

Úvod

Hlavní náplní tohoto textu je programování. Mohlo by se zdát, že v dnešní době, kdy si již nedovedeme představit život bez počítačů, je pojem programování zcela jasný téměř každému. Bohužel tomu tak není. Dokonce si troufám tvrdit, že blíže pravdě je tvrzení opačné. Před nějakými dvaceti lety, kdy se počítače díky klesající ceně začaly postupně šířit z úzce specializovaných pracovišť k dalším uživatelům, musel v podstatě každý, kdo chtěl s počítačem komunikovat, počítačům také rozumět. Neobešel se bez dosti podrobných znalostí instrukcí operačního systému, bez poměrně hlubokých vědomostí o nemnoha tehdy existujících aplikacích. A pokud chtěl s pomocí počítače řešit nějaký problém, téměř vždy se musel nejdříve naučit programovat.

Dnes je situace zcela jiná. Téměř každý, kdo chce dnes řešit nějaký problém pomocí počítače, může postupovat tak, že si z nabídky na trhu vybere hotový produkt, ten zakoupí a pak mu nezbyvá než doufat, že tento software příslušné problémy řešit „umí“. Takový uživatel toho o počítačích (tím méně o programování) příliš vědět nepotřebuje. Stačí umět mačkat sugestivně popsaná tlačítka, či klikat myší na příslušný text a nechat se hýčkat plovoucími nápovědami. Počítač něco provádí, odněkud něco „tahá“, někam něco ukládá a pak poskytne očekávaný výsledek.

Tento ideální případ však téměř nikdy nenastane. Především lze snad ke každému problému najít hned několik softwarových produktů, které by ho mohly řešit. Budoucí uživatel pak stojí před dilematem, co si vlastně pořídit a v našich zeměpisných šířkách dosti často vymýšlí také to, jak koupí (jemně řečeno) „šikovně obejít“. Pak má značné problémy s instalací (to se dnes bohužel často stává i s legálně nabytým produktem), s tím, že systém „padá“, dělá hrubé chyby, či nedává výsledky, které uživatel očekával. Po několika minutách (hodinách, dnech...) marných pokusů s víceméně náhodným mačkáním tlačítek, nastavováním nejrůznějších parametrů apod. se mnozí ponoří do „počítačových“ problémů, které dnes mnoho lidí označuje jako „programování“. Začíná se zjišťovat, který soubor kam přkopírovat, aby se systém rozběhl, jak si ho nakonfigurovat, aby byl použitelný, který ovladač jak aktualizovat, aby se výsledky slušně zobrazily či vytiskly apod. Většinou na to uživatel sám nepříjde a začíná shánět někoho, komu tento systém funguje. Jakmile pak systém jakž takž pracuje, je třeba se prokousat množstvím funkcí a možností rozsáhlého systému, abychom záhy zjistili, že velkou většinu z nich nepotřebujeme, a ty, kvůli nimž jsme systém kupovali, nepracují tak, jak jsme si původně představovali. Mnohdy si pak ani neuvědomíme, že jsme vynaložili značné finanční prostředky proto, aby nám na řešení našich původních problémů nezbyl vlastně žádný čas.

V dnešní době bohužel stále méně lidí rozumí softwaru, který kupují. Právě takoví uživatelé jsou pak často překvapeni, když zjistí, že zdánlivě neřešitelný problém vyřeší nevelký jednoúčelový program, vyžadující několik dní (někdy jen hodin) práce šikovného „skutečného“ programátora.

Tento text je tedy zaměřen na algoritmizaci a programování. Probírané algoritmy jsou řešeny v prostředí Borland Delphi. Principy, na kterých jsou tyto algoritmy založeny, však fungují, ať jsou zapsány v kterémkoliv programovacím jazyku. Právě proto bychom si tedy měli všimnout spíše programátorských technik a způsobu uvažování. Právě takové schopnosti přezívají soudobý hardware i konkrétní implementace programovacích jazyků a jsou pro programátora tou nejcennější devizou.

Autor

1. Základní pojmy

Program je posloupnost **příkazů**, které popisují řešení nějakého problému.

Každý program vyžaduje **autora**, který tento program sestaví či napíše a **procesor**, který program provede. Každý program vykonává jisté operace s nějakými objekty, jejich výsledkem je transformace těchto objektů. Objekty, které program zpracovává, se nazývají **data**. V programu musí být především popsány vhodným způsobem vlastnosti dat, se kterými je program schopen pracovat, a operace, které je s nimi schopen provádět. Tento popis nazýváme **deklarací**. Provádění programu se skládá z provádění jednotlivých **příkazů**. Počet příkazů, pořadí jejich provádění či vzájemná návaznost jsou většinou závislé na zpracovávaných datech. Tyto skutečnosti jsou v programu ošetřovány **větvením** a **cykly**.

Je-li procesor schopen provádět vždy jen jeden příkaz, může s provádění následujícího příkazu začít až po splnění příkazu předcházejícího – jedná se o **sekvenční procesor**. Je-li procesor schopen vykonávat více příkazů současně, mohou být některé příkazy prováděny paralelně – jedná se o **paralelní procesor**.

V tomto širokém slova smyslu může být programem jakýkoli pracovní postup. Data mohou být brambory, okurky, vejíčka atd, programem je kuchařský předpis na výrobu bramborového salátu. Procesor – kuchař – je v tomto případě paralelní, neboť je schopen současně vařit brambory, vejíčka a dejme tomu připravovat majonézu.

Programy určené pro počítač jsou však delší a složitější. Nejen proto, že většinou popisují složitější činnosti, ale především proto, že jsou určeny pro procesor bez inteligence, který sice umí instrukce přesně a rychle provádět, ale nikdy není schopen chápat jejich význam. Není proto schopen domýšlet to, co nedomyslel autor programu. Program tedy vyžaduje pečlivost, musí být napsán přesně a většinou vyžaduje i přípravné práce či úvahy. Obecně lze práci na programu rozdělit do několika etap:

1. Definice problému: Je třeba si ujasnit, co má vlastně program řešit. Tento krok není obecně tak triviální, jak by se mohlo na první pohled zdát. Je důležitý především v situacích, kdy zadavatel programu není zároveň jeho programátorem. Zadavatel programu by měl specifikovat:

- a) **Vstupní data** – jejich rozsah, strukturu a množiny přípustných hodnot.
- b) **Způsob zpracování (je-li zadavateli znám)** – známé vzorce, funkční závislosti atd.
- c) **Specifikace výstupních dat** – struktura, formáty, výstupní zařízení atd.
- d) **Požadavky na ošetření očekávaných výjimečných situací** – např. reakce programu na zadání chybných vstupních dat atd.

Forma definice problému závisí na jeho složitosti. Většinou je slovní s použitím terminologie a symboliky obvyklé v dané aplikační oblasti.

2. Stanovení principů řešení a hrubé rozvržení programu: V jednodušších případech to znamená stanovení vhodných metod, které nespécifikoval zadavatel, a způsob jejich návaznosti. Ve složitějších případech vhodné rozvržení problematiky na relativně samostatné podproblémy, které budou řešeny samostatnými programovými moduly a způsobů komunikace mezi těmito moduly.

3. Algoritmizace – vytvoření algoritmu řešení: Tento krok spočívá v detailním rozpracování problému na kroky. Jednotlivé kroky mohou zpočátku představovat i složité problémy, postupně se však rozkládají na podproblémy stále jednodušší. Programátor se zde řídí zásadou „**co nemusíš udělati dnes, odlož na zítřek**“. Obecně lze říci, že tato etapa končí rozkladem na problémy, které jsou z hlediska programátora elementární. (programování „shora dolů“). O tom, zda ten či onen krok v této fázi je „elementární“, rozhoduje samotný programátor. Např. zprogramování řešení soustavy lineárních rovnic Gaussovou eliminační metodou může být pro někoho problém elementární, pro někoho neřešitelný. Forma zápisů při algoritmizaci je zcela v rukou programátora. Dříve byly často používány tzv. vývojové diagramy, či bloková schémata, jejich nevýhodou však je absence prostředků pro popis datových struktur. I jejich popis by však měl být nedílnou součástí této fáze prací. Často se proto používá přímo konstrukcí daného programovacího jazyka s tím, že „elementární“ problémy jsou popsány slovně a budou doeditovány až při kódování programu.

4. Editace programu: Algoritmus je třeba rozložit na kroky, které jsou elementární z hlediska počítače, tj. zapsat v některém programovacím jazyce. Jedná se vlastně o "poslední zjemnění" abstraktních operací prováděných v předchozím kroku.

