object (pocitac)
procedure Urci_Typ_procesora;
virtual;
destructor Done;
end;
patecko = ^atecko;


styri86 = object (pocitac)
procedure Urci_Typ_procesora;
virtual;
end;
Pentium = object (pocitac)
procedure Urci_Typ_procesora;
virtual;
end;
(*********)
constructor
pocitac.Init(abity:tbity;pama
t : string);
begin
sirka_zbernice := abity;
velkost_pamate := pamat;
end;
procedure Pocitac.Pipni;
begin
write(#7);
delay(50) ;
write(^G);
end;


procedure
Pocitac.Urci_Typ_procesora;
begin
write(' rôzny ');
end;
(*********)
procedure Pocitac.Pocitaj;
var ch : char;
procedure Vypis;
begin
write('Typ procesora : ');
```

```
Urci Typ_procesora;
write('; Zbernica : ')';
case sirka_zbernice of
rozna : write('rozna ');
osem : write(' 8 ') ;
sestnast : write(' 16 ');
tridsatdva : write(' 32 ');
sestdesiatstyri : write(' 64
');
end;
write(' bitov; Pamat : ');
writeln(velkost_pamate);
end;
begin
Vypis;
Pipni;
delay(5000);
end;
(*********)
procedure
Sinclair.Urci_Typ_procesora;
begin
write(' Z80 ');
end;
procedure
atecko. Urci_Typ_procesora;
begin
write(' 80286 ');
end;
destructor atecko.Done;
begin
end;
procedure
styri86.Urci_Typ_procesora;
begin
write (' 80486 ') ;
end;
procedure
Pentium.Urci_Typ_procesora;
begin
write(' Pentium ');
end;
(*********)
var pcl : pocitac;
pc2 : Sinclair;
pc3 : patecko;
pc4 : Styri86;
pc5 : Pentium;


BEGIN clrscr;
pcl.Init(rozna,´lubovolna´);
pc2. Init (osem,´64 Kb´) ;
pc3:=New(patecko,Init(ses-tnast, ´l MB´));
pc4.Init(tridsatdva,´ 4 MB´);
pc5.Init(sestdesiatstyri,´32
MB´);
pcl.Pocitaj;
pc2.Pocitaj;
pc3^.Pocitaj;
pc4.Pocitaj;
```

```
pc5.Pocitaj;
Dispose(pc3,done) ;
END.
```

---

2. časť

Prišli sme na to, že OOP je skutočne niečo nové a zložité. Preto sa budeme k mnohým veciam vracať a vysvetľovať si ich z iného ohľadu tak, aby sme si ich zautomatizovali a hlavne aby sme ich pochopili. Ak ste sa pozerali na výpis programu OOP.PAS, ste si istotne niektoré nové pravidlá pri písaní zdrojového zápisu. Možnopovažujete tento program za bezúčelný a, samozrejme, vedeli by ste ho napísať inak a jednoduchšie klasickými technikami.

Bohužiaľ, v začiatkoch je to bežné, lebo krása OOP sa prejaví pri podstatne zložitejších a rozsiahlejších programoch. Ale na to musíte prísť sami... (Ja som sa bránil OOP niekoľko rokov, až Friendly Pascal a Delphi ma k nim priviedli. Škoda strateného času.) Preto nezúfajte, ak niečo nep ochopíte hneď. Neskôr v súvislosti s inými princípmi OOP sa vám všeličo objasní a zistíte, že bez OOP by to ani nešlo!
Ale poďme k dalšej teórii o OOP.

Definícia objektu, vytvorenie inštancie objektu
Objekt je v podstate nový dátový typ, jeho definícia sa teda uskutoční rovnako ako definícia iného používateľského dátového typu.

Je preň zavedené kľúčové slovo **object** , ktorého použitie je veľmi podobné použitiu kľúčového slova *record* na definíciu záznamu. Medzi kľúčové slová **record** a **end** sa uvedú deklarácie d átových položiek a hlavičiek procedúr a funkcií, ktoré v OOP predstavujú metódy objektov definovanej triedy. Vráťme sa k nášmu príkladua definujme objekt *pocitac*:

```
pocitac = object
     sirka_zbernice  :  tbity;
     velkost_pamate  :  string;
     constructor Init(abity  :  tbity;
            pamat  :  string);
     procedure Pipni;
     procedure Urci_Typ_procesora;
            virtual;
     procedure Pocitaj;
end;
```

Takto sme nadefinovali objekt, ktorý má d ve dátové položky - *sirka_zbernice* a *velkost_pamate* - a štyri metódy - *Init, Pipni, Urci_Typ_procesora, Pocitaj* . Význam použitých slov *constructor* a *virtual* si vysvetlíme neskôr. Ak chceme definovať objekt, ktorý je potomkom iného objektu, uvedieme meno pr edka do okrúhlych zátvoriek za slovo object:

```
atecko = object(pocitac)
   Urci_Typ_procesora; virtual;
end;
```

Takto nadefinovaný objekt atecko je potomkom objektu pocitac. My už vieme, že zdedil **všetky** vlastnosti (dáta) a sc hopnosti (metódy) objetku pocitac. Kód jednotlivých metód sa uvedie až za deklaráciou objektu. Pri definovaní metódy sa uvedie meno objektu oddelené bodkou:

```
procedure Pocitac.Pocitaj;
      begin
      .
      .
      .
end;
```

Inštanciu objektu vytvoríme tak, že deklarujeme premennú, ktorej typom bude meno definovaného

objektu:

```
var pc1  :  pocitac;
```

Teraz je premenná *pc1* inštanciou objektu pocitac.

Prístup k položkám obj ektu, rozsahy platnosti
Všetky dátové položky a metódy niektorej inštancie sú z ľubovoľného miesta programu, kde je táto inštancia platnou premennou, prístupné pomocou bodkového zápisu rovnako ako v zázname record. Ak teda použijeme v hlavnom programe zápis

```
pc1.Pocitaj;
```

vyvoláme tým metódu *Pocitaj* príslušnej inštancie pc1 objektu pocitac. Rovnakým spôsobom je možné pristupovať k dátovým položkám objektu, teda

```
pamat := pc1.velkost_pamate;
```

je platný zápis, pokiaľ je premenná *pamat* typu string. Vnútri objektu sú dáta a metódy prístupné jednoduchým zápisom bez bodky:

```
procedure Pocitac.Pocitaj;
      begin
      .
      .
      Urci_Typ_Procesora;
      .
      .
end;
```

Pre rozsahy platnosti jednotlivých objektov platia bežné pravidlá iných dátových typov a premenných. Zvláštnosťou je, že všetky **dátové položky definované v predkovi sú platné aj v potomkovi a netreba ich v potomkovi znova deklarovať. Dátovú položku nie je možné v potomkovi predefinovať!** Ak teda v potomkovi uvedieme rovnaké meno dátovej položky ako v predkovi, dôjde ku kolízii a prekladač vyhlási chybu!

Pre metódy platí rovnaké pravidlo o rozsahu platnosti ako pre dátové položky. Rozdiel je však v tom, **že metódy definované v predkovi je možné v potomkovi predefinovať**! Ak v potomkovi použijeme meno metódy, ktorá je deklarovaná v predkovi, prekladač to akceptuje a metóda je predefinovaná. Počínajúc týmto objektom, v hierarchii objektov je platná novo nadefinovaná metóda. Táto možnosť sa v OOP veľmi využíva. Pre statické, teda nie virtuálne metódy platí, že metóda, ktorá predefinováva (prekrýva metódu predka), sa môže líšiť počtom a typom parametrov. Toto neplatí pre virtuálne metódy.

Prístup k metódam predka
Veľmi často potrebujeme v programe zavolať metódu predka, teda nie tú, ktorá ju predefinovala. Môžeme použiť dva spôsoby:
a) zápis plného mena metódy, teda

```
meno_predka.meno_metody;
```

tento spôsob je platný vo všetkých verziách Turbo Pascalu,
b) p oužitie kľúčového slova *inherited*, ktoré bolo doplnené vo verzii TP 7.0, teda takto:

```
inherited meno_metody;
```

Pokiaľ sa tento zápis objaví v definícii potomka, vyvolá vykonanie metódy definovanej v jeho predkovi.

Virtuálne metódy, konštruktory
Virtuálne metódy sú prostriedkom na využitie polyformizmu - mnohotvarosti objektov. Čiastočne sme si to naznačili v predchádzajúcej časti. Teraz si vysvetlíme pravidlá prekrývania virtuálnych metód. Ak

je už raz metóda definovaná ako virtuálna, musí byť ak o virtuálna definovaná v každom ďalšom potomkovi. Pri virtuálnej metóde nie je možné zmeniť počet ani typ parametrov. Záhlavie virtuálnych metód musí byť úplne rovnaké po celej hierarchii dedičnosti.

Ak je v danom objekte definovaná aspoň jedna virtuálna metóda, musí byť v objekte nevyhnutne definovaná metóda s kľúčovým slovom **constructor** . Konštruktor je zvláštna metóda, pri ktorej vyvolaní sa vytvorí spojenie medzi tabuľkou virtuálnych metód a aktuálnou inštanciou objektu. Toto spojenie sa potom využíva pri volaní niektorej z virtuálnych metód daného objektu. Ak zabudneme v objekte aspoň s jednou virtuálnou metódou zadeklarovať konšruktor, prekladač vyhlási chybu. **Konštruktor musí byť vyvolaný pred prvým volaním niektorej virtuálnej metódy objektu. Opomenutie toho spôsobí chybu za behu programu, ktorá nie je hlásená a jej prejavy bývajú rôzne.**

Dynamické alokácie objektov, deštruktory

Pretože objekty a ich inštancie (premenné) sú rovnocenné bežným dátovým typom a premenným, môžeme s nimi rovnako aj zaobchádzať. Teda je možné získavať ukazovatele na objekty, dynamicky ich alokovať a rušiť.

Dynamická alokácia objektov sa v technike OOP veľmi často používa a okrem iného zrýchľuje beh programu. Pre alokáciu objektov bola rozšírená syntax príkazov **New** a **Dispose**. Príkaz *New* má dve formy:

procedurálnu

```
New( ukazateľ [, konštruktor])
```

a funkčnú

```
ukazateľ = New(typ_ukazateľa [,konštruktor]).
```

Druhým parametrom je v oboch príkladoch konštruktor alokovaného objektu s prípadnými parametrami. Tento konštruktor sa po vykonaní alokácie automaticky vykoná. Samozrejme, ak objekt nepoužíva virtuálne metódy, konštruktor ako parameter sa neuvádza.

Prvý parameter v prípade procedurálnej verzie je premenná typu ukazovateľ, do ktorej sa priradí adresa alokovaného objektu. Tento parameter sa teda musí odovzdávať odkazom.

V prípade funkcionálneho volania má prvý parameter význam typu vytváraného objektu. Vlastná adresa je vrátená ako funkčná hodnota.

Uvoľnenie dynamicky alokovaného objektu z pamäte sa vykonáva pr íkazom **Dispose** :

```
Dispose( ukazateľ [, deštruktor]).
```

Prvý parameter je ukazovateľ na rušený objekt, druhý je tzv. deštruktor. Deštruktor je zvláštna metóda, ktorá sa definuje pomocou kľučového slova **destructor**:

