EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Vysoká škola báňská – Technická univerzita Ostrava

POČÍTAČOVÁ TECHNIKA II

učební text

Milan Heger, Pavel Švec

Ostrava 2012

Recenze: Ing. Jiří Franz, Ph.D.
RNDr. Miroslav Liška, CSc.

Název: Počítačová technika II
Autor: M. Heger – P. Švec
Vydání: první, 2012
Počet stran: 150
Náklad: 20

Studijní materiály pro studijní obory B3922. Ekonomika a management v průmyslu, FMMI a B3922 Management jakosti, fakulty FMMI.

Jazyková korektura: nebyla provedena.

Určeno pro projekt:

Operační program Vzděláváním pro konkurenceschopnost
Název: Personalizace výuky prostřednictvím e-learningu
Číslo: CZ.1.07/2.2.00/07.0339
Realizace: VŠB – Technická univerzita Ostrava
Projekt je spolufinancován z prostředků ESF a státního rozpočtu ČR

ISBN 978-80-248-2560-1

OBSAH

PŘEDMLUVA

Učební podpora, kterou právě držíte v ruce, je cíleně zaměřena na výuku algoritmizace a programování pro studenty těch studijních oborů, které nejsou specializované na výuku informačních technologií. Primárním cílem tedy nebude vychovat studenta jako programátora, ale snaha o prohloubení analytického a logického myšlení, snaha o dosažení takových dovedností v programování, aby mohly být úspěšně a efektivně využity při řešení složitých problémů spojených se studiem odborných předmětů a závěrečných prací.

Tomuto cíli je podřízen obsah a forma výuky, která vznikla na základě mnohaletých pedagogických zkušeností, a která rovněž vychází z analýzy odezvy studentů. Svou podstatou se poněkud liší od formy běžných učebnic programování a je hlavně zaměřena na získání praktických dovedností.

V každé kapitole je nejprve probíraná tématika osvětlena populární formou a teprve postupně jsou nabyté vědomosti rozšiřovány o další a další teoretické základy.

Významná část probírané látky je doplněna výukovými programy, které by na základě řízené animované výuky měly vést k snadnému pochopení jednotlivých postupů při programování. Snahou autorů dokonce bylo, aby základní problematika programování mohla být pochopena již po práci s těmito programy.

Přečtením této podpory (stejně tak jako přečtením jakékoliv jiné učebnice o programování) sice student získá určité povědomí o dané problematice, ale programovat se určitě nenaučí. Dovednost napsat funkční a smysluplný program je nerozlučně spjata s mnoha hodinami strávenými před počítačem. Dobře programovat se dá naučit jen neustálým programováním. A zde si je třeba také uvědomit, že programování většinou není jen napsání několika naučených příkazů tak, aby jim počítač rozuměl a následně je vykonal, ale je nezbytně nutné provést co nejkvalitnější analýzu problému, který má být počítačem řešen a následně vymyslet postup, jak problém efektivně vyřešit.

A nyní jedno soukromé, ale velmi významné doporučení:

Mnozí z vás, kteří teď stojíte před úkolem studijně zvládnout předmět s tématikou, s kterou jste se v životě nesetkali, a která v mnohých dokonce vzbuzuje obavy si teď řeknou - to je konec. Ale není to pravda, je to naopak začátek, který můžete snadno využít k rozvoji vašeho kreativního myšlení a logického úsudku. Bude lépe, když budete programování brát jako složitější logickou hru a budete z ní mít radost a zábavu. Vždyť kdo vás bude ještě někdy v životě tak bezmezně poslouchat jako počítač?

POKYNY KE STUDIU

Počítačová technika II

Pro předmět 3. semestru oboru B3922. Ekonomika a management v průmyslu, FMMI a B3922 Management jakosti, FMMI jste obdrželi studijní balík obsahující:

- integrované skriptum pro distanční studium, které obsahuje i pokyny ke studiu
- CD-ROM s doplňkovými animačními výukovými programy vybraných částí kapitol

Prerekvizity

Pro studium tohoto předmětu se předpokládá absolvování předmětu Počítačová technika I.

Cílem předmětu

je seznámení se základními pojmy algoritmizace a programování, jejichž praktické využití povede k získání specifických návyků nutných pro přístupy k algoritmizaci.

Po prostudování modulu by měl student být schopen prakticky realizovat jednoduché i složitější programové aplikace zaměřené na technickou praxi. Studenti budou umět analyzovat technický problém z hlediska algoritmizace. Budou znát problematiku programování v programovacím jazyku Borland® Delphi™ 2005.

Pro koho je předmět určen

Modul je přednostně zařazen do bakalářského studia oboru B3922. Ekonomika a management v průmyslu, FMMI a B3922 Management jakosti, ale může jej studovat i zájemce z kteréhokoliv jiného oboru.

Skriptum se dělí na části, kapitoly, které odpovídají logickému dělení studované látky, ale nejsou stejně obsáhlé. Předpokládaná doba ke studiu kapitoly se může výrazně lišit, proto jsou velké kapitoly děleny dále na číslované podkapitoly a těm odpovídá níže popsaná struktura.

Při studiu každé kapitoly doporučujeme následující postup:

- Pozorně přečíst teoretickou část kapitoly.
- Ihned si na počítači vyzkoušet všechny, byť jen dílčí příklady.
- Vytvořit všechny programy, které jsou v zadání úloh k řešení a snažit se je tvůrčím způsobem modifikovat.

K orientaci v textu vám mohou sloužit následující ikony:

Čas ke studiu: xx hodin

Na úvod kapitoly je uveden čas potřebný k prostudování látky. Čas je orientační a může vám sloužit jako hrubé vodítko pro rozvržení studia celého předmětu či kapitoly. Čas strávený nad každou kapitolou bude značně závislý na množství příkladů, které budete řešit samostatně na počítači a hloubce jejich propracování.

Cíl: Po prostudování tohoto odstavce budete umět

- popsat ...
- definovat ...
- vyřešit ...

Nejprve se seznámíte s cíli, kterých máte dosáhnout po prostudování této kapitoly. Jde o konkrétní dovednosti, znalosti a praktické zkušenosti, které studiem kapitoly získáte.

Výklad

Následuje výklad studované látky, zavedení nových pojmů, jejich vysvětlení, vše doprovázeno obrázky, tabulkami, příklady a odkazy na výukové programy s animacemi.

Shrnutí pojmů

Na závěr kapitoly jsou stručně zopakovány významné pasáže a pojmy, které si máte osvojit.

Otázky

Pro ověření, že jste dobře a úplně látku kapitoly zvládli, máte k dispozici několik teoretických, ale i praktických otázek.

Úlohy k řešení

Protože většina teoretických pojmů tohoto předmětu má bezprostřední význam a využití v programátorské praxi, jsou Vám nakonec předkládány i praktické úlohy k řešení. V nich je hlavní význam předmětu, a to schopnost aplikovat čerstvě nabyté znalosti při řešení reálných situací.

Klíč k řešení

Výsledky zadaných úkolů jsou uvedeny na CD v podobě řešených programů. Používejte je až po vlastním vyřešení úloh, jen tak si samokontrolou ověříte, že jste obsah kapitoly skutečně úplně zvládli.

CD-ROM

Informace o doplňujících výukových programech, si může student vyvolat z CD-ROMu připojeného k tomuto materiálu.

Úspěšné a příjemné studium s touto učebnicí Vám přejí autoři výukového materiálu

Milan Heger a Pavel Švec

1. NA MÍSTO ÚVODU TROCHU MATEMATIKY A LOGIKY

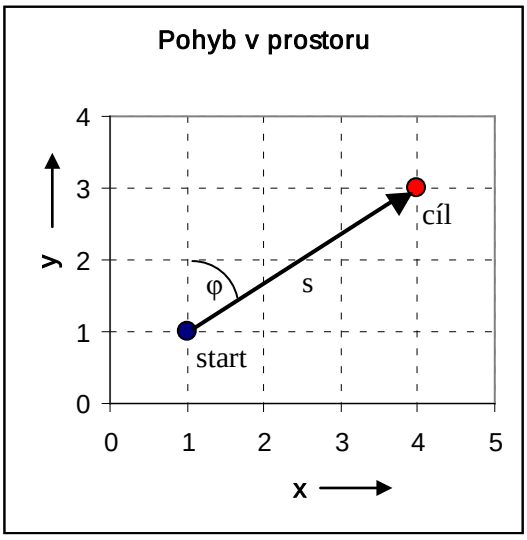
Čas ke studiu: 0.5 hodin

Cíl: Po prostudování tohoto odstavce nebudete sice umět nic víc, než jste uměli, ale získáte názor na to:

- co je to programování.
- co to jsou příkazy v programování.
- jak pomocí omezeného počtu příkazů vyřešit zadanou úlohu.

Výklad

Můžeme si představit, že máme psa. Aby nás poslouchal, potřebujeme ho naučit správně reagovat na určité povely (jako například lehni, sedni, k noze, aport apod.). Nyní dostaneme za úkol pomocí vhodných povelů přesunout psa z jednoho místa, označeného jako „start“, na obecné místo v prostoru, které označíme jako „cíl“ (dle obr. 1.1).



Obr. 1.1 – Schéma ….

Jednoduché by se jevilo použít jen dva povely, které zaručují nejkratší trasu přesunu (pes je v místě startu a je otočen směrem nahoru):

1. otoč se doprava o úhel $\varphi = \arctan\left(\frac{dx}{dy}\right)$, v našem případě: $\varphi = \arctan\left(\frac{3}{2}\right)$

2. projdi vzdálenost $s = \sqrt[2]{y^2 + x^2}$, v našem případě: $s = \sqrt[2]{2^2 + 3^2}$

To však pes a dokonce ani člověk bez přístrojů nedokáže, a proto nám nezbývá, než se snažit dosáhnout cíle jednoduššími, snadno proveditelnými povely. Minimální počet takovýchto povelů, které musíme psa naučit, jsou dva. I když je možností více, použijeme třeba následující (naučíme psa vykonat tyto dva povely):

- *krok vpřed (pes udělá jeden krok)*
- *otoč se doprava (pes se otočí o 90^o doprava)*

Jenže pokus přemístit psa do cíle povely dva kroky vpřed, otoč se doprava a tři kroky vpřed by byl s velkou pravděpodobností neúspěšný. Naproti tomu dospělý člověk by tento úkol dokázal splnit snadno. Pes zde bude mít nepřekonatelný problém s určením počtu kroků.
Spojíme-li inteligenci člověka s precizním provedením dvou, výše uvedených povelů, je možné vhodnou volbou sekvence (sledu) povelů ovládat psa tak, aby bylo cíle úspěšně dosaženo.

Sled prováděných povelů by pak mohl vypadat například takto (viz obr. 1.1):

- krok vpřed
- krok vpřed
- otoč se doprava
- krok vpřed
- krok vpřed
- krok vpřed

} sekvence povelů

Jenže cest ze startu do cíle je velké množství. Jedna je podobná, nejprve se vydat podél osy x a pak podle osy y. Trasy jsou co do délky stejné, nastane však problém s otočením psa do požadovaného směru.

Nabízejí se dvě řešení:

1. naučit psa další povel - „otoč se doleva“, jenže takto by nám pravděpodobně v praxi stále počet povelů přibýval podle toho, jak bychom od psa vyžadovali další a další dovednosti.

- krok vpřed
- krok vpřed
- krok vpřed
- otoč se doleva
- krok vpřed
- krok vpřed

} sekvence povelů

2. zde se držme zcela odůvodněného přísloví „starého psa novým kouskům nenaučíš“ a zkusme striktně použít jen ty dva původní příkazy:

- krok vpřed
- krok vpřed
- krok vpřed
- otoč se doprava
- otoč se doprava
- otoč se doprava
- krok vpřed
- krok vpřed

} sekvence povelů

Cíle je rovněž dosaženo, ovšem za cenu delší sekvence, avšak menšího počtu základních povelů. (Nový povel je zde nahrazen vícenásobným opakováním již známých povelů).

Vyzkoušejte si vytvořit jinou sekvenci povelů v závislosti na jiném rozmístění startovního a cílového bodu v rastru, přičemž respektujte počáteční natočení psa.

Nyní si řeknete: co mají povely pro psa společného s počítači a programováním?
Na první pohled nic. Ve skutečnosti mnoho. Pokud jste pochopili způsob ovládání psa a podařilo se vám vytvořit sekvenci povelů, která dovedla psa k cíli, umíte s trochou nadsázky programovat. Programování není totiž nic jiného, než účelná tvorba sekvence vhodně volených příkazů, kterým vedle vás rozumí i počítač. Určitě z vás v této chvíli ještě nejsou programátoři, po kterých by sáhla každá softwarová firma, ale učinili jste první krok k tomu, abyste byli schopni postupně vytvářet složitější a složitější programy, které vám mohou usnadnit a zkvalitnit vaši odbornou práci, anebo vás jen naučí logicky myslet. I to není málo.

Otázky

1. Kolika způsoby je možno přemístit psa uvedenými povely z místa startu do cíle?
2. Jak byste zde definovali pojem povel?
3. Co si představujete pod pojmem optimalizace postupu řešení?

Úlohy k řešení

1. Vyzkoušejte si vytvořit jinou sekvenci povelů v interaktivní animaci umístěné na CD. Jde o animaci označenou jako ***1a Povely***.
2. Jak by vypadala optimální sekvence příkazů, kdyby byly stávající povely rozšířeny o povel „otoč se o 45° doprava?

2. ALGORITMY A ALGORITMIZACE

2.1. Algoritmus

Čas ke studiu: 1.0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat pojem algoritmus.
- vytvořit jednoduché algoritmy.
- stanovit základní vlastnosti algoritmů.

Výklad

Pokud jste pročetli minulou kapitolu a podařilo se vám úspěšně přemístit psa na základě správné sekvence povelů ze startu do cíle, pak jste vytvořili svůj první počítačový program. Aby se vám to podařilo, museli jste se nejprve seznámit se zadáním úlohy.

Tak vznikl *cíl* úlohy, kterého je třeba dosáhnout. Dále jste museli provést *analýzu* daného problému a teprve na jejím základě vytvořit postup pro dosažení cíle.

Těch postupů Vás mohla napadnout celá řada (v našem případě z předchozí kapitoly jít přímo směrem k cíli, jít nejprve ve směru osy x až do dosažení hodnoty souřadnice x rovné hodnotě souřadnice x cíle, pak postupovat směrem k cíli ve směru osy y až do dosažení cíle, atp.). Volba těch uskutečnitelných postupů je limitována určitými omezujícími podmínkami. Zde se omezíme jen na *prostředky*, které máme k dispozici – počet a význam příkazů (povelů psovi). I po tomto omezení může ještě zůstat celá řada postupů, které vedou k cíli, je proto účelné vybrat takový, který nejlépe splňuje určitá kritéria (nejkratší, nejlevnější, nejspolehlivější postup apod.). Postup se tedy snažíme *optimalizovat*.

Bude-li navržený postup splňovat ještě navíc určité podmínky, které si probereme později, můžeme tento postup odborně nazvat ***algoritmem***.

V tuto chvíli pro naše účely bude dostačující, když budeme pojem ***algoritmus*** chápat jako matematicko-logický postup, který je použit pro řešení zadaného problému.

Algoritmem však nelze nazvat libovolný postup vytvořený pro řešení úkolu. Algoritmus má splňovat i několik podmínek, musí tedy mít své určité specifické vlastnosti.
Pro jednoduchost si můžeme algoritmus představit jako návod (např. i kuchařku) pro realizaci nějakého složitějšího postupu (technologické operace).

Návod za výměnu poškozeného kola automobilu by mohl vypadat například takto:

1. Zajistěte vozidlo proti samovolnému rozjezdu.
2. Povolte montážní matice poškozeného kola.
3. Zvedněte vozidlo na straně poškozeného kola.
4. Vyšroubujte všechny montážní matice poškozeného kola.
5. Poškozené kolo sundejte a nahraďte ho kolem rezervním.
6. Našroubujte montážní matice vyměněného kola.
7. Vozidlo spusťte na zem.
8. Dotáhněte montážní matice vyměněného kola.

Z uvedeného návodu je patrno, že je určen osobám s určitou zkušeností, která má i značné znalosti v oblasti montáží i konstrukce vozidel. Návod je složen jen ze sekvence prostých příkazů, jejichž vykonání předpokládá určitou inteligenci vykonavatele. I to lze považovat za algoritmus, který popisuje problém v určité vyšší hierarchické úrovni a tedy obecnější rovině. Může být použit k základní formulaci postupů, které vedou k vyřešení problému.

Návod za výměnu poškozeného kola automobilu by mohl vypadat například i takto:

1. Vozidlo zajistěte zatažením ruční brzdy a zařazením prvního rychlostního stupně.
2. Jste-li na silnici, pak vyjměte výstražný trojúhelník z jeho místa určeného pro transport a umístěte ho na silnici do jízdního pruhu, ve kterém stojí vaše vozidlo, a to 50 m za vozidlo. Nejste-li na silnici, nechte výstražný trojúhelník na svém místě v zavazadlovém prostoru a pokračujte při montáži bodem 3.
3. Máte-li ve vozidle bezpečnostní klíny, vyjměte dva bezpečnostní klíny z určeného místa. Jeden zasuňte zepředu pod zadní kolo vozidla, a to na té straně vozidla, kde se nenachází poškozené kolo. Druhý zasuňte zezadu pod zadní kolo vozidla, a to na té straně vozidla, kde se nenachází poškozené kolo. Nemáte-li ve vozidle bezpečnostní klíny, pokračujte při montáži přímo bodem 4.

4. Vyjměte klíč z jeho místa určeného pro transport.
5. Klíčem povolte o čtvrt otáčky všechny montážní matice poškozeného kola.
6. Vyjměte zvedák z jeho místa určeného pro transport, položte na zem pod vozidlo, a to na té straně vozidla, kde se nachází poškozené kolo a současně do místa, aby se opěrná část zvedáku při zvedání opřela do místa na karosérii určeného ke zvedání vozidla.
7. Zvedejte vozidlo tak dlouho, dokud se spodní část poškozeného kola nachází méně než 5 cm nad zemí.
8. Vyšroubujte všechny montážní matice poškozeného kola.
9. Poškozené kolo sundejte tahem směrem k sobě.
10. Vyjměte rezervní kolo z jeho místa určeného pro transport a na jeho místo vložte poškozené kolo.
11. Rezervní kolo nasuňte na montážní šrouby, na kterých bylo nasazeno poškozené kolo, a potlačte ho na doraz.
12. Našroubujte montážní matice vyměněného kola tak, aby všechny zapadly do svých lůžek.
13. Klíčem všechny matice dotáhněte.
14. Spusťte zvedák tak, aby vozidlo dosedlo na všechna kola a zvedák se uvolnil.
15. Klíčem dotáhněte všechny matice.
16. Zvedák a klíč ukliďte na jejich místa určená pro transport.
17. Vyndejte oba klíny zajišťující kolo vozidla a ukliďte je na místa určená pro transport.
18. Zůstal-li na místě výstražný trojúhelník, ukliďte ho na jeho místo určené pro transport, nezůstal-li tam, zapamatujte si, že musíte v nejbližším obchodě s autodílny koupit nový a umístit ho na jeho místo určené pro transport.

Tento návod je již rozsáhlejší a podrobnější. Vypadá, že je určen laikovi, který nemá žádné znalosti, jak se poškozené kolo vozidla vyměňuje. Na rozdíl od předchozího návodu obsahuje elementární a hlavně zcela přesné příkazy, které se nedají vysvětlit několika způsoby (vykonavatel se sám nerozhoduje jak operaci vykonat).

Hned příkaz 1. Jednoznačně říká, že je nutno zařadit první rychlostní stupeň a nedává vykonavateli možnost vybrat si jinou rychlost. Stejně dobře by posloužila i zpátečka, ale

příkaz typu zařaď I. rychlostní stupeň nebo zpátečku by stroj nedokázal provést, protože se sám není schopen rozhodnout. Proto *algoritmus musí jednoznačně udávat, co má v dané chvíli vykonavatel provést.*

Příkaz 2. dokonce nabízí dvě varianty postupu, ale rozhodnutí, kterou variantu v dané chvíli použít nenechává na vykonavateli, ale *jednoznačně směr postupu určuje na základě splnění předem definované podmínky*. Podobně příkaz 18., byť vzbuzuje spíše úsměv, je zásadní, protože neopomene ošetřit všechny možné varianty, které mohou během řešení úlohy nastat. Jak by se jinak vypořádal vykonavatel (stroj, počítač) s tím, že má uklidit předmět, který neexistuje. Většina chyb v algoritmech je právě takového charakteru, že při jeho tvorbě nebyly uvažovány veškeré možné varianty, které mohou v reálné praxi nastat. I tomuto podrobnému algoritmu bychom mohli vytknout nějaké maličkosti, které by však v konečném důsledku mohly ohrozit úspěch celého postupu. Ten například předpokládá, že máme s sebou rezervní kolo a to je navíc zcela funkční a nahuštěno na požadovanou hodnotu. Podobně zde není ošetřeno jak postupovat, kdyby nebyl funkční zvedák nebo nebylo možné našimi silami montážní matice povolit. I nenalezení klíče na uvolnění kol na stanoveném místě (byť by se někde ve vozidle nacházel) by mělo zcela fatální následky.

Ty a další požadavky, které jsou kladeny na vlastnosti algoritmů, můžeme nejlépe vysledovat na následujícím (úmyslně špatném) algoritmu.

Návod za výměnu poškozeného kola automobilu (nevhodný jako algoritmus):

1. Vozidlo zajistěte zatažením ruční brzdy a zařazením prvního rychlostního stupně nebo zpátečky.
2. Jste li na silnici, pak vyjměte ze zavazadlového prostoru výstražný trojúhelník a umístěte ho 50 m za vozidlo.
3. Vyjměte dva bezpečnostní klíny z určeného místa a zasuňte je zepředu pod zadní kolo vozidla.
4. Máte nasazené ochranné brýle?
5. Povolte mírně montážní matice poškozeného kola.
6. Vyjměte zvedák z jeho místa určeného pro transport, položte na zem pod vozidlo do místa, aby se opěrná část zvedáku při zvedání opřela do místa na karosérii určeného ke zvedání vozidla.

7. Opatrně zvedejte vozidlo tak dlouho, dokud se poškozené kolo nenachází minimálně 5 cm nad zemí.
8. Vyšroubujte montážní matice poškozeného kola.
9. Poškozené kolo sundejte.
10. Vyjměte rezervní kolo z jeho místa určeného pro transport.
11. Rezervní kolo nasuňte na montážní šrouby, na kterých bylo nasazeno poškozené kolo, a potlačte ho.
12. Našroubujte montážní matice vyměněného kola.
13. Klíčem všechny matice dotáhněte.
14. Spusťte zvedák tak, aby vozidlo dosedlo na všechna kola a zvedák se uvolnil.
15. Klíčem všechny matice dotáhněte nejlépe do kříže.
16. Zvedák a klíč pečlivě uschovejte.
17. Vyndejte oba klíny a ukliďte je na místa určená pro transport.
18. Zůstal-li na místě výstražný trojúhelník, ukliďte ho na místo určené pro transport, nezůstal-li tam, zapamatujte si, že musíte v nejbližším obchodě s autodílny koupit nový a umístit ho na místo určené pro transport.

Jak již bylo řečeno, v příkazu 1. je nevhodně použit výraz „zařazením prvního rychlostního stupně nebo zpátečky“. Správný výraz nesmí nikdy dávat na výběr z více možností.

Příkaz 2. nedává kromě vzdálenosti, žádný bližší údaj o poloze pro umístění výstražného trojúhelníka. Ve správném algoritmu je příkaz doplněn o text „umístěte ho na silnici do jízdního pruhu, ve kterém stojí vaše vozidlo“.

Příkaz 3. nabádá, aby byly „vyjmuty dva bezpečnostní klíny z určeného místa“, aniž jsou v povinné výbavě vozidla, a tedy nemusí se ve vozidle vůbec nacházet. Navíc není určeno jak klíny aplikovat. Pokud bychom dali klíny třeba jen zezadu nebo z boku kola, mohlo by auto po zvednutí popojet a spadnou ze zvedáku.

Příkaz 4. vlastně není žádným příkazem. Je jen otázkou a otázka sama o sobě do algoritmu nepatří.

Příkaz 5. Vypadá logicky správně, ale problém je ve slově „mírně“. Slova jako málo, moc, silně, barevněji apod. nemohou být v algoritmu použita a musí být vyjádřená jednoznačně (číselně). Ve správném algoritmu nalezneme výraz „…povolte o čtvrt otáčky…“. A aby to

nebylo málo, nalezneme tu ještě nepřesnost typu „Povolte … montážní matice …“, ale není zde jasné které matice (všechny nebo jen některé?).
V příkazu 6., byť vypadá zcela korektně, chybí dodat, na které straně vozidla máme zvedák umístit. Jde o podobnou chybu jako v příkazu 2.
I příkaz 7. obsahuje více chyb. Jednak zde již bylo upozorněno na neurčité výrazy a zde se objevuje slovo „opatrně“ které nelze reálně interpretovat. Druhá neurčitost je ve výrazu „…dokud se poškozené kolo nenachází 5 cm nad zemí“. Není zde jasné, která část kola je myšlena. Pokud by byla interpretována jako střed kola, pak pokud by bylo kolo větší než 10 cm, nikdy bychom kolo nevyměnili.
Příkaz 8. rovněž neupřesňuje, o které matice se jedná (jako v příkazu 5.).
Příkaz 9. neurčuje, jakým způsobem kolo sundat.
Příkaz 10. Je formálně v pořádku, ale na rozdíl od formulace ve správném algoritmu se někde ztratilo poškozené kolo. Pokud v následných příkazech nebude tato otázka ošetřena, zůstane kolo na místě opravy.
V příkazu 11. vypadá, že by mohl nastat problém s výrazem „potlačte ho“. Není známo kam a jak silně apod. Ještě horší by byl výraz „tlačte ho“, v tomto případě bychom navíc nevěděli, kdy máme přestat tlačit a to by vedlo k zásadní chybě, když bychom stále tlačili a nedostali se tak už vlastně k žádné další činnosti. Každá činnost tohoto typu musí být limitována splněním určité podmínky (například: tlačte, dokud se kolo po celé ploše nedotkne bubnu kola). Ale i zde musíme být opatrní a aplikovat ještě nějaké jiné podmínky ukončení prováděné činnosti (třeba pro případ, že se nám kolo nepodaří zatlačit do takové polohy, aby byla splněna předchozí podmínka).
Podobné nepřesnosti se nacházejí i ve zbývajících příkazech, ale ty byste již lehce nalezli při srovnání správného a špatného algoritmu.

Možná si řeknete, že jde o malichernosti a zbytečné formality, ale věřte, že Váš názor vyplývá z přirozené lidské vlastnosti automatického využívání zkušeností při činnosti a vlastní kreativity. Prostě myslíte lidsky a je to pochopitelné, jenže zde se budete muset naučit myslet jinak, z pozice stroje perfektně a rychle vykonávajícího jen příkazy. Budete muset být schopni přemýšlet jakoby na nižší intelektuální úrovni.
Podle vámi vytvořeného algoritmu budete muset Vy, nebo někdo jiný, sestavit počítačový program a podle toho programu bude počítač vykonávat jednotlivé operace v přesně určeném

pořadí tak, jak jste mu je zadali a nebude při tom využívat žádné zkušenosti, sám se nebude rozhodovat ani myslet tvůrčím způsobem.

Je to tedy jen na Vás, jak se Vám podaří vystihnout podstatu problému, jak jeho řešení správně a efektivně zakomponovat do algoritmu, jak se dokážete vžít do toho, že Vámi definovanou činnost bude slepě provádět stroj se svými přesně definovanými schopnostmi a jak dokážete odhalit všechna úskalí a možné varianty, které mohou při výsledné aplikaci algoritmu nastat.

Vlastnosti algoritmů

Mnohé vlastnosti algoritmů jsme již odhalili srovnáním správného a špatného algoritmu v předchozí části kapitoly.

Je zřejmé, že algoritmus musí mít definovaný začátek, tedy první příkaz, kterým začíná a také poslední příkaz, kterým je ukončen. Postup od začátku ke konci algoritmu probíhá v logicky daném sledu příkazů, kdy po vykonání jednoho příkazu je *jednoznačně* dán příkaz následující. Provedení jednoho příkazu a přechod na další nazýváme *krokem*.

Vzhledem k tomu, že jsme algoritmus vytvářeli za účelem vyřešení určité úlohy, musí být počet kroků od začátku do konce *konečné* číslo. Ba co více, kroků musí být jen tolik, že při aplikaci algoritmu obdržíme výsledek v čase, který jsme ochotni tolerovat. K čemu by nám byl algoritmus, který by sice předčil svou přesností všechny ostatní, když výsledek řešení úlohy by nám byl znám za třeba 150 let? Ruku na srdce, čekali byste půlhodinku na výsledek z kalkulačky? Významnou a častou chybou při vytváření algoritmů je chyba při formulaci podmínky ukončení nějakého opakujícího se sledu operací. To má za následek, že není splněna podmínka konečného počtu kroků.

Každý algoritmus vždy musí poskytovat výsledky řešení, správný algoritmus zajišťuje i správné výsledky řešení (Výsledkem návodu na výměnu kola vozidla byla uskutečněna úspěšná výměna kola).

Navíc musí algoritmus zajistit *reprodukovatelnost* výsledků. To znamená, že při stejných vstupních údajích musí bezpodmínečně vždy poskytnout stejné výsledky řešení.

Vývoj algoritmu nemusí být levnou záležitostí, a proto by měl být vytvořen tak, aby zahrnoval řešení celé skupiny úloh, lišících se jen vstupními údaji. Napsat postup na výpočet druhé odmocniny z čísla 6,27 by sice bylo možné, ale sami uznáte, že smysluplnější bude

napsat algoritmus na výpočet druhé odmocniny z libovolného zadaného čísla z definičního oboru funkce.

Algoritmus musí být *přesný* a *srozumitelný*, protože cílem není algoritmus jen vytvořit, cílem je dokázat algoritmus aplikovat v konkrétních podmínkách. Nepřesnosti a špatná srozumitelnost vede k chybám při realizaci (nebudeme-li návodu rozumět nebo budou-li v něm nepřesnosti, budeme mít problémy i při tvorbě počítačového programu).