5. Ověření správnosti programu: Program, který nějakým způsobem transformuje data, ještě nemusí být správný. Ověření správnosti spočívá v ověření výsledků programu při všech situacích, které mohou nastat. To však zvláště u rozsáhlejších programů většinou není možné. Program proto ověřujeme jen na vhodné množině zkušebních vstupních dat.

Programovací jazyky: Nejstaršími a v hierarchii programovacích jazyků nejnižšími jsou jazyky strojové a jejich symbolické verze – jazyky symbolických adres – assembly. Program zapsaný v tomto jazyku je posloupností příkazů – instrukcí, které může počítač přímo provádět (každý odpovídá jedné operaci procesoru). Takový program je velmi výhodný pro počítač, práce programátora je však velmi obtížná, neboť musí porozumět mnohdy dlouhým posloupnostem symbolických instrukcí. Jedním z hlavních důvodů obtížnosti programování v těchto jazycích je jejich nestrukturovanost. Tento nedostatek se snaží postupně odstraňovat tzv. vyšší programovací jazyky, pro které je charakteristické hledání a zavádění vhodných operačních, řídicích i datových struktur. Ty zvyšují srozumitelnost programovacího jazyka a usnadňují tak programování. Program zapsaný ve vyšším programovacím jazyku je určen pro technicky realizovaný procesor, který má program provést. Realizace procesoru, který „rozumí“ vyššímu programovacímu jazyku, pomocí běžného počítače, se nazývá **implementace** jazyka na počítači a spočívá ve vytvoření vhodného programovacího prostředí. Jeho součástí je vhodný nástroj pro psaní programu – **editor**, prostředek pro převod programu z vyššího jazyka do tzv. relativního kódu – **překladač (kompilátor)**, popř. program umožňující přímou interpretaci programu počítačem – **interpret**, prostředek usnadňující hledání chyb v programu – **debugger** a konečně sestavovací program – **linker**, jehož výsledkem je program spustitelný přímo v operačním systému.

Tento text je zaměřen především na algoritmizaci často se vyskytujících úloh. Úlohy budou řešeny v Pascalu, a to v konkrétní implementaci firmy Borland – v prostředí Delphi.

2. Základní prvky programu

Každý program je zapsán pomocí posloupnosti symbolů. Každý vyšší programovací jazyk je určen množinou povolených symbolů a pravidly, jak tyto symboly seskupovat – **syntaktickými pravidly**. Aby program mohl fungovat, musí být každá jeho část **syntakticky správná**. Nejjednoduššími syntakticky správnými posloupnostmi jsou **speciální symboly** (většinou operátory a závorky), **klíčová slova** (např. while, do, repeat atd) a **identifikátory**. Ty mohou být standardní, tj. definované jazykem (sin, sqrt, real apod). nebo lze použít identifikátory, které definuje programátor (Obvod, DelkaStrany apod.).

Klíčová slova jsou slova, která si programovací jazyk vyhradil pro svou potřebu a nesmí být používány v jiných souvislostech. Některá důležitá klíčová slova Pascalu postupně probereme. V programu jsou tato slova editorem automaticky zvýrazňována tučným písmem.

Každý program pracuje s konstantami, proměnnými a jejich typy, skládá se z deklarací, procedur, funkcí, výrazů a příkazů. Pascal na rozdíl od některých jiných jazyků nerozlišuje velká a malá písmena. To znamená, že např. Repeat, repeat, REPEAT a rPEAT je stále jeden a tentýž příkaz. To je např. v jakékoli verzi jazyka C jev zcela nevídaný. Jednotlivé příkazy je třeba oddělovat středníkem.

Kromě takovýchto zásad, jejichž dodržování je nutné pro chod programu, existují ještě pravidla, která se sice nemusí striktně dodržovat, ale která mnohdy usnadní programátorovi život a zvyšují čitelnost programu. Obecný návod, jak má program vypadat, neexistuje. Vzhled programu závisí na zvyklostech programátora a je na každém z nich, které zásady mu budou vyhovovat a které bude dodržovat. Zde jsou některé z nich:

- I když Delphi nerozlišuje velká a malá písmena, je dobré psát identifikátory a příkazy „jedním stylem“ - buď všechna písmena malá, nebo první velké, ostatní malá, popř. všechna velká (poslední možnost je však poněkud nepohodlná).
- Každý příkaz je dobré psát na samostatný řádek. Týká se zvláště delších příkazů. Program je jednak přehlednější, jednak se tím usnadňuje ladění. Delší příkaz je vhodné rozdělit na několik řádek, a to zvláště v případě, chceme-li program tisknout na tiskárně. Zde je třeba pamatovat na to, že části řádků přesahující svislou čáru editovacího okna nebudou vytištěny. V případě dělení příkazu je vhodné pečlivě uvážít, kde příkaz rozdělit a na dalším řádku „odskočit“ tabelátorem.
- Slova **begin** a **end** píše na samostatné řádky. Za každé **begin** okamžitě napište **end** a až poté si rozhrňte text a vyplňujte blok programu mezi těmito „programovými závorkami“. Nezapomenete tak nikdy „závorku zavřít“. Chybějící „**end**“ kompilátor neodpouští a zvláště ve složitějších programech se velmi obtížně hledají.
- Každý vnořený blok je vhodné posunout doprava vůči bloku předchozí úrovně (nejlépe tabelátorem).
- Jména proměnných, procedur a funkcí volte v souladu s jejich významem. Šíření textem zde není na místě, zkratkovité identifikátory jsou často matoucí.
- Nešetřeme komentáři.

Základními stavebními kameny programu jsou:

Konstanty: Jsou datové objekty, jejichž hodnota se během vykonávání programu nemění (stránka má 60 řádků, $\pi=3.1415926\dots$ atd).

Proměnné jsou datové objekty, jejichž hodnota se během programu může měnit.

Procedury a funkce popisují dílčí algoritmy, v nichž konkrétní data mohou být zastoupena formálními parametry.

Deklarace slouží k pojmenování a zavedení dalších součástí programu – konstant, proměnných a jejich typů, procedur a funkcí. Podle účelu rozlišujeme deklaraci **informativní**, která informuje překladač o existenci nějakého objektu (typu, proměnné, funkci...) a **definiční**, která přikazuje překladači příslušný objekt vytvořit. Z hlediska rozsahu platnosti mluvíme o deklaraci **globální**, která je účinná pro celý program, a **lokální**, která je účinná jen pro část programu.

Výraz je operační struktura, popisující výpočet hodnoty. V programovacím jazyce mají podobný tvar jako v matematice s tím, že nemohou používat speciální matematické znaky (řecká písmena, odmocnítko, vodorovnou zlomkovou čáru atd.).

Příkaz je programová jednotka popisující jednotlivé výpočetní akce a jejich návaznosti.

Konstanty: je možné používat tak, že uvedeme příslušnou hodnotu přímo v místě, kde se použije, např:

```
while Radek<60 do TiskRadku;  
for Počet:=1 to 28 do CtiData;  
Obvod:=3.1415926*Prumer;  
Titulek:='Karel Čapek: RUR';
```

Takový přímý zápis hodnoty v programu se nazývá **literál** (v uvedeném příkladu 60, 28, 3.1415926). Hodnotu je však možno pojmenovat – tj. konstanty **deklarovat** a místo literálu pak používat identifikátor konstanty:

```
const DelkaStrany = 60;           {číselná konstanta v desítkové soustavě}  
PocetStudentu = 28;  
Pi = 3.1415926;  
Adresa = $A2CC {číselná konstanta v šestnáctkové (hexadecimální) soustavě}  
Znak = 'a';                      {znaková konstanta - přímý zápis}  
Tabelator =#9                    {znaková konstanta - zápis pomocí ASCII kódu}  
Autor ='Karel Čapek';            {řetězcová konstanta}  
Nazev='RUR';
```

Výše uvedený fragment programu pak vypadá následovně:

```
while Radek<DelkaStrany do TiskRadku;  
for Počet:=1 to PocetStudentu do CtiData;  
Obvod:=Pi*Prumer;  
Titulek:=Autor+'; '+Nazev;
```

Používáním identifikátorů konstant se zvyšuje srozumitelnost programu (vhodně volené identifikátory prozrazují sémantický význam konstanty). Kromě toho se usnadňuje modifikace programu. Napíšeme-li rozsáhlý program pro práci s textem, který předpokládá stránku s 60 řádky a používáme-li literály, vyžaduje modifikace programu pro

padesátirádkové stránky mnoho přepisů. Při „ručním“ přepisu hrozí, že některý výskyt přehlédneme, při „automatickém“ se naopak vystavujeme riziku, že opravíme i výskyt literálu, který nesouvisí s počtem řádků na stránce (náhodou můžeme pracovat se seznamem šedesáti studentů). Jestliže příslušnou konstantu pojmenujeme, stačí opravit jedinou hodnotu u její definice.