```
Destructor Done;
```

Deštruktor vykonáva uvoľnenie dynamicky alokovaného objektu, pričom zabezpečí, že aj v prípade využitia mechanizmu polyformizmu bude **vždy** uvoľnené presne správne množstvo byteov pamäte. Robí sa to, samozrejme, pomocou tabuľky virtuálnych metód. Deštruktor musíme použiť vždy, ak máme objekt obsahujúci virtuálne metódy a chceme s ním zaobchádzať dynamicky. Telo deštruktora môže zostať v niektorých prípadoch prázdne.

Pozrime sa teraz podrobnejšie na náš prvý objektovo orientovaný program OOP.PAS:
Pre OOP je typic ké, že všetky možné funkcie programu sú nadefinované v deklaračnej časti a samotné telo programu, teda jeho výkonná časť obsahuje iba niekoľko riadkov. Pri práci s knižnicou Turbo Vision sa stretnete s hlavným telom programu, ktoré bude obsahovať iba tri (!) riadky. Všetko ostatné sa nachádza v deklaračnej časti. Aj náš program má veľmi bohatú deklaračnú časť. Po definícii využívania unitu *Crt* nadeklarujeme nový výčtový typ *tbity* , v ktorom je päť položiek. Skutočné objekty sa začínajú nadeklarovaním objektu *pocitac*. Jeho
dátové položky a metódy už poznáme. Všimnime si ešte raz, že v definovaní objektu sú uvedené iba hlavičky procedúr, ktoré priberajú úlohu metódy. Ich deklarácie sú na inom mieste programu.

Za definíciou objektu pocitac nasleduje deklarovanie nových objektov: *Sinclair, Atecko, Styri86, Pentium* . Každý z týchto objektov je potomkom objektu pocitac, lebo jeho meno je uvedené v zátvorkách. Čo sa nachádza v telách jednotlivých objektov? No predsa iba meno metódy *Urci_Typ_procesora,* ktorú chceme

pre definovať pre každého potomka zvlášť. Všimnime si, že už nemusíme uvádzať v potomkoch dátové položky a ostatné metódy. Tie sa automaticky zdedia od predka *pocitac* . Čo to konkrétne znamená? No to, že aj objekt *Atecko,* aj jeho "bratia" majú dáta *sirka_zbernice, velkost_pamate a metódy Init, Pípni a Pocitaj*, hoci nie sú definované v ich telách!
Všimnime si zápis

```
patecko = ^atecko;
```

Ide o deklaráciu typu *patecko* , čo nie je nič iné ako ukazovateľ na typ *atecko*. Toto som uviedol ako príklad na spomínanú dynamickú alokáciu objektu atecko. Po deklarácii potomkov nasledujú vlastné definície metód jednotlivých objektov.

Začneme pekne po poriadku - všeobecným objektom *pocitac* a konštruktorom *Init* . Vidíme, že Init nastaví dátové položky. Spravidla sa v konštruktoroch zadáva naplnenie dátových položiek, aby sme s nimi mohli ďalej pracovať. Metóda *Pocitac.Pipni* je zrejmá. Metóda *Pocitac.Urci_Typ_procesora* je tá, ktorú zadefinujeme ako virtuálnu, a preto ju budeme v každom potomkovi predefinovávať. V tomto prípade ide o všeobecný počítač, a tak deklarujeme, aby táto metóda tohto objektu vypísala "rôzny". Najmohutnejšia metóda je *Pocitac.Pocitaj*.

Obsahuje procedúru Vypis, ktorá volá vo svojom tele všeobecnú virtuálnu metódu *Urci_Typ_procesora*. Príkazy na výpis pomocných textov sú jasné. V tele metódy *Pocitaj* je zavolaná procedúra *Vypis* a metóda *Pipni*. Takto sme nadeklarovali všetko o objekte pocitac.

Pristúpme teraz k deklarácii metód ostatných objektov - potomkov. Už vieme, že nemusíme deklarovať tie metódy, ktoré sú nadeklarované v predkovi. Na správnu funkciu programu stačí, ak predefinujeme metódu, ktorá opisuje typ procesora. Preto v jednotlivých potomkoch - v *objektoch Sinclair, Atecko, Styri86*
a Pentium - predefinujeme metódu *Urci_Typ_procesora*. Ostatné metódy jednoducho zdedíme!

Nesmieme zabudnúť na deklaráciu deštruktora *Done*, ktorý obsahuje objekt *Atecko* , aby sme zabezpečili správne uvoľnenie pamäte.

Na záver deklaračnej časti programu definujeme premenné *pc1* až *pc5*, ktoré sú takého typu, aký sme nadekl arovali, teda pc1 je typu pocitac, pc2 typu Sinclair, pc3 typu patecko atď. Hovoríme, že sme vytvorili inštancie jednotlivých objektov. Vlastné telo programu obsahuje výmaz obrazovky a po ňom volanie konštruktorov jednotlivých inštancií s konkrétnymi para metrami.

Keďže chceme objekt *Atecko* alokovať dynamicky, musíme uviesť zápis s kľúčovým slovom New. Teraz si všimnime hlavnú myšlienku OOP. Vyvoláme metódu *Pocitaj* pre jednotlivé inštancie bez toho, aby sme ju definovali pre každú inštanciu zvlášť! A tu je ten objektovo orientovaný pes zakopaný! Už nemusíme
definovať metódu *Pocitaj* pre rôzne objekty, ktoré sú potomkom objektu *pocitac* . Stačí, ak sme ju nadefinovali raz v predkovi, a potomkovia ju iba zdedia! A toto je v štruktúrovanom programovaní nemožné! Jej správnu funkciu pri vyvolaní z rôznych inštancií zabezpečuje práve virtualita metódy *Urci_Typ_procesora*. Neveríte?

Ak vám nie je niečo jasné, nezúfajte! Upravte zdrojový text tak, že zmažete slovo "virtual" pri metóde *Urci_Typ_procesora* , program preložte alebo krokujte! Čo sa stane? Na konci programu nesmieme zabudnúť na uvoľnenie alokovanej pamäte príkazom *Dispose* . Ak by sme tak neučinili, pamäť by ostala aj po skončení programu obsadená a až do resetu počítača by ju nemohol využiť iný program. A to by sme nechceli. Nabudúce si vysvetlíme ďašie zásady pri tvorbe objektových programov.

---

3. časť

Dnes sa zameriame na niektoré ďalšie postupy pri tvorbe OOP programov, pripomenieme si a bližšie rozoberieme už skôr spomínané problémy. (Opakovanie je matkou múdrosti!)

Prístup k dátovým položkám objektu
Je všeobecne uznávaným pravidlom, vyplývajúcim z filozofie OOP a hlavne z princípu zapuzdrenia, že prístup k dátovým položkám objektu sa realizuje pomocou špeciálnych metód, a nie priamo. Meniť a čítať hodnoty dátových položiek objektu smú iba metódy tohto objektu. Nerešpektovaním tohto pravidla síce môžeme v určitých prípadoch skrátiť zdrojový text programu, ale pripravíme sa tým o výhodu vyplývajúcu z toho, ako sa v OOP poníma súvislosť dát a kódu. Ak je vďaka zapuzdreniu

jednoznačne určené, s ktorými dátami môže ktorá metóda pracovať, znížI sa alebo priamo vylúči nebezpečie, že použijeme nevhodnú procedúru na nevhodné dáta. Táto kontrola bola predtým len a len na programátorovi.

Nešetrite na metódach
Pridaním metód do vášho programu zväčší zdrojový text objem. Inteligentný linker (spojovací program, ktorý je súčasťou prekladača Turbo Pascalu) však vo výslednom kóde vypustí tie metódy, ktoré v programe nie sú nikdy volané. Nemusíte preto váhať nad tým, či každý program použije alebo nepoužije metódu vášho objektu. Nepoužité metódy vás nestoja nič, pokiaľ ide o veľkosť .EXE súboru - ak nie sú použité, tak v ňom jednoducho nie sú! Preto môžeme napísať zdrojový text s väčším počtom metód, čím sa stáva univerzálnejším.

Písanie konštruktorov a deštruktorov
Ako sme si už povedali, konštruktory a deštruktory sú špeciálne metódy, ktorých použitie je potrebné v prípade, že objekt obsahuje virtuálne metódy (a ak hovoríme o deštruktore, tak v prípade, ak inštancia objektu obsahuje dynamicky vzniknuté položky). Pretože konštruktor treba volať pred prvým použitím niektorej z virtuálnych metód, je výhodné vyvolávať ho ako vôbec prvú metódu daného objektu. (Tak sa to napokon rieši automaticky v prípade dynamickej alokácie objektu pomocou rozšírenej syntaxie príkazu **New**.)

Logickým pokračovaním je sústrediť do konštruktora všetky akcie súvisiace s inicializáciou objektu, teda nastaviť počiatočné hodnoty všetkých dátových položiek objektu tak, ako je potrebné pre nasledujúcu správnu činnosť programu. Deštruktor by mal byť virtuálny. Vysvetlenie prečo, vyplynie z ďalšieho textu, tu sa obmedzíme na jednoduché konštatovanie.

Objektovo orientované unity a rozšíriteľnosť objektov
Ako vieme, ani jeden väčší program sa nezaobíde bez unitov - knižníc. Mechanizmus samostatných pripojiteľných unitov spolu s OOP dáva možnosť písať kvalitatívne novú obdobu klasických unitov. Týmto spôsobom je vytvorená aj jednotka Turbo Vision, ktorou sa budeme zaoberať onedlho.

Myslime na to, že píšeme unit nielen pre seba, ale ho možno budú chcieť využívať aj iní programátori. To predsa nie je nič nezvyčajné, ide o tvorivú činnosť v prospech ostatných. (Či to postavíte na komerčnej báze, alebo nie, to už je iná vec!) Ak píšeme taký objektovo orientovaný unit, uvedieme v sekcii INTERFACE definíciu všetkých objektov, ktoré chceme sprístupniť používateľovi unitu. Telá týchto objektov spolu s definíciami objektov, ktoré pred používateľom zostanú neviditeľné, uvedieme v sekcii IMPLEMENTATION.

Ak potom dodáme používateľovi takto vytvorenú jednotku v preloženom tvare .TPU, chránime tým svoj zdrojový text (a nápad), pokiaľ ho nechceme zverejniť. Na druhej strane však používateľa nepripravujeme o možnosť rozširovať alebo pozmeniť funkciu nami napísaného kódu. To umožňuje jednak mechanizmus dedičnosti, jednak vlastnosť polymorfizmu.

Používateľ totiž môže definovať svoj objekt, ktorý je potomkom niektorého objektu definovaného v našom unite, a to aj vtedy, keď nemá jeho zdrojový text. Pre takto definovaný objekt bude platiť všetko, čo sme dosiaľ povedali o mechanizme dedičnosti. Ak je určitá metóda definovaná v objekte skrytom niekde v .TPU súbore ako virtuálna a používateľ ju vo svojom potomkovi predefinuje, "skrytý" predok o tom dostane informáciu a každé prípadné volanie tejto metódy v iných svojich metódach vykoná korektne, teda zavolá tú správnu, novú verziu metódy. Z toho vyplýva, že ak je kód objektov v unite vhodne napísaný, používateľ môže bez zásahu do zdrojového textu meniť unitu funkciu týchto objektov a prispôsobovať ju svojim požiadavkám. Je to veľký krok smerom k čo najväčšej využiteľnosti hotových programových modulov a veľký pokrok v porovnaní s klasicky chápanými knižnicami funkcií.

Statické versus virtuálne metódy
Ak uvážime to, čo sme uviedli v predchádzajúcom odseku, je zrejmé, že treba starostlivo uvážiť, či urobíme tú-ktorú metódu virtuálnou, alebo nie.

Ak nadefinujeme niektorú metódu ako statickú, teda bez použitia kľúčového slova virtual, dosiahneme tým väčšiu rýchlosť jej vykonania. Táto metóda sa totiž vykoná priamo, bez predchádzajúcich pomocných akcií, lebo pri preklade sa do cieľového kódu na miesto volania takejto funkcie zapíše priamo jej adresa. Táto rýchlosť je však zaplatená tým, že sme používateľa pripravili o akúkoľvek možnosť prispôsobiť funkciu metódy svojím potrebám (o ktorých nemožno vylúčiť, že budú aspoň nepatrne odlišné od toho, čo sme predpokladali v čase písania metódy).

Ak robíme, naopak, niektorú z našich metód virtuálnou, bude situácia presne opačná. Používateľ má síce možnosť v niektorom potomkovi funkciu metódy zmeniť, aby pritom nenarušil

fungovanie ostatných metód, ale vyvolanie takejto metódy bude pomalšie. Dochádza k nemu v podstate v dvoch fázach: najpv sa pomocou odkazu siahne do tabuľky virtuálnych metód pre skutočnú adresu virtuálnej metódy, a až potom sa prevedie vlastný skok na kód tejto metódy.

Ako sa teda pri tej-ktorej konkrétnej metóde rozhodnúť? Jedna z odporúčaných taktík spočíva v tom, že väčšinu metód nadefinujeme virtuálne. Statické metódy zavádzame, len keď je potrebná časová optimalizácia programu, a to tam, kde sme si istí, že používateľ nepocíti potrebu modifikovať nami navrhnutú funkciu metódy.

Na tomto mieste je potrebné vrátiť sa ku skoršej poznámke o tom, že deštruktory by mali byť definované ako virtuálne. Je to preto, že spravidla nemôžeme vylúčiť, či používateľ nepridá v niektorom potomkovi nášho objektu ďalšiu dynamicky alokovanú položku. A pretože deštruktor je najprirodzenejšie miesto, kde tieto dynamické položky môžu byť uvoľnené, mali by sme používateľovi dať možnosť zmeniť funkciu deštruktora tak, aby uvoľňoval aj jeho nové položky. Preto by deštruktor mal byť virtuálny, aj keď nevyhnutné to, samozrejme, nie je.

Vzájomná priraditeľnosť objektov

Ak máme dve premenné typu objekt, pričom jedna je inštanciou predka, druhá je inštanciou potomka, môžeme vykonať nasledujúce priradenie:

```
predok:=predok
potomok:=potomok
predok:=potomok
```

Naopak, nesprávne je priradenie:

```
potomok:=predok
```

Ak sa vrátime k nášmu príkladu, sú správne tieto priradenia:

```
var pc1, pc2 : pocitac;
    at1, at2 : atecko ;

begin
   pc1:=pc2;
   at1:=at2;
   pc1:=at2;
end.
```

Vysvetlenie takéhoto poňatia kompatibility typov spočíva v mechanizme dedičnosti. Tento mechanizmus zabezpečuje, že každý potomok má všetky dátové položky svojho predka a môže mať niektoré navyše. Nimi sa definícia objektu potomka rozširuje. Je teda jasné, že pri priradení potomka predkovi môžeme bez problémov naplniť všetky položky predka.

Pri opačnom priradení nemôžeme vylúčiť, že v potomkovi zostanú niektoré položky nedefinované, preto priradenie potomok:=predok nie je dovolené.

Rovnaké pravidlá platia aj pre vzájomné priraďovanie hodnôt ukazovateľov na objekty. Ukazovateľ na určitý objekt môže vždy ukazovať aj na potomkovský objekt. Táto vlastnosť je veľmi výhodná, umožňuje nám napríklad vytvárať všeobecné fronty objektov. Ak je formálnym parametrom procedúry či funkcie objekt, môže byť skutočným parametrom aj ľubovoľný jeho potomok.

Kľúčové slová Private, Public a Protected

Počínajúc verziou 7.0 Turbo Pascalu a produktom Delphi, Pascal obsahuje nové kľúčové slová *Private, Public* (TP 7.0) a *Protected* (Delphi). V predchádzajúcich verziách totiž nebolo vôbec možné nejakým spôsobom obmedziť viditeľnosť vlastností definovaného objektu. Teraz môžeme pomocou týchto špeciálnych slov rozdeliť položky objektu do troch skupín:

verejné – *public :* bežne dostupné z ktoréhokoľvek miesta programu

chránené – *protected :* dostupné iba pre tzv. spriatelené objekty, teda objekty z určitej definovanej skupiny,

osobné – *private :* dostupné iba z objektu, v ktorom sú definované.

Možno si teraz poviete, na čo je to dobré. Samozrejme, pri programovaní v Turbo Vision by sme sa bez toho zaobišli. Ale v Delphi sú tieto slová "vždy a všade", preto je dobré, aby sme ich poznali. O tom, ako pracujú, si povieme neskôr.

Teraz máte pocit, že som vás napchával teóriou o ničom a všetkom bez praktického využitia. Je to normálne, aj šoférovať ste sa učili najprv teoreticky, až po zvládnutí pravidiel ste išli na praktickú jazdu. Ale aby sme si ukázali praktické využitie teoretických znalostí, napíšeme si nabudúce nejaký objektovo orientovaný programík.

---

4. časť

Dnes sa pozrieme na OOP z inej strany - zo strany grafiky. OOP sa skutočne najviac využíva v grafike a v práci s ňou.

Vytvorme si najzákladnejší grafický objekt Figure, ktorý bude mať vlastnosti (dátové položky): súradnice x a y typu integer, booleovskú premennú *Visible*, ktorá bude definovať, či má byť objekt viditeľný alebo nie, a tieto schopnosti (metódy): Init, ktorá inicializuje celý objekt, *GetX* a *GetY*, ktoré vracajú aktuálnu pozíciu objektu, *IsVisible* udáva, či je objekt viditeľný, a metódy *Draw* na vykresľovanie, *Show* na ukazovanie, *Hide* na schovávanie a *MoveTo* na presun objektu. Zápis takéhoto objektu **Figure** bude vyzerať takto:

```
Figure = object
  X,Y : integer;
  Visible : boolean;
  procedure Init(InitX, InitY);
  function GetX : integer;
  function GetY : integer;
  function IsVisible : boolean;
  procedure Draw;
  procedure Show;
  procedure Hide;
  procedure MoveTo(NewX, NewY: integer);
end;
```

Tento spôsob zápisu je nám už známy, a preto si vytvoríme prvého potomka objektu Figure - objekt Point. Jeho delarácia je:

```
Point = object(Figure)
  procedure Draw;
  procedure Show;
  procedure Hide;
  procedure MoveTo(NewX, NewY: integer);
end;
```

Už vieme, že nemusíme uvádzať všetky dátové položky a metódy, tie objekt Point jednoducho zdedí od objektu Figure. Uvádzame len tie metódy, ktoré budeme predefinovávať.
Je dobrým zvykom prezentovať hierarchiu objektových typov v podobe unitu. Objektové typy sú deklarované v interfaceovej časti unitu, definície metód sa uvádzajú v neverejnej implementačnej časti. Na listingu č.1 je výpis unitu MyPoints.

Podrobne ho preštudujte a snažte sa pochopiť čo najviac, na tomto unite budeme stavať svoje základy a príklady. Ak niektorému príkazu Turbo Pascalu nerozumiete, pozrite sa do manuálu alebo helpu. Ide spravidla o príkazy z jednotky Graph. Jednotku prepíšte v prostredí IDE a preložte pod menom MyPoints. Nezabudnite nastaviť cestu k jednotke Graph alebo ju prekopírujte z adresára BGI do vášho pracovného adresára, kde máte zdrojové texty! Pri prepisovaní sa snažte pochopiť, ako jednotka pracuje. Nič na tom, že metóda Figure.Draw je prázdna. Objekt Point ju predefinuje podľa svojho.

Na listingu č. 2 je výpis unitu MyMouser, ktorý zabezpečuje prácu s myškou počítača. Aj táto

jednotka je objektová. Objekt **Mouse** je takisto potomkom objektu Figure. Má veľmi podobný zápis ako objekt Point, len mu pribudli nové dátové položky *LeftButton*, *RightButton* a *CentreButton* typu boolean a niekoľko metód – Query, Left, Centre a Right. Implementačná časť je trochu zložitejšia ako pri jednotke MyPoints. Využívajú sa služby interruptu 33 BIOS-u počítača na prácu s myšou. Pre podrobnejšie informácie nazrite do príslušnej literatúry alebo to berte ako fakt. V interfaceovej časti sú deklarované aj dve premenné – logická **MousePresent** indikujúca prítomnosť myši a číselná **MouseButtons**, udávajúca počet tlačidiel myši. Veľmi dôležitou je metóda **Query** (otázka, Česi majú na to výstižnejší výraz "dotaz"), ktorá podá komplexné informácie o stave myšky. Na konci tohto unitu je jeho inicializačná časť. Tento unit tiež preložte pod názvom *MyMouser*.

Prečo to všetko? No aby sme si mohli demonštrovať možnosti oboch jednotiek (ale hlavne MyPoints) v našom prvom aplikačnom programe MYSKA! Jeho výpis je na listingu č. 3.

Že je akýsi krátky? Ja som vám hovoril, že hlavné časti programu vytvoreného v OOP bývajú len niekoľkoriadkové. Použijeme nami vytvorenú unitu MyMouser, ktorá zase volá unitu MyPoints. Premenná M je typu Mouse. Vymažeme obrazovku, zinicializujeme premennú (inštanciu) objektu M na súradnice 320 a 100 a volaním procedúry Show kurzor myši zobrazíme na obrazovke. Až do stlačenia ľubovoľného klávesu program volá metódu Query a na obrazovku vypisuje polohu kurzora myšky, stav stlačenia tlačidiel, počet tlačidiel a prítomnosť myšky. (Tu sa môžeme trochu pohrať - ak máme myš, kde je možné prepínať medzi dvoj- alebo trojtlačidlovým režimom, vyskúšajme to!) Nakoniec sa myška skryje a program sa ukončí.

A na záver ako "pikošku" uvádzame výpis programu MyDemo na listingu č.4. Napíšte, preložte a skúšajte zmeniť parametre! (Znova upozorňujem na nastavenie správnych ciest pri preklade.) Krokujte a laďte! A uvidíte, v čom je krása objektov!

LISTING Č. 1

```
unit MyPoints;

interface

uses Graph;

const FigColor = White;
       { Figure Color - farba objektu }

type
  Figure = object
    X, Y: Integer;
    Visible: Boolean;
   procedure Init(InitX, InitY: Integer);
    function GetX: Integer;
    function GetY: Integer;
    function IsVisible: Boolean;
    procedure Draw;
    procedure Show;
    procedure Hide;
  procedure MoveTo(NewX, NewY: Integer} ;
  end;

  Point = object (Figure)
    procedure Draw;
    procedure Show;
    procedure Hide;
    procedure MoveTo(NawX, NewY:
Integer) ;
  end;

implementation

{ ---------------- }
(  Implementacia metod Figure:
```

```
)
 {------------------}
procedure Figure.Init (InitX, InitY;
 Integer) ;
begin
  X := InitX;
  Y := InitY;
  Visible := False; end;

function Figure. GetX: Integer; begin
  GetX := X;. end;

function Figure.GetY: Integer; begin
  GetY := Y; end;

function Figure.IsVisible: Boolean; begin
  IsVisible := Visible; end;

procedure Figure. Draw;
begin
end;

procedure Figure . Show;
begin
  Visible := True;
  SetColor (FigColor) ;
  Draw;
end;

procedure Figure.Hide; begin
  Visible := False;
  SetColor(GetBkColor);
  Draw;
end;

procedure Figure. MoveTo(NewX, NewY:
 Integer);
begin
  Hide;
  X := NewX;   Y := NewY;
  Show;
end;