Nauka zabývající se tvorbou algoritmů se nazývá **algoritmizace**.

Shrnutí pojmů

Definice algoritmu dle ČSN 36 9001/1-*1987* [18] *„Postup řešení problému splňující tyto podmínky: rezultativnost, konečnost, determinovanost, hromadnost.“*

Definice algoritmu dle A. A. Markova [17] - "*Algoritmus je přesný předpis definující výpočtový proces vedoucí od měnitelných výchozích údajů až k žádaným výsledkům.“*

Algoritmizace - nauka zabývající se tvorbou algoritmů. Někdy také proces vytváření algoritmů (programů).

Vlastnosti algoritmů:

Rezultativnost - aplikace algoritmu musí dát výsledek po konečném počtu kroků, a to pro všechny vstupní údaje, které patří do množiny přípustných vstupních údajů, kterou bývá zvykem označovat jako *definiční obor algoritmu*. To znamená, že realizace algoritmu vždy musí vést k jednoznačnému výsledku a následně k ukončení řešení.

Determinovanost - při realizaci algoritmu musí být v každém kroku jednoznačně určena nejen činnost, která je právě prováděná, ale i činnost, která má bezprostředně následovat. To znamená, že v žádném kroku nesmí být pochyb o dalším postupu.

Hromadnost - vytvořený algoritmus není použitelný jen pro určité konkrétní vstupní údaje, ale pro libovolné vstupní údaje z definičního oboru algoritmu. To znamená, že vytvořený algoritmus musí být popisem řešení celé skupiny příbuzných úloh, které se od

sebe liší pouze vstupními údaji.

Reprodukovatelnost - při aplikaci vytvořeného algoritmu musí být výsledek řešení vždy stejný, pokud jsou použity stejné vstupní údaje z definičního oboru algoritmu.

Správnost - věcná správnost algoritmu je nutnou podmínkou úspěšné aplikace algoritmu. To znamená, že význam má pouze takový algoritmus, který dává věcně správné řešení. Správnost algoritmu je tedy samozřejmostí.

Srozumitelnost - algoritmus musí být formulován tak srozumitelně, aby mu rozuměl každý, kdo s ním bude pracovat (jeho tvůrce s odstupem času, programátor, který podle něj bude vytvářet počítačový program apod.)

Otázky

1. Co znamená pojem algoritmus?
2. Co chápeme pod pojmem algoritmizace?
3. Jaké vlastnosti musí mít algoritmus?

Úlohy k řešení

1. Doplňte návod na výměnu kola o pasáž, která by ošetřila i případ, že nemáme s sebou rezervní kolo nebo to je nefunkční případně nenahuštěno na požadovanou hodnotu. Podobně ošetřete jak postupovat, kdyby nebyl funkční zvedák nebo nebylo možné našimi silami montážní matice povolit.
2. Srovnejte příkazy 12. – 18. správného a špatného algoritmu a zjistěte, které nepřesnosti se ve špatném algoritmu nacházejí, a vysvětlete, jak vznikly.

2.2. Vývojové diagramy

Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět:

- různým způsobem popsat algoritmus.
- správně použít symboly vývojových diagramů.
- vytvářet jednoduché vývojové diagramy.

Výklad

Protože jednou z významných vlastností algoritmu je jeho srozumitelnost a přehlednost, je vhodné pro jeho zápis zvolit jednu z mnoha metod, jež byly za tímto účelem vytvořeny.

My jsme se zde již seznámili s jednou z nich, šlo o ***slovní vyjádření***. Tato metoda je přirozená, vhodná i pro komunikaci s osobou neznalou problematiky algoritmizace a programování. Pro větší přehlednost je vhodné jednotlivé kroky očíslovat, a to hlavně k snadnější definici skoku na určený krok, který má následovat. Je to však metoda, která není příliš přehledná a velmi těžko se při její aplikaci zjišťuje správnost, jednoznačnost, zda jsou ošetřeny všechny možné varianty a další podmínky funkčnosti algoritmu. Protože formulace mohou obsahovat libovolná vyjádření, nemusí být zajištěna ani dobrá srozumitelnost algoritmu.

V některých případech je výhodné použít ***matematický zápis***. Jeho největší devizou je, že je jednoznačný. Nevýhodou pak, že je jeho použití omezeno na řešení matematických úloh a sám o sobě nebývá podrobný, což neumožňuje přímé zadání do počítače při programování.

I když existuje ještě celá řada metod jako například ***rozhodovací tabulky*** nebo třeba přímo ***počítačový program***, my se budeme z velké části zabývat metodou, která využívá tzv. ***vývojové diagramy***.

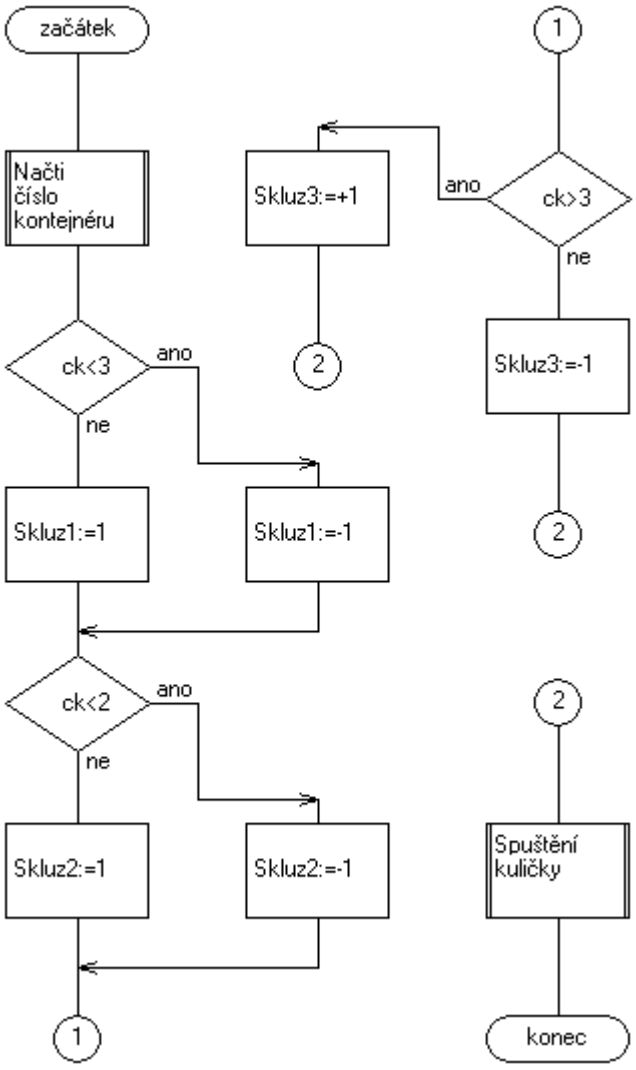
Vývojové diagramy mají dlouhou a úspěšnou historii. Jejich prapůvod sahá až do starověké minulosti. V zásadě jde o *symbolický algoritmický jazyk*, který se hojně používá pro názorné grafické zobrazení algoritmů.

Tento *symbolický algoritmický jazyk* se skládá z přesně definovaných symbolů (značek), které mají jednoznačný význam (sémantika) a pravidla, jak je používat (syntaxe). Jeho výhodou je

to, že umožňuje velice názorně vyjádřit postup řešení dané úlohy s vyznačením všech možných variant, které mohou nastat.

Vývojový diagram je tedy přesně definovaný grafický způsob zápisu algoritmů, v němž jsou k vyjádření jednotlivých příkazů použity dnes již nadnárodní normou definované symbolické značky, které se mezi sebou spojují orientovanými spojnicemi, jež vyznačují směr postupu algoritmu.

Může tak být využit nejen k vývoji a popisu algoritmu, ale může stejně tak dobře sloužit k dokumentačním účelům, jako komunikační prostředek mezi programátory a analytiky, ale hlavně jako podklad pro sestavení počítačového programu již v konkrétním programovacím jazyku.



Obr. 2.1 – Ukázka vývojového diagramu

Analytik je odborník, který provádí analýzu daného problému a na jejím základě vytvoří postup řešení ve formě např. vývojového diagramu.

Programátor je odborník, který pak algoritmus podle tohoto vývojového diagramu napíše v zadaném programovacím jazyku a vytvoří tak vlastní program.

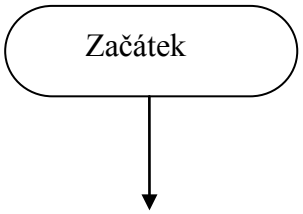
Pro tvorbu vývojových diagramů platí nová česká státní norma ČSN ISO 5807 "Zpracování informací. Dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy programu a diagramy zdrojů systému" (vydáno1.1.1996) [19].

Norma obsahuje celou řadu symbolických značek a pravidel práce s nimi (Obr. 2.1). V praxi si vystačíme s několika obecnými značkami, které si zde uvedeme. Další jsou vesměs značkami rozšiřujícími význam těch obecných a můžete se s nimi blíže seznámit ve výše uvedené normě. Pro kreslení vývojových diagramů je možno použít některý specializovaný program, ale pro naše účely dostatečně poslouží tvary implementované v programu MS Word, které naleznete v záložce Vložit – Ilustrace – Tvary – Vývojové diagramy.

POZOR!
Spojnice značek (hrany) nemusí být označeny šipkou, pokud postup algoritmu směřuje zleva doprava nebo seshora dolů.

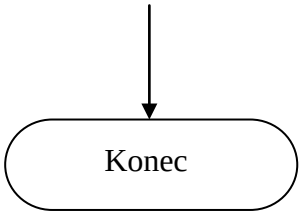
Začátek a konec algoritmu.

Každý vývojový diagram musí obsahovat značku začátku a značku konce algoritmu. Pro tento účel jsou určeny tzv. mezní značky, které obecně představují vstup z vnějšího prostředí do programu nebo naopak výstup z programu do vnějšího prostředí. Pokud jsou použity k označení začátku algoritmu, pak je nutno do značky vepsat slovo „Začátek“ a opatřit značku jedinou hranou, a to tak, aby vystupovala ze spodní části značky a směřovala dolů (Obr. 2.2). Do značky zároveň nesmí vstupovat žádná hrana.



Obr. 2.2 – Mezní značka označující začátek algoritmu

Pokud jsou použity k označení konce algoritmu, pak je nutno do značky vepsat slovo „Konec“ a opatřit značku jedinou hranou, a to tak, aby vstupovala do horní části značky a směřovala nahoru (Obr. 2.3). Ze značky zároveň nesmí vystupovat žádná hrana.



Obr. 2.3 – Mezní značka označující konec algoritmu

Uvnitř vývojového diagramu se mimo jiné nacházejí tzv. sekvenční bloky. Označují se tak proto, že je možné se do nich dostat pouze z předchozích prvků a po provedení požadované operace může algoritmus postoupit výlučně na jeden následující prvek (značku) vývojového diagramu. Zajišťují tak sekvenční postup algoritmem.

Zde si uvedeme dva základní sekvenční bloky:

- zpracování
- data (vstup nebo výstup údajů)

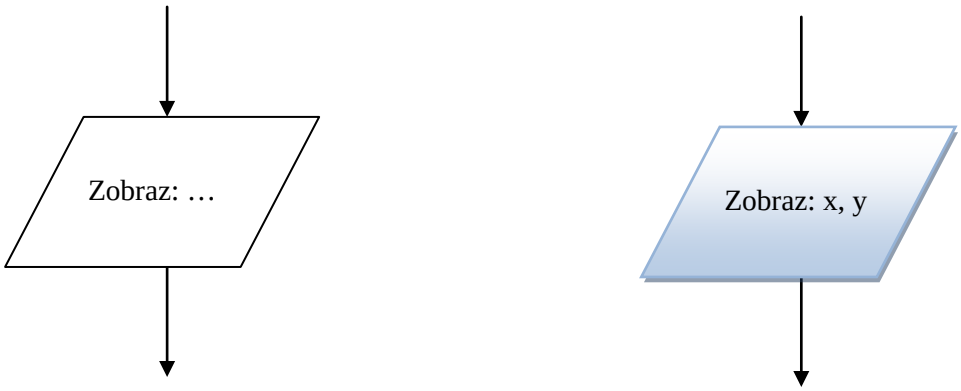
Symbol „zpracování“ (Obr. 2.4) je zásadním symbolem a reprezentuje jakékoliv provedení definované činnosti (operace) nebo skupiny činností (operací). Přitom dochází k transformaci vstupních údajů na výstup. Většinou jde o matematické nebo logické operace (např. vynásobení dvou hodnot nebo třeba negace logického výrazu).

Obr. 2.4 – Značka označující „zpracování“

Norma povoluje i několik vstupů do libovolné strany symbolu, my však pro přehlednost využijeme možnost přivedení vstupních údajů pouze jedním vstupem, byť se může jednat o *spojnici výslednou*. Výstup symbolu musí být zásadně vždy jen jeden a musí být spojen zase pouze se vstupem jediného následujícího symbolu. Na obr. 2.4 je vidět, že vstup symbolu může být složen z výstupu dvou předcházejících symbolů (plná a čárkovaná čára). Nejde však

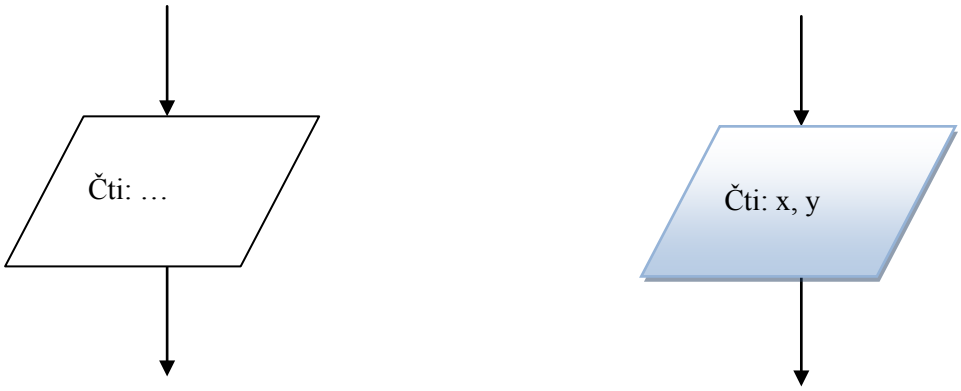
o fyzický spoj, vstupní údaje se zde ani nesčítají, představte si jen, že v jednu chvíli jsou v daném bloku zpracovány údaje z jednoho předcházejícího bloku (plná čára) a jindy podle toku zpracování algoritmu zas z druhého předcházejícího bloku (čárkovaná čára).

Symbol „data“ (Obr. 2.5) je rovněž zásadním symbolem. Na rozdíl od předchozího reprezentuje *dvě* významné operace. V obou případech jde o zajištění komunikace mezi počítačem a obsluhou za běhu programu (vstupně - výstupní operace s daty).
Každý algoritmus musí vyprodukovat výstupní údaje a ty je třeba nějakým způsobem z počítače „dostat“ ve formě srozumitelné obsluze (číslo, tabulka, graf, které mohou být zobrazeny na monitoru nebo vytištěny na tiskárně (zobrazovací zařízení), případně v datové podobě uloženy do souborů na paměťových mediích (HD, flash disky, CD a pod). V tom případě symbol „data“ znázorňuje výstup údajů z programu převážně na zobrazovací zařízení a do značky se píše slovo „Zobraz“ a charakteristika výstupních údajů (obsah proměnné apod.).



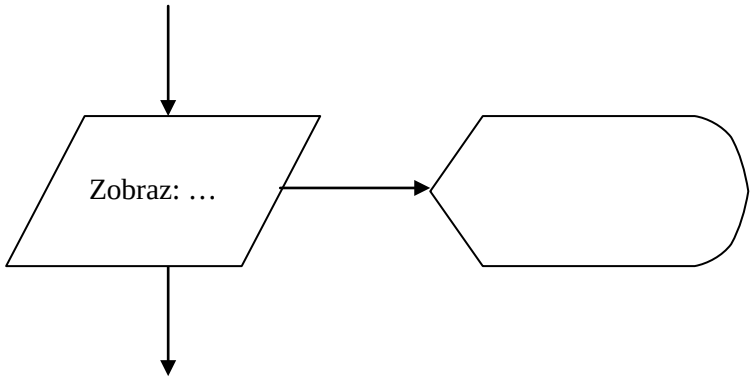
Obr. 2.5a – Značka označující „data“ - výstup

Naopak, i když to nemusí být bezpodmínečně nutné, je třeba počítači dodat potřebné vstupní údaje pro zpracování algoritmem, což se děje prostřednictvím vstupních zařízení počítače (klávesnice, myš a jiná čtecí zařízení), případně v datové podobě jsou přečteny ze souborů na paměťových mediích (HD, flash disky, CD apod.). V tom případě symbol „data“ znázorňuje vstup údajů do programu ze vstupních zařízení počítače a do značky se píše slovo „Čti“.



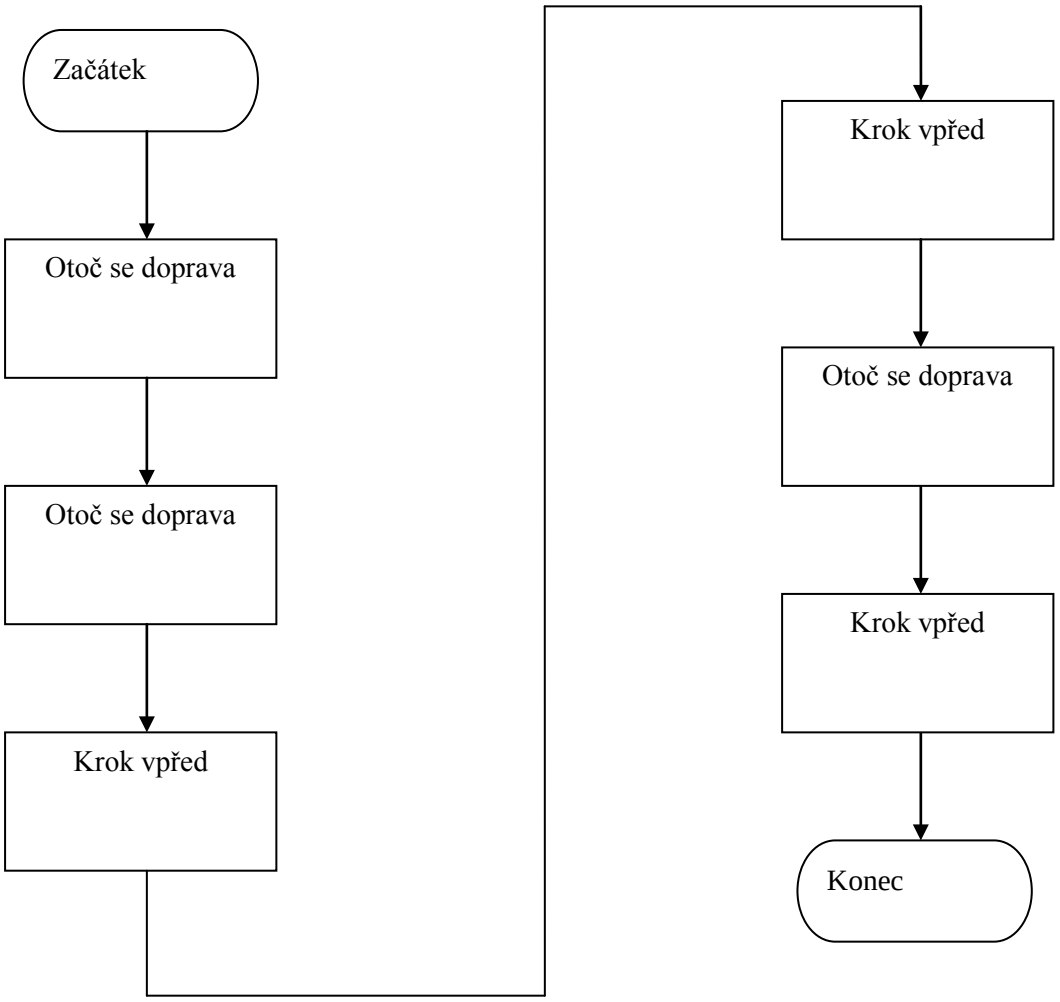
Obr. 2.5b – Značka označující „data“ - vstup

Specifikace použitého vstupního nebo výstupního zařízení buď vyplývá z řešené úlohy, nebo je-li to nutné, může být provedena komentářem, případně paralelním přiřazením odpovídajícího symbolu, jak ukazuje obr. 2.6, kde je znázorněn monitor nebo jiné zařízení pro vizuální zobrazení dat.



Obr. 2.6 – Značka označující „data“ – vstup na monitor

Doposud uvedené symboly charakterizují takové kroky algoritmu, které představují sekvenci příkazů (povelů) jdoucích postupně po sobě od začátku až po konec. Pokud si ještě vzpomínáte na úkol přesunout psa pomocí dvou naučených povelů z jednoho místa na druhé, pak nám doposud uvedené symboly naprosto postačí pro vytvoření funkčního vývojového diagramu. Jednoduchý příklad je uveden na obrázku 2.7.



Obr. 2.7 – Vývojový diagram povelu psovi

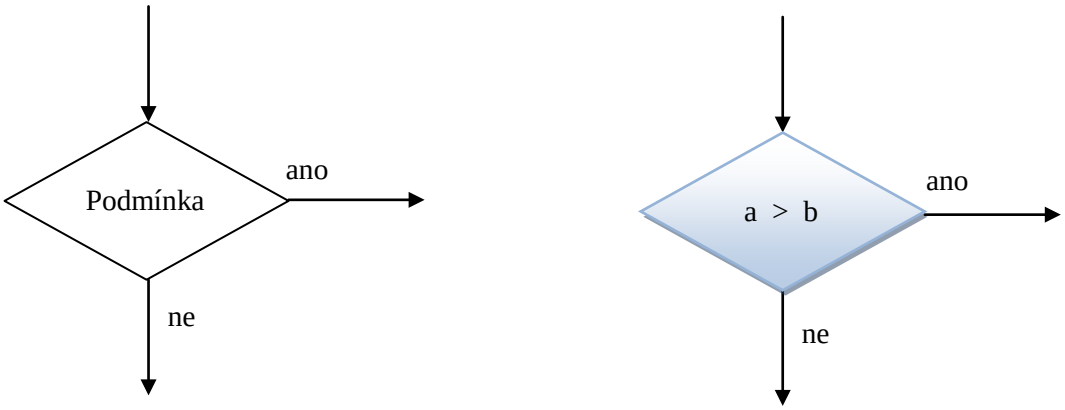
Větvení algoritmu

Uvedené symboly nám však v žádném případě nebudou vystačovat pro realizaci vývojového algoritmu popisujícího dříve zmíněný návod na výměnu poškozeného kola automobilu. Příkaz 1. je možno bez problémů znázornit symbolem (nebo dvěma) „zpracování“. Jenže hned druhý příkaz způsobuje rozdělení toku provádění algoritmu do dvou větví na základě logického stavu podmínky v daném okamžiku.

Příkaz 2.: Jste-li na silnici, pak vyjměte výstražný trojúhelník z jeho místa určeného pro transport a umístěte ho na silnici do jízdního pruhu, ve kterém stojí vaše vozidlo, a to 50 m za vozidlo. Nejste-li na silnici, nechte výstražný trojúhelník na svém místě v zavazadlovém prostoru a pokračujte při montáži bodem 3.

Jedna větev je realizována, stojíme-li na silnici (aplikujeme výstražný trojúhelník a následně pokračujeme příkazem 3.), druhá větev se realizuje, nestojíme-li na silnici (neuděláme nic a rovněž následně pokračujeme příkazem 3.).
Podobná situace nastává při realizaci příkazu 18. Ovšem s tím rozdílem, že tentokrát v obou větvích vývojového diagramu budeme muset provést nějakou činnost.
Příkaz 18.: Zůstal-li na místě výstražný trojúhelník, ukliďte ho na jeho místo určené pro transport, nezůstal-li tam, zapamatujte si, že musíte v nejbližším obchodě s autodílny koupit nový a umístit ho na jeho místo určené pro transport.
Jedna větev se aplikuje v případě, že již nebyl trojúhelník na původním místě (zapamatovat si, že je nutno koupit nový trojúhelník), druhá v případě, že se na původním místě nacházel (úklid trojúhelníku).
Z toho vyplývá, že musíme mít při vytváření vývojových diagramů k dispozici takový symbol, který umožní rozvětvení algoritmu na základě rozhodnutí podle jednoznačně definované podmínky.

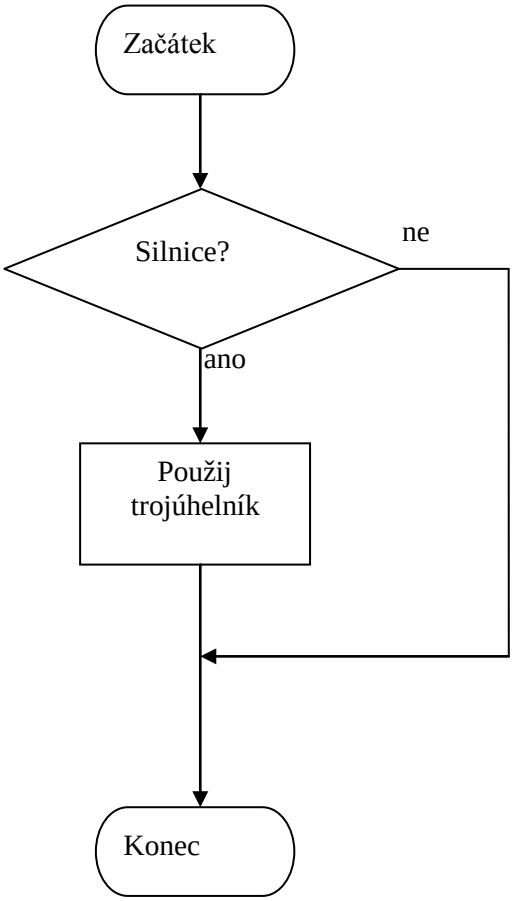
Značka „rozhodování“ na obr. 2.8 je určena k rozvětvení algoritmu a představuje rozhodovací



Obr. 2.8 – Značka označující „rozhodování“

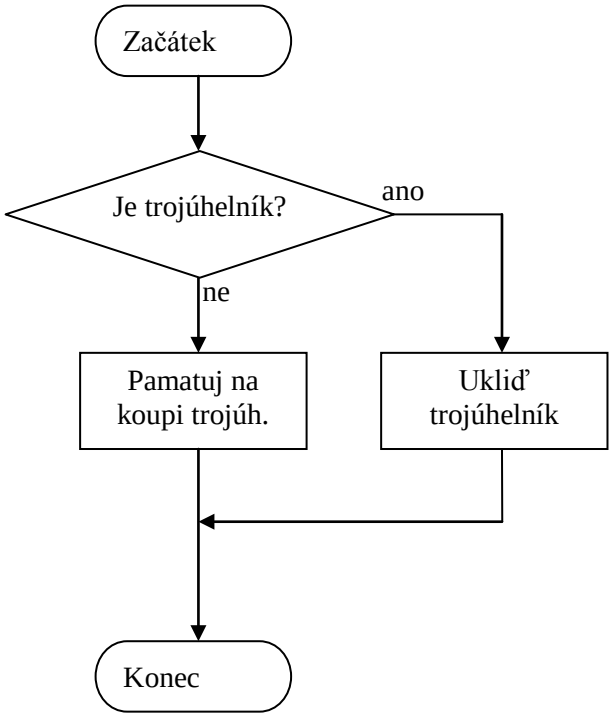
funkci. Rozhodovací podmínka se píše dovnitř symbolu. Je-li podmínka splněna, pak se ubírá tok algoritmu po výstupní spojnici označené slovem „ano“ nebo znaménkem „+“, není-li splněna, tok algoritmu postupuje po výstupní spojnici označené slovem „ne“ nebo znaménkem „-“. Přitom není důležité, z kterého rohu symbolu výstupní spojnice vystupují, a které jsou označeny jako „ano“ či „ne“. Vychází se zde jen z grafické přehlednosti algoritmu. Značka má pouze jeden vstup, byť může být realizován výslednou spojnicí, podobně jako v případě předchozích tří symbolů. Vývojový diagram popisující příkaz 2. návodu na výměnu

poškozeného kola automobilu je uveden na obr. 2.9. Z obrázku je patrno, že v rozhodovacím bloku je tok algoritmu rozdělen do dvou nezávislých větví, které obsahují nějaký příkaz (nebo i sled příkazů), ale jedna větev dokonce nemusí obsahovat příkaz žádný. Ta představuje jen přesun toku algoritmu do jiného místa vývojového diagramu. Konstrukce, kde by obě větve neobsahovaly žádné příkazy je sice možná, ale bezúčelná (Dokonce její naprogramování by mohlo způsobit určité potíže).