Proměnné: Každá proměnná je určitého **typu**, kterým je specifikována množina přípustných hodnot a operace, které lze s těmito hodnotami provádět. Před použitím proměnné je třeba deklarovat její jméno a typ. Často používané typy jsou standardní, další typy může definovat programátor.

Jazyk Pascal je přísně typový jazyk. Za všech okolností přísně hlídá datové typy proměnných a nedovolí, aby se s proměnnou prováděla operace, která neodpovídá jejímu typu. Někdy však bývá výhodné, aby se obsah proměnné jednoho typu umístil do proměnné jiného typu. Například **číslo** 12 jako výsledek operace 3.4 vložit do **textu** - např. do textové zprávy o tomto výsledku. Takové operace nazýváme **přetypování**. Přetypování proměnné je možné provést v podstatě třím způsobem a budeme se jím zabývat později.

Typ proměnné může být **jednoduchý** (např. x je reálné číslo) nebo **strukturovaný** (např. A je bod v prostoru - uspořádaná trojice reálných čísel, záznam o studentovi obsahuje jméno, příjmení, datum narození a prospěch u maturity).

Přehled předdefinovaných typů:

Jednoduché

ordinální	boolean
(celočíslné)	byte
	char
	integer
	word
	longint
reálné	real
(čísla, která mají	double
desetinnou část)	extended
řetězcové	char(znak)
(znaky, symboly)	string
	(posloupnost znaků)

strukturované

pole	array
záznam	record
množina	set
soubor	file
ukazatel	pointer

(neuvádíme zde kompletní výčet, jen typy nejpoužívanější).

Ordinální typy jsou uspořádané množiny hodnot. Každá hodnota kromě první má tedy svého předchůdce a každá kromě poslední svého následovníka. Kromě výše uvedených předdefinovaných typů lze použít typy definované programátorem – výčtový typ a typ interval. Na zjištění aktuální hodnoty v množině lze použít funkci **Ord**. Mezi ordinální typy patří i Char (znak). Množina hodnot typu char je uspořádána podle rozšířeného souboru znaků ASCII. Ordinalita znaku je v souladu s pozicí znaku v tabulce ASCII a funkce Ord pro tento typ tedy vrací ASCII kód znaku. Na zjištění hodnoty předchůdce proměnné ordinálního typu se používá funkce **Pred**, na zjištění následovníka **Succ**.

Datový typ **boolean** má pouze dvě hodnoty – **True** (popř. **1**) pro stav pravda, **False** (nebo **0**) pro stav nepravda. Tomuto datovému typu lze přiřadit buď konkrétní hodnotu – např. `Prvocislo_Nalezeno:=False`, nebo výsledek vyhodnocení logického výrazu – např. `Delitelno_Sedmi:=(x mod 7 = 0)`. Logické výrazy nacházejí uplatnění zejména v podmíněných příkazech a příkazech cyklu.

Předdefinované ordinální typy

	rozsah	počet bytů
Byte	0 .. 255	1
ShortInt	-127 .. 127	1
Integer	-32768 .. 32768	2
Word	0 .. 65535	2
DWord	0 .. 4294836225	4
LongInt	-2147483647 .. 2147483647	4

Delphi připouští zadávání ordinálních hodnot v hexadecimálním formátu. Pro zadávání hexadecimální hodnoty se používají cifry 0..9 a dále A (10), B (11), C (12), D (13), E(14), F (15). Při specifikaci je nutné před vlastní hodnotu vložit znak \$, např. hexadecimální vyjádření \$A23D odpovídá dekadickému 41533, neboť $41533 = 10 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16 + 13$.

Ordinální typy definované programátorem:

Interval – definuje rozsah hodnot ordinálního typu. Specifikuje nejmenší a největší hodnotu v intervalu. Obě hraniční konstanty musí být stejného ordinálního typu. Hodnota proměnné typu interval nesmí za běhu programu překročit předepsané meze.

Výčet – definuje množinu hodnot výpisem identifikátorů. Ordinalita identifikátoru je dána jeho pozicí v seznamu. První identifikátor má ordinální hodnotu nula.

Při definici typu programátorem je žádoucí volit název tak, aby z něj byl patrný sémantický význam typu. Navíc doporučuji tako první znak použít T tak, aby bylo zřejmé, že jde o typ (tuto konvenci totiž používá i Delphi)

Příklad deklarace ordinálních typů a proměnných:

```

Type   TBarevna_Slozka   = Byte;                      {standardní jednoduché typy}
          TCele cislo       = Integer;
          TDelka            = Word;

                                     {jednoduché typy definované programátorem}
          TMalePismo       = 'a'..'z';                      {typ interval - znakový}
          TJednociferneCislo = 1..9;                         {typ interval - číselný}
          TMalaSamohlaska   = ('a','e','i','o','u');         {výčtový typ - znakový}
          TJednocifernePrvocislo = (2,3,5,7);               {výčtový typ - číselný}

Var   Red, Green          :TBarevnaSlozka ;
          Blue               :Byte
          CeleCislo          :TCeleCislo;

```

MalePismo :TMalePismo;
 Jednociferne Cislo :TJednociferneCislo;
 MalaSamohlaska1,
 MalaSAmohlaska2 :TMalaSamohlaska;
 JednocifernePrvocislo :TJednocifernePrvocislo;

Reálné typy

	rozsah	platné číslice	počet bytů
Single	$\langle 1.5 \cdot 10^{-45}; 3.4 \cdot 10^{38} \rangle$	7	4
Real	$\langle 2.9 \cdot 10^{-39}; 1.7 \cdot 10^{38} \rangle$	11	6
Double	$\langle 5 \cdot 10^{-324}; 1.7 \cdot 10^{308} \rangle$	15	8
Extended	$\langle 3.4 \cdot 10^{-4932}; 1.1 \cdot 10^{4932} \rangle$	19	10

Řetězcové typy

Datový typ **Char** (znak) má velikost 1 byte a uchovává jeden znak. Znamená to, že lze použít 255 znaků. Znakové konstanty se vyjadřují tak, že se znak uzavře do apostrofů, tj. např. 'A', 'l', '&'. Znak 'l' v tomto případě znamená znak pro cifru jedna, nikoli číselnou hodnotu. Každému znaku odpovídá číselná hodnota typu byte a naopak. Toto vzájemné přiřazení je dáno tzv. ASCII tabulkou (American Standard Code for Information Interchange). Prvních 128 znaků ASCII tabulky je univerzálních, horních 128 znaků je vyhrazeno pro národní znakové sady.

Rádobyvtipní autoři nejrůznějších spisků o programování často uvádějí, že kompletní ASCII tabulka je oblíbenou součástí programátorských příruček, neboť zabere dost místa a její vygenerování nedá moc práce. Byla by to pravda v případě, že by ASCII tabulka byla jenom jedna. Pak bychom ji byli skutečně schopni nahradit několikařádkovým programkem, který by nám ji vygeneroval (v kapitole 8 to také uděláme). Ovšem jen pro češtinu existuje dnes asi desítka nejrůznějších ASCII tabulek. Pro zobrazení správné znakové sady je třeba vybrat tu „správnou“ tabulku, což bohužel české mutace žádného dostupného (a většinou velmi drahého) software nedokážou. Proto tak často při použití češtiny dochází k zobrazování nejrůznějších nesmyslných znaků na místech, kde mají být zobrazeny znaky s diakritikou. Programek, který by v daném textu rozpoznal správné kódování a zvolil správnou tabulku, je velmi jednoduchý a sám o sobě zabere několik řádek (i ten si dále sestavíme). Potřebuje jediné – ASCII tabulky všech „češtin“, které naši „mutátoři“ používají. Pro vývojáře je dnes bohužel lukrativnější do textového editoru programovat kancelářskou sponku, která běhá přes dokument a divoce poulí očima, než aby se starali o to, aby text vytvořený jejich mutantem byl čitelný i v mutantu konkurenčním.

Za současné situace by deset-patnáct stránek obsahujících kompletní tabulky všech používaných kódování češtiny bylo rozhodně cennějších než devadesát procent takzvané počítačové literatury, která dnes v prodejnách zbytečně zabírá místo knihám. Takový seznam však v současnosti k dispozici není a skuteční programátoři se ho asi hned tak nedočkají. A tak potíže s češtinou ve všech myslitelných aplikacích přetrvávají již několik desetiletí a zřejmě nás všechny přejijí.