 {------------------}
 { Implementacia metod Point:
 }
 {------------------}

procedure Point.Draw;
begin
  PutPixel(X, Y, GetColor);
end;

procedure Point.Snow;
begin
  Visible := True;
  SetColor(FigColor);
  Draw;
end;

procedure Point.Hide;
```

```
begin
  Visible := False;
  SetColor(GetBkColor);
  Draw;
end;

procedure Point.MoveTo(NewX, NewY:
Integer);
begin
  Hide;
  Init(NewX, NewY);
  Show;
end;

END.
```

LISTING Č. 2

```
unit MyMouser;

INTERFACE

uses MyPoints, Dos;
type
  Mouse = object (Figure)
    LeftButton, CentreButton,
RightButton: Boolean;
    procedure Init(InitX, InitY:
Integer);
    procedure Show;
    procedure Hide;
    procedure MoveTo(NewX, NewY:
Integer);
    procedure Query;
    function Left: Boolean;
    function Centre: Boolean;

     function Right: Boolean;
end;

var MousePresent: Boolean;
           { Indikuje pritomnost mysi }
    MouseButtons: Byte;
         { Udava pocet tlacidiel mysi }

IMPLEMENTATION

var Regs: Registers;

procedure Mouse.Init(InitX, InitY:
Integer);
begin
  Query;
  Figure.Init(InitX, InitY);
  regs.ax:=4;
  regs.cx:=InitX;
  regs.dx:=InitY;
  intr($33,regs);
end;

procedure Mouse.Show;
```

```
begin
  Visible:=true;
  regs.ax:=4;
  regs.cx:=X;
  regs.dx:=Y;
  intr($33,regs);
  regs.ax:=1;
  intr($33,regs);
end;

procedure Mouse.Hide;
begin
  Visible:=false;
  regs.ax:=2;
  intr($33,regs);
end;

procedure Mouse.HoveTo(NewX, NewY:
Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

procedure Mouse.Query;
begin
  regs.ax:=3;
  intr($33,regs);
  X:=regs.cx;.
  Y:=regs.dx;
  LeftButton:=(regs.bx and 1)<>0;
  RightButton:=(regs.bx and 2)<>0;
  CentreButton:=(regs.bx and 4)<>0;
end;

function Mouse.Right: Boolean;
begin
  Right:=RightButton;
end;

function Mouse.Left: Boolean;
begin
  Left:=LeftButton;
end;

function Mouse.Centre: Boolean;
begin

  Centre:=CentreButton;
end;

BEGIN
{Inicializacna cast}
  regs.ax:=0;
  intr($33,regs);
  MousePresent:=regs.ax<>0;
  MouseButtons:=regs.bx;
  END.
```

LISTING Č.3

```
program Myska; (demonstruje cinnost
MyPoints a MyMouser}

uses MyMouser, Crt;

var M :Mouse;

BEGIN
  ClrScr;
  M.Init(320, 100);
  M.Show;
  while nôt keypressed do
   with M do
    begin
     query;
     write(#13, X:4, Y:4, Left:6,
Centre:6, Right:6,
           MouseButtons:3,
MousePresent:6 );
   end;
  M.Hide;
END.
```

LISTING Č. 4

```
program MyDemo;

uses Crt, Graph, MyPoints;

const N = 600;  { Skuste zmenit tieto
koeficienty !!! }
      K = 4;
      L = 2;

var B: Array[1..N] of Point;
    i: integer;
    dX, dY: integer;
    gd, gm: integer;

BEGIN
  gd := detect;
  InitGraph(gd,gm,´´);
  for i := l to N do
B[i].Init(320+round(i/L*cos(i/K)),240+ro
und(i/L*sin(i/K)));
  for i := l to N do B[i].Show;
  while not keypressed do
   begin
    dX := random(20) - 10;   {aj tu!}
    dY := random(20) - 10;
    for i := l to N do with B[i] do
MoveTo(GetX+dX, GetY+dY);
    delay(500);
   end;

  for i := 1 to N do B[i].Hide;
  CloseGraph;
END.
```

5.časť

Minule sme si vytvorili knižnicu MyPoints, kde sme definovali objekt Figure a jeho priameho potomka Point. Vieme, že Point zdedil všetky vlastnosti objektu Figure a tie, ktoré mu nevyhovujú, sme predefinovali podľa svojich potrieb. Hierarchiu objektových typov obvykle budujeme tak, že k skôr definovaným typom (Figure, Point) pridávame ďalšie ako ich potomkov. Predstavme si teda, že chceme pridať bezprostredného potomka objektu Figure - objektový typ Circle:

```
type
   Circle = object(Figure)
     Radius: integer;
     procedure Init(InitX,InitY,InitRadius : integer);
     procedure Draw;
     procedure Show;
     procedure Hide;
     proedure MoveTo(NewX, NewY);
   end;
```

Aj tento objektový typ zdedí všetky datové položky a metódy od svojho predka. Je však bohatší o ďalšiu vlastnosť - polomer (radius), takže mu niektoré zdedené metódy nevyhovujú. Môžeme to riešiť klasicky, teda predefinovaním metódy, napr.:

```
procedure Circle.Draw;
begin
   Graph.Circle(X,Y,radius);
end;
```

Na rozdiel od typu Point musíme predefinovať aj metódu Init:

```
procedure Circle.Init(InitX, InitY, InitRadius);
begin
   Figure.Init(InitX, InitY);
   Radius:= InitRadius;
end;
```

Všimnime si, že predefinovaním metódy jej pôvodný význam nestrácame, naopak, môžeme ho s výhodou použiť. Musíme ale rozlišovať príslušnosť identifikátorov jednotke (Graph.Circle) a objektovému typu (Figure.Init).

Metódy, ktoré sme dosiaľ používali, sa nazývajú statické. Ich väzby sme si už povedali v predchádzajúcich častiach. Ale pri dedení statických metód môže dôjsť k problémom. Pozrime sa na unit MyPoints ešte raz. Všimnime si, že metódy Show, Hide a MoveTo objektový typ Point nezdedí od typu Figure napriek tomu, že sú formálne úplne rovnaké! Líšia sa iba v tom, že metódy typu Figure volajú Figure.Draw a metódy typu Point volajú Point.Draw. Práve kôli tomu ich musíme vždy predefinovať.

Predstavme si, že v definícii objektového typu Point tieto metódy neuvedieme, a teda sa zdedia v pôvodnom tvare. Ak teraz budeme volať pre nejakú inštanciu (napr.Bod) typu Point napr. metódu Show, bude toto volanie realizované ako Figure.Show, ktoré volá Figure.Draw. Tá nám rozhodne bod nenakreslí. Overte si zapoznámkovaním metód v jednotke MyPoints a krokujte! Je to skutočne tak. (Ak teraz nechápete, čo som vlastne povedal, prečítajte si to niekoľkokrát a snažte sa tomu porozumieť a overte si to na príkladoch, aj mne to trvalo pekne dlho! Jednoduchšie to už v časopise vysvetliť neviem.).

Nedalo by sa to urobiť jednoduchšie a efektnejšie? Ale áno, použijeme už dávnejšie spomínané virtuálne metódy. Aj ich princíp sme si už hovorili.

Ak by sme chceli obohatiť jednotku MyPoints o virtuálne metódy, vyzerala by časť INTERFACE takto:

```
type
  Figure = object
     x,y : integer;
```

```
    visible: boolean;
    constructor Init(InitX,InitY : integer);
    function GetX: integer;
    function GetY: integer;
    function IsVisible : boolean;
    procedure Draw; virtual;
    procedure Show;
    procedure Hide;
    procedure MoveTo(NewX, NewY: integer);
  end;

  Point = object(Figure);
    procedure Draw; virtual;

    (všimnite si, že tu už nič nie je!}

  end;
```

Vidíte, ako sa zjednoduší zápis objektových potomkov, ak sa používajú virtuálne metódy. Všetky časti objektu sme si už spomínali, ale teraz ich začneme používať v praxi.

Už nie je potrebné deklarovať totožné metódy Show, Hide a MoveTo u objektu Point, aj keď volajú metódu Draw. Na základe virtuality program vždy použije tú správnu metódu Draw, ktorá prislúcha patričnému objektu.

Aby naša knižnica MyPoints nebola taká chudobná, deklarujeme ďalších potomkov - kružnicu (Circle) ako potomka Figure, a oblúk (Arc) a elipsu (Ellipse) ako potomkov Circle. (Čisto teoreticky sú to už vnuci objektového typu Figure). Postavíme ju na základe virtuálnej metódy Draw. Jej výpis je na listingu č.1.

Aby sme ju otestovali, použijeme demonštračný program MYFIGDEM.PAS na listingu č.2. A teraz si predstavte situáciu, že vám niekto dal knižnicu MyFigure len v tvare .TPU, teda bez zdrojového textu, a vy chcete pridať nového potomka - úsečku Line. Ako na to? V klasickom použití Turbo Pascalu je to nemožné, ba dokonca ani pri použití objektovo orientovaného programovania so statickými metódami by to nešlo. Ale pri použití virtuálnych metód to nie je nič jednoduchšie. Ukážeme si to!

Vytvoríme si jednotku NewFigs, kde použijeme jednotku MyFigure. V interfejsovej časti deklarujeme nový objekt Line ako potomka Figure (listing č.3). Na jeho deklaráciu stačí konštruktor Init, datové položky dX a dY typu integer, virtuálna metóda Draw a dve funkcie - GetdX a GetdY. To ostatné, čo je v zložených zátvorkách tento objekt zdedí od svojho predka, ale ja som to schválne napísal pre vašu ilustráciu, čoho nás objekty ušetrujú. V implementačnej časti unitu deklarujeme jednotlivé metódy. Vzhľadom na virtualitu metódy Draw ani tu nie je nutné opakovať identické a zdediteľné metódy Show, Hide a MoveTo.

Použitie tejto knižnice NewFigs demonštruje program NewFigsDemo (NFIGSDEM.PAS - listing č.4). Je skoro rovnaký ako predchádzajúci príklad MYFIGDEM.PAS z listingu č.2, len má pridané tri riadky, týkajúce sa úsečky Line.

Ak správne opíšete tento program, pohráte sa s jeho parametrami a budete sledovať jeho činnosť na obrazovke, isto vás napadne, že by sa to akosi dalo urobiť tak, aby sa tie objekty pohybovali plynule. Ale, ale, moji zlatí, to už sú začiatky animácie! A to je ďalší bod pre objekty. O tom nabudúce.

LISTING č. 1

```
unit MyFigure;

   { Upraven verzia jednotky MyPoints}

INTERFACE

uses Graph, Crt;

const FigColor = White;
       { Figure Color - farba objektu }

type
  Figure = object
```

```
    X,Y: Integer;
    Visible: Boolean;
    constructor Init(InitX, InitY:
Integer);
    function GetX: Integer;
    function GetY: Integer;
    function IsVisible: Boolean;
    procedure Draw; virtual;
    procedure Show;
    procedure Hide;
  procedure MoveTo(NewX, NewY:
Integer);
  end;

  Point = object (Figure)
    procedure Draw; virtual;
  end;

  Circle = object (Figure)
    Radius: Integer;
    constructor Init(InitX, InitY,
InitRadius: Integer);
    procedure Draw; virtual;
  end;