Obr. 2.9 – Vývojový diagram příkazu 2. návodu na výměnu poškozeného kola automobilu

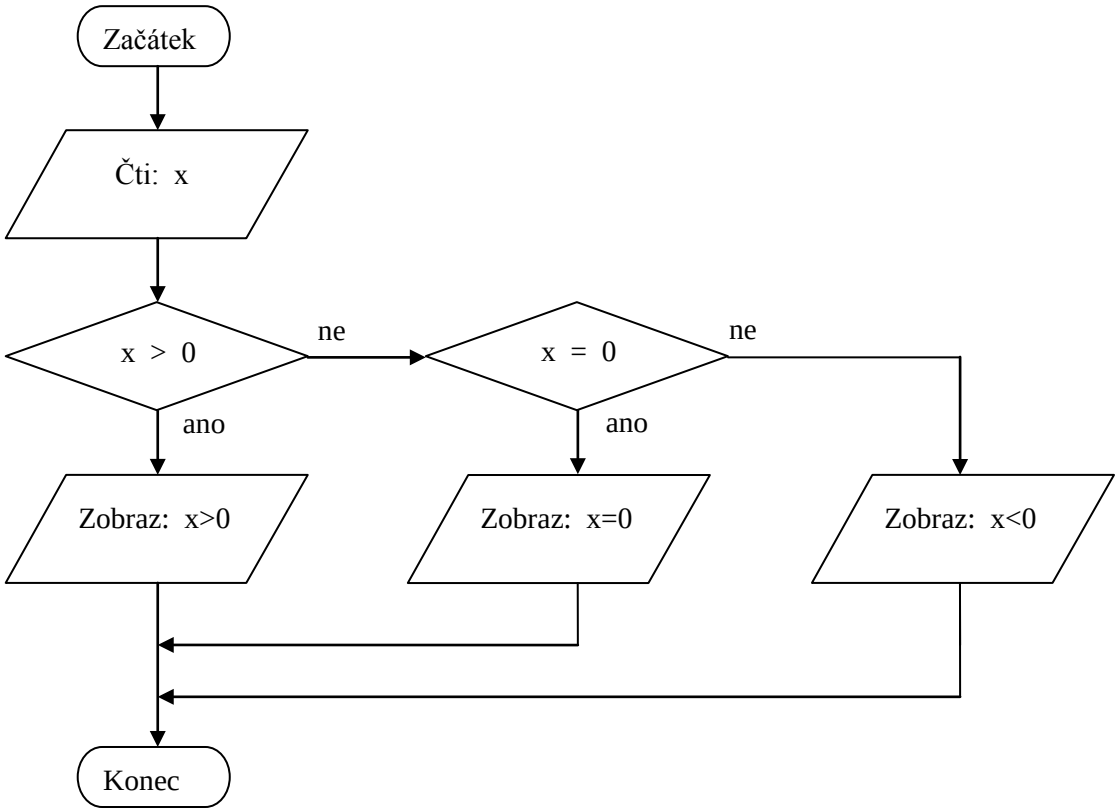
Vývojový diagram popisující příkaz 18. návodu na výměnu poškozeného kola automobilu je uveden na obr. 2.10. Zde si všimněte, že byly přehozeny polohy spojnic pro splnění podmínky „ano“ a pro nesplnění „ne“. Druhá větev mohla stejně tak dobře vycházet z levého rohu značky, pokud by to bylo pro přehlednost lepší.



Obr. 2.10 – Vývojový diagram příkazu 18. návodu na výměnu poškozeného kola automobilu

Protože každá větev může obsahovat libovolné příkazy (značky), pak není vyloučeno, že se v ní znovu vyskytne značka „rozhodování“. Tím mohou vzniknout složitější konstrukce jako například na obr 2. 11.

Zkuste zjistit, k čemu je určen algoritmus z tohoto obrázku a zkuste jej nějakým způsobem modifikovat, ale tak, aby zůstala jeho funkce zachována. Zde je patrna právě grafická přehlednost vývojových diagramů oproti slovnímu zápisu algoritmu. Lépe je zde vidět, zda jsme vyčerpali všechny možnosti, které pro danou vstupní množinu dat mohou nastat. V tomto případě je tato množina (definiční obor algoritmu) definovaná jako množina reálných čísel. Pokuste se na základě tohoto vývojového diagramu vytvořit popis algoritmu s využitím slovního zápisu. Pro inspiraci se vraťte k návodu na výměnu poškozeného kola automobilu. Úloha je stále poměrně jednoduchá, nemělo by vám proto její řešení činit velké potíže.



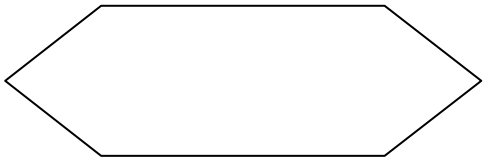
Obr. 2.11 – Vývojový diagram úlohy na větvení algoritmu

Opakování části algoritmu.

Při tvorbě algoritmů se někdy nevyhneme nutnosti opakovaně provádět pořád tu stejnou operaci nebo celou skupinu operací. V některých případech by se technicky dalo odpovídající skupinu operaci napsat několikrát znovu a pak klasicky sekvenčně všechny vykonat za sebou. S rostoucím počtem opakování by však rostla délka algoritmu a také pravděpodobnost chyby při určení kolik skupin operací jsme již napsali. U některých algoritmů dokonce ani dopředu neznáme, kolikrát se bude daná skupina opakovat. Blíže bude tato významná problematika probrána v kapitole o cyklech.

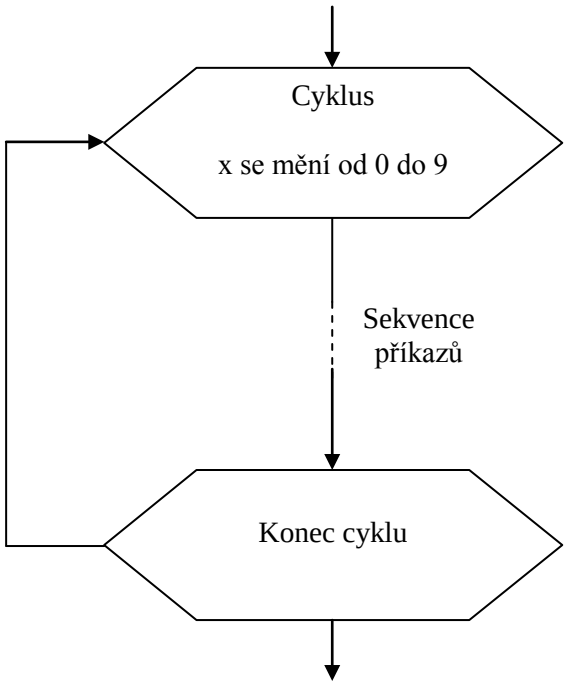
I z hlediska požadavku na tvorbu efektivních algoritmů, by bylo účelné ve vývojových diagramech zohlednit potřebu opakování určité části algoritmu nějakou přehlednou konstrukcí podporovanou příslušnou značkou. Právě značka „Příprava“ (Obr 2.12), která sice obecně označuje přípravnou fázi algoritmu, se většinou používá právě na konstrukce algoritmů se známým počtem opakování určitých operací. Počet opakování nějaké sekvence příkazu budeme označovat jako počet cyklů, kde jeden cyklus můžeme chápat jako jeden průchod

definovanou sekvencí příkazů. Zde je na místě podotknou, že sekvence příkazu je sice ve všech cyklech pořád stejná, avšak aktuální vstupní hodnoty se v každém cyklu mění podle hodnoty *řídicí proměnné cyklu* (opakování operací, aniž by se něco měnilo, by bylo bezúčelné, tedy s výjimkou požadavku na zpoždění běhu programu).



Obr. 2.12 – Značka označující „přípravu“

Pro naše účely zvolíme tu variantu, která pro označení cyklu s pevným počtem opakování využívá dvou identických značek (Obr. 2.13), přičemž jedna s vepsaným slovem „Cyklus“



Obr. 2.13 – Označení cyklu s pevným počtem opakování

označuje zahájení cyklu a bývá dále opatřena parametry cyklu, druhá označuje konec cyklu a zapisuje se do ní slova „Konec cyklu“. Vlastní cyklus je vytvořen spojnicí ze značky konce cyklu do značky zahájení cyklu. Parametry cyklu zprostředkovaně udávají, kolikrát má cyklus proběhnout. Řídicí proměnnou cyklu je zde proměnná „x“.

Uvedeme si zde příklad, který problematiku opakování rychle objasní. Dostali jsme za úkol vypočítat hodnoty funkce sinus pro úhly 0^0, 10^0, 20^0 … 90^0. Když zapneme kalkulačku a

přepneme jí na výpočet goniometrických funkcí s argumentem zadávaným ve stupních, čeká nás poměrně pracný úkol. Navolit na klávesnici kalkulačky první hodnotu argumentu (tedy 0), následuje stisk tlačítka pro výpočet funkce sinus („sin“), vymazání displeje, zadání nového argumentu, stisk tlačítka „sin“ a tak pořád dokola, dokud nemáme vypočtenou poslední hodnotu.

Později si ukážeme, že pomocí počítačového programu se nám výpočet automatizuje a není problém získat výsledky pro hodnoty argumentu třeba s intervalem jeden stupeň za zlomek času, který bychom potřebovali pří výpočtu původního zadání na kalkukačce.

Podprogram.

Často se v praxi můžeme setkat se situací, kdy v rámci řešeného úkolu musíme provést nějakou operaci vícekrát, ale ne jako v předchozím případě bezprostředně za sebou, ale vracíme se k ní vždy po vykonání nějakých jiných činností. Mohli bychom sice i zde tu stejnou operaci zařadit do těch částí algoritmu, kde je jí zapotřebí provést, ale což tak si realizovat algoritmus, který umí provést právě a jen tu opakující se operaci a vždy, když dospěje tok hlavního algoritmu do místa, kde se má ona operace provést, předat její provedení tomuto specializovanému algoritmu. Realizace tohoto specializovaného algoritmu v nějakém programovacím jazyku se nazývá podprogram. Jak již název napovídá, půjde zřejmě také o nějaký program, který je nějakým způsobem součástí jiného (hlavního) programu. Jde opravdu o realizaci skutečného algoritmu, který je schopen řešit nějakou část hlavního

Obr. 2.14 – Značka označující „podprogram“

problému.

Ve vývojových diagramech je vytvořena značka „podprogram“, jejíž vzhled je podobný značce zpracování a můžeme ji vidět na obr. 2.14. Jde o samostatnou část algoritmu, která obsahuje více příkazů (kroků).

Spojka.

Spojka je značka (Obr. 2.15), která sice sama nic neprovádí, ale umožňuje přehledně spojit dvě navazující části vývojového diagramu, které nebylo možné propojit klasickou spojnicí. Představuje tedy přechod z jedné části vývojového diagramu na jinou část. Důvodem může být například přechod na novou stránku, omezení husté spleti čár nebo jejich křížení při kreslení složitých algoritmů.

Obr. 2.15 – Značka označující „spojku“

Jde o značku párovou a odpovídající symboly (většinou čísla) musí u obou obsahovat stejné označení. Spojka na konci přerušení má pouze jeden vstup a spojka na začátku pokračovaní má pouze jeden výstup. Použití spojek je patrno z obrázku 2.1.

Komentáře.

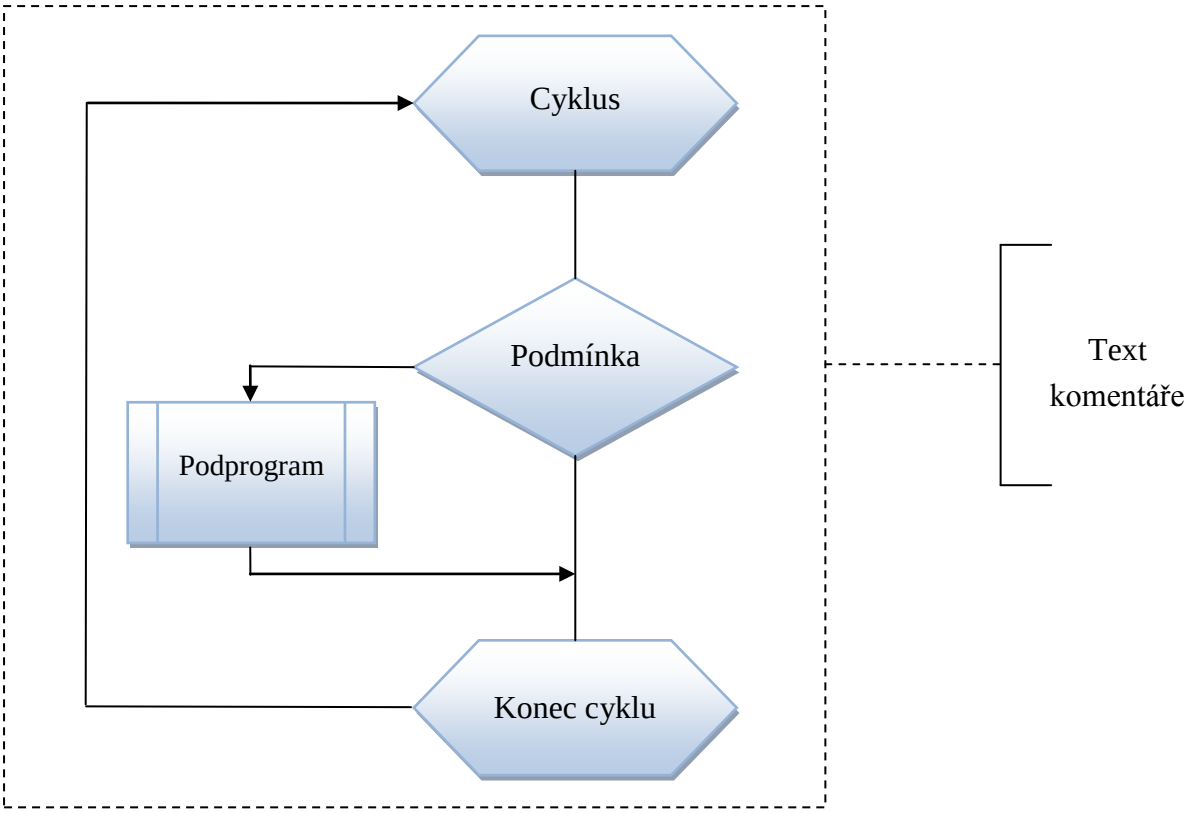
Jak již bylo řečeno, jednou z požadovaných vlastností zápisu algoritmů je jejich srozumitelnost a přehlednost. Právě vhodné využití komentářů nám může tuto vlastnost

Obr. 2.16 – Značka označující „anotaci“

pomoci zajistit.

Symbol k tomu určený se nazývá „anotace“ (obr. 2.16) a používá se k doplnění komentářů (vysvětlujících textů).

Přerušovaná čára symbolu anotace je připojena k příslušnému výkonnému symbolu, nebo může být připojena i k ohraničené skupině symbolů, jež je vidět na obr. 2.17.



Obr. 2.17 – Značka označující „anotaci“ pro komentář ke skupině symbolů

Vývojové diagramy nás budou provázet celým zbytkem učební opory, proto jim byla věnována taková pozornost. Bylo by dobré, kdybyste jim dokonale porozuměli a to nejlépe tak, že si vymyslíte nějaký úkol, a ten se budete snažit vyjádřit vývojovým diagramem. Uvidíte, že to nebude snadná záležitost. Jak budete pronikat do hloubky řešení zdánlivě jednoduchého problému, zjistíte, že do algoritmu musíte zahrnout a ošetřit další a další podmínky, abyste měli jistotu, že algoritmus bude zcela funkční a tedy správně reagující na všechny i málo pravděpodobné situace. Pro inspiraci nahlédněte do nabídky úloh k řešení a vyzkoušejte si je vyřešit. Když to půjde snadno, máte velké šance, že vám vlastní programování nebude činit zvláštní problémy. Možná vás dokonce začne bavit.

Shrnutí pojmů

Vývojový diagram je přesně definovaný grafický způsob zápisu algoritmů, v němž jsou k vyjádření jednotlivých příkazů použity dnes již nadnárodní normou definované symbolické značky, které se mezi sebou spojují orientovanými spojnicemi, jež vyznačují směr postupu algoritmu. Jeho výhodou je to, že umožňuje velice názorně vyjádřit postup řešení dané úlohy s vyznačením všech možných variant, které mohou nastat.

Vývojový diagram slouží jako:

- komunikační prostředek mezi analytiky a programátory,
- prostředek k dokumentaci vytvořeného algoritmu,
- podklad pro tvorbu počítačového programu.

Analytik je odborník, který provádí analýzu daného problému a na jejím základě vytvoří postup řešení ve formě např. vývojového diagramu.

Programátor je odborník, který pak algoritmus podle tohoto vývojového diagramu napíše v zadaném programovacím jazyce vlastní počítačový program.

Metody zápisu algoritmu:

- slovní vyjádření,
- matematický zápis,
- rozhodovací tabulky,
- vývojový diagram,
- metody strukturální a objektové analýzy,
- počítačový program.

Počátek vývojového diagramu musí být označen mezní značkou s nápisem *Začátek*. To odpovídá zahájení činnosti algoritmu. Spojnice, která ze značky *Začátek* vychází, určuje první výkonnou značku. Pokud vychází ze značky jediná spojnice, je další postup jednoznačný. Z některých značek ale může vycházet více spojnic, které nazýváme větve. Ze značky

rozhodování musí vycházet právě a jen dvě větve. Která bude zvolena, o tom jednoznačně rozhodne logická hodnota podmínky, která je zapsaná uvnitř značky. Tok algoritmu může některou část algoritmu proběhnout i opakovaně, avšak v konečném počtu kroků musí dojít na jedinou mezní značku s nápisem *Konec*. Algoritmus tak ukončí svou činnost. Pro zopakování problematiky vývojových diagramů můžete použít animovaný program, který je na CD označený jako ***2a Vyvojovy diagram***.

Otázky

1. Jakými způsoby lze vyjádřit algoritmus?
2. Jaké jsou výhody a nevýhody jednotlivých vyjádření?
3. Co je to vývojový diagram?
4. Jaké základní symbolické znaky vývojových diagramů znáte?

Úlohy k řešení

1. Pokuste se určit význam jednotlivých symbolických značek ve vývojovém diagramu z obr. 2.1.
2. Zjistěte, k čemu je určen algoritmus z obrázku 2.11 a zkuste jej nějakým způsobem modifikovat, ale tak, aby zůstala jeho funkce zachována.
3. Vytvořte vývojový diagram, který by popisoval postup při přecházení silnice.
4. Vytvořte vývojový diagram, který by popisoval postup při jízdě přes křižovatku.
5. Vytvořte vývojový diagram, který by popisoval postup při výpočtu funkce faktoriál.
6. Vytvořte vývojový diagram, který by popisoval postup při výpočtu transponované matice.
7. Vytvořte vývojový diagram, který by popisoval postup při ukládání souboru na externí paměťové médium.
8. Vytvořte vývojový diagram, který by popisoval postup při tabelaci lineární funkce.

3. PROGRAMOVÁNÍ

3.1. Úvod

Čas ke studiu: 0.5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- hierarchii programovacích prostředků.
- základní etapy vývoje programu.
- plánovat a strategicky rozmýšlet tvorbu programu.

Výklad

Právě se nacházíme ve stádiu, kdy už umíme vytvořit algoritmus prakticky pro libovolnou úlohu, kterou je třeba vyřešit. Abychom si námi vymyšlený algoritmus řešení zpřehlednili a nějakým způsobem dokumentovali, dokážeme jej zapsat do vývojového diagramu. Pak stačí jen malé nahlédnutí a ihned si vzpomeneme, jak jsme chtěli zadaný úkol řešit.

Jenže právě to je ono, jak ho vlastně vyřešit? Algoritmus již máme, ale čím ho realizovat? S výměnou kola to bylo jednoduché, prostě se vyhrnuly rukávy a hurá do práce. Jenže teď jsme právě vymysleli algoritmus složité matematické úlohy a řešit ji na kalkulačce by nemuselo být nic jednoduchého.

Nabízí se myšlenka na počítač, tolikrát jsme slyšeli, jak vše bleskurychle a bravurně vypočítá, přistání na Měsíci, dokonce dokáže řídit celý kosmický let, tak proč by si neporadil i s naším algoritmem? Je to asi jediná šance. Jdeme na to! Jenže, jak ten náš algoritmus výpočtu počítači sdělíme?

Bude to asi podobné, jako bychom chtěli něco sdělit cizinci, který sice neumí ani slovo česky, zatímco ve své mateřštině štěbetá a štěbetá. Jenže té zase nerozumíme my. Tak to s ním zkusíme lámanou angličtinou, a ejhle, něco jako angličtinu slyšíme i z jeho úst. Možná, když se oba zdokonalíme v angličtině, budeme si dokonale rozumět.

Ano je to velmi podobné, abstraktní algoritmus nemáme počítači přímo jak sdělit, aby mu porozuměl. My zas většinou nerozumíme oné spleti nul a jedniček, s kterými pracuje on. Jsou tu ale počítačové ***programovací jazyky***, které umožní vzájemnou komunikaci a hlavně umožní uskutečnit náš cíl, kterým je předat náš algoritmus počítači, ať jej provede a poskytne nám výsledky řešení zadané úlohy.

V užším slova smyslu právě přepis algoritmu do programovacího jazyka se nazývá ***programování***.

V širším slova smyslu ***programování*** zahrnuje více činností:

- Analýza problému.
- Vytvoření algoritmu (realizace např. vývojovým diagramem).
- Překlad a editace algoritmu do programovacího jazyka (vzniká tzv. ***zdrojový kód***), editace je většinou prováděná ve specializovaném ***textovém editoru.***
- Překlad zdrojového kódu do jazyka počítače (vzniká tzv. ***strojový kód***), překlad je automaticky realizovaný ***překládačem.***
- Spuštění programu, vyhledání a odstranění chyb programu (tzv. ***ladění programu***).

Výsledkem programování je tedy ***počítačový program***, který lze spustit na počítači, a který po zadání vstupních údajů aplikuje vložený algoritmus a následně vygeneruje žádané výstupní údaje.
Programování používá svou přesně definovanou symboliku, své programovací jazyky, které představují jiné druhy jazyka, než jsou cizí jazyky, jako např. angličtina. Nebude sice úplně snadné do programovacího jazyka zcela proniknout, ale ***vývojová prostředí*** moderních programovacích jazyků vám situaci značně usnadní svým přívětivým chováním a snahou o značné omezení formálních a jednotvárných prací. Na vás prakticky zbude jen ta kreativní činnost tvorby zdrojového kódu.

Shrnutí pojmů

Programovací jazyk je jazyk, kterému rozumí jak programátor, tak počítač.

Zdrojový kód je program napsaný v programovacím jazyku.

Strojový kód je program napsaný v jazyku počítače.

Překládač je program, který překládá zdrojový kód napsaný v programovacím jazyce do strojového kódu, v kterém pracuje počítač.

Počítačový program je algoritmus, který je vyjádřen pomocí přesně definovaných výrazových prostředků daného programovacího jazyka.

Programování je proces vytváření algoritmu a jeho následného vyjádření v programovacím jazyku.

Programátor je člověk vyvíjející program.

Otázky

1. Co je to zdrojový kód?
2. Co je to strojový kód?
3. Co je to překládač?
4. Co je to počítačový program?
5. Co je to programování?
6. Co je to programovací jazyk?
7. Co je to programátor?

Úlohy k řešení

1. Počítače pracují ve dvojkové soustavě, prostudujte z dostupné literatury nebo z internetu princip číselných soustav a způsob převodu mezi dvojkovou a desítkovou soustavou.
2. Informujte se z literatury také o soustavě šestnáctkové.

3.2. Programovací jazyky a vývojové prostředí

Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat pojem programovací jazyk.
- popsat pojmy – editor, překládač, linkér, debugger.
- pracovat s vývojovým prostředím Delphi 2005.

Výklad

Z předchozí kapitoly je jasné, že je nutno naučit se nějaký programovací jazyk. Pokud tedy existuje více programovacích jazyků, tak který zvolit?

Opravdu existuje celá řada programovacích jazyků a liší se hlavně jejich dominantní oblasti použití. To znamená, že, i když se tyto oblasti mnohdy překrývají, byly jednotlivé programovací jazyky vytvořeny pro svůj specifický účel. Některé byly vyvinuty pro vědeckotechnické výpočty, jiné pro zpracování ekonomické agendy, některé pro systémové účely, internet nebo dokonce pro řízení. Jeden z úspěšných programovacích jazyků vyvinutých přednostně pro výuku byl jazyk se jménem Pascal. Postupem času se vyvíjel a tak vedle svého původního účelu byl stále více využíván i při programování běžných problémů z technické praxe. Tak jak se vyvíjely programátorské přístupy a techniky, tak přecházel postupně i Pascal ze strukturovaného jazyka, přes objektově orientovaný až do současné podoby, kterou je moderní produkt s názvem Delphi.

My se konkrétně se zaměříme na Borland® Delphi™ 2005.

Borland® Delphi™ 2005 je softwarový systém (program), který umožňuje vytvářet programy v programovacím vycházejícím z příkazů jazyka Pascal. Je účelným funkčním spojením ***editoru, překládače, linkéru*** a ***debuggeru*** do grafického integrovaného *vývojového prostředí*, které usnadňuje a zrychluje vývoj uživatelského programu tím, že umožňuje přímo z editoru editovaný program přeložit, spustit i odladit.

Editor – program určený k editaci (případně opravám) zdrojového kódu.

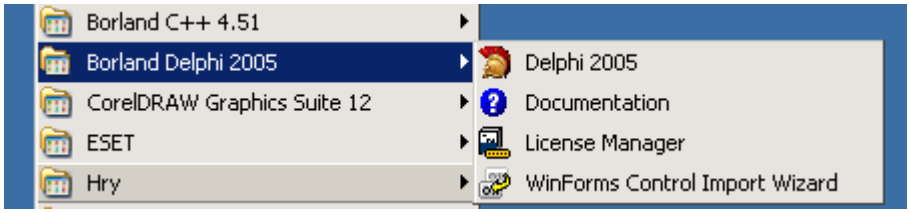
Překládač - program, který překládá zdrojový kód napsaný v programovacím jazyce do strojového kódu, v kterém pracuje počítač.

Linkér – program, který spojuje přeložený zdrojový kód z překládače s ostatními potřebnými již přeloženými programy ve výsledný funkční program spustitelný na počítači.
Debugger – program, který je určen pro usnadnění ladění programu. Usnadňuje nalezení a odstranění logických chyb při běhu programu.

Nyní již nezbývá, než ukojit vaši touhu vytvořit svůj první program. Musíme se ale nejprve seznámit s vývojovým prostředím programu Borland® Delphi™ 2005.

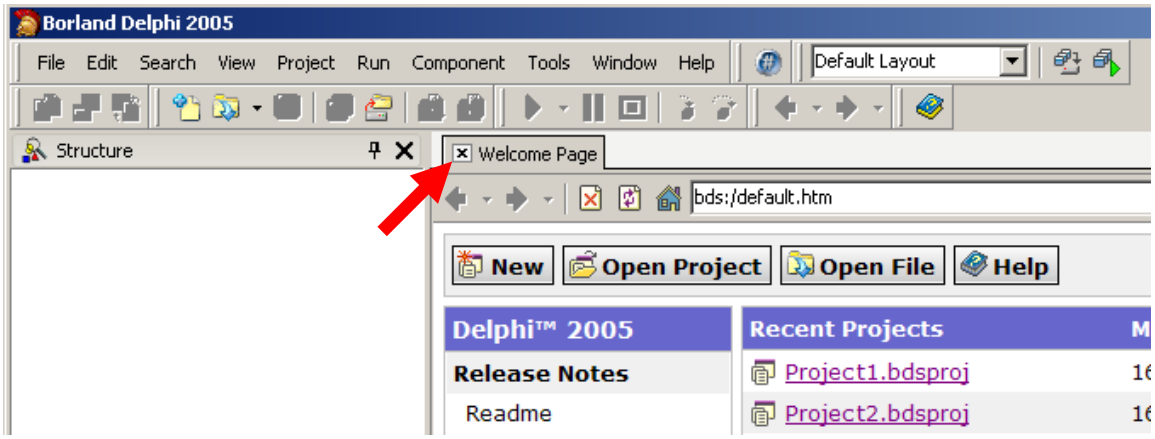
Spuštění programu Borland® Delphi™ 2005

Jde o program jako každý jiný, proto jeho spuštění obstaráme dvojitým kliknutím na tuto ikonku na ploše počítače nebo klepnutím na tlačítko „Start“. Po volbě nabídky „Programy“ si vybereme položku **Borland Delphi 2005**, která nabídne další položky, z nichž volíme **Delphi 2005** (jak je patrno z obr 3.1).



Obr. 3.1 Postup spuštění programu Borland® Delphi™ 2005

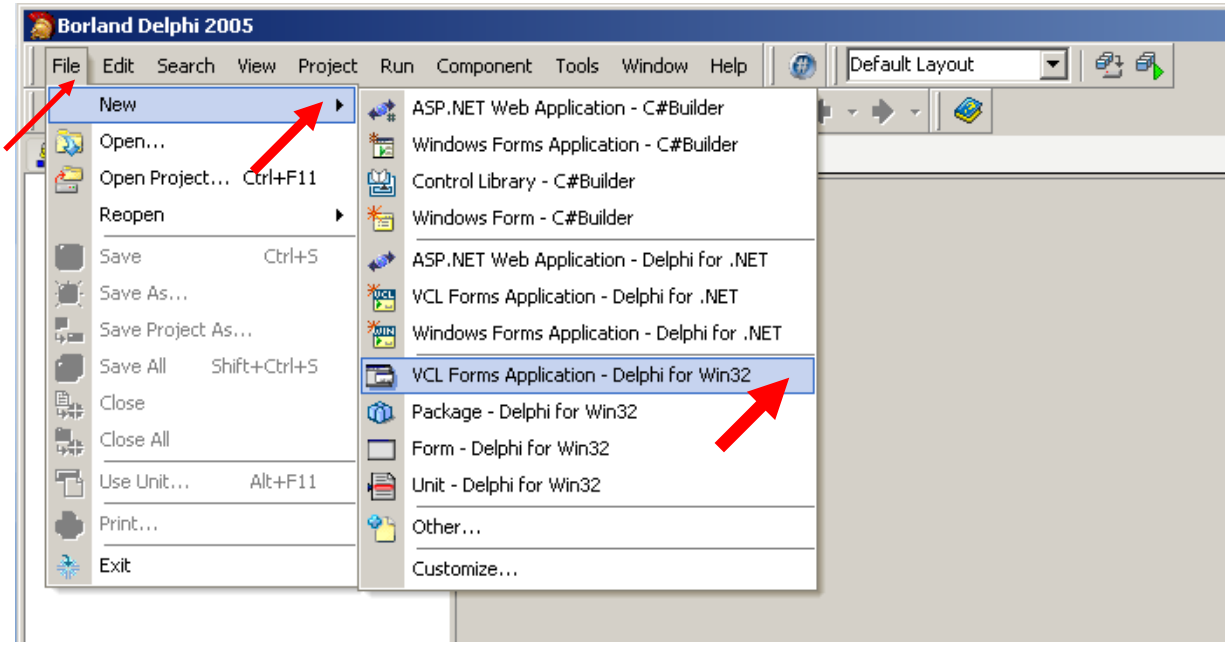
Za několik sekund se již před vámi objeví prostředí Borland® Delphi™ 2005 jehož fragment



Obr. 3.2 Fragment prostředí programu **Borland® Delphi™ 2005**

je uveden na obr. 3.2. Uvítací okno můžeme zavřít kliknutím na jeho uzavírací znak v záložce, jak je vyznačeno v obrázku červenou šipkou.