Při práci s typem Char je třeba rozlišovat reprezentaci velkých a malých písmen a v kódech dekadický a hexadecimální zápis. Číslo musí předcházet znak #: např. 'A' je totéž jako #65 (dekadický ASCII kód) nebo #\$41 (hexadecimální ASCII kód). Znaky s kódy 0 až 31 se

nazývají řídící, protože se původně používaly k řízení dálkopisných operací. Tyto znaky nelze generovat z klávesnice přímo, ale pouze přes rozšiřující klávesu CTRL, po které následuje příslušné písmeno.

Datový typ **String** (řetězec) je posloupnost znaků. Je-li uváděn jako konstanta, musí být ohraničen apostrofy. V paměti je řetězec uložen na tolika bytech, kolik znaků obsahuje, plus jeden. V nultém byte je uložena délka řetězce (počet znaků počítáno od prvního nikoliv od nultého). Deklarujeme-li řetězec na určitou délku (např. `String[15]`) a skutečný obsah je menší, pak paměť obsazují jen naplněné znaky. V programu je proto lépe uvádět deklaraci bez uvedení délky. V programu lze s řetězcem zacházet jako s poli znaků. Často se používá tzv. prázdný řetězec, který neobsahuje žádný znak (v programu jej reprezentují dva bezprostředně po sobě následující apostrofy). Jestliže se náš řetězec jmenuje `Chybove_Hlaseni` a je třeba ohlásit chybu, provedeme přiřazení – např.: `Chybove_Hlaseni := 'Chyba – dělení nulou !'`. Jestliže pak byla chyba napravena, vymažeme obsah tohoto hlášení přiřazením prázdného řetězce: `Chybove_Hlaseni := ''`. Chceme-li do řetězce vložit znak apostrof, musíme uvést dva apostrofy bezprostředně po sobě následující. Chceme-li např. v editovacího okénka s názvem `Edit1` pomocí programu poslat text **John's book**, je třeba provést přiřazení `Edit1.Text := 'John's book'`. Do řetězců lze vkládat i znaky vyjádřené jejich ASCII hodnotou včetně znaků řídících.

Strukturované typy představují více než jednu hodnotu.

Pole – má pevně stanovený počet složek jednoho typu. Jeho definice zahrnuje rozměr pole a typ jeho složek

Záznam – obsahuje specifikaci jednotlivých položek, které mohou být různého typu. Při definici je třeba definovat typ každé položky.

Množina – je definována nad ordinálním typem.

Soubor – je posloupnost položek udaného typu.

Ukazatel – neobsahuje samotná data, pouze jejich adresy v operační paměti počítače. Obvykle je asociován s určitým datovým typem.

Příklad deklarace typů a proměnných:

```

Type   TPoint = array [1..2] of Real;           {bod v rovině}
          TMatrix = array [1..50,1..50] of Real;    {čtvercová matice padesátého řádu}
          TZnaky = set of Char;                     {množina všech znaků}
          TPisma = ['a'..'z','A'..'Z'];              {množina všech velkých a malých písmen}
          TAdresa = record                          {záznam adresy}
              Ulice      :String[20];
              CisloDomu   :Integer;
              Misto       :String[20];
              PSC         :String[5];
          end;
          TStudent = record                          {záznam o studentovi}
              Jmeno       :String[10];
              Prijmeni    :String[15];
              Vek         :Byte;
              Adresa      :TAdresa;
          end;
          TSouborVysledku = file of double;          {soubor reálných hodnot}
          TKartoteka      = file of TStudent;         {soubor záznamů typu student}

```

```

TSoubor      = file;                                {soubor bez udaného typu}
TUkazatel    = ^TStudent;                            {ukazatel na záznam o studentovi}
TSeznamStudentu= array [1..100] of TStudent;
Var A,B:      TPoint;
      Student:  TStudent;
      Index:    Integer;
      x,y:      Real;
      Matrix:   TMatrix;
      Kartoteka: TKartoteka;
      Ukazatel: TUkazatel;
      SeznamStudentu: TSeznamStudentu;

```

Pokud se v definici typu souboru vynechá typ složek (viz TSoubor), je soubor definován bez typu. Takto definované soubory lze použít při vstupně-výstupních operacích na nejnižší úrovni. Lze například číst hodnoty ze souboru bez ohledu na jeho vnitřní formát.

Výrazy představují zápisy posloupnosti operací. Obsahují operandy (konstanty, proměnné a funkce), operátory a okrouhlé závorky. Provedení všech operací je vyhodnocení výrazu, jeho výsledkem je hodnota. Operátory mají různou prioritu a podle počtu operandů je dělíme na unární a binární

Přehled operátorů a relací:

Operátor či relace		Operátor či relace	
přiřazení	:=	sčítání	+
rovnost	=	odčítání	-
nerovnost	<>	násobení	*
menší nebo rovno	<=	reálné dělení	/
větší nebo rovno	>=	celočíslné dělení	div
a	and	zbytek po dělení	mod
nebo	or	následovník	inc
negace	not	předchůdce	dec

Procedury a funkce: systematický přístup k algoritmizaci složitějších problémů spočívá v jeho rozkladu na problémy dílčí. Ve vyšším programovacím jazyku lze tyto dílčí problémy řešit procedurami a funkcemi. Jako příklad uveďme dílčí problém výpočtu souřadnic bodu, ležícího na rovinné křivce zadané parametrickými rovnicemi:

```

Procedure ParamCurve(t:Real;var Q:TPoint);
begin
      Q[1]:=3*sin(5*t);
      Q[2]:=5*cos(3*t);
end;

```

Parametr t zde představuje proměnnou, která je volána **hodnotou**, parametr Q je formální parametr volaný **odkazem**. Pomocí parametrů volaných hodnotou nejčastěji reprezentujeme vstupní hodnoty, parametry volané odkazem lze použít jako výstup z procedury. Funkce je speciálním případem procedury, která má jedinou výstupní hodnotu:

```

Function HarmonicOscill(t:Real):Real;
begin
      HarmonicOscill:=exp(-0.1*t)*sin(5*t);
end;

```

Spolu s předdefinovanými typy lze volat také řadu užitečných předdefinovaných funkcí:

Skupina funkcí či procedur	Název <i>(f)</i> funkce <i>(p)</i> procedura	Popis
Pro ordinální typy	<i>(p)</i> dec	sníží hodnotu parametru o jedničku
	<i>(p)</i> inc	zvýší hodnotu parametru o jedničku
	<i>(f)</i> odd	zjišťuje, zda parametr je lichý
	<i>(f)</i> pred	vrací hodnotu předchůdce
	<i>(f)</i> succ	vrací hodnotu následovníka
Pro konverzi typů	<i>(f)</i> round	zaokrouhlí reálnou hodnotu na celočíselnou
	<i>(f)</i> trunc	zkrátí reálnou hodnotu na celočíselnou
	<i>(f)</i> IntToStr	převede ordinální typ na řetězec
	<i>(f)</i> StrToInt	převede řetězec na ordinální typ
	<i>(f)</i> FloatToStr	převede reálný typ na řetězec
	<i>(f)</i> StrToFloat	převede řetězec na reálný typ
	<i>(p)</i> val	převede řetězec na reálný či ordinální typ
matematické funkce:		abs, arctan, exp, cos, frac, ln, random, sin, sqr, sqrt,

Znak *(p)* resp *(f)* před názvem udává, zda se jedná o proceduru či funkci, neboť zacházení s těmito rutinami se liší (viz dále).

Příkazy popisují jednotlivé akce programu a jejich návaznosti. Příkazy dělíme na **jednoduché** (přiřazovací příkaz, příkaz volání procedury nebo funkce) a **strukturované** (složený příkaz, podmíněný příkaz, příkaz cyklu).

Přiřazovací příkaz mění hodnotu proměnné. Na levé straně je proměnná, jejíž hodnota se mění, na pravé pak proměnná nebo výraz, jehož hodnota má být přiřazena, tj. např. $z := x + 2 * y$, $a := b$. Součástí výrazu může být i volání funkce. Výsledek tohoto volání lze např. přiřadit, procedura takto však použít nelze, neboť ta „přiřazuje“ automaticky do parametrů volaných odkazem. Chceme-li např. do proměnné j typu integer uložit hodnotu i sníženou o jedničku a přitom hodnotu i zachovat, je třeba psát $j := \text{pred}(i)$. Chceme-li touto hodnotou původní hodnotu přepsat, můžeme opět použít funkci $i := \text{pred}(i)$; je však lépe použít **příkaz procedury** a psát jen $\text{dec}(i)$. Těmito příkazy voláme procedury, které ukládají výstupní hodnoty doproměnných volaných odkazem (viz výše uvedená procedura ParamCurve).