  Arc = object (Circle)
   StartAngle, EndAngle: Integer;
   constructor Init(InitX, InitY,
InitRadius: Integer;
                    InitStartAngle,
InitEndAngle: Integer);
    procedure Draw; virtual;
  end;

  Ellipse = object (Circle)
    Radius2: Integer;
    constructor Init(InitX, InitY,
InitRadius, InitRadius2: Integer);
    procedure Draw; virtual;
  end;

IMPLEMENTATION

{ ---------------------- }
{ Implementacia metod Figure:
}
{ ---------------------- }

constructor Figure.InitflnitX, InitY:
Integer);
begin
  X := InitX;
  Y := InitY;
  Visible := False;
end;

function Figure.GetX: Integer;
begin
  GetX := X;
end;

function Figure.GetY: Integer;
begin
  GetY := Y;
end;

function Figure.IsVisible:  Boolean;

begin
   IsVisible := Visible;
end;

procedure Figure. Draw;
begin
```

```
end;

procedure Figure.Show;
begin
  Visible := True;
  SetColor(FigColor);
  Draw;
end;

procedure Figure.Hide;
begin
  Visible := False;
  SetColor(GetBkColor);
  Draw;
end;

procedure Figure.MoveTo(NewX, NewY:
Integer);
begin
  Hide;
  X := NewX;
  T := HewY;
  Show;
end;

{------------------------------}
{ Implementacia metod Point:
}

{------------------------------}

procedure Point.Draw; begin
  PutPixel(X, Y, GetColor);
end;

{------------------------------}
{ Implementacia metod Circle:
}

{------------------------------}

constructor Circle.Init(InitX, InitY,
InitRadius: Integer);
begin
  Figure.Init(InitX, InitY);
  Radius := InitRadius;
end;

procedure Circle.Draw;
begin
  Graph.Circle(X, Y, Radius);
end;

{------------------------------}
{ Implementacia metod Arc:
}

{------------------------------}


constructor Arc.Init(InitX, InitY,
InitRadius: Integer;
                     InitStartAngle,
InitEndAngle: Integer);
begin
  Circle.Init(InitX, InitY, InitRadius);
  StartAngle := InitStartAngle;
  EndAngle  := InitEndAngle;
end;

procedure Arc.Draw;
begin
```

```
  Graph.Arc(X, Y, StartAngle, EndAngle,
Radius);
end;

{------------------------------}
{ Implementacia metod Ellipse:

}

{------------------------------}


constructor Ellipse.Init(InitX, InitY,
InitRadius, InitRadius2: Integer);
begin
  Circle.Init(InitX, InitY, InitRadius);
  Radius2 := InitRadius2;
end;

procedure Ellipse.Draw;
begin
  Graph.Ellipse(X, Y, 0, 360, Radius,
Radius2) ;
end;

END.
```

LISTING č.2

```
program MyFigureDemo;
uses Crt, Graph, MyFigure;

var Kruznica: Circle;
    Obluk: Arc;
    Elipsa: Ellipse;
    gd, gm: integer;
    ErrCode  : Integer;
BEGIN
  gd := detect;
  InitGraph(gd,gm,'');
  Kruznica.Init(50,50,30);
{mente tieto parametre !!!)
  Obluk.Init(150,50,50,220,320);
{mente tieto parametre !!!}
  Elipsa,Init(250,50,20,50);
{mente tieto parametre !!!}
  Kruznica.Show;
  Obluk. Show;
  Elipsa.Show;
  while not keypressed do
   begin
    Kruznica.MoveTo(random(GetMaxX),ran-
dom(GetMaxY));
    Obluk.MoveTo(random(GetMaxX),ran-
dom(GetMaxY));
    Elipsa.MoveTo(random(GetMaxX),ran-
dom(GetMaxY));
    delay(15000);
                {ak chcete, tak aj tu}
   end;
  Kruznica.Hide;
  Obluk. Hide;
  Elipsa.Hide;
  CloseGraph;
END.
```

LISTING č. 3

```
unit NewFigs;


INTERFACE

uses Crt, Graph, MyFigure;

type
  Line = object (Figure)
  { X,Y: Integer; }
    dX, dY : Integer;
  { Visible: Bbolean; }
    constructor Init(InitX, InitY,
InitdX, InitdY: Integer);
    procedure Draw; virtual;
  { procedure Show; }
  { procedure Hide; }
  { function GetX: Integer; }
  { function GetY: Integer; }
    function GetdX: Integer;

   function GetdY: Integer;
 { function IsVisible: Boolean; }
 { procedure MoveTo(NewX, NewY:
Integer); }
  end;

 IMPLEMENTATION

 { ------------------------- }
 { Implementacia metod Line: }

 { ------------------------- }

 constructor Line.Init(InitX, InitY,
 InitdX, InitdY: Integer);
 begin
   Figure.Init(InitX, InitY);
   dX := InitdX;
   dY := InitdY;
end;

procedure Line.Draw;
begin
  Graph.Idne ( X, Y, X + dX, Y + dY );
end;

function Line.GetdX: Integer;
begin
  GetdX:=dX;
end;

function Line.GetdY: Integer;
begin
  GetdY:=dY;
end;

END.
```

LISTING č. 4

```
program NewFigsDemo;
uses Crt, Graph, NewFigs, MyFigure;

var Kruznica: Circle;
    Obluk: Arc;
    Elipsa: Ellipse;
    Usecka: Line;
    gd, gm: integer;

BEGIN
```

```
  gd := detect;
  InitGraph(gd,gm,'');
  Kruznica.Init(50,50,30);
  Obluk.Init(150,50,50,220,320);
  Elipsa.Init(250,50,20,50);
  Usecka.Init(350,50,50,0) ;
  Kruznica.Show;
  Obluk.Show;
  Elipsa.Show;
  Usecka.Show;
  while not keypressed do
   begin
    Kruznica.MoveTo(random(GetMaxX),ran-
dom(GetMaxY));
    Obluk.MoveTo(random(GetMaxX),ran-
dom(GetMaxY)};
    Elipsa.MoveTo(random(GetMaxX),ran-
dom (GetMaxY) );
    Usecka.MoveTo(random(GetMaxX),ran-
dom (GetMaxY) );
    delay(15000);
   end;
  Kruznica.Hide;
  Obluk.Hide;
  Elipsa.Hide;
  Usecka.Hide;
  CloseGraph;
END.
```

---

6. časť

Ak si dobre pozrieme naše posledné zdrojové texty, zistíme, že metóda Draw má rôzne implementácie v závislosti od objektu, z ktorého je volaná. Môžeme povedať, že vďaka virtualite je akási "viactvará". Tomuto javu hovoríme polymorfizmus. A ako vieme, je to jeden zo základných princípov OOP, s ktorým sa budeme odteraz často stretávať.

Objekty, či lepšie povedané ich inštancie, o ktorých sme si doteraz hovorili, boli statické:

```
var
    Bod : Point;
    Kruznica : Circle;
```

Ak pracujeme s objektmi, ktorých počet sa pri behu programu mení, a použijeme na ich reprezentáciu statickú štruktúru, napr. pole, musíme už pri písaní programu odhadovať maximálny počet týchto objektov a podľa toho pamäť alokovať. Tento spôsob sa vo väčšine príkladov stáva nepraktickým, ba dokonca pri zlom odhade počtu objektov môže program kolabovať. Často je výhodnejšie použiť dynamicky alokované objekty, ktorých pamäť uvoľníme, lenčo ich už nebudeme potrebovať.

Takéto objekty sú prístupné prostredníctvom ukazovateľov, tak ako to poznáme pri premenných ostatných typov. Navyše je Turbo Pascal od verzie 5.5 vybavený rozšírením, ktoré umožňuje jednoduchšiu a efektívnejšiu alokáciu a uvoľňovanie dynamických objektov v pamäti.
Ak máme

```
type
    CirclePtr = ^Circle;

var
   uKruznica : CirclePtr;
```

potom na základe príkazu

```
New(uKruznica);
```

je pre dynamický objekt vytvorený priestor v pamäti (v tzv. heape) a ukazovateľu je priradená adresa

tohto priestoru. Ak obsahuje dynamický objekt virtuálne metódy, musí byť inicializovaný volaním konštruktora ešte pred volaním jeho metód:

```
uKruznica^.Init(600,100,30);
```

Ako som spomínal, TP od verzie 5.5 rozširuje syntax procedúry New tak, že je možné vykonať alokáciu priestoru a inicializáciu dynamického objektu jediným príkazom:

```
New (uKruznica, Init(600,100,30));
```

Kompilátor určí správnu metódu Init podľa prvého parametra. Ďalšia podoba funkcie New je:

```
uKruznica = New(CirclePtr);
```

kde v zátvorke je uvedené meno typu ukazovateľa. Funkčná podoba New môže mať aj druhý parameter konštruktor objektu:

```
uKruznica = New (CirclePtr, Init(600,100,30));
```

Rozšírená syntax prispieva k lepšej čitateľnosti programu a má za následok kratší a efektívnejší kód.

Rozšírená syntax sa týka aj procedúry *Dispose*, ktorá dovoľuje prostredníctvom druhého parametra špecifikovať množstvo pamäte, ktoré má byť uvoľnené. Dosiaľ sme zvyknutí používať procedúru *Dispose* takto:

```
type
   StringPtr = ^String;
var
   uRetazec : StringPtr;

   begin
     New(uRetazec);
     uRetazec^:='****************';
     Dispose(uRetazec);
   end.
```

V tomto prípade odvodí procedúra Dispose veľkosť priestoru, ktorý má uvoľniť, z veľkosti hodnoty typu, na ktorý *uRetazec* ukazuje. Tá je pevne daná deklaráciou a v priebehu výpočtu sa nemení. Polymorfné objekty majú však neobyčajne premenlivú veľkosť. Z identifikátora ukazovateľa nemožno odvodiť veľkosť inštancie, na ktorú sa momentálne vzťahuje. Preto je potrebné mať nejaký mechanizmus, ktorý informáciu o veľkosti priestoru pamäte, ktorý má byť uvoľnený, procedúre *Dispose* poskytne. Týmto mechanizmom je mechamizmus deštruktora.

Vieme, že deštruktorom sa nazýva špeciálna metóda, ktorá slúži na uvoľnenie pamäte dynamicky alokovaných objektov. Kľúčové slovo *Destructor* tejto metódy spôsobí, že pri jej vyvolaní generuje kompilátor dodatkový kód, ktorý vyberie z prvého slova tabuľky VMT zodpovedajúceho objektu informáciu o veľkosti objektu. Zistenú veľkosť inštancie odovzdá deštruktor procedúre *Dispose* takto:

```
Dispose(uKruznica, Done);
```

Samotné volanie dešruktora všeobecne nespôsobí uvoľnenie pamäte. Záleží na tom, ako je deštruktor zostavený.

Metóda deštruktora môže byť volaná aj pre statické inštancie bez toho, aby došlo k nejakej chybe alebo kolízii. Napríklad

```
destructor Circle.Done;
begin
    Hide;
    end;
    ...
```

```
Kruznica.Done;
```

V tomto prípade sa volanie deštruktora prejaví zmiznutím objektu z obrazovky. Žiadna pamäť nie je uvoľňovaná, netreba informáciu o jej veľkosti, deštruktor neurobí v tomto smere nič. Ale pretože zvyčajne je ťažké predpovedať, či nebude niekedy v budúcnosti potrebné, aby boli pôvodne statické objekty používané ako dynamické, je vhodné vybaviť každý objektový typ vlastným deštruktorom.

Podobne ako iné metódy aj deštruktory môžu byť dedené. Deštruktory môžu byť statické aj virtuálne. Pretože však rôzne objektové typy vyžadujú rôzne deštruktory, odporúča sa, aby boli deštruktory vždy viruálne.

Deštruktor môže byť aj prázdny:

```
destructor Figufe.Done;
begin
end;
```

Z praktických dôvodov mávajú niektoré objekty aj niekoľko deštruktorov.
Možno sa pýtate, prečo sa vôbec zaťažujeme takými vecami. Bohužiaľ, bez nich by sme nemohli pristúpiť ku skutočnému programovaniu v grafickom móde. A pripomínam, že Windows sú celé postavené na grafike.
Vráťme sa však k polymorfizmu.
Polymorfným objektom rozumieme objekt, ktorý môže v priebehu programu nadobúdať hodnoty svojich potomkov. Turbo Pascal dovoľuje s takýmito objektmi pracovať. Túto možnosť oceníme najmä v situácii, keď chceme vytvoriť objekt zložený z nejakých iných objektov. Pri práci na riešení našej úlohy jednoduchej animácie si takýto objekt, obrázok zložený z iných obrázkov, nazveme PieChart a stretneme sa s ním neskôr.
A aby ste nepovedali, že dnes nemáte nijaký zdrojačik, na listingu č. 1 je program, ktorý demonštruje polymorfné objekty. Využíva naše knižnice, takže ich musíte mať prístupné. Ako?
Ááále, to už predsa viete!