Protože máme v úmyslu vytvořit svůj první nový program, musíme pro to vytvořit odpovídající prostředí. V menu vybereme položku „File“ (Soubor), následně zvolíme položku „New“ (Nová) a následně výběr ukončíme položkou VCL „Forms Application – Delphi for Win32“, což má za následek otevření grafického integrovaného vývojového prostředí pro tvorbu aplikací (programu) pracujících pod operačním systémem Windows. Postup je patrný z obrázku 3.3.



Obr. 3.3 Postup vytvoření nové aplikace **Borland® Delphi™ 2005**

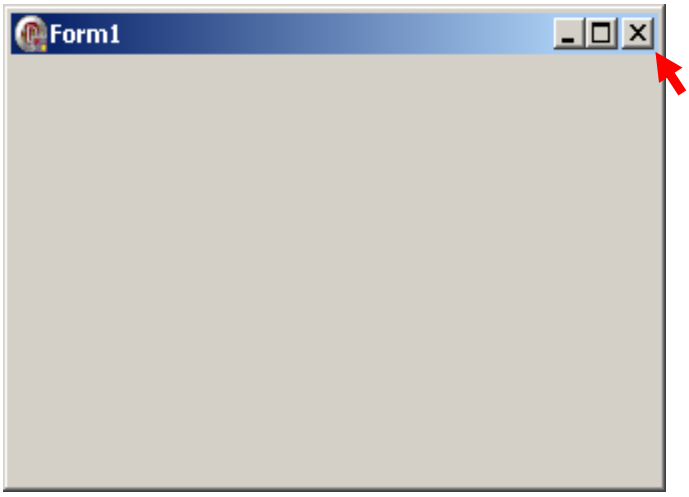
Nyní máme před sebou skutečné vývojové prostředí (Obr. 3.6), které si jen stručně popíšeme, abychom si již mohli vyzkoušet náš první program.
Těsně pod titulkovým pruhem se nachází „Menu“, z kterého můžeme vybírat jednotlivé položky. Důležité ovládací prvky jsou uspořádány v panelu pod Menu. *Nás budou ….*

Anebo ne, zkusíme si první program spustit už nyní. Spuštění vytvořeného programu se dá provést několika způsoby:

- stiskem tlačítka F9 na klávesnici

- kliknutím myši na tlačítko označené podobným symbolem jako například na MP3 přehrávači 
- kliknutím myši na položku „Run“ v menu a pak z nabídky výběrem položky také označené slovem „Run“ (běh programu).

Nyní již vývojové prostředí zajistí překlad zdrojového kódu do kódu strojového a spuštění programu. Výsledek je vidět na obrázku 3.4.



Obr. 3.4 Výsledek (výstup) našeho prvního programu

Řeknete si:

- Vždyť to nic nedělá, jen se objevilo nějaké okno a co s tím?
- A kde je nějaký program?
- Co se vlastně programovalo? Přece jsme nic takového nedělali.

Odpověď:

Opravdu výsledek našeho prvního programu nic nedělá. Nemá vlastně co řešit, protože jsme mu nezadali žádný algoritmus, který by měl vykonat. Přesto se vytvořilo vcelku estetické okno, které již známe z Windows, a to skutečně není málo.

Vývojové prostředí se již postaralo o to, abychom se nemuseli zabývat formálními činnostmi, abychom měli vytvořený kvalitní design naši budoucí aplikace a jen se soustředili na implementaci vlastního algoritmu.

A kde je tedy program, který jsme spustili? Nalezneme jej po kliknutí myši na záložku „Code“, která je umístěna ve spodní části vývojového prostředí, vedle záložky „Design“, jejímž výběrem si naopak zobrazíme opět design budoucí aplikace.

Každý programovací jazyk má svá přísná a nekompromisní pravidla. Také v Delphi se musíme zcela podřídit jeho pravidlům. Povinnou minimální formální strukturu programu v Delphi, tak jak jsme ji získali kliknutím myši na záložku „Code“, vidíme na obr. 3.5.

Stručně si ji popíšeme, i když ji vývojové prostředí Delphi vytvoří vlastně za vás.

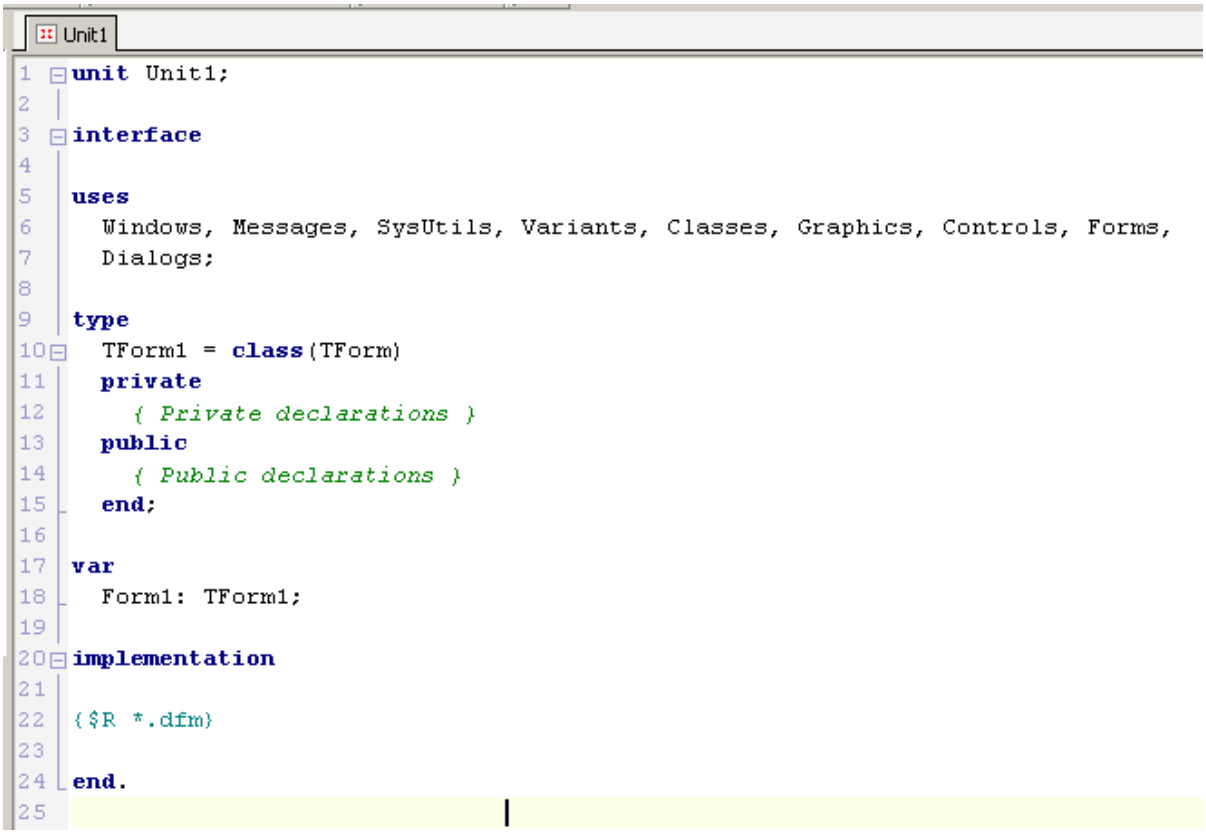
Struktura programu

Vytvořili jsme vlastně programovou jednotku, proto první řádek začíná slovem „unit“ (jednotka) a názvem jednotky, který je implicitně *Unit s číslem* (v našem případě Unit1). Řádek je povinné ukončen středníkem. Interface znamená rozhraní (nebude nyní vysvětleno). Další položkou bývá „uses“, která uvádí použité programové jednotky (někdy nazývané knihovny). Jejich skladba nemusí odpovídat našemu příkladu. V zásadě jde o programy, které již byly vývojáři dříve vytvořené pro zjednodušení programování. Pokud tedy do výčtu uvedeme nějakou takovou jednotku, můžeme pak používat její funkce, aniž bychom je museli programovat. Jednotlivé jednotky je nutno oddělit čárkami a za poslední je rovněž povinnost umístit středník.

Konstrukci uvedenou slovem „type“ zde uvedeme jen poznámkou, že obsahuje část zdrojového kódu, kde se určují (definují) typy různých ***datových typů***.

V oddílu „var“ jsou deklarovány ***proměnné*** (variable) tím, že je jim přiřazen konkrétní ***identifikátor*** (název) a ***datový typ***.

Po slově „implementation“ již následuje vlastní tělo programu, tedy výkonná část programu. Zde se budeme snažit implementovat náš algoritmus. Konec zdrojového kódu patří slovu „end.“, za kterým je zde povinně tečka. Co je napsáno za tečkou, na to není brán zřetel.



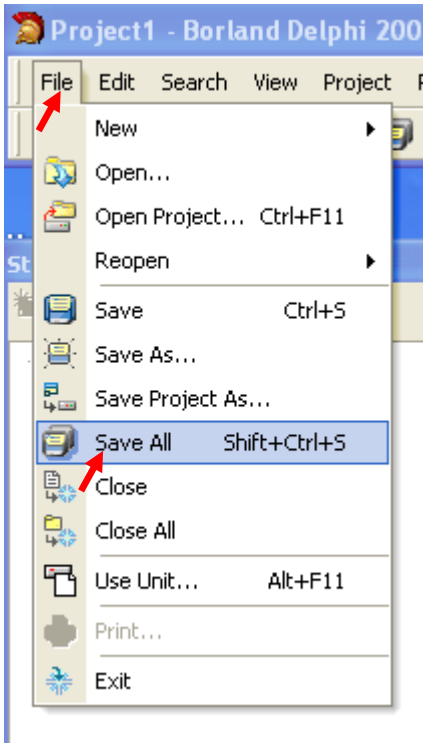
Obr. 3.5 Výpis zdrojového kódu automaticky vytvořeného v prostředí programu Delphi

Ukládání a otevírání projektů v Delphi

Náš program sice ještě nic nedělá, ale je žádoucí každý program již v této podobě uložit na nějaké záznamové medium. Na rozdíl od programu například v Pascalu, kde jsou veškeré potřebné údaje pro jeho spuštění umístěné jen v textu zdrojového kódu, pak tím, že v Delphi programové aplikace podporují grafické prostředí Windows, jsou veškeré potřebné informace rozloženy do několika souborů různých typů. Vlastní zdrojový kód je doplněn o další grafické, zdrojové a systémové informace, takže již nehovoříme o programu, ale aplikace v Delphi se nazývá projekt, a ten je uložen do těchto souborů nesoucích název projektu, případně jednotky (unit) a lišících se příponou:

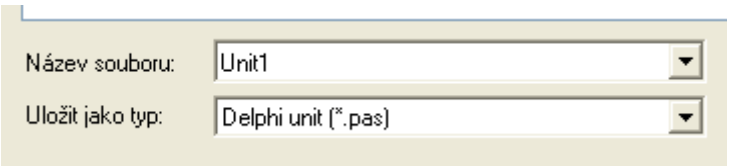
Hlavním souborem projektu je soubor s koncovkou DPR, vlastní zdrojový kód (unit) se nachází stejně jako programy v Pascalu v souboru s koncovkou PAS. Definice formuláře jsou ukládány do souboru s koncovkou DFM, soubor se zdroji má koncovku RES a nastavení parametrů je uchováno v souboru s koncovkou CFG. Po překladu se ještě navíc vytvoří

spustitelný soubor aplikace, který má koncovku EXE, a který je určen pro spouštění aplikace již bez vývojového prostředí tak, jako každé jiná aplikace (Word, Excel apod.).

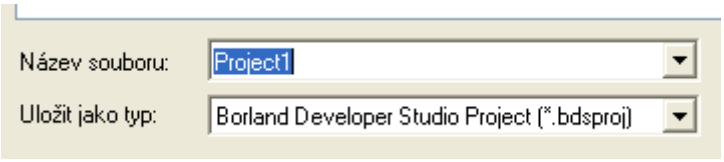


Obr. 3.6 Uložení projektu

Námi vytvořený nový program je jen v paměti počítače, k uložení celého projektu je třeba použít volbu File z Menu a následně zvolit položku Save All (Obr. 3.6). Stejný účinek má i tlačítko s několika disketami  . V obou případech je vyvolán klasický dialog pro uložení souboru. Je vhodné vytvořit pro každý projekt vlastní složku. Všechny soubory se uloží až po obsluze dvou dialogů.



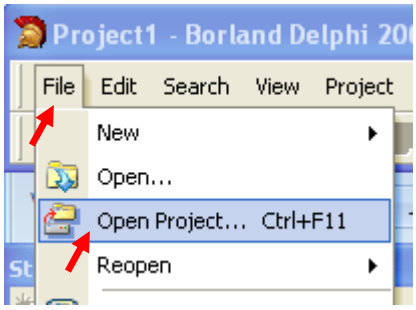
První má za následek uložení souborů Unit1.dfm a Unit1.pas.



Druhý je použit pro uložení soborů Project1.dpr, Project1.res, Project.cfg a dva další podpůrné soubory.

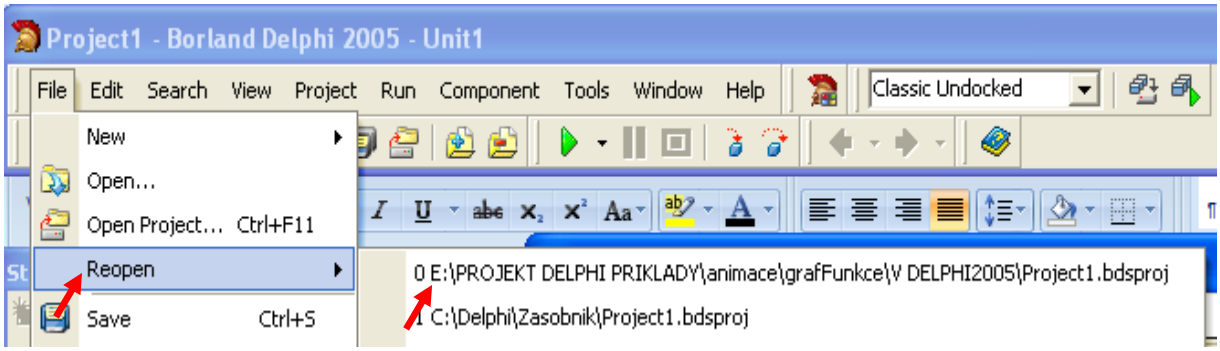
Po překladu nebo spuštění projektu se do složky umístí i soubor Project1.exe.

Dbejte úzkostlivě na to, aby byly opravdu uloženy úplně všechny soubory. Ztráta jakéhokoli z nich nedovolí otevření a další úpravy projektu ve vývojovém prostředí. Během vývoje projektu průběžně ukládejte výsledky své práce. Vhodné je, dílčí funkční stádia uložit do oddělených složek označených datem.

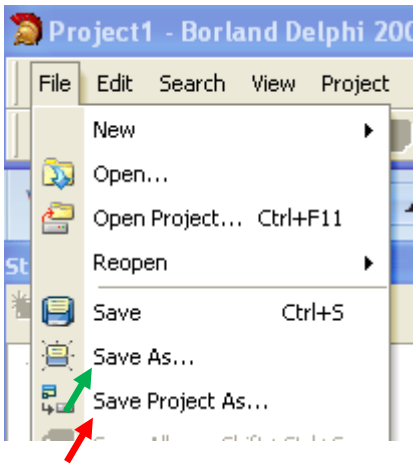


Pro otevření uloženého projektu ve vývojovém prostředí je možno jednak použít volbu File z Menu a následně zvolit položku Open Project..., nebo klávesové zkratky Ctrl+F1. Rovněž je možno využít tlačítko se složkou a diskem.  . Vždy bude vyvoláno komunikační okno pro

otevření souboru. Naším úkolem je pak jen zvolit odpovídající složku, kde jsou soubory projektu uloženy a vybrat soubor Projekt1.dpr.



Volba Reopen nabídne výběr z posledně vyvíjených projektů v tomto vývojovém prostředí. Stačí si jen najít ten, který chceme upravovat a kliknutím myši otevřít.



Při využití volby Save All, jsou všechny soubory opět uloženy do původního umístění. Chceme-li soubory přejmenovat (z praktických důvodů nedoporučujeme) nebo je uložit do jiné složky, je nutno použít postupně voleb Save As… (označeno zelenou šipkou) a Save Project As… (červenou šipkou), které jsou také dostupné z volby File z Menu. I zde nezapomeňte, že musíte uložit nejen soubory označené Unit1.*, ale také soubory označené Project1.*

Jak pracuje program vytvořený v Delphi.

Již se neobejdeme bez trochy teorie, abychom si vysvětlili, jak program v programovacím jazyku Delphi vlastně pracuje. V Delphi vzniká tzv. ***program řízený událostmi*** *(event driven).* O co tedy jde? Je-li program spuštěn, navenek nic nedělá, dokud nezaznamená nějakou událost (např. stisk klávesy na klávesnici). Pokud je v programu návod, jak má být událost ošetřena, počítač ji podle tohoto algoritmu zpracuje. Pokud se tam žádný návod nenachází, pak událost ignoruje.

Tak tomu bylo i při spuštění našeho prvního programu. Objevilo se okno, ale mohli jsme dělat cokoli s výjimkou několika manipulací s myší a nic se nedělo. A přece se okno dalo tažením myši přesunout, dalo se minimalizovat kliknutím do minimalizačního tlačítka, libovolně

zmenšovat a zvětšovat a dokonce zavřít, vše podle pravidel, která již známe z prostředí Windows, a to aniž jsme cokoli programovali. Je to tím, že v Delphi se pracuje s třídami, jejichž jednou z vlastností je i dědičnost. Avšak vysvětlení této problematiky již přesahuje rámec této publikace.

Jak lze vyvolat určitou událost už víme, ale jak do programu vložit algoritmus její obsluhy nám zatím zůstává záhadou. Pomůže nám, když si vzpomeneme na symboly vývojových diagramů. Jeden z nich byl označen jako „podprogram“ někdy se podprogram označuje slovem ***procedura***. V podstatě jde o samostatný program, řešící nějakou specifickou a často se opakující problematiku. Opatříme-li ho označením, že jde o proceduru a jedinečným názvem, můžeme mu předat řešení toho, co umí a následně využít jeho výsledky, kdykoliv to potřebujeme.

Právě ***procedura*** je vhodným programovacím nástrojem na ošetření událostí. Kdykoli se objeví definovaná událost, je předána její obsluha příslušné proceduře, v té je zpracována a systém dále čeká na další událost.

Do těla programu tedy nemůžeme přímo psát vytvořený algoritmus (tak jak to bylo běžné například v programovacím jazyce Pascal), ale budou do něj postupně vkládány procedury se svým názvem, vlastnostmi a údaji odpovídajícími konkrétní události. Proto se tyto procedury odborně nazývají ***procedury událostí***. Vložení prázdné procedury události do vymezeného místa programu je samočinně provedeno vývojovým prostředím dle volby programátora. Vytvořené algoritmy se pak vkládají do těla vytvořené procedury událostí.

Jako příklad je zde uveden formální fragment programu, který znázorňuje umístění procedury ošetření kliknutí (*Click*) na tlačítko (s označením *Button1*), které je umístěné v ploše okna (*Form1*) zobrazeného na monitoru. Ten musí být umístěn do těla programu (mezi *implementation* a *end.*).

```
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
   begin
      // Zde se vkládá zdrojový kód vytvořeného algoritmu, který řeší, co se má provést,
      //když nastala událost stisku tlačítka (klik myši na tlačítko) umístěného v okně.
   end;
end.
```

Výrazu {$R *.dfm} nevěnujte pozornost, jde jen o direktivu překládači a pro naše účely nemá žádný význam.

Button1 obdobně jako celá řada jiných objektů v Delphi se nazývá ***komponenta***. Vzhledem k tomu, že o komponentách bude ještě řeč později, omezíme se zde jen na stručný a méně přesný komentář, že komponenty představují ovládací prvky (tlačítka, zaškrtávací políčka, přepínače, editační okna apod.), které využívají aplikace pod operačním systémem MS Windows (OS MS Windows) pro komunikaci s obsluhou (vyvolání události, vstupy a výstupy údajů) nebo pro jiné činnosti.

Možná vás zarazily tmavě zelené nápisy v těle procedury. Ne, nejde o přepis algoritmu. Jsou to jen komentáře. *Mimochodem naučte se komentáře používat i při psaní jednoduchých programů, jednou až budete psát ty složitější, budete určitě odměněni za ten dobrý zvyk.*

Ale aby překládač nebral komentáře jako nesmyslný program, je nutno mu sdělit, že jde jen o komentář, kterému nemusí vůbec věnovat pozornost a může pokračovat za komentářem dál v překladu. Zde bylo použito dvou lomítek (//), která překládači říkají, že do konce řádku je jen komentář. Proto lomítka musí být i u druhého komentáře na druhém řádku. Dlouhý komentář na více řádku je výhodnější zapsat do složených závorek { }.

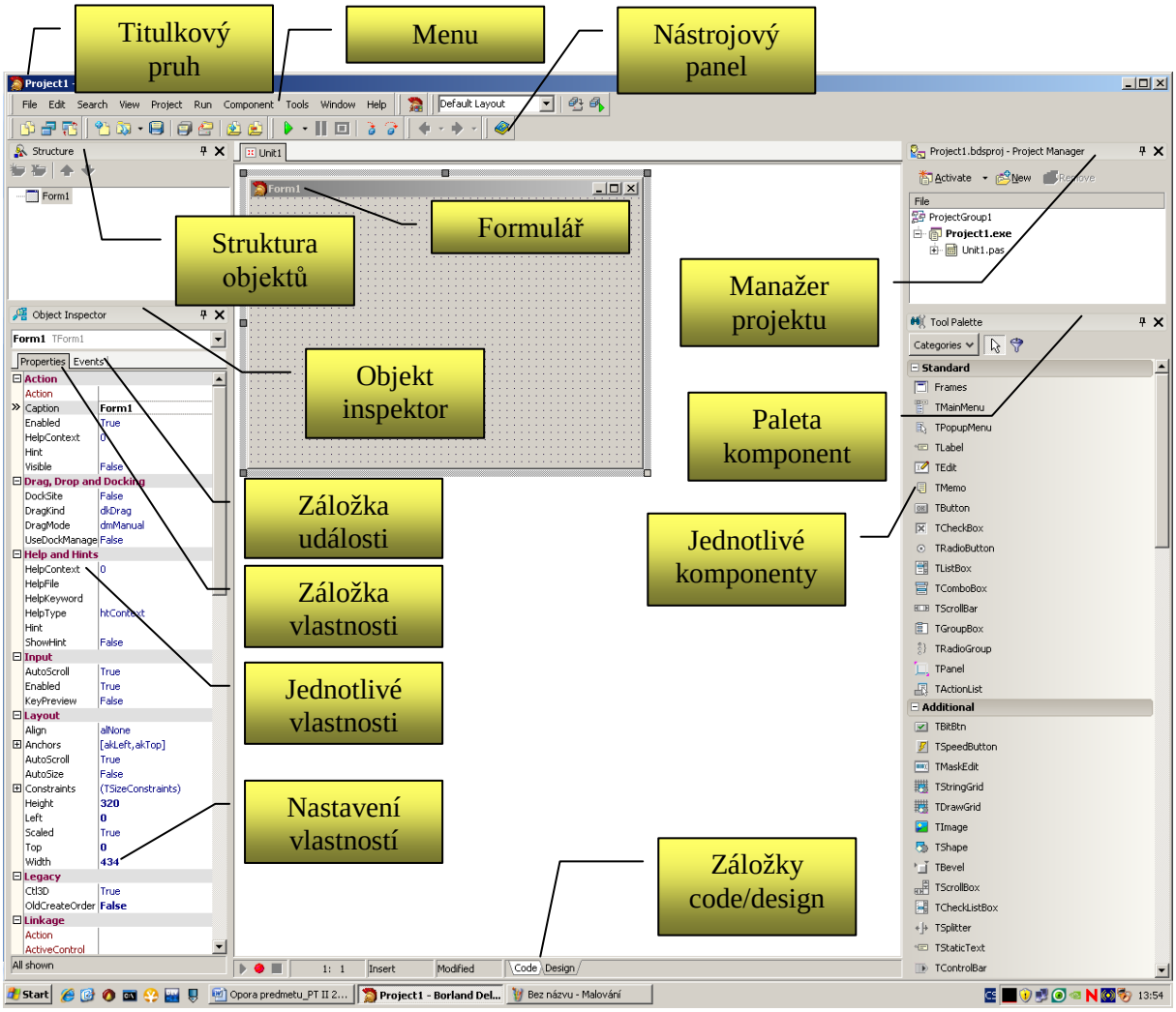
Náš předešlý komentář lze tedy zapsat i takto:

```
{ Zde se vkládá zdrojový kód vytvořeného algoritmu, který řeší, co se má provést,
  když nastala událost stisku tlačítka (klik myši na tlačítko) umístěného v okně. }
```

POZOR!

Mimo tuto proceduru události v těle programu, vygeneruje automaticky uživatelské prostředí ještě v hlavičce programu v části označené „type“ informaci o existenci a datovém typu této procedury události. Protože jsou obě části formálně svázány, vymazaní jedné z nich způsobí chybu při překladu programu. To bývá velmi častá chyba při editaci programu. Pokud jsme nějakou takovou proceduru události nechali vývojovým prostředím vygenerovat, není vhodné ji mazat celou, lépe je vymazat její obsah (mezi begin a end;) a pokud překládač zjistí, že vlastně není využita, sám ji korektně odstraní.

Popis vývojového prostředí



Obr. 3.6 Vývojové prostředí programu **Borland® Delphi™ 2005**

Pokud nechceme při spuštění programu zůstat jen u prázdného okna, je nejvyšší čas seznámit se s funkcemi a strukturou vývojového prostředí (obr. 3.6).

Základním prvkem je zde „formulář“, který je implicitně označen *Form a číslo* (v našem případě Form1). Jeden se vytváří automaticky při otevření vývojového prostředí s novým projektem (tak jak jsme již v úvodu udělali). Tento formulář se nám vlastně po spuštění programu objeví na monitoru jako klasické okno pod OS MS Windows. Proto je možné s tímto oknem pracovat jako s každým jiným. Není-li to zakázáno, můžeme oknem posouvat

tahem myši, můžeme měnit jeho rozměry táháním za hrany, můžeme ho minimalizovat, maximalizovat a konečně i zavřít, čímž zároveň ukončíme činnost programu.

Shrnutí pojmů

Editor – program určený k editaci (případně opravám) zdrojového kódu.

Překládač - program, který překládá zdrojový kód napsaný v programovacím jazyce do strojového kódu, v kterém pracuje počítač.

Linkér – program, který spojuje přeložený zdrojový kód z překládače s ostatními potřebnými již přeloženými programy ve výsledný funkční program spustitelný na počítači.

Debugger **–** program, který je určen pro usnadnění ladění programu. Usnadňuje nalezení a odstranění logických chyb při běhu programu.

Program řízený událostmi *(event driven) je* program, který navenek nic nedělá, dokud nezaznamená nějakou událost (např. stisk klávesy na klávesnici). Pokud je v programu návod, jak má být událost ošetřena, počítač ji podle tohoto algoritmu zpracuje. Pokud se tam žádný návod nenachází, pak událost ignoruje.

Otázky

1. Co je to vývojové prostředí a k čemu je určeno?
2. Jak se vytváří nový projekt?
3. Jak se projekt ukládá na záznamové médium?
4. Jak se projekt otevírá ze záznamového média?

Úlohy k řešení

Podrobně se seznamte s vývojovým prostředím programu Borland® Delphi™ 2005 a procvičte si vytvoření nového projektu, otevření a uložení projektu s kontrolou uložených souborů.

4. KOMPONENTY

4.1. Charakteristika komponent

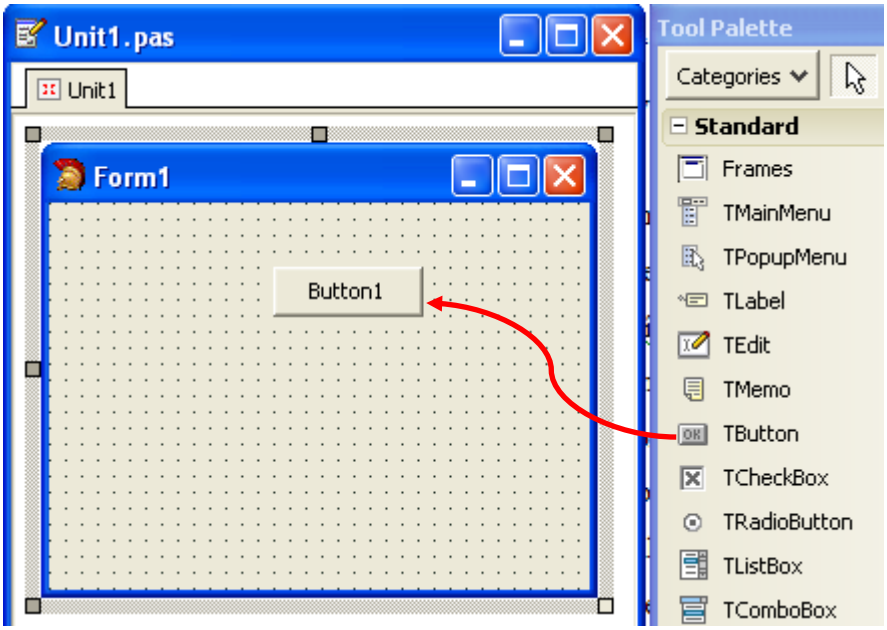
Čas ke studiu: 0,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- popsat základní charakteristiky komponent.
- vkládat a rušit komponenty.
- manipulovat s komponentami v rámci formuláře.

Výklad

Vedle již dříve zmíněné roletové nabídky (Menu) a panelu ovládacích nástrojů (Toolbar) se obvykle vpravo objeví okno označené „Paleta nástrojů“ (Tool Palette), které obsahuje kategorizovaný seznam komponent, o kterých již byla dříve zmínka. Nyní si o nich řekneme něco více.