Příkaz volání procedury aktivuje proceduru určenou jejím identifikátorem. Pokud jsou v příkazu procedury uvedeny parametry, musí korespondovat s parametry v deklaraci procedury, např:

```
t:=0.2;
ParamCurve(t,Q);
```

Z deklarace procedury ParamCurve (viz předchozí strana) je zřejmé, že proměnná t je volána hodnotou a Q odkazem. Bod Q tak bude naplněn souřadnicemi pro aktuální nastavenou hodnotu parametru $t=0.2$.

Příkaz volání funkce: Na rozdíl od volání procedury získáme voláním funkce hodnotu, která musí být přiřazena (funkce vrací hodnotu). Typ proměnné, do které přiřazujeme, musí být kompatibilní s typem, udaným v deklaraci funkce, např:

```
t:=0.2;
y:=HarmonicOscillator(t);
```

– pro nastavenou hodnotu $t=0.2$ vrátí funkce `HarmonicOscillator` reálnou hodnotu, kterou přiřazujeme proměnné `y`.

Strukturované příkazy: Složený příkaz, podmíněný příkaz, příkaz cyklu tvoří konstrukce jiných příkazů, které se provádějí sekvenčně tak, jak jsou ve struktuře zapsány. Konstrukce příkazů ohraničují klíčová slova **begin** a **end**, jednotlivé příkazy se oddělují středníky. Tyto příkazy probereme průběžně na konkrétních příkladech.

3. Struktura programu

{do těchto závorek lze psát poznámky, které budou překladačem ignorovány}
(také sem lze psát poznámky, které budou překladačem ignorovány,*
*navíc lze takto "zabalit" {více} {množinových} {závorek} *)*
// po tomto dvojlomítku je překladačem ignorován řádek až do jeho konce

```
Program Struktura;
{odkazy na samostatné programové jednotky}
uses Knihovna1, Knihovna2;
{globální deklarace programu}
type .....;
const.....;
var ...a,b,.....;
```

```
procedure Global1;
{lokální deklarace}
type .....;
const.....;
var ...c,d, .....;
```

```
{procedura vložená do procedury Global1}
Procedure Lokal;
{lokální deklarace}
type .....;
const.....;
var ...e,f, .....;
begin
{tělo procedury Lokal. Zde jsou použitelné lokální proměnné e, f a globální
proměnné a,b, nepoužitelné lokální c,d. Nelze volat ani Global1, ani Global2}
end;
begin
{tělo procedury Global1. Zde jsou použitelné lokální proměnné c,d a
globální proměnné a,b

Nepoužitelné lokální e,f. Nelze volat Global2, lze volat Lokal}
end;
```

```

procedure Global2;
{lokální deklarace}
type .....;
const .....;
var ...g,h, .....;
  begin
    {tělo procedury Global2. Zde jsou použitelné lokální proměnné g,h a globální
    proměnné a,b. Nepoužitelné lokální c,d,e,f. Lze volat Global1, nelze volat Lokal}
  end;

begin
{Tělo programu. Zde jsou použitelné pouze globální deklarace. Lze volat Global1, Global2,
nelze volat Lokal}
end;

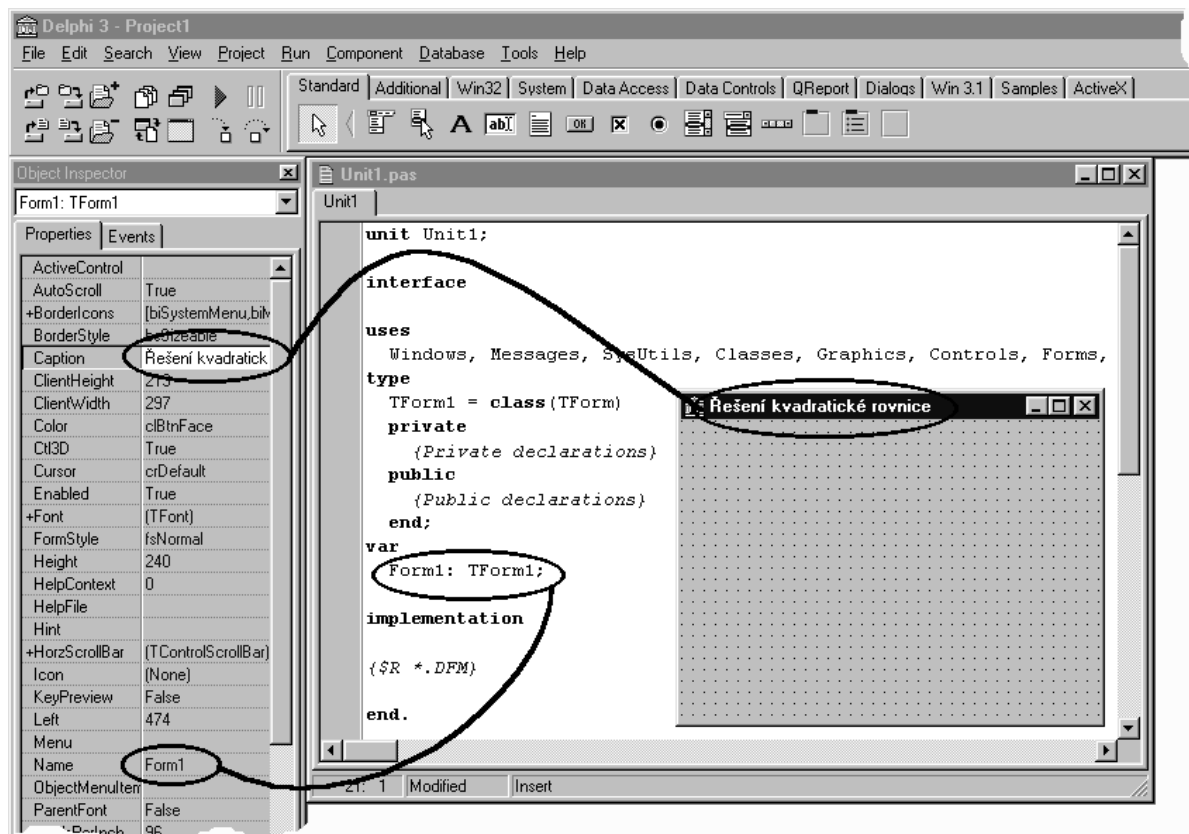
```

Pascalovský program obsahuje blok deklarací typů, konstant a proměnných, dále blok deklarací procedur a funkcí a tělo programu. V bloku procedury nebo funkce mohou být deklarovány lokální typy, konstanty a proměnné, ale též lokální procedury a funkce. Úroveň vnoření těchto deklarací není omezena. Jak uvidíme dále, v Delphi nebudeme prakticky vůbec zasahovat do programu, budeme „pouze“ doplňovat procedury v jistých knihovnách, výše naznačenou strukturu však zde můžeme uplatnit. Každá procedura či funkce v Pascalu může mít totiž obecně stejnou strukturu, jako celý program s jedinou výjimkou - uvnitř procedur nemůžeme použít seznamy knihoven, tj. klíčové slovo *uses*.

4. Prostředí Borland Delphi

Delphi je vývojový prostředek, který v sobě spojuje silnou sadu vizuálních nástrojů pro tvorbu jednotlivých částí aplikace s robustními programovými nástroji a výkonným kompilátorem. V prostředí Delphi vytváříme **projekt** a v něm jednoduchým způsobem **objekty** (formuláře, okna, tlačítka apod.), které budou reagovat na **události** (stisk klávesy, kliknutí nebo tažení myši apod). Na připojeném obrázku (viz následující strana) je prostředí vytvořené při spuštění Delphi. Vidíme zde menu, pod ním ikony spojené s často používanými operacemi (otevření souboru, přenos na disketu...). Následuje vícestránková paleta komponent (Standard, Additional...), s jejíž pomocí lze do formuláře dodávat komponenty (na obrázku je z úsporných důvodů lišta zkrácena). Dále by se mělo objevit okno **Object Inspector**. Pokud tomu tak není, otevřeme jej z menu **View/Object Inspector**. Konečně je to samotný formulář. Formulář je objekt, jehož vlastnosti lze prohlížet a také měnit v Object Inspectoru. V příkladu na připojeném obrázku je změněn titulek formuláře – vlastnost **Caption** (implicitně Form1) a dále rozměry. Ty lze měnit tažením okrajů pomocí myši, nebo zadat číselně (vlastnosti **ClientHeight**, **-Width**). Udělali jsme zatím velmi málo práce, zatím jsme nic neprogramovali, ale přesto máme plně funkční aplikaci. Spustíme-li nyní program (tlačítkem s modrou šipkou nebo z menu **Run/Run**, objeví se náš (zatím prázdný) formulář, se kterým můžeme zacházet jako s každým objektem v prostředí Windows (přesouvat, měnit velikost „taháním“ za okraje, minimalizovat, maximalizovat, zavřít). Všimněte si: Změnili jsme titulek formuláře, nezměnili jsme však jeho jméno pro program, která je nám zatím skryt. Jméno je stále **Form1** a formulář (jako proměnná) je typu **TForm1**, jak si můžeme přečíst v záhlaví Object Inspektoru. Jméno proměnné můžeme změnit pomocí **Name**, není to však zatím nutné a nebudeme to dělat.