```
LISTING l
program PolymorphicObjectDemo;

uses Crt, Graph, MyFigure;

var Kruznica: Circle;
    Obluk: Arc;
    Elipsa: Ellipse;
    P: ^Figure;
    r, FX, FY: integer;
    gd, gm: integer;


begin
  gd := detect;
  InitGraph(gd,gm,'');
  Kruznica.Init(0,0,30);
  Obluk.Init(0,0,40,120,420);
  Elipsa.Init(0,0,20,50);
  P := @Kruznica;
  FX := 320;
  FY := 240;
  while not keypressed do
begin
    r := random(lO);
    if r=0 then P. = @Kruznica;
    if r=l then P = @0bluk;
    if r=2 'then P = @Elipsa;
    FX := FX + random(9)-4;
    FY := FY + random(9)-4;
    P^.MoveTo(FX, FY);
    delay(5000);
    P'.Hide;
   end;
  CloseGraph;
end.
```

7. časť

Teraz, keď už je po prázdninách, môžeme sa zasa naplno venovať našim zlatým objektom. A aby sme pokročili k hlavnému cieľu – ku grafickým objektom a k manipulácii s nimi - vytvoríme si naozajstnú funkčnú a kompletnú knižnicu objektových typov a práce s nimi. Pomenujeme ju MyFigs a jej zdrojový text MYFIGS.PAS.

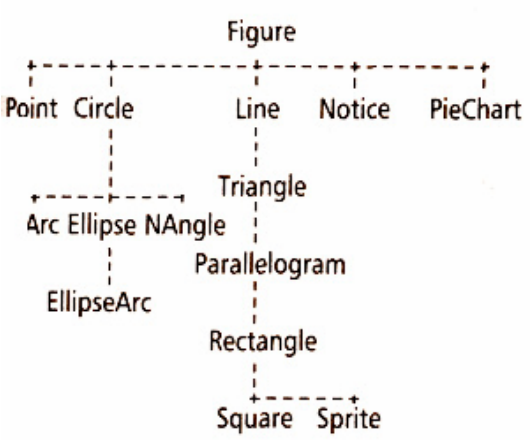
Jej interfejsová časť je na listingu č.1. Obsahuje ďalšie naše knižnice MYPOINTS a MYMOUSER. MyPoints sme si vytvorili, takže ju máme. MyMouser som doplnil o tri metódy pre kvalitnejšiu prácu s myšou, a tak jej nová verzia spolu s kompletným výpisom zdrojového textu MyFigs vrátane komentárov sa nachádza v redakcii. Nie je možné ich z priestorových dôvodov uverejniť.

Objektový typ Figure v nej bude značne rozšírený. Najdôležitejšie zmeny sú:

a) každému objektu je pridaná jeho farba,
b) pre každý objekt je zavedená expanzia a rotácia,
c) je pridaná metóda Drag, ktorá umožňuje manipulovať s objektmi pomocou klávesnice a myši.

Každú novú vlastnosť a schopnosť obrázku (farba, expanzia, rotácia, posuv) zavedieme už pri objektovom type Figure, aj keď tu práve nemajú zmysel. Je to však nevyhnutné z dôvodu dedenia, ako je nám známe z minulých častí.

Všeobecne sa treba snažiť, aby metód, ktoré treba predefinúvať, bolo čo najmenej a aby sa rovnaký kód neopakoval niekoľkokrát.



Obr. 1 Štruktúra jednotky MyFigs

Tak napríklad rotáciu doprava implementujeme pomocou rotácie doľava, ale o opačný uhol a zmenšenie pomocou zväčšenia s inverzným koeficientom mierky.

Tak sa už ďalej v hierarchii nemusíme starať o rotáciu doprava a o zmenšovanie, tieto metódy sa zdedia bez zmien.

Takisto nemusíme nikde predefinúvať metódu **Drag** (nemýliť s **Draw!**), lebo je v nej zabudované ovládanie všetkých schopností obrázka - pohyb, rotácia a expanzia.

Hierarchická štruktúra knižnice MyFigs je na obrázku č. 1. Pozrime sa teraz na jednotlivé časti podrobnejšie.

Circle a potomkovia

Prarodičom všetkého je objektový typ Figure, ktorého veľmi dobre poznáme. Aj Point je nám známy. Vetvu guľatín začínajúcu objektovým typom *Circle* rozširujú typy *Arc* (oblúk), *Nangle* (n-uholník), *Ellipse* (elipsa) a *EllipseArc* (elips. oblúk). Všimnime si rotácie pri *Circle*. Je to jediný typ, ktorý zároveň rotuje aj nerotuje (v skutočnosti nerotuje, ale nepoznať to). Jeho potomkovia Arc a NAngle rotujú, avšak Ellipse a EllipseArc nie. Spolu s Notice sú to jedinné nerotujúce objekty v celej hierarchii MyFigs. To preto, lebo pascalovská knižnica Graph tieto rotácie nepodporuje. To, že elipsa a text nerotujú, však nie je vždy na škodu,(uvidíme na príklade "gúľania očami" v programe PANAK).

Line a potomkovia

Máme niekoľko možností, ako túto vetvu vytvoriť. Už úsečku možno implementovať dvoma spôsobmi: buď určíme dva krajné body alebo stačí jeden bod a vektor, ktorý je rozdielom medzi týmito bodmi. Použijeme tento druhý spôsob a celá vetva je reprezentovaná vektorovo, lebo je to vhodné nielen na

rotáciu a expanziu, ale aj na posun objektov. Na rozdiel od prvého spôsobu nemusíme totiž predefinúvať nikde MoveTo. Logická nadväznosť jednotlivých objektov je tu zrejmá.

Sprite (zložený obrázok)
Osobitnú pozornosť zasluhuje objektový typ Sprite, ktorý sa trochu líši od ostatných objektov. Jeho prítomnosť je daná tým, že nie všetko možno vyjadriť pomocou elemetárnych objektov. Na tvorbu spritov slúži zvláštny editor spritov a možno ho získať ako shareware alebo freeware rôzne po BBS-kách alebo cédečkách.

Notice (poznámka - text)
Tento objekt je potrebný, ak chceme akékoľvek nápisy reprezentovať na obrazovke ako objekty. Aby sa tento objekt dal skutočne univerzálne použiť, sú v ňom ako položky definované všetky vlastnosti textu, ktoré jednotka Graph podporuje. V rôznych aplikáciách potrebujeme rôzne fonty aj rôzny stred textu. Rotovať text zatiaľ nemožno, ani v TP 7.0 nie, avšak možno ho mať vertikálne aj horizontálne. Expanzia textu je našťastie možná, avšak len celočíselne, takže sa text nezväčšuje plynule.

Absenciu rotácie a plynulej expanzie môžeme obísť nadefinovaním textu ako jednotlivé inštancie PieChartu po jednotlivých písmenách, ale to vám isto hneď prišlo na um!

PieChart
je prostriedok, ako skladať zložitejší objekt z jednoduchších, alebo doslovne, ako poskladať koláč (-ový nákres) z jednotlivých jeho častí. *PieChart* je reprezentovaný pomocou relatívnych súradníc jednotlivých komponentov oproti globálnemu stredu, ktorý je rozhodujúci na rotáciu a expanziu. Lepšie sa to ozrejmí na príklade PANAK. Rotácia a expanzia celku je realizovaná tak, že sa aplikuje tak na relatívne súradnice, ako aj na jednotlivé objekty, z ktorých je koláčový nákres (PieChart) zložený. Je dobré uvedomiť si, že PieChart možno skladať tak zo statických, ako aj dynamických častí alebo kombinovane. Väčšinou je dobré urobiť to dynamicky na heapu. Syntax je trochu zložitejšia, ale dá sa na to zvyknúť. Ak chceme napríklad vytvoriť panáka, urobíme to podľa listingu č. 2.

PieChart v panákovi vytvoríme tak, že sa najprv inicializuje ako zoznam obsahujúci nula komponentov a jednotlivé časti sa pridávajú postupne volaním metódy Add.

Za povšimnutie stojí rotácia panáka už pri jeho vytváraní. Je totiž výhodnejšie pridávať vlasy na rovnaké miesto pri priebežnom otáčaní hlavy, než počítať súradnice postranných vlasov pomocou trigonometrických funkcií (fuj...). Pri panákovi pracujeme s PieChartom ako celkom.
Záverom dodajme, že ak chceme mať na obrazovke niekoľko rovnakých PieChartov, napr. panákov, tak je lepšie vytvoriť nový objektový typ, ktorý zdedíme od PieChartu a do jeho konštruktora dáme všetky príkazy Add. Potom môžeme jednoducho nadeklarovať niekoľko objektov vytvoreného objektového typu.

Takže prepíšte alebo stiahnite z redakcie, preložte a skúšajte. Je to krása!
Pozn: Pri preklade unitu MyFigs treba definovať, či chceme preklad pre celé alebo reálne hodnoty. To nadefinujeme v IDE Turbo Pascalu v položke Option - Compiler - Conditional defines.

---

8. časť (chýba listing)
Ak ste si stiahli z redakčného webu potrebné súbory a preložili ste si panáčika, iste vás napadlo, že pokiaľ ide o animáciu, ešte to nie je to pravé orechové. To preto, že panák sa síce zväčšuje a zmenšuje, posúva všetkými smermi alebo rotuje, ale nechodí! Áno, prvok chodenia, teda kroku, dáva animácii ten pravý nádych. Preto si dnes ukážeme, ako naše objekty rozhýbeme a rozchodíme.
Podstatu plynulého pohybu možno rozdeliť na dva problémy:
- valivý pohyb na kolesách,
- presun dvoch alebo viacerých končatín v jednom cykle – krok.

Valivý pohyb na kolesách by sme aj teraz dokázali zrealizovať. Typickým príkladom je autíčko. Najprv nadeklarujeme koleso ako potomka PieChartu, podobne ako u panáčika. Toto koleso sa bude na základe pohybu myši alebo povelu z klávesnice otáčať požadovaným smerom. Už z predchádzajúcej časti vieme, že kruh je jediný objekt, ktorého otáčanie nevidíme. Aby otáčanie kolesa bolo zjavné, vyplníme koleso čiarami, ktoré budú predstavovať osi - špice, tak ako napr. na bicykli alebo na starom veteráne. Potom deklarujeme autíčko, ktoré je tiež potomkom PieChartu. Samozrejme, musí obsahovať aj už deklarované kolesá. Karosériu auta vytvoríme z úsečiek a oblúka. Vlastné telo programu obsahuje voľby veľkosti kolies a počtu osí kolesa a metódy Init, Show, Drag, a Done. Kompletný výpis programu AUTICKO.PAS je na listingu č. 1. Ostatné prvky pohybu, zmenšenie

či zväčšenie autíčko zdedilo od svojich predkov, takže toto je vaše jediné auto, ktoré si môžete zväčšovať podľa momentálnych potrieb.

Problém kroku je obdobný ako v bežnom svete. Najprv musí mať náš objekt nejaké nožičky. Tie mu vykreslíme pomocou grafických procedúr, ako sme na to zvyknutí. Zložitejšie to je už s grafickým vyjadrením kroku. Tu musíme nadefinovať postupné prekresľovanie polohy nôh so striedaním pravej a ľavej nohy. Nesmieme pritom zabudnúť, že pohyb sa skladá z odklonu a rotácie nožičky o určitý uhol, a to všetko v jednom jedinom kroku. Aby to bolo možné, vytvoríme si jednotku MyFeet.

Jednotka MyFeet je dodatkom k našej MyFigs, umožňujúcej objekty rozchodiť. Implementuje jednak akúsi šablónu pre "dvojnožcov" v podobe objektového typu BiPed, a jednak šablónu pre "n-nožce" v podobe objektového typu NPed. Ako tieto objektové typy vyzerajú, vidíte na listingu č. 2.

Typ BiPed obsahuje ukazovatele na ľavú a pravú nohu, maximálny uhol odklonu nohy od zvislej polohy, uhol rotácie nohy v jednom kroku cyklu chôdze a ukazovateľ na procedúru, ktorá sa vykonáva pri každom kroku cyklu chôdze (napr. pípnutie).
Pri type NPed sa dvojica nôh pridáva jednoducho ako jeho komponenty typu BiPed. Pri chôdzi "n-nožca" sa preberie zoznam komponent a pri tých, ktoré sú typu BiPed, sa vykoná patričná rotácia končatín.

Najvhodnejším predstaviteľom typu BiPed je kuriatko. Má dve nožičky, ktorými prepletá a vie sa pohybovať dopredu aj dozadu. A má aj zobáčik, ktorý môže otvárať, prípadne vydávať pípavý zvuk. Na listingu č. 3 je program KURCA.PAS, ktorý demonštruje práve spomínané vlastnosti (pohyb a zvuk) kuriatka.

Typickým predstaviteľom viacnožca je stonožka. Ako sme už naznačili, využíva obdobné metódy na vykreslenie pohybu viacerých končatín naraz v jednom smere. Na listingu č. 4 je program STONOZKA.PAS, ktorý demonštruje kráčanie stonožky. Keďže stonožka nepípa, táto naša vie "grúliť" očami.

Jadro programu má zase voľby veľkosti stonožky a počtu článkov tela. Ak určíme extrém, kde stonožka má iba jeden článok, veľmi sa objektovo (nie fyzicky!!!) podobá na kuriatko.

Kroky obidvoch zvieratiek sa ovládajú z klávesnice, a to súčasným stlačením klávesov pravý Shift + Enter (krok dopredu) a ľavý Shift + Enter (krok dozadu).

Ak sa vám zdajú programy pomerne zložité a teórie je málo, predpokladám, že už máte určité skúsenosti. A ak nie, tak skúšajte meniť rôzne parametre a laďte a krokujte, skúšajte a testujte. Lebo len takto sa dá naučiť programovať.