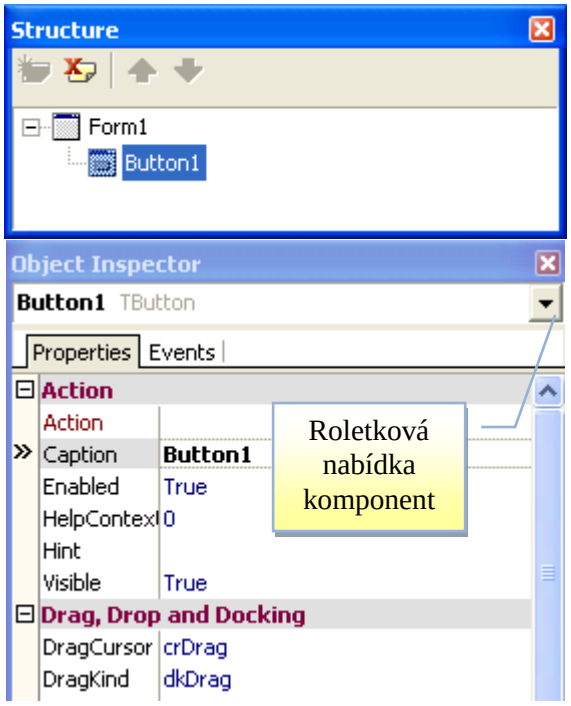


Obr. 4.1 Přetažení komponenty do formuláře

Komponenta je nástroj (malý program), který když chceme použít, musíme ho umístit do formuláře (obr. 4.1). Postup je „Window-sovský“ jednoduchý. Vybereme si určitou komponentu (třeba již zmíněné tlačítko). Protože jde o třídu tlačítek, začínají názvy všech komponent velkým písmenem T. Vybereme tedy komponentu s označením TButton. Funkci jednotlivých komponent můžeme vytušit z přidělené ikony. Ikonku „přichytíme“ myší a táhneme ji do

požadovaného místa ve formuláři, kde ji „upustíme“. Komponenta se ve formuláři rozvine do odpovídajícího objektu. Nežádoucí komponenty ve formuláři se zbavíme tak, že ji nejprve označíme myši a následně ji zrušíme stiskem tlačítka „Delete” z klávesnice. Každá přidaná komponenta do formuláře způsobí, že se automaticky doplní údaje v hlavičce zdrojového kódu (unit), kde se v sekci **type** uvede datový typ komponenty.

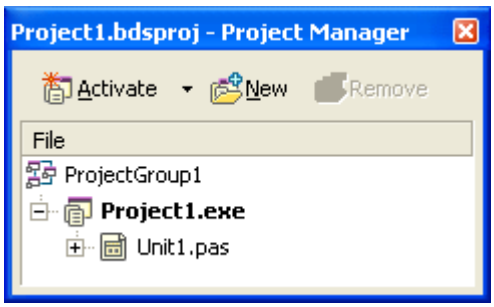
Každá komponenta je definována svou činností (metodami – Metods), kterou je schopna po aktivaci provést, vlastnostmi (Properties), které je možno měnit a událostmi (Events), které je schopna ošetřit.



Obr. 4.2 Okna Structure a Object Inspector

Vlevo se obvykle nachází okno nazvané „Inspektor objektů“ (Object Inspector). V tomto okně se zobrazují aktuální vlastnosti a události vybrané (například ve formuláři myší označené) komponenty (obr. 4.2).

Záložkami Properties a Events lze přepínat mezi zobrazením vlastností nebo událostí vybrané komponenty. Můžeme si je prohlížet, ale hlavně nastavovat. Výběr (označení) komponenty je možno provést i podle názvu v roletkové nabídce okna „Inspektora objektů“ (Object Inspector), která obsahuje abecední seznam použitých komponent. Význam takového způsobu výběru je opodstatněný hlavně tehdy, když se některé komponenty ve formuláři překrývají, takže výběr kliknutím myši je nemožný.



Obr. 4.3 Project Manager

Přehled všech komponent, které jsme umístili do formuláře, je uveden ve stromové struktuře okna označeného slovem „Structure“, který je obvykle umístěn nad oknem Inspektora objektů (Object Inspector). Klikneme-li zde myší na název některé komponenty, ta se nám označí i ve formuláři a tím se rovněž zobrazí její vlastnosti příp. události v okně Inspektora objektů (Object Inspector).

Vpravo, nad oknem Palety komponent (Tool Palette) se obvykle nachází okno (obr. 4.3) s názvem Manažer projektu (Project Manager). Manažer projektu je nástroj vývojového prostředí, který obvykle slouží k zobrazení vztahů mezi soubory otevřeného projektu. Jeho význam roste s rostoucí složitostí struktury projektu, avšak pro naše účely nebude zapotřebí se jim hlouběji zabývat.

Vizuální komponenty (ty, které se po spuštění aplikace zobrazí v okně aplikace), které jsme vložili do formuláře, můžeme snadno vizuálně modifikovat (měnit jejich vlastnosti). Vyzkoušejte si, že je lze „uchopit“ pomocí myši a přenést do jiného místa formuláře, kliknutím označit a následně měnit jejich rozměr tažením za značky. Všimněte si, že se při těchto operacích zároveň budou měnit hodnoty odpovídajících vlastností v inspektoru objektů. Přesunutím komponenty se změní alespoň jedna z vlastností „*Top*“ nebo „*Left*“, které určují souřadnice levého horního rohu komponenty ve formuláři. Pozor na dvojité kliknutí na komponentu. Mohla by vás překvapit reakce počítače. Pravděpodobně by vám zmizel formulář a objevil by se zdrojový kód, v kterém by se vytvořila nějaká procedura událostí dané komponenty. Pokud se vám něco takového stane, proceduru se nesnažte smazat, ale jednoduše do ní nic nepište a projekt spusťte (třeba klávesou F9). Procedura se sama korektně odstraní.

Shrnutí pojmů

Paleta nástrojů (Tool Palette), obsahuje kategorizovaný seznam komponent.

Komponenta je nástroj (malý program), který když chceme použít, musíme ho umístit do formuláře.

Každá komponenta je definována svou ***činností*** (metodami – Metods), kterou je schopna po aktivaci provést, ***vlastnostmi*** (Properties), které je možno měnit a ***událostmi*** (Events), které je schopna ošetřit.

V okně (Object Inspector) se zobrazují aktuální vlastnosti a události vybrané komponenty.

Přehled všech komponent, které jsou umístěny do formuláře, je uveden ve stromové struktuře okna označeného slovem ***Structure***.

Manažer projektu (Project Manager) je nástroj vývojového prostředí, který obvykle slouží k zobrazení vztahů mezi soubory otevřeného projektu.

Otázky

1. Co je to paleta nástrojů?
2. Co je to komponenta?
3. Jak se vkládá do formuláře?
4. Jak se korektně komponenta odstraní?
5. Co je to Inspektor objektů a k čemu slouží?

Úlohy k řešení

1. Vytvořte si formulář s různými komponentami a vyzkoušejte si měnit jeho design.

4.2. Základní vlastnosti komponent

Čas ke studiu: 0,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat a rozdělit základní vlastnosti komponent.

Výklad

Základní vlastností všech komponent je vlastnost *„Name“* je to jedinečné jméno komponenty, které se objeví ve stromové struktuře okna „Structure“. Při vkládání komponent do formuláře se automaticky generují názvy komponent s rostoucím pořadovým číslem (Button1, Button2 …, Label1, …). I když je možné tyto názvy změnit, tak pro naše účely to nebudeme dělat. Pokud byste přece jen chtěli komponenty pojmenovat vystižnějším názvem, doporučujeme v prvních třech písmenech charakterizovat typ komponenty a teprve následně přidat vystižný název dle určení komponenty v projektu. Názvy v Delphi nesmí obsahovat mezery a českou diakritiku, takže přijatelným se jeví např. označení BtnProvedVypocet (vidíme, že jde o tlačítko, po jehož „stisku“ se má provést nějaký výpočet). U všech komponent můžeme ještě

nalézt vlastnost „*Tag*“, což je určitá celočíselná speciální položka, kterou můžeme měnit a mohla by sloužit k identifikaci komponenty.

Obr. 4.4 Ukázka „ShowHint“

Všechny vizuální komponenty mají ještě vlastnost „*Visible*“, která nese dvouhodnotovou informaci, zda je objekt viditelný (hodnota true) nebo není vidět (false), vlastnost „*ShowHint*“, která povoluje (true) nebo nepovoluje (false) zobrazit na několik sekund bublinovou nápovědu při umístění kurzoru myši nad komponentu a vlastnost „*Hint*“, která obsahuje vlastní text bublinové nápovědy (v ukázce na obr. 4.4 byla vlastnost „*Hint*“ nastavena na „To je komponenta tlačítko“). Jsou pro ně rovněž typické již výše zmíněné vlastnosti „*Top*“ a „*Left*“, které určují polohu komponenty (číselné hodnoty rostou zleva doprava a shora dolů) a dvě vlastnosti určující rozměr komponenty „*Width*“ (šířka komponenty) a „*Height*“ (výška komponenty).

Další vlastnosti komponent jsou specifické pro určité skupiny nebo dokonce i jednotlivé komponenty podle činnosti, kterou v programu vykonávají, a probereme je později.

Určité vlastnosti komponent (poloha, rozměr) je možno měnit v průběhu návrhu přímo myší ve formuláři, mnohem víc vlastnosti nastavíme v inspektoru objektů. Vlastnosti jde dokonce nastavovat i za běhu aplikace (softwarově).

Shrnutí pojmů

Základní vlastnosti komponent

Name je jedinečné jméno komponenty.

Tag je určitá celočíselná speciální položka.

Visible nese dvouhodnotovou informaci, zda je objekt viditelný nebo ne.

Hint obsahuje text bublinové nápovědy.

ShowHint povoluje nebo nepovoluje zobrazit na několik sekund bublinovou nápovědu při umístění kurzoru myši nad komponentu.

Top a ***Left*** určují polohu levého horního rohu komponenty.

Width určuje šířku komponenty.

Height určuje výšku komponenty.

Otázky

1. Kterou vlastností ovlivňujeme polohu komponenty?
2. Kterou vlastností ovlivňujeme velikost komponenty?
3. Kterou vlastností ovlivňujeme viditelnost komponenty?
4. Kterou vlastností ovlivňujeme bublinovou nápovědu komponenty?

Úlohy k řešení

1. Vytvořte si formulář s různými komponentami a inspektoru objektů proveďte modifikace jejich vlastností. Po spuštění aplikace zkontrolujte funkce Visible, ShowHint, Hint apod.

4.3. Základní události komponent

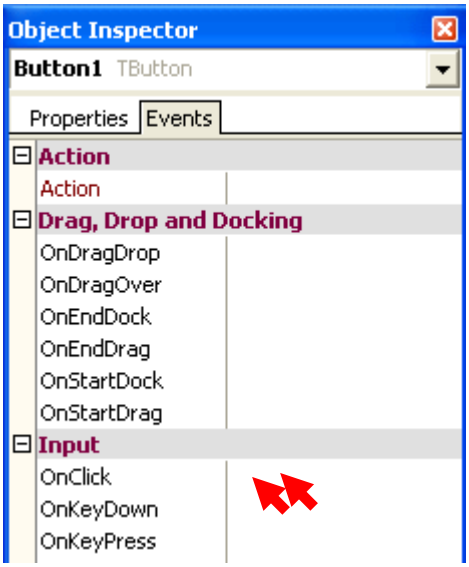
Čas ke studiu: 1,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- popsat základní události komponent.
- použít proceduru událostí.

Výklad

Jak jsme si již dříve vysvětlili, aby naše aplikace vyvíjela nějakou činnost, musí být vyvolána nějaká událost a zároveň musí být v programu nějaká procedura události, která ji obslouží. Podobně jako seznam vlastnosti jednotlivých komponent se liší, také není stejný seznam událostí, které jsou jednotlivé komponenty schopny obsloužit. Proto si zde uvedeme jen ty, s kterými se při výuce nejčastěji setkáte.



Obr. 4.5 Vložení procedury události

„*OnClick*“ - identifikuje událost kliknutí myší na komponentu, nejčastěji se používá pro kliknutí na komponentu Button. Postup na ošetření události a tím vytvoření procedury události v programu je jednoduchý. Vybereme komponentu, v inspektoru objektu (Object Inspector) klikneme na záložku událostí (Events), najdeme odpovídající událost (zde „*OnClick*“) a provedeme „dvojklik“ v řádku OnClick v pravém sloupci (jak je znázorněno na obr. 4.5). V inspektoru objektů se objeví název procedury

události, která bude volána po příchodu události kliknutí na tlačítko Button1. Zároveň se v těle zdrojového kódu vytvoří procedura s identifikací v hlavičce v sekci **type**:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
....
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Stejným postupem můžete aplikovat do programu další procedury události.

„*OnDblClick*“ identifikuje událost dvojitého kliknutí myší na komponentu, nelze jej použít pro tlačítko, ale jeho využití může být například ve spojení s formulářem. Pak obdržíme následující proceduru události:

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

end;
```

„*OnChange*“ identifikuje událost změny položky v komponentě. Pro komponenty pomocí kterých například ručně zadáváme vstupní údaje, může být využití této události účelné.
Pak můžeme například obdržet následující proceduru události (pro komponentu Edit):

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
end;
```

„*OnMouseMove*“ identifikuje událost pohybu myši nad komponentou a současně dává k dispozici souřadnice její polohy v rámci komponenty a stav tlačítka Shift. Může být například použita ke kreslení do formuláře (pro kreslení je však vhodnější komponenta Image).
Procedura události pro komponentu Form1 pak bude vypadat takto:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
end;
```

„*OnMouseDown*“ identifikuje událost stlačení tlačítka myši nad komponentou a současně dává k dispozici souřadnice polohy myši v rámci komponenty, které tlačítko myši bylo stlačeno a stav tlačítka Shift. Využití je obdobné jako u předchozí události. Procedura události pro komponentu Form1 pak bude vypadat takto:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
end;
```

„*OnMouseUp*“ identifikuje událost uvolnění tlačítka myši nad komponentou a současně dává k dispozici souřadnice polohy myši v rámci komponenty, které tlačítko myši bylo stlačeno a stav tlačítka Shift. Využití je obdobné jako u předchozí události. Procedura události pro komponentu Form1 pak bude vypadat takto:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
end;
```

„*OnCreate*“ identifikuje událost vytvoření objektu. Často je využívána ve spojení s formulářem, kde slouží k indikaci spuštění aplikace, kdy se vytváří formulář. Do takto vytvořené procedury událostí se vepisují příkazy, které je vhodné vykonat hned po spuštění programu (počáteční nastavení). Procedura události pro komponentu Form1 pak bude vypadat takto:

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

„*OnClose*“ identifikuje událost uzavření objektu. Nejčastěji je využívána ve spojení s formulářem, kde slouží k indikaci ukončení aplikace (konce programu). Většinou se pak do procedury události vkládají příkazy zprávy o ukončení programu s volbou, zda souhlasíme. Procedura události pro komponentu Form1 pak bude vypadat takto:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
end;
```

Možná, že by pro vás mohly být zajímavé i následující události. Pokuste se zjistit, jak by se daly využít:

„*OnHide*“	indikace skrytí objektu,
„*OnShow*“	indikace zviditelnění objektu,
„*OnKeyPress*“	indikace stisku alfanumerické klávesy z klávesnice
„*OnKeyDown*“	indikace stisku jakékoliv klávesy z klávesnice
„*OnKeyUp*“	indikace uvolnění jakékoliv klávesy z klávesnice

Shrnutí pojmů

„*OnClick*“	indikace události kliknutí myší na komponentu,
„*OnDblClick*“	indikace události dvojitého kliknutí myší na komponentu,
„*OnChange*“	indikace události změny položky v komponentě,
„*OnMouseMove*“	indikace události pohybu myši nad komponentou,
„*OnMouseDown*“	indikace události stlačení tlačítka myši nad komponentou,

„*OnMouseUp*“	indikace události uvolnění tlačítka myši nad komponentou,
„*OnCreate*“	indikace události vytvoření objektu,
„*OnClose*“	indikace události uzavření objektu,
„*OnHide*“	indikace skrytí objektu,
„*OnShow*“	indikace zviditelnění objektu,
„*OnKeyPress*“	indikace stisku alfanumerické klávesy z klávesnice,
„*OnKeyDown*“	indikace stisku jakékoliv klávesy z klávesnice,
„*OnKeyUp*“	indikace uvolnění jakékoliv klávesy z klávesnice.

Otázky

1. Které události můžeme použít při práci s myší?
2. Která událost je vhodná pro počáteční nastavení parametrů programu?

Úlohy k řešení

1. Vytvořte si formulář s různými komponentami a vyzkoušejte, jaké procedury událostí se vytvoří při dvojkliku na ně.

4.4. Základní komponenty

Čas ke studiu: 2,0 hodin

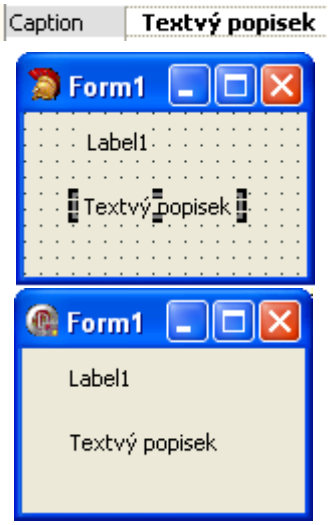
Cíl: Po prostudování tohoto odstavce budete umět

- definovat a rozdělit základní komponenty.
- nastavovat základní vlastnosti komponent.
- využít odpovídající komponenty pro daný účel.

Výklad

V předchozí kapitole jsme se seznámili s několika základními vlastnostmi a událostmi komponent. Abychom je však mohli využívat, musíme znát jejich funkci (význam). V Delphi se vyskytuje velké množství komponent. I když je jejich počet závislý na verzi a typu vývojového prostředí (úroveň využití) a typ Delphi 2005 Personal je na nejnižší úrovni, přesto by výčet a podrobné vysvětlení všech komponent, kterými disponuje, zabralo neúměrné množství času a místa. Komponenty si navíc může vytvářet sám programátor a mnoho jich najdete na internetu. To nás nutí zabývat se jen těmi nejdůležitějšími komponentami a studium zbývajících tak necháme na každém z vás.

Nejprve si probereme komponenty, které jsou určené pro výstup nebo vstup textových údajů.



Obr. 4.6 Komponenta Label

Label – komponenta, jejíž název bývá překládán jako „návěští“ nebo „popisek“. Jejím úkolem tedy bude zobrazování nějakého textu. Nejdůležitější vlastnosti této komponenty je vlastnost „*Caption*“. Hodnota této vlastnosti obsahuje text, který má být zobrazen jak je vidět na Obr.4.6. Další vlastnosti komponenty Label umožňují modifikovat vzhled komponenty Patří mezi ně:

„*Color*“ – určuje barvu pozadí v rámci komponenty

„*Font*“ – určuje font písma, nastavení se může provádět (podobně jako např. ve Wordu) v dialogovém okně.

„*WordWrap*“ – určuje, zda může být text v komponentě zalomený (volba true) nebo ne (volba false).

„*Transparent*“ - umožňuje nastavit průhlednost komponenty (true). Neprůhlednou komponentu zajistíme hodnotou (false).

„*Autosize*“ – nastavuje možnost automatického přizpůsobení velikosti komponenty v závislosti na délce textu (true). Vlastní určení rozměru komponenty vyžaduje nastavení na hodnotu (false).

„*Alignment*“ - umožňuje nastavit horizontální zarovnání textu. Můžeme vybírat z těchto tří možností: vlevo, vpravo a na střed.

„*Layout*“ - umožňuje nastavit vertikální zarovnání textu. Můžeme vybírat z těchto tří možností: nahoru, dolu a na střed.

„*Cursor*“ - umožňuje nastavit tvar kurzoru, vyskytuje-li se nad komponentou.

„*Enabled*“ – vlastnost by se dala charakterizovat jako aktivita komponenty.

Další významné vlastnosti jako je „*Name*“, „*Left*“, „*Top*“, „*Height*“, „*Width*“ a „*Visible*“ již byly vysvětleny dříve, neboť mají obecný charakter. Další textové komponenty budou mít také mnohé vlastnosti shodné s komponentou Label, proto se zde omezíme jen na podstatné rozdíly nebo nové vlastnosti.

Edit - komponenta, jejíž název již napovídá k čemu je určena. Podobně jako předchozí komponenta může sloužit jako výstup textových údajů, ale umožňuje navíc zadávat ručně text jakožto vstupní údaje určené ke zpracování.

Novými vlastnostmi zde jsou:

„*Text*“ - jde o základní vlastnost obdobně jako u komponenty Label vlastnost Caption. V hodnotě této vlastnosti se nachází text zobrazený v komponentě.

„*ReadOnly*“ – vlastnost jen pro čtení. Pokud je nežádoucí, aby obsluha mohla měnit text v komponentě, pak má-li tato vlastnost hodnotu true, je změna zakázána.

„*PasswordChar*“ - udává znak, který se bude zobrazovat místo zadávaných znaků (při zadávání hesla)

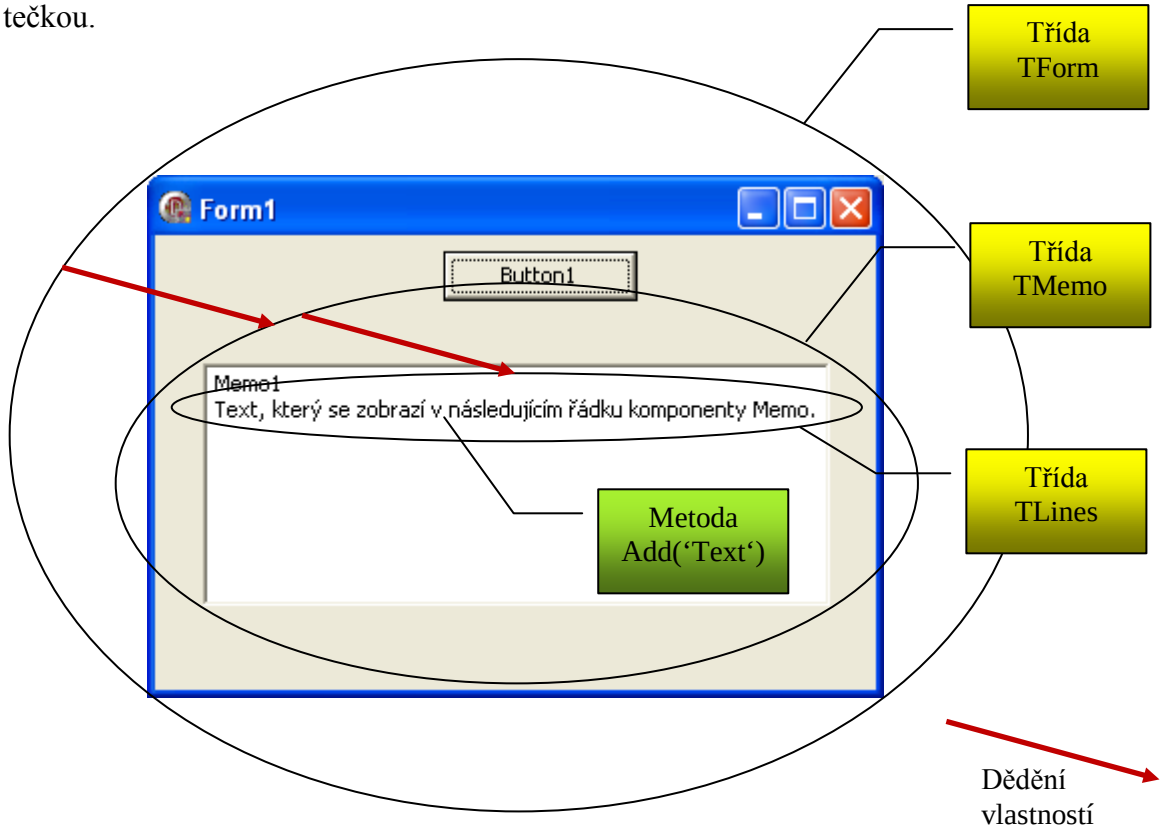
„*CopyToClipboard“* metoda, která zkopíruje obsah Editu do schránky.

Memo – komponenta s významem blízkým komponentě *Edit*. Slouží tedy jednak jako textový výstup počítače, ale můžeme do ní psát text podobně jako například do aplikace Poznámkový blok nebo Word. Vlastnosti má obdobné jako *Edit*, ale navíc disponuje spoustou nových

vlastností, které jsou přístupné pouze za běhu aplikace a je proto nutné umístit je přímo do nějaké procedury události zdrojového kódu. Konstrukce těchto příkazů vychází ze syntaxe Delphi při práci s třídami a na první pohled se může zdát složitá. Na jedné, záměrně složitější ukázce si celý problém ukážeme (obr. 4.7):

```
Form1.Memo1.Lines.Add('Text, který se zobrazí v následujícím řádku komponenty Memo.');
```

Touto konstrukcí v zásadě voláme podprogram, který má za úkol zapsat (přidat) na poslední řádek komponenty Memo1, následující text: „Text, který se zobrazí v následujícím řádku komponenty Memo.“. V příkazu ještě vidíme Form1 nalevo od Memo1, což říká, že Memo1 je komponentou (je vložena do) formuláře Form1. Oddělení jednotlivých položek je povinně tečkou.



Obr. 4.7 Vysvětlení struktury příkazu pro komponentu Memo

Podobně jak jsme vložili do komponenty nový řádek textu, můžeme za běhu programu modifikovat libovolnou vlastnost komponenty. Z toho plyne, že při vložení komponenty do formuláře se vytvoří implicitně komponenta se zděděnými vlastnostmi rodiče (nadřazené třídy) a se svými specifickými vlastnostmi. Značné množství vlastností můžeme v editačním

režimu změnit pomocí inspektora objektů. A nakonec i za běhu programu můžeme vlastnosti komponent měnit softwarově.

Abychom to ale dokázali, musíme se naučit základní příkaz jazyka Delphi, který se nazývá „***Přiřazení***“ a jeho značka je povinně „:=“ (dvojtečka a rovnítko bez mezery).

Uvedeme si zde ukázku příkazu přiřazení pro případ, že máme v úmyslu změnit barvu komponenty Memo1 na žlutou (podklad písma).

```
Form1.Memo1.Color := clYellow;
```

Tuto konstrukci si velmi dobře zapamatujte, neboť odtud se s ní budete stále setkávat. Všimněte si také, že na konci se nachází středník. Překládač podle něj pozná, že je zde konec příkazu, protože při troše fantazie by mohl předcházející příkaz vypadat i takto a být přitom funkční:

```
Form1.Memo1.Color
 :=
 clYellow
 ;
```

Tento zápis by ovšem byl značně nepřehledný, v žádném případě se jím nenechte inspirovat.

Co příkaz vlastně znamená? Mohli bychom jej přečíst asi takto: Přiřaď vlastnosti „*Color*“ komponenty ***Memo1*** hodnotu žluté barvy.“ Delphi mají předdefinovaných několik základních barev jako pojmenované konstanty, které můžeme snadno použít, tak jak je uvedeno v ukázce. První dvě písmena jsou povinná (cl – zkr. color) a po nich bez mezery následuje anglický název barvy:

```
clAqua        clBlack   clBlue    clCream   clDkGray  clFuchsia
clGray        clGreen   clLime    clLtGray  clMaroon  clMedGray
clMoneyGreen  clNavy    clOlive   clPurple  clRed     clSilver
clSkyBlue     clTeal    clWhite   clYellow
```

Pokud by nám oněch 22 barev nestačilo, je tu ještě jedná možnost - poskládat barvu z RGB složek (červená, zelená, modrá), tak jak jsou skládány na obrazovce monitoru nebo televize, abyste získali několik milionů barev. Jak se to dělá, se ale dozvíme až při studiu grafických funkcí. Pro dnešek postačí, když se na svůj monitor podíváte přes silné zvětšovací sklo a

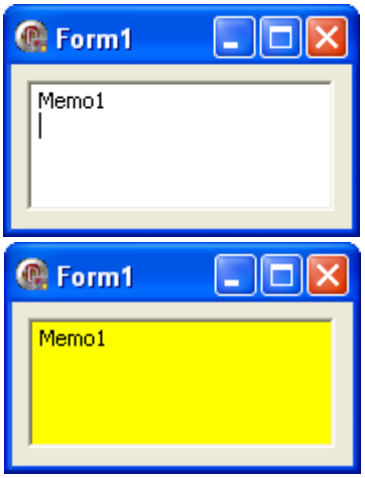
pokud žádné nemáte, namočte si mírně ruku a zcela jemným třepnutím vytvořte na monitoru malé kapičky. Třeba uvidíte ony tři jmenované barvy.
Jak bude vypadat příkaz, již víme, ale musíme jej vložit do nějaké procedury událostí, o které je známo, že vzniká dvojitým kliknutím na žádanou událost dané komponenty. Pak již stačí námi vytvořený příkaz napsat do těla této procedury.

A nyní už konečně máme dostatek informací, abychom vytvořili svůj první program, který opravdu něco dělá.

1) Otevřeme nový projekt.
2) Do prázdného formuláře vložíme komponentu Memo1.
3) V inspektoru Objektů se přes záložku Properties dostaneme na události komponenty memo1. Zde se vždy ještě jednou přesvědčte, že se jedná o vlastnosti komponenty, kterou máte na mysli.
4) Najdeme událost OnClick, kde v pravém sloupci provedeme dvojitý klik „dvojklik“.
5) Ve zdrojovém kódu se objeví procedura události s názvem `TForm1.Memo1Click.`
6) Do těla vytvořené procedury vložíme příkaz přiřazení: `Form1.Memo1.Color := clYellow;`
7) Projekt spustíme tlačítkem se zeleným trojúhelníčkem nebo funkční klávesou F9
8) Až se objeví okno naší aplikace, klikneme na komponentu (objekt) Memo.
9) Plocha určená pro psaní by měla zežloutnout.