Čas, po který na naši aplikaci pracujeme, lze rozdělit do dvou skupin. První je **design time** („vývojový čas“) – je to čas, kdy naši **aplikaci vytváříme** (dodáváme komponenty na formulář, měníme jejich vlastnosti, vypisujeme procedury) a aplikace **není spuštěna**. Za druhé je to **run time** („čas běžící aplikace“), tj. čas, kdy **aplikace je spuštěna** a kdy testujeme její chování. V této době není možné naši aplikaci rozumně modifikovat. Za běhu aplikace není k dispozici Object inspector, formuláři nelze dodávat ani odebírat komponenty či modifikovat jejich vlastnosti (ledaže by tyto operace prováděla sama běžící aplikace, což



možné je). Pro potřeby ladění jsou za běhu programu k dispozici zdrojové soubory a je možné do nich zasahovat. Tyto zásahy však nelze doporučit – pak totiž zdrojový kód jednak neodpovídá běžící aplikaci, což znemožňuje korektní práci debuggeru a za druhé se takto provedené změny projeví stejně až po ukončení běhu programu a nové kompilaci.

Naši běžící aplikaci můžeme ukončit dvojím způsobem: jednak **korektně** – tj. spuštěním příslušné procedury, kterou jsme v aplikaci pro ukončení naprogramovali, jednak **resetem** (položka menu **Run/Program reset** v Delphi). Tímto způsobem lze aplikaci ukončit kdykoli – např. pokud se vinou chyby v programu dostala do nekonečného cyklu a není možné ji ukončit korektně.

Ukončíme tedy náš běh naší aplikace (naše jednoduchá aplikace se korektně ukončí kliknutím na tlačítko, standardně uzavírající okno ve Windows) a vraťme se do design time. Odsuňme formulář poněkud stranou. Pod ním objevíme editovací okno, ve kterém je (zatím prázdná) knihovna s názvem **Unit1.pas**. Má část **interface** (rozhraní), která umožňuje komunikaci s dalšími částmi programu. Jsou zde odkazy na další knihovny (**uses ...**) a informativní deklarace našeho (zatím prázdného) formuláře. Část **implementation** je také prázdná a je určena pro definiční deklarace. Z menu **View\Units** lze také zobrazit hlavní program, který má jméno **Project1.dpr**:

```

program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

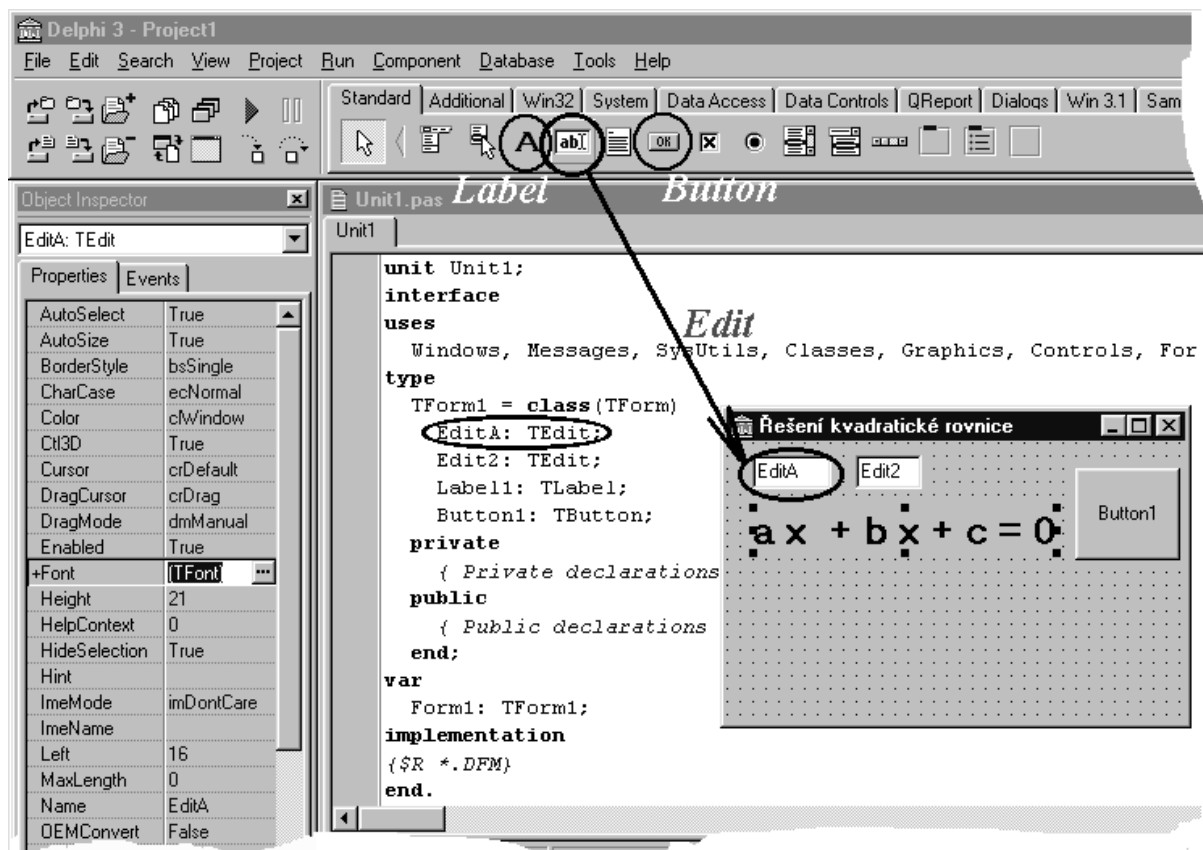
```

Bez podrobnějších znalostí však do něj není radno zasahovat.

Vývoj programu sestává jednak z tvorby uživatelského rozhraní a jednak z prací na jeho funkčnosti. Uživatelské rozhraní je prostředek, který umožňuje vzájemnou komunikaci mezi programem a uživatelem. Uživatelská rozhraní mohou mít různou formu a složitost, programy mohou fungovat nejrozličnějším způsobem. Každý program obecně však musí nějakým způsobem získat data, zpracovat je a výsledek tohoto zpracování poskytnout uživateli. Vše si ukážeme na jednoduchém příkladu řešení kvadratické rovnice.

Příklad 1: Sestavme program na řešení kvadratické rovnice. Nejdříve navrhne uživatelské rozhraní. Sestavíme ho z objektů **Edit**, **Label** a **Button** v liště **Standard**, které umístíme na formulář. Výběr provedeme myší a „natažením“ na formulář. Na obrázku (viz následující strana) vidíme dvě natažená editovací okna (objekt **Edit**), do kterých bude uživatel zadávat koeficienty kvadratické rovnice $ax^2 + bx + c = 0$ (na připojeném obrázku chybí ještě třetí). Okna mají implicitní jména Edit1, Edit2 atd. Zde je již první přejmenováno pomocí vlastnosti **Name** v Object Inspectoru na **EditA**, další dvě doporučujeme přejmenovat na EditB, EditC. Text obsažený v těchto oknech určuje vlastnost **Text**. Doporučujeme ji v Objekt Inspektoru změnit tak, aby po spuštění programu byla programu zadána korektní úloha. Podobně je na formulář natažen objekt **Button**. Jmenuje se **Button1** (**Name**) a má nápis **Button1** (**Caption**). Nápis doporučujeme změnit tak, aby informoval uživatele o funkci tlačítka (např. **Výpočet**). Symbolický zápis kvadratické rovnice dodáme na formulář pomocí objektu **Label** (chybějící exponent je třeba doplnit dalším objektem Label s vhodně změněnou velikostí a umístěním). Všimněte si, že pro všechny objekty, které takto dodáváme na formulář, jsou do zdrojového kódu automaticky „dodávány“ jejich informativní deklarace. V případě potřeby jsou interaktivně měněna jejich jména (viz změnu z Edit1 na EditA). Tyto změny provádí Delphi automaticky a neznalému uživateli nedoporučujeme do těchto změn zasahovat.