```
LISTING Č. 1
program Auticko;

{ Auto, ktore jazdi iba doprava a ktoremu sa automaticky otacaju
kolesa }
{ podla ubehnutej vzdialenosti }

uses MyFigs, Graph, Crt;

type
  KolesoPtr = ^Koleso;

  Koleso = object(PieChart)
    Radius: Number;
    constructor Init(InitX, InitY, InitRadius: Number;
                     InitN: Byte);
end;

AutoPtr = ^Auto;

Auto = object(PieChart)
  LKoleso, PKoleso: KolesoPtr;
  LAlpha, PAlpha: Real;
  dAngle: Integer;
  SX, SY: Number;   { Pozicia auta pri poslednom otacani kolies }

  constructor Init(InitX, InitY, D: Number;
                   InitLKoleso, InitPKoleso: KolesoPtr);
```

```
  procedure Rotate; virtual;
  procedure Draw; virtual;
end;

constructor Koleso.Init(InitX, InitY, InitRadius: Number;
                        InitN: Byte);
var I: Byte;
begin
  PieChart. Init(InitX, InitY);
  Radius := InitRadius;
  Add(New(CirclePtr, Init(LightRed,0,0,Radius)) ,0,0);
  for I := 1 to InitN do
   begin
    Add(New(LinePtr,Init(Random(16),0,0,0,2*Radius)),0,-Radius);
    RotateLeft(180 div InitN);
   end;
end;

constructor Auto.Init(InitX, InitY, D: Number;
                        InitLKoleso, InitPKoleso: KolesoPtr);
var LR, PR, LR2, PR2, R2: Number;
begin
  PieChart.Init(InitX, InitY);
  LKoleso := InitLKoleso;
  PKoleso := InitPKoleso;
  SX := InitX;
  SY := InitY;
  LR := LKoleso^.Radius; -
  PR := PKoleso^.Radius;
  LAlpha := 180/Pi/LR;
  PAlpha := 180/Pi/PR;
LR2 := 2*LR + 2*D;
PR2 := 2*PR + 2*D;
R2 := (LR2 + PR2)/2;

  Add(LKoleso, -D - LR, 0);
  Add(PKoleso, D + PR, 0) ;
  Add(New(LinePtr, Init(14,0,0,2*D,0)) ,-D,0);
  Add(New(LinePtr, Init(14,0,0,D,0)),-LR2,0);
  Add(New(LinePtr, Init(14,0,0,-D,0)),PR2,0);
  Add(New(ArcPtr, Init(14,0,0,R2,0,180)),-LR2+R2,0);
  dAngle := round(180/Pi*arctan((PR-LR)/(PR+LR+2*D)));
  RotateLeft(dAngle);
end;

procedure Auto.Draw;
begin
  if GetColor<>GetBkColor then Rotate;
  PieChart.Draw;
end;

procedure Auto.Rotate;
var T: Real;
  NX, NY: Number; A: Integer; begin
  NX := GetX; NY := GetY;
  T := VectorLength(NX-SX,NY-SY); { Ubehnut vzdialenost }

  A := VectorAngle(PKoleso^.GetX-LKoleso^.GetX,PKoleso^.6etY-
LKoleso^.GetY)-dAngle
       -VectorAngle(NX-SX,NY-SY); { uhol medzi starou a novou pozi-
ciou }

  sound(round(10+T*5)) ;
  LKoleso^.RotateRight(round(LAlpha*T));
  PKoleso^.RotateRight(round(PAlpha*T));
  RotateRight(A);
```

```
  nosound;
  SX := NX;
  SY := HY;
end;

var A: Auto;
    LK, PK: KolesoPtr;
    LR, LO, PR, PO, Roz: Byte;
    gd.gm: integer;

BEGIN
  Write(' Polomer laveho kolesa : '); ReadLn(LR);
  Write(´ Pocet os laveho kolesa : '); ReadLn (LO);
  Write(' Polomer praveho kolesa : '); ReadLn(PR);
  Write(' Pocet os praveho kolesa: '); ReadLn(PO);
  Write(' Rozpatie kolies       : '); ReadLn(Roz);
  gd := detect;
  InitGraph(gd,gm,'');
  LK := New(KolesoPtr, Init(0,0,LR,LO));
  PK := New(KolesoPtr, Init(0,0,PR,PO)); »
  A.Init(200,200,Roz,LK,PK);
  A.Show;
  A.Drag(10);
  A.Done;
  CloseGraph;
END.
```

LISTING Č. 2

```
unit MyFeet; {umoznuje rozchodit objekty}

INTERFACE

uses MyFigs;

type StepProcedure = Procedure;


type
  BiPedPtr = ^BiPed;

  BiPed = object (PieChart)
  { Color: Byte; }
  { X,Y: Number; }
  { Visible: Boolean; }
  { PieSlices: PieSlicePtr; }
  LFoot, RFoot: PieSlicePtr;
  MaxFootAngle, RotAngle: Integer;
  StepProc: StepProcedure;
  constructor Init(InitX, InitY: Number;
                   InitLFoot, InitRFoot: FigurePtr;
                   FeetRX, FeetRY: Number;
                   InitMaxFootAngle, InitRotAngle: Integer;
                   InitStepProc: StepProcedure);

 { destructor Done; virtual; }
 { procedure Drav; virtual; }
 { procedure Show; virtual; }
 { procedure Hide; virtual; }
 { function GetX: Number; }
 { function GetY: Number; }
 { function IsVisible: Boolean; }
 {procedure MoveTo(NewX, NewY: Number); }
 { procedure Drag(DragBy: Number); virtual; }
 { procedure Expand(ExpandBy: Real); virtual; }
 { procedure Contract(ContractBy: Real); virtual; }
 { procedure RotateLeft(Alpha: Integer); virtual; }
 { procedure RotateRight(Alpha: Integer); virtual; }
 { procedure Add(Item: FigurePtr; RX, RY: Number); }
    procedure ChangeFeet;
    procedure StepRight; virtual;
    procedure StepLeft; virtual;
end;

  NPedPtr = ^NPed;

  NPed = object (PieChart)
```

```
  { Color: Byte; }
  { X,Y: Number; }
  { Visible: Boolean; }
  { PieSlices: PieSlicePtr; }
    StepProc: StepProcedure;
    constructor Init(InitX, InitY: Number;
                     InitStepProc: StepProcedure);
  { destructor Done; virtual; }
  { procedure Draw; virtual; }
  { procedure Show; virtual; }
  { procedure Hide; virtual; }
  { function GetX: Number; }
  { function GetY: Number; }
  { function IsVisible: Boolean; }
  { procedure MoveTo(NewX, NewY: Number); }
  { procedure Drag(DragBy: Number); virtual; }
  { procedure Expand(ExpandBy: Real); virtual; }
  { procedure Contract(ContractBy: Real); virtual; }
  { procedure RotateLeft(Alpha: Integer); virtual; }
  { procedure RotateRight(Alpha: Integer); virtual; }
  { procedure Add(Item: FigurePtr; RX, RY: Number); }
    procedure StepRight; virtual;
    procedure StepLeft; virtual;
end;
```

LISTING Č. 3

```
program Kurca;

uses MyFigs, MyFeet, Graph, Crt;

type
  NohaPtr = ^Noha;

  Noha = object (MyFigs. Triangle)
    procedure Draw; virtual;
  end;

  KuraPtr = ^Kura;