Výpis funkční procedury je zde:

```
procedure TForm1.Memo1Click(Sender: TObject);
begin
 Form1.Memo1.Color := clYellow;
end;
```



Obr. 4.8 Ošetření události

Na obrázku 4.8 je ukázka aplikačních oken po spuštění a po průchodu výše uvedenou procedurou události inicializovanou kliknutím na komponentu Memo1.
Možná jste si všimli, že při psaní příkazu do těla procedury vám stále naskakovaly po stisku tečky nějaké nabídky. Možná jste je již využili, a pokud se v nich objevilo to slovo, které jste

chtěli napsat, tak neváhejte, v nabídce jej označte myší a stiskněte klávesu Enter. Příkaz bude doplněn a nehrozí riziko, že složitá slova napíšete nesprávně. Nyní si to vyzkoušejte. Určitě se vám nelíbí nápisy Memo1 a Form1, které sice informují o typu komponenty, ale z estetického hlediska hyzdí vaše první dílo. …

Pokud nechcete o své programy přijít, nezapomeňte je vždy včas uložit a teď už je nejvyšší čas. Vzpomenete si ještě, jak se to dělalo? Pokud jste si vzpomněli správně, tak by to měla být sekvence kliknutí na File – SaveAll. V dialogovém okně vytvořte novou složku s vystihujícím názvem (bez diakritiky) a všechny soubory nadvakrát do vytvořené složky uložte. Můžete zkontrolovat, zda se ve složce nenachází příliš málo souborů. …

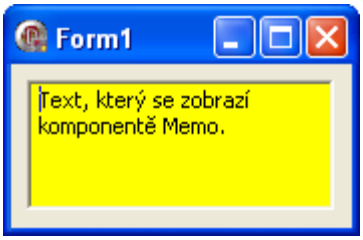
A teď již ke změně těch hyzdících nápisů, ale pozor, nápisy nejsou jména (vlastnost „*Name*“) komponent, byť jsou jejich hodnoty identické. U komponenty „Memo“ tedy musíme změnit hodnotu vlastnosti „*Text*“, ale ve vlastnostech formuláře položku „*Text*“ nenajdeme. Zde použijeme, stejně jako jsme použili u komponenty „Label“, vlastnost „*Caption*“ (titulek).

Obě vlastnosti jsou sice přístupné z inspektora objektů, ale proč to nezkusit opět softwarově?

Nejprve se zaměříme na úpravu textu v komponentě Memo1 a využijeme nabídek, které nám počítač při tvorbě příkazů bude nabízet:

1) Do stejné procedury, kde máme příkaz pro změnu barvy, napíšeme název formuláře – Form1, a když uděláme tečku, objeví se nabídka obsahující i položku Memo1, kterou můžeme vybrat rovněž dvojitým kliknutím. Memo1 se zařadí za Form1.
2) Znovu doplníme tečku. I zde se objeví nabídka, která je ale tak objemná, že naši hledanou vlastnost nevidíme. Stačí ale napsat písmeno „t“, a pokud by to nestačilo, tak ještě další, dokud neuvidíme vlastnost „Text“ a tu vybereme.
3) Této vlastnosti musíme přiřadit nějakou hodnotu. V našem případě to může být nějaký libovolný text. Přiřazení již známe a víme, že se označuje dvojtečkou a rovnítkem (napíšeme jako pokračování příkazu) a nyní následuje náš text. Text v Delphi, aby ho překládač poznal, musí být zepředu i zezadu uzavřen do jednoduchých uvozovek, které získáme nejjednodušeji přepnutím do USA klávesnice a stiskem klávesy s paragrafem. Mezi uvozovky napíšeme svůj text, například: 'Text, který se zobrazí komponentě Memo.'.
4) Tak by to mělo ve zdrojovém kódu vypadat: `Form1.Memo1.Text:='Text, který se zobrazí v komponentě Memo.';`

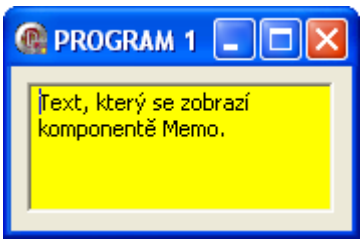
5) Po spuštění programu a kliknutí na komponentu Memo1 by se vám mělo objevit asi takové okno, jak zde vidíte.
6) Pokuste se podobným způsobem změnit titulek formuláře. Mohl by třeba být změněn na PROGRAM 1.



Pokud se vám to podaří, dostanete poslední složitější úkol. Již nechceme klikat na komponentu Memo1, ale chceme, aby se texty přepsaly samy již při spuštění aplikace. Vzpomínáte si, jak jsme si již řekli, že při spuštění aplikace se nejprve musí vytvořit formulář a událost, která se při tom aktivuje, se nazývá „*OnCreate*“? Musíme tedy vytvořit odpovídající proceduru události, buďto tak, jak jsme se již naučili (dvojité kliknutí na událost „*OnCreate*“ v inspektoru objektů, nebo v tomto případě je řešením i dvojité kliknutí přímo na kus prázdné plochy formuláře „Form1“. Nyní máme ve zdrojovém kódu procedury dvě. Protože příkazy, které jsme vytvořili, není třeba měnit. Přeneseme je jen z původní procedury do té nové. V těle původní procedury již žádný kód není, proto je zbytečná. Ale my už víme, že ji nemůžeme sami smazat. Projekt spustíme tak jak je. Mělo by se nám objevit okno, které aniž jsme cokoliv provedli, bude už mít komponentu „Memo1“ se žlutým podkladem a změněné texty. Když náš program ukončíme, neměli bychom již tu prázdnou proceduru ve zdrojovém kódu nalézt.

V následující ukázce je zdrojový kód procedury události a názvem `TForm1.FormCreate` a na obrázku 4.9 výsledný design vytvořeného programu.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Memo1.Color := clYellow;
  Form1.Memo1.Text := 'Text, který se zobrazí komponentě Memo.';
  Form1.Caption := 'PROGRAM 1';
end;
```



Obr. 4.9 Výsledný design vytvořeného programu

Shrnutí pojmů

Label – komponenta, jejíž název bývá překládán jako „návěští“ nebo „popisek“. Jejím úkolem tedy bude zobrazování nějakého textu.

Edit - komponenta, jejíž název již napovídá k čemu je určena. Podobně jako předchozí komponenta může sloužit jako výstup textových údajů, ale umožňuje navíc zadávat ručně text jakožto vstupní údaje určené ke zpracování.

Memo – komponenta s významem blízkým komponentě *Edit*. Slouží tedy jednak jako textový výstup počítače, ale můžeme do ní psát text podobně jako například do aplikace Poznámkový blok nebo Word.

S těmito základními komponentami a také s několika dalšími se můžete seznámit v animovaném programu na CD. Název programu je ***4a Komponenty***.

Otázky

1. Které komponenty se používají pro vstupně výstupní operace?
2. Které vlastnosti se k tomu používají?

Úlohy k řešení

1. Vyzkoušejte si softwarově měnit různé vlastnosti komponent umístěných na formuláři (viditelnost, barvu, rozměry, umístění apod.).
2. Pokročilejší a zvídavější studenti si mohou vytvořit program, který při pokusu o kliknutí na tlačítko změní jeho polohu ve formuláři. Bude zde nutno ošetřit proceduru události „*MouseMove*“ u vybraného tlačítka. Pro jeho přemístění bude nutno použít změnu vlastnosti „*Botton.Left* a *Botton.Top*. Pro vygenerování nové náhodné polohy tlačítka použijte funkci „Random(x), která vygeneruje náhodná celá čísla od 0 do (x-1).

5. PROMĚNNÉ, KONSTANTY, DATOVÉ TYPY

5.1. Proměnné

Čas ke studiu: 1,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat a rozdělit proměnné.
- jednotlivé datové typy.

Výklad

Doposud jsme se bez těchto pojmů obešli, ale chceme-li, aby program dokázal zpracovat zadané vstupní údaje a poskytl nám řešení ve formě výstupních údajů, budeme muset problematice formálního vyjádření dat v počítači věnovat dostatečnou pozornost.

Všechna data, s kterými počítač pracuje, musí být nějak jednoznačně označena a po nějakou nezbytnou dobu uchována v paměti počítače.

Tato data se buď po celou dobu běhu programu nemění (jsou konstantní) nebo se mohou kdykoli (na popud odpovídající instrukce programu změnit (jsou proměnné).

Z tohoto důvodu programovací jazyk Delphi umožňuje vytvářet v paměti úložiště dat označovaných jako „proměnné“ (angl. variable) a „konstanty“ (angl. constant).

Proměnné

Každá proměnna musí mít v programu svůj ***identifikátor***, kterým je pojmenování proměnné a musí mít udán svůj „***datový typ***“.

Pojmenování (název-jméno) proměnné je plně v rukou programátora. U složitějších programů by měl název odpovídat určení proměnné (např. PocetLahvi), u jednoduchých si obvykle vystačíme s jedním písmenem. Existuje sice celý výčet pravidel, co nesmí název proměnné obsahovat (např. českou diakritiku, některé jiné znaky, číslo na počátku apod.), ale budete-li se řídit našim příkladem, bude program přehlednější a zároveň si nevytvoříte chyby. Tím, že

každá proměnná slouží pro uchování specifické hodnoty aplikace a určuje i specifické místo v paměti počítače, musí mít i jedinečný název.

POZOR!
Překládač programovacího jazyka Delphi nerozlišuje velká a malá písmena, takže identifikátor „PocetLahvi“ je pro něj identický s identifikátorem „pocetlahvi“ nebo „PoCeTlAhVi“.

Trochu jiné to je s datovým typem proměnné. Zde je na výběr z omezené, přísně definované množiny datových typů.
Co to ten datový typ je a proč je v Delphi tak důležitý?
Existují programovací jazyky, které datové typy nepoužívají, ale je spousta jazyků a mezi ně patří i Delphi, které bez určení datového typu nedokončí překlad. Již z praxe víme, že informace, s kterými se v životě setkáváme, mohou mít různý charakter (typ). Mohou to být nějaké textové informace (např. věty), jen písmena (morseovka, čtení z tabule u očního lékaře), čísla, barvy, nebo dokonce dvouhodnotové informace typu ano/ne (true/false). Je pochopitelné, že každý typ informace je nutno nějak jinak zpracovat a také při ukládání například do paměti počítače zabere různě velké místo. A to je důvod, proč je v Delphi tak důležité znát konkrétní typ každé proměnné, neboť jen to umožní zpracovat zadané informace efektivně.

Právě efektivnost zpracování a ukládání dat v paměti počítače na druhou stranu rozšiřuje počet datových typů, jejichž základní přehled je uveden v tabulce 4.1.

Kategorie	**Datové typy**	**Rozsah**	**Přesnost**	**Místo v paměti**
Logické	Boolean	True, False	-	1 bit
Celočíselné	ShortInt	-128 až 127	celé číslo	8 bitů (1B)
	SmallInt	-32768 až 32767	celé číslo	16 bitů (2B)
	Integer (také) LongInt	-2147483648 až 2147483647	celé číslo	32 bitů (4B)
	Int64	-2^{63} až $2^{63}-1$	celé číslo	64 bitů (8B)

	byte	0 až 255	celé číslo	8 bitů (1B)
	Word	0 až 65535	celé číslo	16 bitů (2B)
	LongWord	0..4294967295	celé číslo	32 bitů (4B)
Desetinné	Real	$5.0 \cdot 10^{-324}$ až $1.7 \cdot 10^{308}$	15–16 číslic,(-)	8 B
	Real48	$2.9 \cdot 10^{-39}$ až $1.7 \cdot 10^{38}$	11–12 číslic,(-)	6 B
	Single	$1.5 \cdot 10^{-45}$ až $3.4 \cdot 10^{38}$	7–8 číslic,(-)	4 B
	Double	$5.0 \cdot 10^{-324}$ až $1.7 \cdot 10^{308}$	15–16 číslic,(-)	8 B
	Extended	$3.6 \cdot 10^{-4951}$ až $1.1 \cdot 10^{4932}$	19–20 číslic,(-)	10 B
	Comp	$2^{63}+1$ až $2^{63}-1$	19–20 číslic,(-)	8 B
	Currency	–922337203685477.5808 až. 922337203685477.5807	19–20 číslic,(-)	8 B
Textové	char	1 znak	-	8 bitů (1B)
	AnsiString	max 2^{31} znaků	-	2^{31} B
	string*	max 2^{31} znaků	-	2^{31} B
	ShortString	max 255 znaků	-	255B
* *typ string je implicitně překládán jako AnsiString, direktivou překládače {$H–} může být překládán jako ShortString.*				

Tabulka 4.1. Přehled základních datových typů

Řekněme, že pro naše potřeby si vystačíme s následujícími typy:

- boolean
- integer
- real
- char
- string

„boolean“ – jde o logickou proměnnou, která může nabývat jen dvou stavů - true (pravda) a false (nepravda). Tento typ proměnné se používá tam, kde je nutno se rozhodnout na základě splnění nebo nesplnění nějaké podmínky, při logických operacích a tam, kde je například nějaká vlastnost objektu splněna nebo nesplněna, jako například vlastnost „Visible“ konponenty (konponenta je viditelná nebo není viditelná).

„integer“ – jde o celočíselnou proměnnou, ve které může být uložena hodnota typu celé číslo. I když by se dalo v mnoha případech celočíselné hodnoty přechovávat v proměnných desetinného datového typu, jsou případy, kdy je nutnost použití proměnné typu integer nevyhnutelné a pokud není použita celočíselná proměnná, vygeneruje překládač chybové hlášení. Například chceme li vypočítat faktoriál čísla nebo chceme-li opakovat určitou část programu, určitě se neobejdeme bez proměnné typu integer.

„real“ – jde o proměnnou, do které lze uložit hodnotu desetinného čísla. Při výpočtech matematických vzorců a funkcí se neobejdeme bez tohoto datového typu. Pro finanční výpočty je určen speciální typ pod názvem „currency“, který je typický čtyřmi číslicemi za desetinnou tečkou.

POZOR!
Čekali jste určitě výraz „za desetinnou čárkou“, ale při psaní neceločíselných hodnot v programu Delphi, musíte důsledně dodržovat tečku „.“ jako znak oddělující celočíselnou část od zbývající časti čísla. Čárku překládač chápe jako oddělovač (např. proměnných apod.) a proto nemůže být použita jako desetinná čárka.

„char“ – proměnná, která může reprezentovat právě jeden znak. Znakem označujeme písmeno, číslici, různé znaky jako +, -, *, / a podobně. Protože proměnná typu char zabírá v paměti 1 byte, tedy 8 bitů, z toho plyne, že proměnna může obsahovat jeden z 2^8, což je 255 různých znaků. Jednotlivé znaky byly seřazeny do tzv. ASCI tabulky, kde jsou jim přiřazeny stálé číselné hodnoty.
Proměnná typu „char“ se používá pro práci se znaky např. při vyhledávání textu apod.

„string“ – typ proměnné, která reprezentuje nějakou textovou informaci (tzv. textový řetězec). Délka textového řetězce je z praktického hlediska takřka neomezena, avšak často bývá do 255 znaků. Práce s texty je vedle výpočtových úloh jedna z nejvýznamnějších úloh programových aplikací (např. Word apod.).

Ostatní datové typy jsou významné hlavně při snaze programátora o vytvoření efektivní aplikace, která zabírá co nejmenší místo v paměti počítače a zároveň probíhá v co nejkratším čase.

Podrobnosti týkající se proměnných můžete najít v animovaném programu z CD, a to pod názvem ***5a Typy promennych***.

Shrnutí pojmů

1. „boolean“ – jde o logickou proměnnou, která může nabývat jen dvou stavů - true (pravda) a false (nepravda).
2. „integer“ – jde o celočíselnou proměnnou, ve které může být uložena hodnota typu celé číslo.
3. „real“ – jde o proměnnou, do které lze uložit hodnotu desetinného čísla.
4. „char“ – proměnná, která může reprezentovat právě jeden znak.
5. „string“ – typ proměnné, která reprezentuje nějakou textovou informaci (tzv. textový řetězec).

Otázky

1. Co je to identifikátor?
2. Jaký je rozdíl mezi proměnnými a konstantami?
3. Co je to datový typ a jaké známe?

Úlohy k řešení

1. Seznamte se i s jinými datovými typy z tabulky a zvažte, kde by se daly s výhodou použít.
2. Seřaďte jednotlivé datové typy dle toho, jaké přesnosti výpočtu může být s nimi dosaženo.

5.2. Deklarace proměnných

Čas ke studiu: 0,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- deklarovat jednotlivé typy proměnných.

Výklad

Jak již bylo řečeno programovací jazyk Delphi vyžaduje znát typ všech proměnných před překladem programu. Hovoříme o deklaraci proměnných (variables), která se provádí v sekci **var** hlavičky programu. V tomto případě hovoříme o „globálních“ proměnných, které jsou definovány tak, aby mohly být používány ve všech částech programu včetně použitých podprogramů.

Vedle globálních proměnných můžeme tedy očekávat i proměnné „lokální“. Lokální znamená místní a jejich deklarace se provádí jen pro určitou část programu, například pro nám již dobře známou proceduru události. Pokud si vzpomínáte, žádná procedura události se nevytvořila s žádnou hlavičkou, ve které by byl výraz **var**. Je to proto, že zatím žádnou proměnnou nepotřebovala. Pokud tedy budeme nějakou lokální proměnnou v proceduře potřebovat, musíme před tělo procedury, a tedy před **begin** napsat výraz **var**, čímž si sekci **var** vytvoříme. Postup je pak shodný jako při deklaraci proměnných globálních.

Vlastní deklarace spočívá ve vytvoření názvu proměnné a určení jejího datového typu. Mezi obě položky je nutno pro překladač vložit znak dvojtečky „:“ a konec příkazu označit znakem středník „;“.

```
var
  PrvniCelociselnaPromenna : integer;
  PrvniDesetinnaPromenna : real;
  PrvniZnakovaPromenna : char;
  PrvniTextovaPromenna : string;
  PrvniLogickaPromenna : boolean;
```

Máme-li více proměnných stejného datového typu, můžeme je vypsat jako výčet oddělený čárkou:

```
var
  a, b, c : integer;
  x, y    : real;
```

POZOR!

Všimněte si, že zde nejsou definovány mezery mezi jednotlivými položkami ve zdrojovém kódu. Překládač mezery ignoruje, proto můžete mezery využít ke zpřehlednění programu. Nesmíte však vkládat mezery do klíčových slov (slov, která jsou rezervována v jazyku Delphi), názvu proměnných, procedur a podobně. Totéž platí pro přechod na nový řádek, kdy rozdělení má stejný následek jako mezera.

Shrnutí pojmů

Vlastní deklarace spočívá ve vytvoření názvu proměnné a určení jejího datového typu. Mezi obě položky je nutno pro překládač vložit znak dvojtečky „:“ a konec příkazu označit znakem středník „;“.

Otázky

1. Jakým způsobem se deklarují proměnné?
2. V které části programu se deklarují?

Úlohy k řešení

Vytvořte několik proměnných různých datových typů a program spusťte. Pokud po překladu zaznamenáte chyby, zkontrolujte formální zápis.

5.3. Konstanty

Čas ke studiu: 0,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- pracovat s konstantami v programu.
- definovat typové konstanty.

Výklad

Na rozdíl od proměnných se hodnota konstant po celou dobu běhu programu nemění. Konstanty jsou definovány konkrétní hodnotou, a ta je na trvalo uložena do paměti počítače. Konstanty musíme rovněž definovat v hlavičce programu. Protože při automatické tvorbě prázdného programu vývojovým prostředím žádné konstanty nejsou využity, není ani vytvořena odpovídající sekce. Tu musíme vytvořit sami. Tak jako proměnné sdružuje sekce **var**, konstanty vyžadují **const** jako označení sekce konstant. Pak již stačí konstantu pojmenovat, určit její konkrétní hodnotu a vložit ji do sekce **const.**

```
const
  KonstantaPI = 3.14;
  Kon1 = ‘konstanta’;
  Kon2 = 17;
```

Mezi jménem a hodnotou konstanty musí být vždy znaménko rovnítka „=“.

Za zmínku stojí, že existují ještě tzv. ***typové konstanty***. Zapisují se do sekce **const**, jsou určeny jménem, typem i konkrétní hodnotou.

```
const
  TypovaConst : integer = 0;
```

Typová konstanta je vlastně proměnná, která má již při deklaraci přidělenou počáteční hodnotu, a proto se tato hodnota může při běhu programu měnit.

Shrnutí pojmů

Konstanty jsou definovány konkrétní hodnotou, která se po celou dobu běhu programu nemění.

Typová konstanta je vlastně proměnná, která má již při deklaraci přidělenou počáteční hodnotu, a proto se tato hodnota může při běhu programu měnit.

Otázky

1. Co je to konstanta a jak se definuje?
2. Co to jsou typové konstanty?

Úlohy k řešení

Deklarujte několik konstant a typových konstant a překladem ověřte správnost řešení úkolu.

5.4. Přidělení hodnot proměnným a typovým konstantám

Čas ke studiu: 0,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- využít příkaz přiřazení pro přidělení hodnot proměnným.
- využít příkaz přiřazení pro přidělení vlastností komponentám.

Výklad

Přidělení počáteční hodnoty typové konstantě (ale i proměnné) se nazývá ***inicializace***.
Jak se provádí inicializace typové proměnné, jsme si již vysvětlili, ale jak provést její změnu za běhu programu? Není tak dávno, co jsme se zmínili o tzv. příkazu přiřazení, který jsme označovali dvojsymbolem „:=“. Přiřazení jsme používali ke změně vlastností komponent.
Form1.Caption := ‘Název formuláře‘;

Tento výraz můžeme přečíst jako „Titulku formuláře Form1 přiřaď text „Název formuláře“. Středník ukončuje příkaz a je povinný. Příkaz můžeme použít i pro změnu typové konstanty nebo proměnné. Syntaxe takového přiřazení by mohla vypadat například takto:

TypovaKonstanta := 12345;
a := 3;
b := 17;
c := a;
x := 15.76;
y := KonstantaPI;
Kon1 := ‘libovolný text’;
	Na levé straně příkazu je proměnná, kterou máme v úmyslu měnit a po symbolu přiřazení, tedy na pravé straně je hodnota nebo proměnná odpovídajícího typu. Na pravé straně mohou být i složité výrazy odpovídající např. matematickým vzorcům, funkcím a podobně.

POZOR!

Zde je třeba si zapamatovat, že při přiřazování musí být až na malé výjimky stejné datové typy na obou stranách. Výjimku tvoří případ, kdy proměnné typu „real“ přiřazujeme hodnotu typu integer (výsledek přiřazení bude tedy typem „real“).

Shrnutí pojmů

Inicializace - přidělení počáteční hodnoty typové konstantě nebo proměnné.

Pro změnu typové konstanty nebo proměnné se využívá příkaz přiřazení.
Příklad: X := 12;

Otázky

1. Jak se přiděluje hodnota proměnným a typovým konstantám?
2. Co musí být přitom dodrženo?

Úlohy k řešení

1. Vložte do nějaké procedury události vašeho programu několik přiřazení pro proměnné různého datového typu a spuštěním programu ověřte správnost.

5.5. Operace s proměnnými a funkce

Čas ke studiu:

Cíl: Po prostudování tohoto odstavce budete umět

- využít matematické operace při programování.
- využít matematické funkce při programování.
- využít logické funkce při programování.
- využít funkce s textovými řetězci při programování.

Výklad

Číselné proměnné resp. *konstanty* prezentují v každém okamžiku nějakou konkrétní číselnou hodnotu. Proto matematické operace s čísly mohou být nahrazeny stejnými operacemi s proměnnými resp. konstantami.

Na oba typy proměnných (integer, real) lze aplikovat mnoho společných matematických operací, avšak existují rozdíly v závislosti na tom, zda výsledek matematické operace bude přiřazen do proměnné celočíselné nebo desetinné. Nejběžnější matematické operace a funkce jsou shrnuty v následujících tabulkách.

Pro správnou interpretaci tabulek je nutné přijat předpoklad, že bylo zvoleno a deklarováno několik proměnných:

```
var
  a, b, c, n : int;
  x, y : real;
  L : boolean;
```

Základní matematické operace:

Název operace	**Operátor**	**Poznámka**
Součet	+	a := b + c; x := x + y;
Rozdíl	-	a := b - c; ; x := x - y;
Násobení	*	a := b * c; x := x * y;
Podíl(*)	/	x := x / y; x := a / b;
Celočíselné dělení(**)	div	a := b / c;
Zbytek po celočíselném dělení	mod	a := b mod c;
Změna znaménka	-	x := -x;

(*)Platí pro datový typ real, (**)platí pro datový typ integer

Základní matematické funkce:

Název funkce	**Operátor**	**Poznámka**
Druhá mocnina	sqr	y := sqr (x);
Druhá odmocnina	sqrt	y := sqrt (x);
Sinus	sin	y := sin (x);
Cosinus	cos	y := cos (x);
Arkus tangens	arctan	y := arctan (x);

Absolutní hodnota	abs	y := abs (x);
Exponent	exp	y := exp (x);
Přirozený logaritmus	ln	y := ln (x);
Vrací celou část desetinného čísla	int	y := int (x);
Vrací desetinnou část desetinného čísla	frac	y := frac (x);
Generování náhodného čísla 0..1 (real)	random	x := random;
Generování náhodného čísla 0..n (integer)	random(n)	a := random(n);

Priority jednotlivých operací souhlasí s prioritami známými z matematiky. Můžeme je modifikovat i několikanásobným použitím klasických závorek „( )“. Nemůžeme zde však použít jiný typ závorek jako je „[ ]“ ani „{ }“, které mají v Delphi jiný význam.

Následuje ukázka správných zápisů matematických operací a funkcí v Delphi:

Klasická ukázka přiřazení. a := b+1;	Proměnné „a” přiřaď hodnotu proměnné „b”, a přičti jedničku.
Přiřazení s využitím sama sebe. a := a - 1;	Proměnné „a” přiřaď její vlastní hodnotu a odečti jedničku.
Výpočet funkční hodnoty matematického výrazu. x := sin(c*ln(y));	Proměnné „x” přiřaď hodnotu funkce sinus ze součinu hodnoty proměnně „c“ a přirozeného logaritmu hodnoty proměnné „y“.
Výpočet jednoho kořene kvadratické rovnice. x := (-b+sqrt(sqr(b)-4*a*c))/(2*a);	Proměnné „x” přiřaď… Vztah je natolik složitý, že jeho slovní vyjádření by bylo nepřehledné.
Výpočet objemu válce. ObjemValce := PI*sqr(r)*v;	Proměnné „ObjemVálce” přiřaď součin hodnoty proměnné „v“ s druhou mocninou hodnoty proměnné „r“ vynásobené hodnotou konstanty „PI“. Podrobné vysvětlení: Zde jsme využili jednu z veřejných konstant programovacího jazyka Dephi, která má název „PI“ a její obsah odpovídá hodnotě 3.14… . Číselná hodnota poloměru podstavy

	je nutno v předstihu přiřadit do proměnné „r“ a výšku válce do proměnné „v“. Nejdříve je však třeba deklarovat obě proměnné, tak jako proměnnou „ObjemValce“, a to jako proměnné typu **real**.

Nyní si ukažme i několik nesprávně zapsaných příkazů:

var
  a:integer;
  x:real;
.
.
a := x;
	Do proměnné typu integer nelze přiřadit obsah proměnné typu real (opačně to je možné). Při překladu by byla nahlášena chyba: Incompatible types: ‚Integer‘ and ‚Real‘

y := sin(sqrt(x);	Nesouhlasí počet levých a pravých závorek. Přo překladu by byla nahlášena chyba: ‚)‘ expected but , ; ‘ found
c := a+b+c	Výraz je správně ale chybí ukončení příkazu středníkem. Pozor – překládač bude identifikovat chybu až v následujícím příkazu, pokud jim nebude „end“, před kterým není středník povinný.

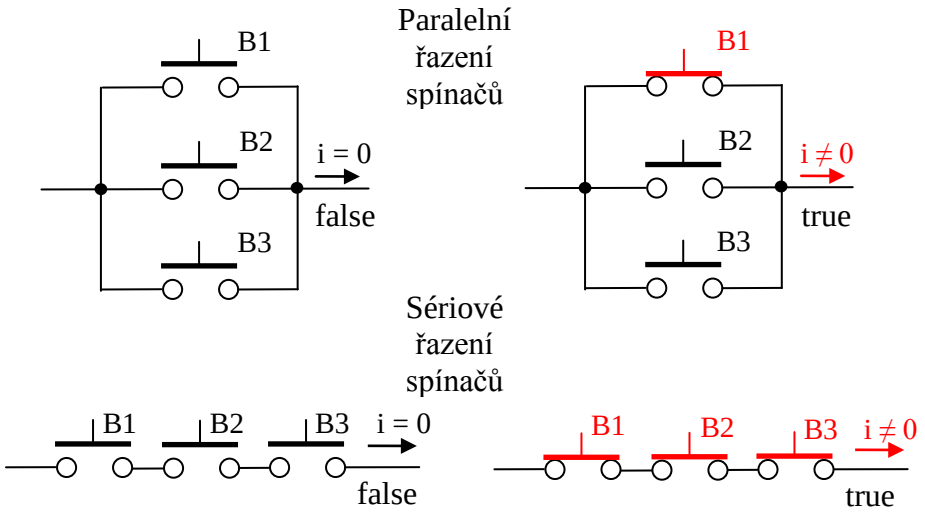
Logické proměnné reprezentují v libovolném okamžiku nějakou booleovskou hodnotu (true/false). Operace s logickými proměnnými představují speciální matematické operace, které se nazývají ***logické operace***. Základní a zároveň nejznámější logické operace budeme nuceni i my používat při složitějších programových konstrukcích. Nejprve si deklarujeme tři logické proměnné (typu boolean).

var
B1,B2 B3 : boolean;

Základní logické operace:

Název operace	**Operátor**	**Poznámka**
Logický součet	or	B1 := B2 or B3;
Logický součin	and	B1 := B2 and B3;
Logická negace	not	B1 := not(B1);

Operace logického součtu říká, že je-li alespoň jedna proměnná (obecně nemusí jít jen o proměnnou) v logickém součtu pravdivá (true) je vždy výsledek pravdivý. Výstižnou interpretací logického součtu je paralelní zapojení spínačů (obr 5.1). Stačí, aby byl spojen



Obr. 5.1 Prezentace logických funkcí prostřednictvím paralelního a sériového zapojení spínačů

kontakt libovolného spínače nebo i několik současně, aby obvodem mohl procházet elektrický proud.

Operace logického součinu říká, že musí být všechny proměnné (obecně nemusí jít jen o proměnnou) v logickém součinu pravdivé (true), aby byl výsledek pravdivý. Výstižnou interpretací logického součinu je sériové zapojení spínačů (obr 5.1). Musí být spojeny kontakty všech spínačů, aby obvodem mohl procházet elektrický proud.

Operace logické negace říká, že proměnná bude mít po negaci opačnou pravdivostní hodnotu, něž původně měla. Byla-li pravdivostní hodnota proměnné „true“, bude mít po negaci hodnotu „false“ a opačně.

Pro programování „rozhodování“ budou mít nezastupitelnou úlohu „relační“ neboli „porovnávací“ operátory

Základní relační operátory:

Název operace	**Operátor**	**Poznámka**
Je rovno	=	a = b
Je různé	<>	a <> b
Je větší	>	a > b
Je menší	<	a < b
Je větší nebo rovno	>=	a <= b
Je menší nebo rovno	<=	a <= b

Při použití těchto operátorů nedochází ke změnám žádné z proměnných. Vzhledem k tomu, že jejich využití bude probráno později, nejsou zde prezentovány celé příkazy, ale jen jejich formální zápis v Delphi.

POZOR!
Všimněte si, že tentokrát není na konci řádku středník, jak jsme byli doposud zvyklí. Je to proto, že nejde o příkazy, ale o podmínky, na které můžeme odpovědět, že je pravdivá (true) nebo nepravdivá (false). Později si ukážeme, že na základě pravdivostní hodnoty výsledku posouzení podmínky, se bude moci počítač podle vhodně napsaného programu jednoznačně rozhodnout, které příkazy bude dál provádět. Relace budou tedy součásti složitějších příkazů.

Znakové proměnné reprezentují v libovolném okamžiku nějaký znak. Operace se znaky nebudou pro nás tak frekventovanými příkazy. Budeme se více soustředit na operace s celými znakovými řetězci, přesto si zde uvedeme některé základní operace se znaky. Nejprve si deklarujeme znakovou proměnnou (typu char).