Na dalším obrázku je pak kompletní rozhraní pro náš program. Editovací okna pro vstup mají jména EditA, EditB a EditC, pro reálný výstup EditX1, EditX1, pro komplexní výstup pak EditRe1, EditIm1, EditRe2, EditIm2. Všechny texty jsou dodány pomocí objektů Label. Mají pouze informativní charakter, v programu se na ně nebudeme odvolávat, proto jsme je nepřejmenovávali.



V Object Inspectoru jsme zatím měnili pouze vlastnosti – **Properties**. Jsou zde však i události – **Events**. Klepneme-li na ně, můžeme zadávat akce, kterými má program (přesněji řečeno jednotlivé objekty) reagovat na příslušné události. Tímto způsobem se postaráme o funkčnost programu. Z obrázku je patrné, že objektu **Button1** jsme zadali akci **Reseni** jako reakci na událost **OnClick** (tj. zřejmě zmáčknutí knoflíku). Pozor – jméno **Reseni** chápe program jako identifikátor (v tomto případě jméno procedury). Nelze tedy použít diakritiku, mezery apod. Protože tuto proceduru náš projekt zatím nezná, reaguje Delphi tím, že ji založí.

V této chvíli Delphi požaduje kód procedury **Reseni**. Tato procedura může mít, jak již bylo řečeno, obecně stejnou strukturu, jako celý Pascalovský program a může vypadat třeba takto:

```

procedure TForm1.Reseni(Sender: TObject);
{** deklarace proměnných **}
var    a,b,c,      {koeficienty kvadr. rovnice}
        d,          {diskriminant}
        X1,X2,      {reálné řešení}
        XRe,XIm     {komplexní řešení}:      Double;
        Kod         {kód pro chybové hlášení}: Integer;
        TextString   {převodní řetězec}:      String;
{**** tělo procedury řešení ****}
begin

```



```

{procedura Val(S:string;var x:Double;var ErrorCode) převádí řetězec na číselnou hodnotu.
    Proběhne-li převod bez chyby, je ErrorCode=0}
    {okno EditA je objekt, jeho obsah - text je vlastnost, na kterou se odvoláme přes tečku}
    Val(EditA.Text,a,Kod);
    {při chybném převodu ohlásí chybu a ukončí proceduru}
    if Kod>0 then begin ShowMessage('Chybné a');exit;end;
    Val(EditB.Text,b,Kod); if Kod>0 then begin ShowMessage('Chybné b');exit;end;
    Val(EditC.Text,c,Kod);if Kod>0 then begin ShowMessage('Chybné c');exit;end;
    d:=b*b-4*a*c;
    if d>=0 then {je-li diskriminant nezáporný}
    begin {složený příkaz pro výpočet reálných kořenů}
        X1:=(-b+sqrt(d))/2/a; X2:=(-b-sqrt(d))/2/a; {vypočítej reálné kořeny}
        {procedura Str převede číselnou hodnotu na řetězec, parametry u čísla specifikují
            počet cifer - celkem pět toho dvě desetinné}
        Str(X1:5:2,TextString);
        EditX1.Text:=TextString; {výpis výsledku do příslušného okna}
        Str(X2:5:2,TextString);EditX2.Text:=TextString;
        EditRe1.Text:="";EditIm1.Text:=""; {vymazání oken určených pro komplexní kořeny}
        EditRe2.Text:="";EditIm2.Text:="";
    end {konec složeného příkazu pro nezáporný diskriminant}
    else begin {složený příkaz pro záporný diskriminant}
        XRe:=-b/2/a;XIm:= sqrt(-d)/2/a; {výpočet reálné a imagiární složky kořenů}
        Str(XRe:5:2,TextString); {výpisy do příslušných oken}
        EditRe1.Text:=TextString;EditRe2.Text:=TextString;
        Str(XIm:5:2,TextString);
        EditIm1.Text:=TextString;EditIm2.Text:=TextString;
        EditX1.Text:="";EditX2.Text:=""; {vymazání oken určených pro reálné kořeny}
    end {konec složeného příkazu pro záporný diskriminant}
end; {**** konec procedury řešení ****}

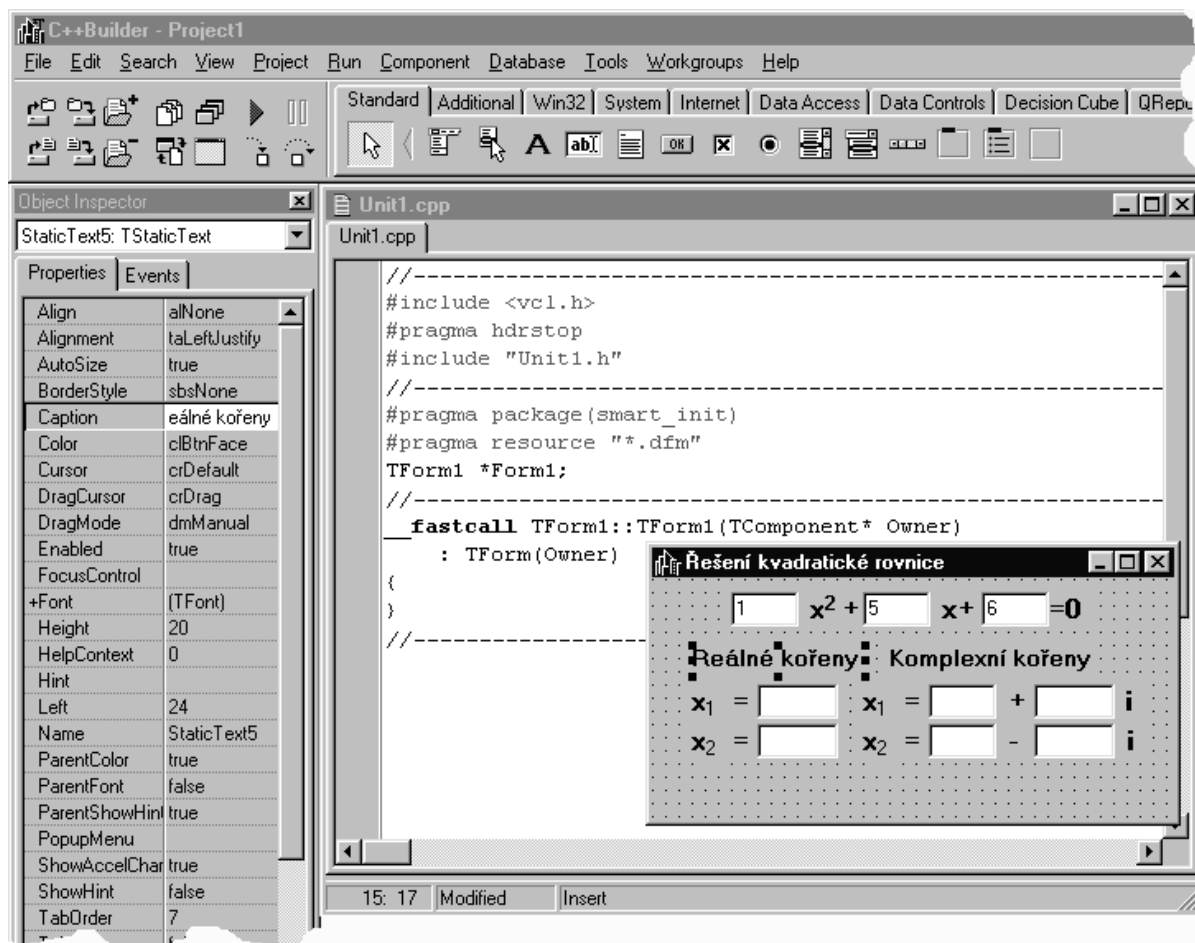
```

Modifikace příkladu: Editovací okna EditA,EditB,EditC lze do formuláře umístit přímo místo koeficientů, proceduru **Reseni** pak lze spojit nikoli s událostí onClick na Button1, ale např. onChange na tato editovací okna. Řešení se pak provede okamžitě při libovolné změně zadání a objekt Button je pak zbytečný. To však nelze doporučit při složitějších výpočtech, které vyžadují delší čas.