  Kura = object (BiPed)
    constructor Init(InitX, InitY, InitR: Number;
                     InitStepProc: StepProcedure);
  end;

var C: Kura;
    nh2, nd2: Triangleptr;
    MM: Byte;
    gd,gm: integer;

procedure Noha.Draw;
begin
  Graph.Line(round(X),round(Y),round(X+dX),round(Y+dY));
  Graph.Line(round(X+dX),round(Y+dY),round(X+dX2),round(Y+dY2));
end;

procedure beep; far;
var j:integer;
begin
  if random>0.9 then
  begin
    nh2^.hide;
    nd2^.hide;
    nh2^.rotateleft(55);

   nd2^.rotateright(55);
   nh2^.show;
   nd2^.show;              ,
   for j:=0 to 100 do sound(3000+j*20);
   delay(150);
   nh2^.hide;
   nd2^.hide;
   nh2^. rotateright (55);
   nd2^.rotateleft(55);
   nh2^.show;
   nd2^.show;
 end
```

```
 else delay(30);
 nosound;
end;

constructor Kura.Init(InitX, InitY, InitR: Number;
                      InitStepProc: StepProcedure);
const a:real = 1;
     h = 7;
var p2,12: FigurePtr;
r1,r2,r3: Real;
C1, C2 : CirclePtr;
begin
  r1 := InitR;
  r2 := 2/3*InitR;
  r3 := InitR/5;
  p2:=New(NohaPtr,Init(12,0,0,0,r2,r2/2,r2));
  12 :=New(NohaPtr,Init(12,0,0,0,r2,r2/2,r2));
  BiPed.Init(InitX,InitY,12,p2,0,r1,35,5,InitStepProc);
  C1 := New(CirclePtr,Init(14,O,O,r1));
  Add(C1,0,0);
  C2 := New(CirolePtr,Init(14,0,0,r2));
  Add(C2,(r1+r2)*cos(a),-(r1+r2)*sin(a));
  Add (New (CirclePtr, Init (LightMagenta, O, O, r3)), (r1+r2 * l.5) *cos (a), -
(r1+r2*1.5)*sin(a));
  nh2: =New (TrianglePtr, Init (LightGray, O, O, -h/8, -h/2, h, O));
  Add(nh2,(r1+r2)*cos(a)+r2+l,-(r1+r2)*sin(a));
  nd2:=New(TrianglePtr,Init(LightGray,0,0,h,0,-h/8,h/2));
  Add(nd2, (r1+r2)*cos(a)+r2+l,-(r1+r2)*sin(a));
end;

BEGIN
  gd := detect;
  InitGraph(gd,gm,'');
  setbkcolor(Blue);
  C.Init(40,200,15,beep);
  C.Show;
  repeat
    C.Drag(lO);
    MM := mem[0:$0417];              { Ovladanie Shiftov }
    if (MM and $01) <> O then C.StepRight else
      if (MM and $02) <> O then C.StepLeft;
  until (MM and $03) = 0;
  C.Done;
  CloseGraph;
EMD.
```

LISTING Č. 4

```
program Stonozka;

uses MyFigs, MyFeet, Graph, Crt;

type
  NohaPtr = ^Noha;

  Noha = object (Triangle)
    constructor Init(InitR: Nunber};
    procedure Draw; virtual;
  end;

  HlavaPtr = ^Hlava;

  Hlava = object (PieChart)
    constructor Init(InitX, InitY, InitR: Number; Right: Boolean);
  end;

  ClanokPtr = ^Clanok;

  Clanok = object (BiPed)
    constructor Init(InitX, InitY, InitR: Number);
  end;

  Stonoh = object (NPed)
    constructor Init(InitX, InitY, InitR: Number;
                        PocetClankov: Integer;

                     InitStepProc: StepProcedure);
end;
```

```
var LH, RH: HlavaPtr;     { Left Read & Right Head Pointer }
    MM: Byte;
    gd.gm: integer;


procedure nic; far;
begin
end;

procedure hlavotoc; far;

begin
  RH^.rotateright(45);
  LH^.rotateright(45);
end;

constructor Noha.Init(InitR: Number);
begin
  Triangle. Init (12,0,0,0, InitR, InitR/3, InitR);
end;


procedure Noha.Draw;
begin
  Graph.Line (round (X), round (Y) , round (X+dX), round (Y+dY)) ;
  Graph.Line(round(X+dX) ,round(Y+dY),round(X+dX2),round(Y+dY2));
end;

constructor Hlava.Init(InitX, InitY, InitR: Number; Right: Boolean);
var D: Real;
begin
  PieChart.Init(InitX, InitY);
  Add(New(CirclePtr, Init(14,0,0,InitR)),0,0);
  if Right then D := InitR/3 else D := -InitR/3;
  Add(New(CirclePtr, Init(13,0,0,InitR/4)),D,-D);
end;

constructor Clanok.Init(InitX, InitY, InitR: Number);
var NL, NR: NohaPtr;
begin
  NL := New(NohaPtr, Init(InitR) );
  NR := New(NohaPtr, Init(InitR) ) ;
  BiPed.Init(InitX,InitY,NL,NR,0,InitR,20,5,nic) ;
  Add(New(CirclePtr, Init(14,0,0,InitR)),0,0);
end;

constructor Stonoh.Init(InitX, InitY, InitR: Number;
                        PocetClankov: Integer;
                        InitStepProc: StepProcedure);
var I: Integer;
   Clan: ClanokPtr;
begin
  NPed.Init(InitX, InitY, InitStepProc);
  for I := 1 to PocetClankov do
    begin
      Clan := New(ClanokPtr, Init(0,0,InitR));
      if I = 1
        then begin LH := New(HlavaPtr, Init(0,0,InitR,False));
              Clan^.Add(LH,-2*InitR,0); end;
      if I = PocetClankov
        then begin RH := New(HlavaPtr, Init(0,0,InitR,True)) ;
              Clan^.Add(RH,2*InitR,0); end; Add(Clan,I*InitR*2,0);
   end;
end;

var C: Stonoh;
    R, N: Byte;

BEGIN
  Write(' Polomer tela stonozky(max 30): '); ReadLn(R);
  Write(' Počet clankov stonozky(max 100): '); ReadLn(N);
  if R>30 then R:=30;
  if N>100 then N:=100;
  gd := detect;
  InitGraph(gd,gm,'');
  C.Init(100,100,R,N,hlavotoc);
```

```
  C.Show;
  repeat
    C.Drag(lO);
    MM := mem[0:$0417];
    if (MM and $01) <> O then C.StepRight else
      if (MM and $02) <> O then C.StepLeft;
  until (MM and $03) = 0;
  C.Done;
  CloseGraph;
END.
```

---

9. časť

Skoro po celý rok sme sa stretávali s najmodernejšími prvkami programovania – s objektmi. Objektovo orientované programovanie, familiárne nazývané oopéčko, zasiahlo tak hlboko do práce programátorov, že, div sa svete, aj programovanie v assembleri už má prvky objektového programovania.

Verím, že aj vám prirástli objekty k srdcu, pocítili ste ich silu a krásu. Priznávam, že v časopise nie je možné z priestorových dôvodov vysvetliť všetko, ako by sa žiadalo, a preto, hlavná časť tohto seriálu (tá, ktorá už nebude vidieť) bude vaše samoštúdium. Pozrime sa na to prakticky:

Keď som začínal s programovaním, mojou métou bolo zvládnuť štrukturované programovanie. Napísal som v ňom veľa užitočných malých programíkov alebo, lepšie povedané, utilitiek, napr. na konverziu textov, DBF súborov, programov na ovládanie tlačiarní a iné. O objektoch som už aj vtedy počul, ba aj kadečo čítal, zdali sa mi však nepochopiteľné, komplikované a hlavne nevyužiteľné. Keď som po určitom čase pristúpil k môjmu prvému veľkému projektu (bol to program, ktorý evidoval došlý materiál na opravu, vytváral baliace listy, zratúval náklady na opravu, objednával náhradný materiál a vykonával ďalšie s tým súviasiace úkony), začínal som klasickým štruktúrovaným "bezobjektovým" programovaním. Keďže nijaký väčší celok sa nepíše na prvý raz (k dnešnému dňu prešiel projekt 98! väčšími zmenami, úpravami a opravami), dostal som sa do stavu, keď som mal napísaných zhruba 3000 riadkov zdrojového textu v niekoľkých unitách a bolestne som si uvedomil, že tadiaľto cesta nevedie. Unity boli vzhľadom na počet procedúr také rozsiahle, že boli neprehľadné. Neostávalo nič iné ako začať znova, ale inak.

Začal som študovať objekty, ale už pri štúdiu som myslel na konkrétne využitie v mojom programe. Najprv som ich nenávidel, tak som si ich skúšal na čiastočných procedúrkach. Zrazu som zistil, že keď som prácne nadefinoval vstupnú kartu materiálu ako objekt, jej variácie sú potom veľmi jednoduché. Objekty mi uľahčovali programovanie. Dnes sa ani nepamätám, ako som ten-ktorý objekt vlastne vytvoril, a keď upravujem tento môj prvý projekt, stačí mi len vedieť, aké má daný objekt vlastnosti a schopnosti, tie nám už dobre známe objektové veličiny. Predefinujem, pridám, opravím, skompilujem a ide sa ďalej. Dnes má môj program 10 000 riadkov zdrojáku v 17 unitoch. Len vďaka objektom je funkčný a moje kolegyne ho stale používajú. A to všetko v textovom režime.

Dnes mám objekty rád. Je pravda, že sem-tam nerozumiem, prečo sa niektorý objekt správa tak, ako sa práve správa, ale to sa stáva asi každému programátorovi. Jedno viem isto – bez objektov sa slušný program dnes nedá napísať. Zvlášť program, ktorý pracuje s grafikou, ako sme si ukázali na niekoľkých príkladoch.

Keď som začínal s programovaním pod Windows, znalosť objektov sa mi prenáramne hodila. Ten, kto nepozná základy objektov, nech do windovského programovania ani nenazerá. Nebude rozumieť a nič poriadne ani nevytvorí. Prečo? Lebo Billove Okná sú jedna veľká kopa grafických objektov!

Budete mi oponovať. Na scénu dnes predsa nastupuje nový trend – vizuálne programovanie. Asi najslávnejším predstaviteľom sú Delphi. Je to také ťahanie myškou sem a tam, vkladanie prvkov do formulára, tu editačné okno, tam tabuľka, kompilácia a hotovo! Veru áno, ale len niekoľko veľmi jednoduchých aplikácií.

Jednoduché spájanie komponentov do hotového programu veľmi zjednodušuje návrh projektu, ale odozvy na jednotlivé udalosti musíme už dopísať do zdrojového programu sami. A viete, že je to aj správne? Veď čo by to bolo za programovanie, keby sa len ťahalo a "lepilo". To by predsa mohol programovať každý. Hmm…, niežeby som to ľudom neprial, ale každý by mal robiť svoju prácu. Pekár pečie chlieb, murár stavia domy, lekár lieči, učiteľ učí, opravár opravuje, programátor nech programuje.

Delphi je odroda Turbo Pascalu. Kto túži programovať vo Windows, nech začne s Delphi. Ale pozor, je celé objektové! O tom, že k programovaniu sa dá pristupovať rôzne, svedčí aj e-mail, ktorý som dostal nedávno od známeho z Ameriky:

“Na svete prestalo svietiť slnko. Všetci sa obrátili na IBM, aby problém vyriešili. Keďže nič nevymysleli, povedali, že to je softvérový problém, a postúpili to Microsoftu. A čo urobil Bill? Zaviedol tmu ako priemyselný štandard.”

A tak cieľom celého časopisového kurzu objektovo-orientovaného programovania bolo uviesť vás do veľkého sveta ozajstného programovania. Lebo programovanie je celoživotná láska. Láska nekonečná, z ktorej sa nevystupuje.

Miroslav Oravec