Kvadratická rovnice: [Zdrojový kód](#)

[Spustitelný kód](#)

Než opustíme tento příklad, učinme ještě poznámku o prostředí **Borland C++ Builder**. Toto prostředí vypadá, jak je patrné z obrázku na následující straně, naprosto stejně. Stejný Object inspector, stejné zacházení s objekty. Mezi prostředím Delphi a C++ lze dokonce bez obtíží objekty přenášet přes Clipboard. Unita má nicméně jinou příponu, předpřipravený kód je v C++, také uživatelské procedury je třeba vyplňovat v C++. Níže uvedená procedura funguje stejně, jako výše uvedená pascalovská, a je dobrou příležitostí k demonstraci faktu, že práce v tomto prostředí je velmi obdobná:



// místo pro poznámky je uvozeno dvojíým lomítkem

```
void __fastcall TForm1::ReseniKvadratickeRovnice(TObject *Sender)
{ // pascalovské begin zde zastupuje levá množinová závorka
  // deklarace
  double a,b,c,d,X1,X2,XRe,XIm; //typ předchází jménům proměnných
  // převod řetězce na číslo zde zařizuje funkce StrToFloat, k vlastnostem objektů
  // přistupujeme přes ->
  a=StrToFloat(EditA->Text); b=StrToFloat(EditB->Text); c=StrToFloat(EditC->Text);
  d=b*b-4*a*c; // přiřazovacím příkazem je jen rovnítko,
               // relace rovnost má rovnítko dvojité, tj. ==
  if (d>=0)    //zde "chybí" pascalovské then
  { X1=(-b+sqrt(d))/2/a; X2=(-b-sqrt(d))/2/a; // nezáporný diskriminant
    EditX1->Text=FloatToStr(X1);EditX2->Text=FloatToStr(X2);
    // řetězcové konstanty jsou v úvozovkách, nikoli v apostrofech
    EditRe1->Text="";EditIm1->Text=""; EditRe2->Text="";EditIm2->Text="";
  } //pascalovské end zastupuje pravá množinová závorka
  else
  { XRe=-b/2/a;XIm= sqrt(-d)/2/a; //záporný diskriminant
    EditRe1->Text=FloatToStr(XRe);EditIm1->Text=FloatToStr(XIm);
    EditRe2->Text=FloatToStr(XRe);EditIm2->Text=FloatToStr(XIm);
    EditX1->Text="";EditX2->Text="";
  } //konec větve pro záporný diskriminant
} // konec procedury ReseniKvadratickeRovnice
```

5. Strukturované příkazy

Strukturované příkazy tvoří, jak již bylo řečeno, konstrukce jiných příkazů, které se provádějí sekvenčně tak, jak jsou ve struktuře zapsány. Konstrukce příkazů ohraničují klíčová slova **begin** a **end**, jednotlivé příkazy se oddělují středníky.

V příkladu na řešení kvadratické rovnice jsme použili dva často se vyskytující strukturované příkazy – složený příkaz a příkaz podmíněný.

Složený příkaz tvoří konstrukce obsahující další příkazy, které se mají provádět sekvenčně, a to v posloupnosti, v jaké jsou zapsány. Složený příkaz je ohraničen klíčovými slovy **begin** a **end**, jednotlivé příkazy v něm jsou vzájemně odděleny středníky. Složené příkazy jsme již použili v příkladu na řešení kvadratické rovnice v předchozí kapitole.

Další skupinou strukturovaných příkazů jsou **příkazy podmíněné**, které k provedení vybírají pouze jeden (nebo také žádný) ze svých vnitřních příkazů. Do této kategorie patří příkaz **if** a **case**.

Příkaz **if** již známe z předchozí kapitoly. Jeho obecné schéma je

```
if <výraz> then <příkaz1> else <příkaz2>;
```

Výraz musí být booleovského typu. Je-li **True**, provede se výraz, který následuje za **then** (příkaz1), je-li **False**, provádí se alternativa za **else** (příkaz2). Příkazy mohou být i složené tak, jako v našem předcházejícím příkladu. Jestliže podmíněný příkaz nemá alternativu **else**, provádí se příkaz bezprostředně následující za strukturovaným příkazem **if**. Příkazy **if** lze do sebe vnořovat, např:

```
if <výraz1> then
    if <výraz2> then <příkaz1> else <příkaz2>;
```

V tom případě obecně platí, že **else** je alternativou k nejbližšímu předcházejícímu **then**, které by nebylo sdruženo s žádným **else**.

Příkaz Case zajišťuje vícenásobné větvení programu a je vlastně zkratkou za mnohonásobné použití příkazu **if**:

```
Case <výraz - selektor> of
    hodnota_1 : <příkaz1>;
    hodnota_2 : <příkaz2>;
    .....
    hodnota_n : <příkaz1>;
end
else <příkaz>;
```

Skládá se z výrazu – tzv. selektoru a seznamu příkazů. Selektor musí být ordinálního typu. Každý z příkazů je opatřen jednou nebo několika hodnotami. Příkazy mohou být jednoduché nebo složené, žádná z hodnot v seznamu se nesmí opakovat. Ordinální hodnoty selektoru musí být maximálně v rozsahu –23768..32768. Program provede příkaz, který je uveden za aktuální hodnotou selektoru. Na místě hodnot mohou být uvedeny i intervaly. V tom případě

se provede příkaz, který je uveden za intervalem do něhož náleží aktuální hodnota selektoru. Alternativa **else** je volitelnou částí příkazu. Pokud selektor nenabývá žádné z uvedených hodnot, provede se alternativa za else. Jestliže alternativa chybí, přikročí se k dalšímu příkazu.

Příklad 1: Sestavme program, který bude simulovat hod hrací kostkou.

Uživatelské rozhraní bude velmi jednoduché a sestojíme ho dle připojeného obrázku. Hod proběhne zmáčknutím knoflíku, jehož titulek byl změněn na Házej. Bílý čtverec - stylizovaná kostka je vytvořen z objektu typu TPanel v liště Standard. Jeho titulek byl vymazán nastavením vlastnosti Caption na prázdný řetězec a jeho barva nastavena na bílou (vlastnost Color nastavována na clWhite). Na tomto panelu jsou umístěny pod sebou tři objekty Static text,



které najdeme v liště Additional. Jejich titulek byl rovněž vymazán podobně jako titulek panelu. Tuto vlastnost však budeme v programu měnit. Pomocí těchto tří objektů totiž budeme na čtverec umisťovat hvězdičky – body kostky podle čísla, které má padnout. Výsledek hodu jsou čísla od jedné do šesti. V proceduře je tedy deklarován typ TVysledekHodu jako tento interval a proměnná VysledekHodu tohoto typu. Výsledek je v proceduře nastavován funkcí Random, která vrací náhodné číslo z intervalu $(0;1)$. Vracená hodnota tedy musí být vynásobena šesti, přičtena jednička a výsledek nakonec zkrácen na celé číslo pomocí funkce Trunc. Tato hodnota pak slouží jako selektor pro příkaz Case, jehož pomocí jsou sestrojovány jednotlivé případy.

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

```
Type TVysledekHodu = 1..6;
```

```
var VysledekHodu:TVysledekHodu;
```

```
begin
```

```
  VysledekHodu:=Trunc(6*Random+1);
```

```
  Case VysledekHodu of
```

```
    1:begin
```

```
      StaticText1.Caption:='';
```

```
      StaticText2.Caption:=' *';
```

```
      StaticText3.Caption:='';
```

```
    end;
```

```
    2:begin
```

```
      StaticText1.Caption:=' *';
```

```
      StaticText2.Caption:='';
```

```
      StaticText3.Caption:=' *';
```

```
    end;
```

```
    3:begin
```

```
      StaticText1.Caption:=' *';
```

```
      StaticText2.Caption:=' *';
```

```
      StaticText3.Caption:=' *';
```

```
  end;
```

```
4:begin
```

```
  StaticText1.Caption:=' * *';
```

```
  StaticText2.Caption:='';
```

```
  StaticText3.Caption:=' * *';
```

```
end;
```

```
5:begin
```

```
  StaticText1.Caption:=' * *';
```

```
  StaticText2.Caption:=' *';
```

```
  StaticText3.Caption:=' * *';
```

```
end;
```

```
6:begin
```

```
  StaticText1.Caption:=' * *';
```

```
  StaticText2.Caption:=' * *';
```

```
  StaticText3.Caption:=' * *';
```

```
end;
```

```
end;
```

```
end;
```

Hrací kostka:

[Zdrojový kód](#)

[Spustitelný kód](#)