```
var
znak : char;
```

Základní znakové operace:

Název operace	**Operátor**	**Poznámka**
Vrací číslo znaku v ASCI	ord	a := ord(znak);
Vrací znak určený číslem v ASCI	char	znak := char(a);
Konvertuje malá písmena na velká	UpCase	Znak := UpCase(znak)

Někdy může nastat potřeba určit pořadové číslo znaku v ASCI tabulce. K tomu nám poslouží přikaz „ord“, opačný postup, tedy určit znak z jeho pořadového čísla v ASCI tabulce je možné využitím příkazu „char“. Příkaz UpCase dokáže změnit malé písmeno na velké (a →A). Nerespektuje však národní zvyklosti a proto je pro češtinu nepoužitelný.

Proměnné typu string reprezentují v libovolném okamžiku nějakou textovou hodnotu. Protože se v praxi často vyskytují úlohy práce s textem (vyhledávání, smazání části textu, vložení textu a podobně), není divu, že byla vytvořena celá řada funkcí a procedur, které mají za úkol programátorovi pomoci při tvorbě programů, které provádějí tyto operace. Pojem ***procedura*** zde budeme chápat jako specializovaný podprogram z knihovny vývojového prostředí, který je nám k dispozici, a který provede definovanou akci a vrátí do hlavního programu výsledky řešení. Proceduru budeme volat jejím názvem a obvykle jí musíme předat vstupní údaje, které má zpracovat. Nejprve si deklarujeme tři proměnné typu textový řetězec (typu string) a další pomocné proměnné.

```
var
  Retezec1, Retezec2, Retezec3, HlText  : string;
  Pol, Poc : integer;
{HlText – hledaný text, Pol – pozice v řetězci, Poc – počet znaků}
```

Základní operace s textovými řetězci.

Název operace	**Operátor**	**Poznámka**
Délka textového řetězce	Length	a := Length(Retezec1);
Smaže definovanou část textu	Delete	Delete(Retezec1, Pol, Poc);
Vytvoří nový řetězec z def. části řetězce	Copy	Retezec2:=Copy(Retezec1,Pol, Poc);
Vyhledá pozici znaku v řetězci	Pos	a := Pos(HlText, Retezec1);
Sloučí řetězce	Concat	Retezec3:= Concat(Retezec1, Retezec2,’Daší text’);
Vloží text na definovanou pozici řetězce	Insert	Insert('+', Retezec1, 3);

Při práci s textovým řetězcem je stěžejní znát jeho délku (počet znaků). Tu zjistíme pomocí funkce s názvem „Length“, která udává počet znaků řetězce jako celočíselnou hodnotu.

Příklad zápisu zdrojového kódu:

var Retezec : **string**; DelkaRetezce : integer; . . Retezec := ‘Řetězec’; DelkaRetezce := Length(Retezec);	Nejprve deklarujeme proměnné „Retezec“ typu string a “DelkaRetezce” typu integer. Pak do proměnné „Retezec“ přiřadíme text ‘Řetězec’. Poslední příkaz je napsán tak, aby byla do celočíselné proměnné „DelkaRetezce” přiřazena hodnota, která odpovídá počtu znaků v proměnné „Retezec“.

Ostatní příkazy nebudeme používat, a proto zde nebudou podrobně popsány. Zájemci se s nimi mohou blíže seznámit v doporučené literatuře.

Ukážeme si zde ještě dvě potřebné operace, které se nám budou v budoucnu docela hodit.

- Spojování znakových řetězců – vedle speciální procedury pro spojování řetězců můžeme úspěšně použít jednoduchý matematický aparát, kterým je součet.

  Příkaz by pak vypadal následovně:

  Retezec3:= Retezec1+ Retezec2,+’Daší text’;

  Příklad: Fragment programu pro sloučení textu.

  Retezec1 := ‘Jméno’;
  Retezec2 := ‘Příjmení’;
  Retezec3 := Retezec1+ Retezec2;

  V proměnné Retezec3 je po ukončení běhu předchozího fragmentu programu textový řetězec ve tvaru: ‘JménoPříjmení’. Pokud bychom chtěli jméno a příjmení oddělit, můžeme do příkazu součtu retězců mezi obě proměnné vložit řetězec, který by obsahoval sadu mezer.

  Třeba takto:

  Retezec3 := Retezec1+ ‘ ‘+Retezec2;

  V proměnné „Retezec3“ by pak byl textový řetězec ve tvaru: ‘Jméno Příjmení’, což je již přijatelná varianta.

- Zjištění jednotlivých znaků v textovém řetězci – uvedeme si zde jen komentovanou ukázku fragmentu programu, která tento problém řeší.

  Retezec1 := ‘Jméno’;
  a := 3;
  znak := Retezec1[a];

  V proměnné „znak“ je po ukončení běhu předchozího fragmentu programu písmeno „é“, tedy třetí znak v textovém řetězci. Pokud bychom postupně měnili hodnotu celočíselné proměnné „a“ od 1 do 5, obdrželi bychom všechna písmena textového řetězce. Jak to provést programově se naučíme v dalších kapitolách.

Shrnutí pojmů

Číselné proměnné resp. *konstanty* prezentují v každém okamžiku nějakou konkrétní číselnou hodnotu.

Logické proměnné reprezentují v libovolném okamžiku nějakou booleovskou hodnotu (true/false).

Znakové proměnné reprezentují v libovolném okamžiku nějaký znak.

Proměnné typu string reprezentují v libovolném okamžiku nějakou textovou hodnotu.

Otázky

1. Které matematické operace a funkce znáte?
2. Které logické funkce znáte?
3. Které operace s textovými řetězci znáte?
4. Jak se spojují textové řetězce?

Úlohy k řešení

1. Vyzkoušejte aplikovat uvedené operace do vašeho programu do procedury události a ověřte překladem správnost.
2. Vytvořte vlastní programy dle animovaných programů z CD, označených názvy ***5b ASCII tabulka*** a ***5c Analyza znaku.***

5.6. Konverze typu proměnných

Čas ke studiu: 0,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- převádět různé číselné datové typy.
- převádět číselné datové typy na texty a opačně.

Výklad

V minulé kapitole bylo řečeno, že je většinou dovoleno uskutečnit přiřazení jen pro shodné typy proměnných. Dost často je však nutné nějak do proměnné přece jen vložit hodnotu, která je v proměnné jiného typu. Pak ovšem nezbývá, než konvertovat výsledek výrazu na pravé straně přiřazení na datový typ, který je shodný s datovým typem proměnné (ale třeba i vlastnosti komponenty) na levé straně přiřazení.
Za tímto účelem bylo vytvořeno několik speciálních funkcí, které tento převod uskuteční.

Převod desetinného čísla na celé číslo:

Název operace	**Funkce**	**Poznámka**
Zaokrouhlení	round	a := round (x);
Odříznutí	trunc	a := trunc (x);

Zaokrouhlení - celočíselná hodnota je získána klasickou matematickou operací zaokrouhlení.
Oříznutí - výsledná celočíselná hodnota je získána odříznutím části za desetinnou tečkou.

Často jsme nuceni převádět číselné hodnoty na texty a opačně. Většina komponent, jako například „Edit“, „Label“, „Memo“ a podobně, využívá pro komunikaci s obsluhou vlastnosti jako je „Text“, „Caption“, „Lines.Add“ a jiné, které vyžadují pouze textová data. Pro převod těchto typů můžeme použít následující funkce programovacího jazyka Delphi.

Převod čísla na text:

Název operace	**Funkce**	**Poznámka**
Převod celočíselné hodnoty na textový řetězec	IntToStr	Retezec1 := IntToStr (CeleCislo);
Převod desetinného čísla na textový řetězec	FloatToStr	Retezec2 := FloatToStr (DesCislo);

A opačně převod textu na číslo:

Název operace	**Funkce**	**Poznámka**
převod textového řetězce na celé číslo	StrToInt	CeleCislo := StrToInt (Retezec1);
převod textového řetězce na desetinné číslo	StrToFloat	DesCislo := StrToFloat (Retezec2);

POZOR!

Převod z čísla na řetězec by měl být bezproblémový, ale opačný převod s sebou nese určitá rizika. Může se stát, že text (např. v komponentě „Edit“ neodpovídá číslu). Často zde zůstává při zpuštění programu nápis (text) např. Edit1. Převod pak skončí nechtěným přerušením běhu programu. Další častou chybou při vstupu dat je použití desetinné tečky, když je v národním prostředí Windows nastavena čárka a opačně (je to nezávislé na požadavku, že ve zdrojovém kódu je striktně vyžadována desetinná tečka).

Vedle těchto základních převodů typu dat existuje ještě celá řada dalších (např. mezi textem a datem nebo časem). Jejich popis již překračuje rámec obsahu tohoto učebního textu a čitatel si je může snadno nastudovat v doporučené literatuře nebo helpu programovacího jazyka Delphi.

Shrnutí pojmů

Většinou je dovoleno uskutečnit přiřazení jen pro shodné typy proměnných někdy je však nutné do nějaké proměnné přece jen vložit hodnotu, která je v proměnné jiného typu. Pak je nutno použít konverzi na odpovídající typ.

Název operace	**Funkce**	**Poznámka**
Zaokrouhlení	round	a := round (x);
Odříznutí	trunc	a := trunc (x);
Převod celočís. hodnoty na textový řetězec	IntToStr	Retezec1 := IntToStr (CeleCislo);
Převod desetinného čísla na textový řetězec	FloatToStr	Retezec2 := FloatToStr (DesCislo);
převod textového řetězce na celé číslo	StrToInt	CeleCislo := StrToInt (Retezec1);
převod textového řetězce na desetinné číslo	StrToFloat	DesCislo := StrToFloat (Retezec2);

Otázky

1. Jak se převádí datový typ real na integer?
2. Jak se převádí datový typ string na číslo?
3. Jak se převádí číselný datový typ na string?

Úlohy k řešení

Vyzkoušejte aplikovat uvedené konverze do vašeho programu do procedury události a ověřte překladem správnost.

Ověřte své znalosti použitím animovaného programů z CD, označeného názvem ***5d Konverze.***

5.7. Výpočtový program

Čas ke studiu: 5,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- měnit vlastnosti komponent dle požadavků.
- vytvářet design aplikace dle požadavků.
- ošetřit události komponent.

Výklad

Nyní jsme již dostatečně teoreticky vyzbrojení k tomu, abychom vytvořili svůj druhý, ale tentokrát poměrně složitý program, který by měl realizovat některé funkce kalkulačky.

Postup návrhu:

Nejprve si musíme přesně definovat, co od aplikace budeme očekávat. Pak do formuláře umístíme potřebné komponenty a upravíme podle potřeb jejich vlastnosti. Při tomto kroku vycházíme nejen z požadavků na funkčnost, ale také na ergonomii a celkový design budoucí aplikace. Nakonec ošetříme požadované události (většinou půjde o kliknutí na tlačítko).

1. Určení vlastností aplikace
   - Možnost zadání dvou číselných hodnot do dvou komponent Edit
   - Určení matematických operací a funkcí se zadanými čísly pomocí tlačítek
   - Výpočet odpovídající operace nebo funkce
   - Zobrazeni výsledku do třetí komponenty Edit
2. Volba komponent a tvorba designu
   - Dle požadavků dle předcházejícího výčtu vytvoříme ve formuláři zadané komponenty
   - Vhodně upravíme jejich polohu a rozměry
   - Vytvoříme popisy v komponentách, aby byla aplikace přehledná a intuitivně ovladatelná
3. Tvorba potřebných procedur událostí

- Automatické vytvoření procedur událostí (snadno provedeme dvojitým kliknutím na příslušné tlačítko)
- Ošetření událostí – vytvoření příkazů pro realizaci matematických operací nebo funkcí a zobrazení výsledků.

TIP!

Je výhodné vytvořit definitivní vzhled jedné z opakujících se komponent (zde jedno vzorové tlačítko) a pak jen stačí ono tlačítko označit a stiskem kláves Ctrl+C uložit do schránky. Následně klikneme do formuláře a několikerým stiskem kláves Ctrl+V vytvoříme potřebný počet zcela identických komponent. Změníme jen nápisy a máme je připraveny pro další práci (někdy nemusí fungovat).

POZOR!

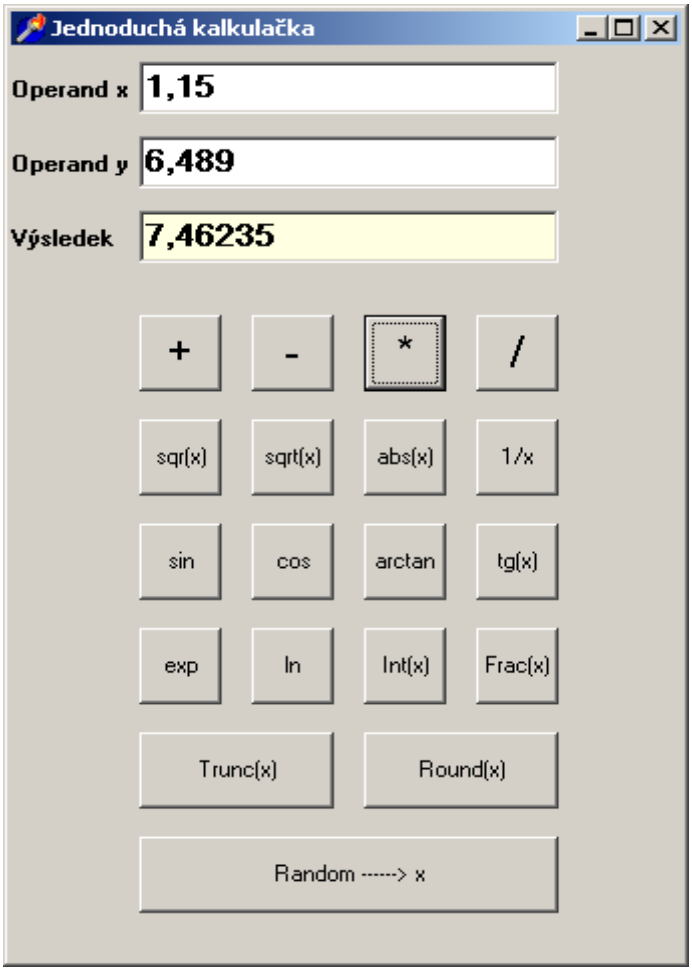
Pokud bychom již před kopírováním komponenty vytvořili její proceduru událostí, kopie by si s sebou nesly povinnost volat stále jen tu jednu a tu samou proceduru události. Východiskem by pak bylo u každé zkopírované komponenty vymazat událost „OnClick“ v inspektoru objektů.

Výčet požadovaných matematických operací a funkcí:

- součet, rozdíl, součin, podíl,
- celočíselné dělení, zbytek po dělení, zaokrouhlení, ořezání desetinné části,
- sinus, kosinus, arkus tangens, tangens, exponent, přirozený logaritmus,
- převrácená hodnota, druhá mocnina, druhá odmocnina, absolutní hodnota,
- náhodné číslo

Pro všechny zde jmenované matematické operace a funkce musíme vytvořit ve formuláři tlačítka a pojmenovat je výstižným názvem.

Návrh podoby aplikace realizující kalkulačku se zadanými parametry můžeme vidět na obr. 5.2.



Obr. 5.2 Návrh designu programu „Kalkulačka“

Dvojitým kliknutím na vybrané tlačítko vytvoříme prázdnou proceduru událostí, do které vložíme několik příkazů, které zajistí přečtení vstupních údajů z Edit1 eventuálně i z Edit2, provede definovanou matematickou operaci nebo funkci a následně vypíše výsledek do Edit3.

Jde o poměrně složité přiřazení, kdy musíme navíc dvakrát měnit datové typy, neboť matematické operace a funkce jsou definovány jen pro číselné hodnoty, kdežto informace v komponentách Edit jsou textového charakteru.

Zde si ukážeme na operaci součtu pouze jednu ukázku, jak by měla celá procedura události vypadat. Výpis celého programu je v sekci výsledky úloh k řešení.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Edit3.Text:=FloatToStr(StrToFloat(Form1.Edit1.Text)
                    +StrToFloat(Form1.Edit2.Text));
end;
```

Shrnutí pojmů

Postup návrhu aplikace:

- Nejprve si musíme přesně definovat, co od aplikace budeme očekávat.
- Pak do formuláře umístíme potřebné komponenty a upravíme podle potřeb jejich vlastnosti.
- Nakonec ošetříme požadované události (většinou kliknutí na tlačítko), to znamená, že vložíme příkazy do procedur událostí.

Otázky

1. Vyjmenujte základní etapy vývoje aplikace?
2. Které vlastnosti komponent nejčastěji měníme při tvorbě vzhledu aplikace?
3. Co vkládáme do procedur událostí?

Úlohy k řešení

1. Vytvořte programovou aplikaci, která by realizovala kalkulačku dle výše uvedených požadavků. Můžete se nechat inspirovat animovaným programem z CD, který je označen názvem ***5e Kalkulacka vyukova***.
2. Vytvořte program pro výpočet obvodu a obsahu vybraných plošných geometrických obrazců. Jako inspiraci použijte animovaný program z CD, který je označen názvem ***5f Vypocty obrazcu***.
3. Vytvořte program pro výpočet objemu a povrch vybraných prostorových geometrických obrazců.

6. ZÁKLADNÍ PROGRAMOVÉ KONSTRUKCE

6.1. Sekvence

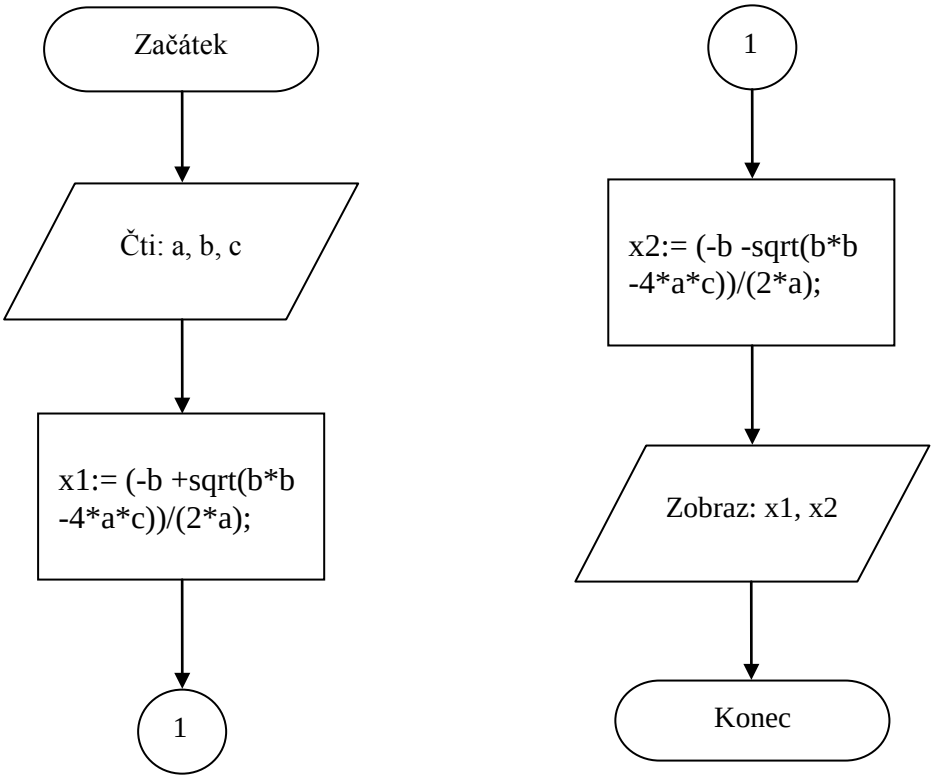
Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat programovou konstrukci – sekvence.
- vyřešit úlohy s programovou konstrukci – sekvence.

Výklad

Nejjednodušší programovou konstrukcí je ***sekvence*.** Jak již název napovídá, půjde o sled výkonných příkazů typu přiřazení nebo příkazy vstupů a výstupů, které jsou postupně za běhu programu vykonávány seshora dolů. S touto konstrukcí jsme se již seznámili v příkladech z minulých kapitol, nyní si takovou konstrukci vyzkoušíme na vývoji programu pro výpočet kořenů kvadratické rovnice. Konstrukce sekvence je nejlépe patrna z následujícího vývojového diagramu:



Nyní máme úkol vytvořit aplikaci v programovacím jazyku Delphi, která by realizovala zde uvedený algoritmus výpočtu kořenů kvadratické rovnice.

Postup řešení:

- Do formuláře vložíme pět komponent třídy Edit. První tři nám budou sloužit jako vstupy pro zadávání parametrů kvadratické rovnice (a, b, c) a druhé dva použijeme pro zobrazení výsledků řešení, tedy kořenů x1 a x2.
- Pro práci se vstupními hodnotami deklarujeme stejnojmenné proměnné typu real, protože parametry a, b, c jsou z množiny reálných čísel.
- Pro hodnoty vypočtených kořenů kvadratické rovnice deklarujeme ještě dvě proměnné typu real, které nazveme ve shodě s vývojovým diagramem x1 a x2.
- Pro spuštění výpočtu po zadání vstupních údajů do komponent Edit1 až Edit3 použijeme kliknutí na komponentu tlačítka, kterou si vytvoříme pro tento účel ve formuláři. Dvojitým kliknutím na komponentu tlačítka Button1 vytvoříme prázdnou proceduru události kliknutí na tlačítko Button1.
- Do ní již napíšeme sled příkazů dle vývojového diagramu.
- Pro přehlednost ještě do formuláře doplníme pět komponent třídy Label, do kterých umístíme popisky názvu jednotlivých vstupů a výstupů. Programově pak vhodně změníme vlastnosti Caption tlačítka i formuláře.

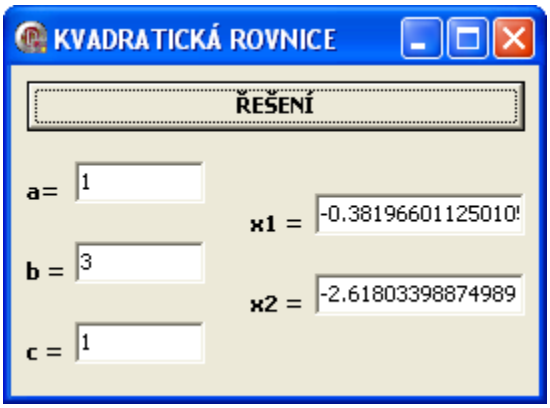
Výsledek naší práce je patrný z následujícího výpisu programu a ukázky podoby pracovního okna.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
```



```
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  x1,x2,a,b,c:real;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Caption:='KVADRATICKÁ ROVNICE';
  Form1.Button1.Caption:='ŘEŠENÍ';
  a:=StrToFloat(Form1.Edit1.Text);
  b:=StrToFloat(Form1.Edit2.Text);
  c:=StrToFloat(Form1.Edit3.Text);
  x1:=(-b+sqrt(b*b -4*a*c))/(2*a);
  x2:=(-b-sqrt(b*b -4*a*c))/(2*a);
  Form1.Edit4.Text:=FloatToStr(x1);
  Form1.Edit5.Text:=FloatToStr(x2);
end;

end.
```

Shrnutí pojmů

sekvence je sled výkonných příkazů typu přiřazení nebo příkazy vstupů a výstupů, které jsou postupně za běhu programu vykonávány seshora dolů.

Otázky

1. Co je to sekvence?
2. Na které úlohy se dá použít?

Úlohy k řešení

Vytvořte program pro výpočet kořenů kvadratické rovnice dle předchozího popisu a vyzkoušejte jeho funkci. Snažte se otestovat bezchybnost programu.

6.2. Větvení

Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat různé typy příkazu větvení.
- aplikovat různé typy příkazu větvení.

Výklad

Při řešení příkladu z minulé kapitoly jsme dospěli k názoru, že pokud chceme, aby program správně fungoval nezávisle na tom, jaké vstupní hodnoty zadáváme, nevystačíme si jen s prostou sekvenci příkazů typu přiřaď nebo čti hodnotu, vypiš hodnotu.

Chyby, které nám vznikly, i když algoritmus vlastního výpočtu byl správný, můžeme rozdělit do dvou kategorií.

1. Program nebyl schopen přečíst vstupní hodnoty, které nevyhodnotil jako datový typ „real“ (celočíselné hodnoty chápe jako real bez udání desetinné části).
2. I když byly vstupní hodnoty formálně v pořádku, stávaly se případy, že výpočet proběhl bez problému a správně (např. pro a = 1, b = 3, c =1) a jindy předčasně skončil s chybovým hlášením (např. pro a = 1, b = 1, c = 1).

Ze vzorce pro výpočet kořenů kvadratické rovnice hned vidíme, kde může nastat chyba:

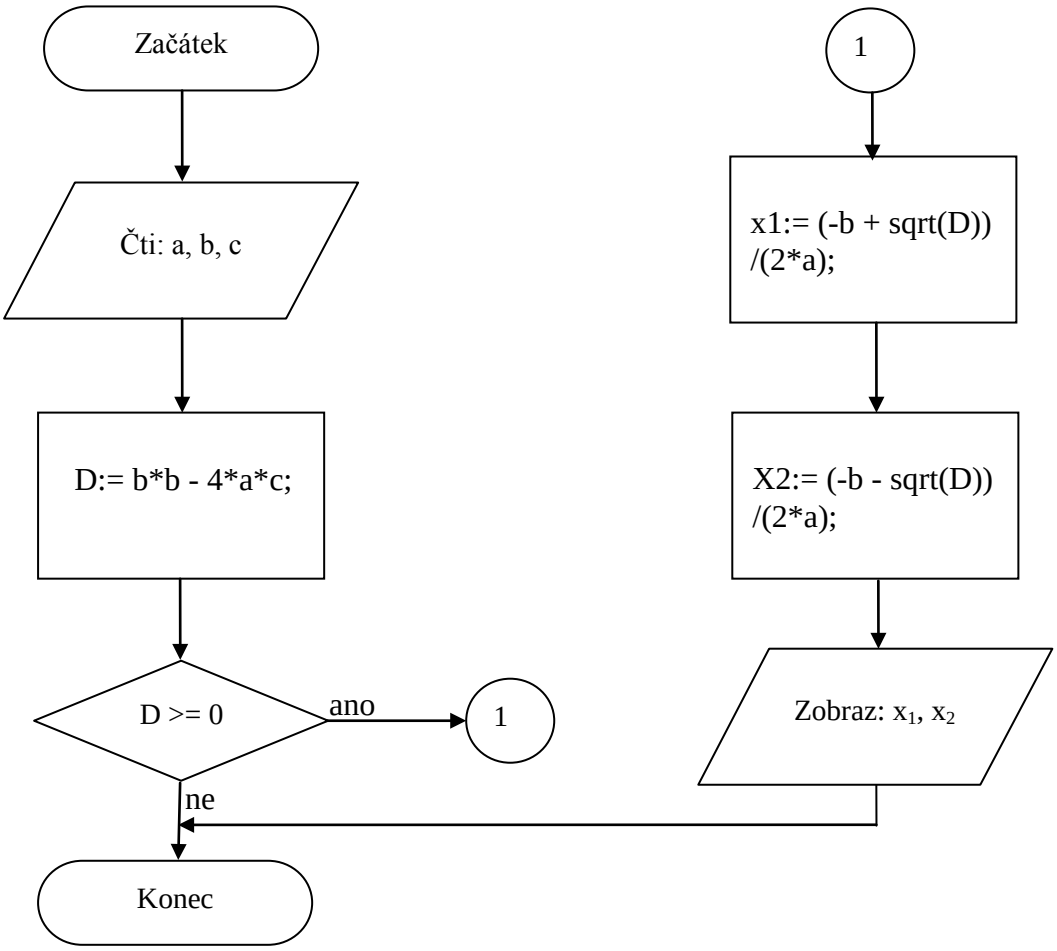
$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- pokud je hodnota $a = 0$
- pokud je výraz $b^2 - 4ac < 0$

První případ bychom sice mohli řešit tak, že nulu do „a“ nikdy nezadáme (jenže co když jí tam zadá někdo jiný?).

Druhý případ je horší. Museli bychom si ručně vypočítat pro každé zadání hodnotu výrazu $b^2–4ac$, a pokud by náhodou byla záporná, výpočet bychom neuskutečnili. Jenže s takovým programem by nikdo nechtěl pracovat, a proto nám nezbývá, než se naučit chyby, které jsme si dokázali definovat a víme, jak a kde vznikají, zpracovat přímo v programu.

Zkusme nejprve ošetřit chybu, která vznikne při podmínce, že $b^2 - 4ac < 0$. Tento výraz bývá zvykem označovat jako diskriminant (označení - D) a z výuky matematiky víme, že kvadratická rovnice má řešení v oboru reálných čísel jen pro $D \geq 0$.
Máme tedy určitou, přesně definovanou podmínku, jejíž splnění či nesplnění nás nutí postupovat dále rozdílným způsobem (jednou ze dvou cest). Když bude podmínka $D \geq 0$ splněna, tedy je pravdivá (true), můžeme pokračovat ve výpočtu. Pokud splněna není (false), nemůže být výpočet uskutečněn.
V programu musí nutně dojít ke větvení do dvou různých tras, které se zas někde před koncem programu musí sejít.
Pro větvení, pokud si vzpomínáte, existuje ve vývojových diagramech speciální značka. Zkusme tedy vývojový diagram využít ke znázornění řešení našeho problému.



V programovacím jazyku Delphi tedy musí zákonitě existovat adekvátní prostředek, který by umožnil rozvětvit program na základě výsledku definované podmínky. Tento prostředek se nazývá příkaz pro větvení a má dokonce i několik variant.

Neúplný příkaz větvení

Jde o příkaz, jehož formální zápis vypadá následovně:

if „podmínka“ **then** „příkaz“;

Můžeme si ho interpretovat takto:
Pokud (if) je splněna „podmínka“, pak (then) proveď *následující* „příkaz“.

Například:

```
if a > b then
```

```
x := a – b;
```

Někdy je však nutno při splnění podmínky vykonat příkazů více. Příkazy se dají slučovat zapsáním mezi klíčová slova **begin** a **end;**

Například:

```
if a > b then
  begin
    x := a – b;
    y := a + x;
  end;
```

sdružený příkaz je chápán jako jeden příkaz

jeden příkaz konstrukce **if - then**

POZOR!

Následující konstrukce má zcela jinou strukturu, a proto se nechová shodně s předchozí:

```
if a > b then
  x := a – b;
y := a + x;
```

jeden příkaz konstrukce **if - then**

druhý příkaz přiřazovací

Příkaz x := a – b; by byl proveden jen tehdy, pokud by byla splněna podmínka $a > b$. Tím by však byla působnost příkazu **if – then** ukončena a příkaz y := a + x; by byl proveden vždy, tedy i při skutečnosti, že $a <= b$. Jde o velmi častou chybu při psaní zdrojového kódu.

Program pro řešení kvadratické rovnice z minulé kapitoly by se použitím příkazu **if – then** změnil následovně (změna je jen v proceduře události):

```
procedure TForm1.Button1Click(Sender: TObject);
var D : real;
begin
  Form1.Caption:='KVADRATICKÁ ROVNICE';
  Form1.Button1.Caption:='ŘEŠENÍ';
  a:=StrToFloat(Form1.Edit1.Text);
  b:=StrToFloat(Form1.Edit2.Text);
  c:=StrToFloat(Form1.Edit3.Text);
  D:= b*b-4*a*c;
  if D>=0 then
```

```
    begin
      x1:=(-b+sqrt(D))/(2*a);
      x2:=(-b-sqrt(D))/(2*a);
      Form1.Edit4.Text:=FloatToStr(x1);
      Form1.Edit5.Text:=FloatToStr(x2);
    end;
end;
```

Zavedli jsme pomocnou proměnnou „D“ typu real, kterou jsme deklarovali přímo v proceduře události v sekci **var.** Tu jsme ovšem museli ručně vytvořit před klíčovým slovem **begin**, které určuje začátek těla procedury. Tím je proměnná „D“ platná pouze v této proceduře a na rozdíl od ostatních proměnných, které jsou definované v hlavičce jednotky (unit), není přístupná z jiných částí programu.

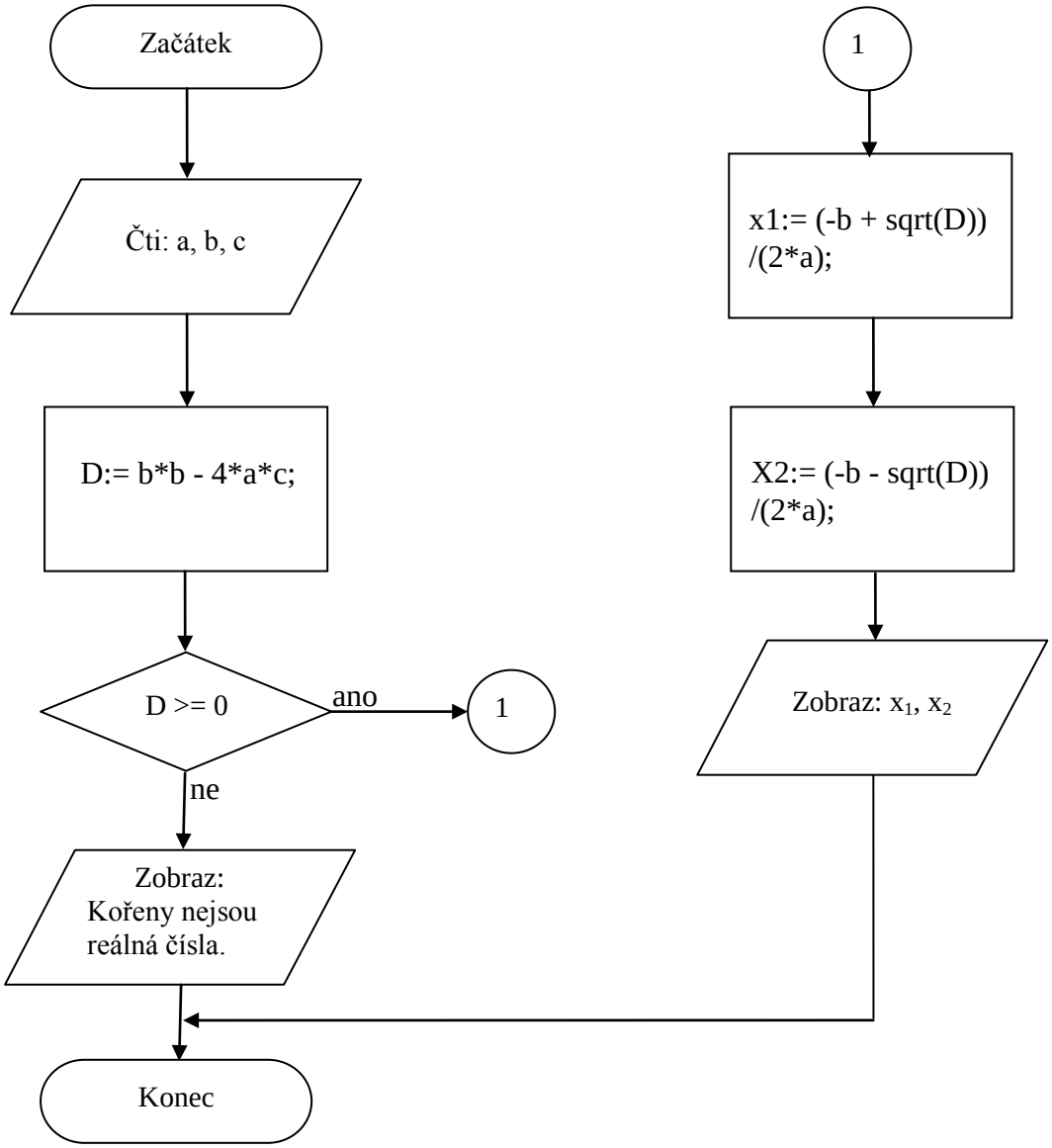
Jak již víme, odborně se taková proměnná označuje jako ***lokální***, ty proměnné, které platí v rámci celé jednotky (unit) se označují jako ***globální***.

Program by měl nyní pracovat pro všechny korektně zadané vstupní údaje. *Ale tímto postupem jsme si vyrobili závažnou chybu.* Než budete číst další text, zkuste na tu chybu přijít. Nebude to úplně jednoduché, budete muset hodně logicky zauvažovat.

Přišli jste na ni?

Pokud budete postupně zadávat takové vstupní údaje, že jsou výsledkem reálné kořeny, bude vše v pořádku, ale zadáte-li vstupní údaje tak, že jsou výsledkem kořeny komplexně sdružené ($D<0$), program sice výpočet neprovede a „nehavaruje“, ale na výstupech poskytne údaje platné z posledního zadání, kdy došlo k regulérnímu výpočtu x_1 a x_2. Chyba tohoto typu může být špatně odhalitelná, proto je třeba věnovat jejímu ošetření velkou pozornost.

Řešení bychom mohli znázornit tímto vývojovým diagramem.



Při splnění hodnoty *D >= 0* by došlo k výpočtu a výpisu výstupních hodnot. Při nesplnění podmínky by program informoval o tom, že kořeny nejsou reálná čísla. Na to ale budeme potřebovat jiný typ příkazu větvení v Delphi.

Úplný příkaz větvení

Jde o příkaz, jehož formální zápis vypadá následovně:

if „podmínka“ **then** „příkaz1“ **else** „příkaz2“;

Můžeme si ho interpretovat takto:

Pokud (if) je splněna „podmínka“, pak (then) proveď „příkaz1“, pokud však není splněna (else) proveď „příkaz2“.

Například:

```
if a > b then
  x := a – b
else
  x := b - a;
```

Máme tedy k dispozici programovou konstrukci, kterou můžeme využít pro výpočet kvadratické rovnice jak pro případ, že kořeny vyjdou reálná čísla, tak pro případ, že vyjdou kořeny komplexně sdružené.

pro $D >= 0$ platí:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

pro $D < 0$ platí pro reálnou část kořenů:

$$x_{Re} = \frac{-b}{2a}$$

pro $D < 0$ platí pro imaginární části kořenů:

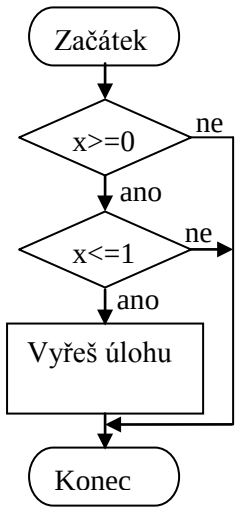
$$x_{Im_{1,2}} = \pm \frac{\sqrt{-D}}{2a}$$

Znalost vzorců a syntaxe úplného příkazu větvení nám umožňuje vytvořit program, který již zvládne výpočet jak reálných tak komplexně sdružených kořenů kvadratické rovnice. Musíme si ještě deklarovat proměnné ReX a ImX a můžeme již doplnit tělo procedury události do následujícího tvaru.

```
procedure TForm1.Button1Click(Sender: TObject);
var D,ReX,ImX: real;
begin
  Form1.Caption:='KVADRATICKÁ ROVNICE';
  Form1.Button1.Caption:='ŘEŠENÍ';
```

```
  a:=StrToFloat(Form1.Edit1.Text);
  b:=StrToFloat(Form1.Edit2.Text);
  c:=StrToFloat(Form1.Edit3.Text);
  D:= b*b-4*a*c;
  if D>=0 then
    begin
      x1:=(-b+sqrt(D))/(2*a);
      x2:=(-b-sqrt(D))/(2*a);
      Form1.Edit4.Text:=FloatToStr(x1);
      Form1.Edit5.Text:=FloatToStr(x2);
    end
  else
    begin
      ReX:=(-b)/(2*a);
      ImX:=(sqrt(-D))/(2*a);
      Form1.Edit4.Text:=FloatToStr(ReX)+'+i'+FloatToStr(ImX);
      Form1.Edit5.Text:=FloatToStr(ReX)+'–i'+FloatToStr(ImX);
    end;
end;
```

Složená podmínka



Často může být podmínka v příkazu větvení složitější. Chceme-li například provést řešení nějaké úlohy pro x <0;1>, pak bychom mohli použít řešení dle následujícího vývojového diagramu. To by znamenalo vytvořit odpovídající konstrukci programu.

To by znamenalo vytvořit následující konstrukci programu:

```
if x >= 0 then
  if x <= 1 then
    VyresUlohu;
```

U složitějších podmínek by se mohlo stát vnořování příkazů **if-then-else** nepřehledným. Alternativním (a efektivnějším) řešením je použití složené podmínky, která využívá logické funkce (**and**, **or**, **not**) pro sloučení jednoduchých, převážně relačních podmínek.

Předchozí úkol při použití složené podmínky by pak vypadal následovně:

```
if (x >= 0) and (x <= 1) then VyresUlohu;
```

Můžeme číst: „pokud je splněna první podmínka a současně druhá podmínka, pak proveď následující příkaz“.

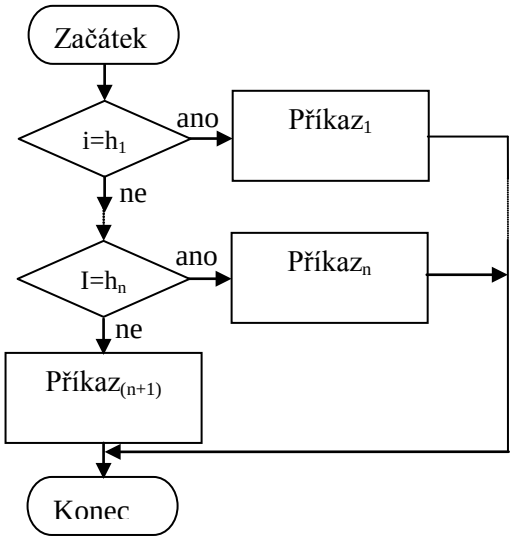
Jak bychom přečetli následující příkaz?

```
if ((x > 0) and (y < 0)) or ((x < 0) and (y > 0)) then
  Form1.Edit1.Text := ‘Součin x·y bude záporné číslo‘;
```

Jestliže je x záporné a současně y kladné, nebo x je kladné a současně y záporné, napiš do komponenty Edit1 text ve znění: „Součin x·y bude záporné číslo“.
Jako pomůcku k řešení kvadratické rovnice bez i s ošetřením záporné hodnoty pod odmocninou je možno použit animovaný program z CD pod názvem ***6a Kvadraticka rovnice***.

Příkaz větvení case-of-else

Posledním příkazem, kterým můžeme rozvětvit program, je příkaz větvení **case-of-else**, který jako jediný umožňuje na základě vyhodnocení proměnné celočíselného typu nebo typu „char“ rozvětvit program do více než dvou větví. Vývojový diagram vysvětlující funkci příkazu **case-of-else** by se dal znázornit například takto:



Jeho formální interpretace v programovacím jazyku Delphi je následující:

```
case „proměnná“ of
  hodnota1: příkaz1;
  hodnota2: příkaz2;
          .
          .
```

```
    hodnotaN: příkazN;
  else
    příkaz;
  end;
```

Přičemž „hodnota“ může být jedno celé číslo nebo znak, může obsahovat výčet celých čísel nebo znaků oddělených čárkou (3, 7, 5, 8), popřípadě interval celých čísel nebo znaků (a..z), kde je udána minimální hodnota, dvě tečky a hodnota maximální.

Řešený příklad

Zadání

Vytvořte program, který na základě hodnoty v proměnné „teplota“, vypíše do komponenty „Edit1“ slovní hodnocení teplotní pohody. Pro řešení využijte příkaz **case-of-else.**

Řešení:

```
case round(teplota) of
  -60..-1 : Edit1.Text := ‘Ukrutná zima‘;
  0       : Edit1.Text := ‘Bod tání‘;
  1,2,3   : Edit1.Text := ‘Příjemná zima‘;
  4..15   : Edit1.Text := ‘Chladno‘;
  16..25  : Edit1.Text := ‘Teplo‘;
  26..35  : Edit1.Text := ‘Letní ideál‘;
  36..60  : Edit1.Text := ‘Nedá se vydržet‘;
else
  Edit1.Text := ‘Jiná planeta‘;
end;
```

Shrnutí pojmů

Neúplný příkaz větvení

Jde o příkaz, jehož formální zápis vypadá následovně:

if „podmínka“ **then** „příkaz“;

Můžeme si ho interpretovat takto:

Pokud (if) je splněna „podmínka“, pak (then) proveď *následující* „příkaz“.

Úplný příkaz větvení

Jde o příkaz, jehož formální zápis vypadá následovně:

if „podmínka“ **then** „příkaz1“ **else** „příkaz2“;

Můžeme si ho interpretovat takto:
Pokud (if) je splněna „podmínka“, pak (then) proveď „příkaz1“, pokud však není splněna (else) proveď „příkaz2“.

Složená podmínka

Často může být podmínka v příkazu větvení složitější. Řešením je použití složené podmínky, která využívá logické funkce (**and**, **or**, **not**) pro sloučení jednoduchých, převážně relačních podmínek.

Jde o příkaz, jehož formální zápis vypadá následovně:

```
if (podmínka1) LogickáFunkce (podmínka2) then VyresUlohu;
```

Příkaz větvení **case-of-else**, umožňuje na základě vyhodnocení proměnné celočíselného typu nebo typu „char“ rozvětvit program do více než dvou větví.

Jeho formální interpretace v programovacím jazyku Delphi je následující:

```
case „proměnná“ of
   hodnota1: příkaz1;
   hodnota2: příkaz2;
                .
                .
   hodnotaN: příkazN;
else
   příkaz;
end;
```

Hodnota1 až hodnota N může být *jedno* celé číslo nebo znak, může obsahovat *výčet* celých čísel nebo znaků oddělených čárkou (3, 7, 5, 8), popřípadě *interval* celých čísel nebo znaků (a..z), kde je udána minimální hodnota, dvě tečky a hodnota maximální.
Jako názornou pomůcku k pochopení problematiky větvení programu můžete použít animovaný program z CD pod názvem ***6b Vetveni programu***.

Otázky

1. Jaký tvar a význam má příkaz větvení?
2. Jaké varianty příkazu větvení znáte a jak se používají?

Úlohy k řešení

1. Do programu pro výpočet kořenů kvadratické rovnice dle předchozího popisu přidejte ošetření případu, že kořeny nevyjdou jako reálná čísla a vyzkoušejte jeho funkci. Snažte se otestovat bezchybnost programu.

2. Pokuste se zjistit, zda není zapotřebí použít i další ošetření hodnot při zadání parametrů kvadratické rovnice. Pokud se Vám podaří odhalit nějaká úskalí, upravte program tak, aby nedošlo k nechtěnému ukončení programu chybou.

3. Napište program, který by na základě znalosti vstupních logických proměnných, které můžete například ovládat pomocí komponent „CheckBox“ (viz ***4a Komponenty***), vygeneroval a zobrazil logickou hodnotu výstupní logické proměnné y = a.b+a$\underline{c}$+$\underline{bd}$.(a+$\underline{c}$), kde $\underline{x}$ je negace x, x.x je logický součin a x+x je logický součet.

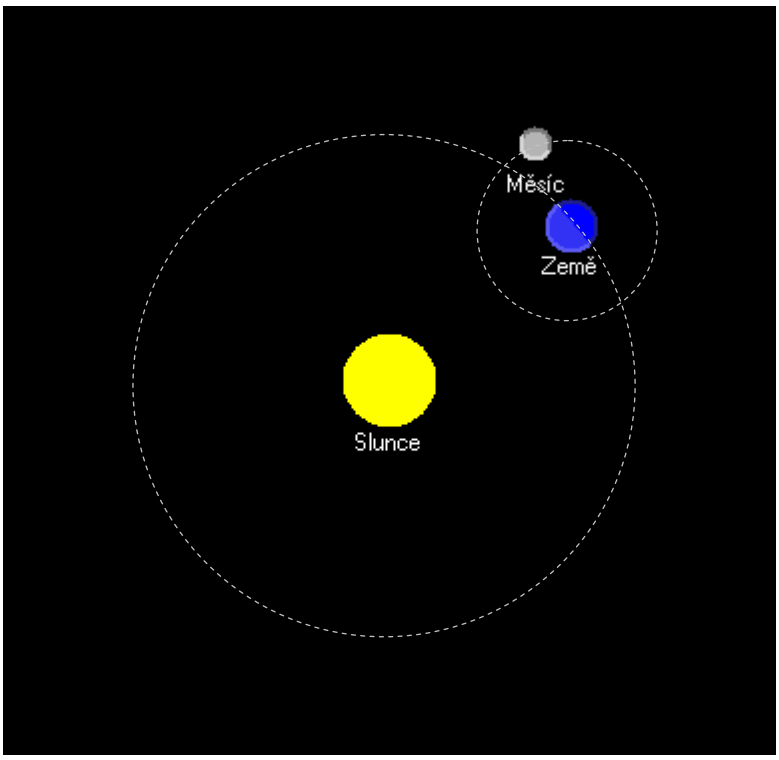
6.3. Cykly

Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat programový cyklus.
- rozdělit cykly.
- vybrat vhodný typ cyklu pro různé aplikace.

Výklad



V běžném životě se setkáváme s jevy, které se po určité době opakují. Můžeme například pozorovat pravidelné střídání dne a noci, jednotlivých ročních období, úplňku a novu měsíce. Při určitém zjednodušení jde ve všech těchto případech o časově pravidelně se opakující úkazy, které lze vyjádřit přesnou periodou opakování. Den se opakuje po 24 hodinách, rok po 365,25 dnech a úplněk měsíce můžeme pozorovat po 29,5 dnech. Tato pravidelná opakování nazýváme cykly (viz animační program z CD pod názvem ***6c Obeh Slunce_Cykly***).

Jako cykly však označujeme i méně pravidelně opakující se děje jako například koloběh vody v přírodě, klimatické doby ledové, záplavy nebo erupce na slunci. Pro naše účely tedy můžeme označovat cyklem v širším slova smyslu jakékoliv jevy nebo činnosti, které se několikrát opakují. Počet opakování je závislý na konkrétním případě, vždy však musí být z hlediska programování konečným číslem. Vždyť i zdánlivě nekonečný cyklus střídání dne a noci je omezen dobou existence naší sluneční soustavy.

Máme-li například vypočítat několik bodů grafu funkce, jsme nuceni několikrát zopakovat tutéž činnost, avšak pro různé hodnoty nezávisle proměnné. Na kalkulátoru by to znamenalo opakovaně zadat nezávisle proměnnou, stisknout tlačítko příslušné funkce a vypočtený výsledek zapsat do připravené tabulky.

Ve vlastním programu bychom sice také mohli v mnoha řádcích zopakovat stejný příkaz pro různé hodnoty nezávisle proměnné, ale nebylo by to přinejmenším účelné.

```
Memo1.lines.Add(sin(0.0));
Memo1.lines.Add(sin(0.1));
Memo1.lines.Add(sin(0.2));
....
```

V praxi se setkáváme se dvěma různými podmínkami pro ukončení cyklu. Často můžeme v návodu číst:

a) „...směs mícháme po dobu dvou minut ...“,

b) „...směs mícháme dokud nezmizí hrudky ...“.

Kdybychom tyto podmínky aplikovali na předešlý příklad výpočtu funkce sinus, mohlo by to vypadat asi takto:

a) „Vypočtěte hodnoty funkce sin(x), pro $0 \leq x \leq 1$, s krokem 0.1“,

b) „Vypočtěte hodnoty funkce sin(x), pro x měnící se s krokem 0.1 od hodnoty 0 do té hodnoty x, kdy hodnota sin(x) překročí číslo 0.8 “.

Z obou zadání je patrno, že případ a) jednoznačně určuje počet opakování v cyklu (2 minuty, 11 cyklů), kdežto v případě b) je cyklus ukončen až po splnění podmínky, která souvisí s hodnotou nezávisle proměnné zprostředkovaně přes nějakou funkci. V případě b) tedy není dopředu znám počet opakování v cyklu.

Tomu jsou přizpůsobeny i příkazy programovacího jazyka Delphi. Rozeznáváme tak příkazy pro realizaci cyklu s pevným počtem opakování a příkazy řízené podmínkou.

a) cyklus s pevným počtem opakování (nepomíněný) – jde o příkaz se syntaxí:

```
for Celociselna_ridici_promenna:= Dolni_mez to Horni_mez do prikaz;
```

```
for i: = 0 to 10 do
```

```
Memo1.Lines.Add(sin(0.1*i));
```

Někdy je účelné použít cyklus s klesající hodnotou řídicí proměnné. Pak se změní syntaxe příkazu takto:

```
for Celociselna_ridici_promenna:= Horni_mez downto Dolni_mez do prikaz;
```

```
for i: = 10 downto 0 do
   Memo1.Lines.Add(sin(0.1*i));
```

Kdyby tento příkaz neexistoval, museli bychom využít předcházející v této úpravě:

```
for i: = 0 to 10 do
   Memo1.Lines.Add(sin(0.1*(10 – i)));
```

Tento cyklus provede příkaz jen jednou (bez opakování):

```
for i: = 1 to 1 do
   Memo1.Lines.Add(sin(0.1* i));
```

a tento cyklus příkaz nevykoná ani jednou:

```
for i: = 10 to 0 do
   Memo1.Lines.Add(sin(0.1*(10 – i)));
```

b) cykly řízené podmínkou (podmíněné) jsou dvojího druhu:

- s testem podmínky na začátku cyklu
- s testem podmínky na konci cyklu

Při testu podmínky na začátku cyklu jsou příkazy v cyklu prováděny po tu dobu, dokud je zadaná podmínka splněna – jde o příkaz se syntaxí:

```
while logicka_podmika do
    prikaz;
```

```
i:=0;
while i <= 10 do
   begin
      Memo1.Lines.Add(sin(0.1*i));
      Inc(i);
   end;
```

Při testu podmínky na konci cyklu jsou příkazy v cyklu prováděny tak dlouho, dokud není zadaná podmínka splněna – jde o příkaz se syntaxí:

```
repeat
   prikaz;
until logicka_podmika;

i:=0;
repeat
   Memo1.Lines.Add(sin(0.1*i));
   Inc(i);
until i >= 10;
```

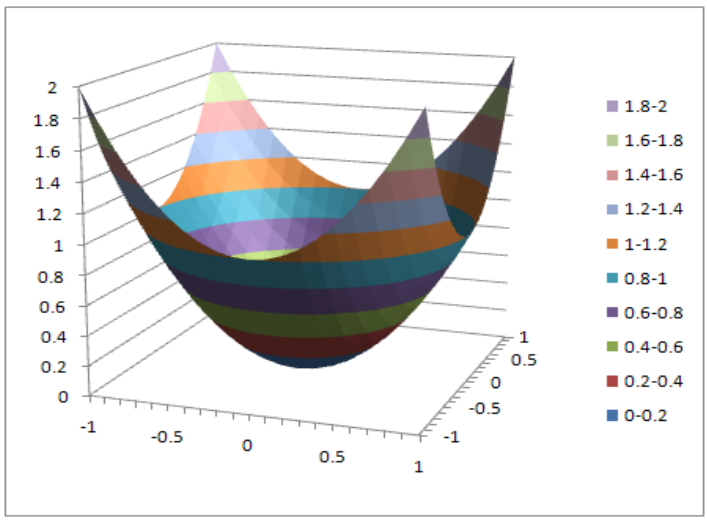
Vnořené cykly

Nejde v zásadě o nic nového. Každý cyklus opakovaně provádí určitou sekvenci libovolných příkazů. Není tedy vyloučeno, že se v této sekvenci objeví příkaz dalšího cyklu. Hovoříme o tzv. vnořeném cyklu. Počet vnoření není prakticky omezen, ale v praxi se nejspíš setkáme s jedním vnořením, případně dvěma. Použijeme-li vnoření u cyklů s pevným počtem kroků, musíme dbát o to, aby měl každý cyklus svou vlastní řídicí proměnnou. Vnořené cykly mohou být i kombinací různých typů (**for, while, repeat**).

Nejlépe si vnořené cykly vysvětlíme na následujícím příkladu:

Máme za úkol vypočítat funkční hodnoty dvojrozměrné funkce definované vztahem,

$$z = f(x,y) = x^2 + y^2$$



Obr. 6.1 Rotační paraboloid

Jde o matematický popis rotačního paraboloidu, který si můžeme představit z obr. 6.1. Funkce je definována pro x i y přes celou množinu reálných čísel, my však musíme na počítači řešit problém pouze v diskrétních bodech s rozumnou roztečí a předem definovaným rozsahem obou nezávislých proměnných. Proto zvolíme

$$x \in \langle -1;1 \rangle \quad a \quad y \in \langle -1;1 \rangle$$

v diskrétních hodnotách s krokem 0,1. Funkční hodnoty ve všech definovaných bodech uložíme do proměnné typu pole (**array**).

Pro realizaci výpočtu, kdy musíme postupně dosazovat do funkce všechny dvojice hodnot x a y - {[-1;1],[-1;-0.9],[-1;0.8],…[i,j]…[1;0.9],[1;1]}, budeme volit dva vnořené cykly s pevným počtem opakování. Jako řídící proměnné cyklů zvolíme celočíselné konstanty „i“ a „j“, které musíme deklarovat jako typ integer. I když to není zcela nezbytné, vytvoříme také proměnné „x“ a „y“, které budou zastupovat nezávislé proměnné funkce, a proto je musíme deklarovat jako datový typ real.

Výsledné hodnoty uložíme do dvojrozměrného pole (**array**) s názvem „z“, které musíme nejdříve deklarovat v sekci programu **var**. Vypočtené funkční hodnoty jsou desetinná čísla, proto pole „z“ musí být typu real. Jak ale správně pole dimenzovat? Již víme, že půjde o pole dvojrozměrné. Proměnná x, tak jako i y se má měnit od -1 po jedné desetině až do hodnoty 1, a to je přesně 21 hodnot. Oba rozměry pole tedy budeme muset dimenzovat na 21 prvků.

Šlo by sice pole dimenzovat od nuly do dvaceti [0..20], ale my musíme nalézt co nejjednodušší vztah mezi proměnnou „i“ a „x“, tak jako mezi proměnnou „j“ a „y“. Proto bude daleko výhodnější, když budeme pole dimenzovat od -10 do 10. Jak bude sekce **var** vypadat po deklaracích proměnných je vidět z následného fragmentu zdrojového kódu:

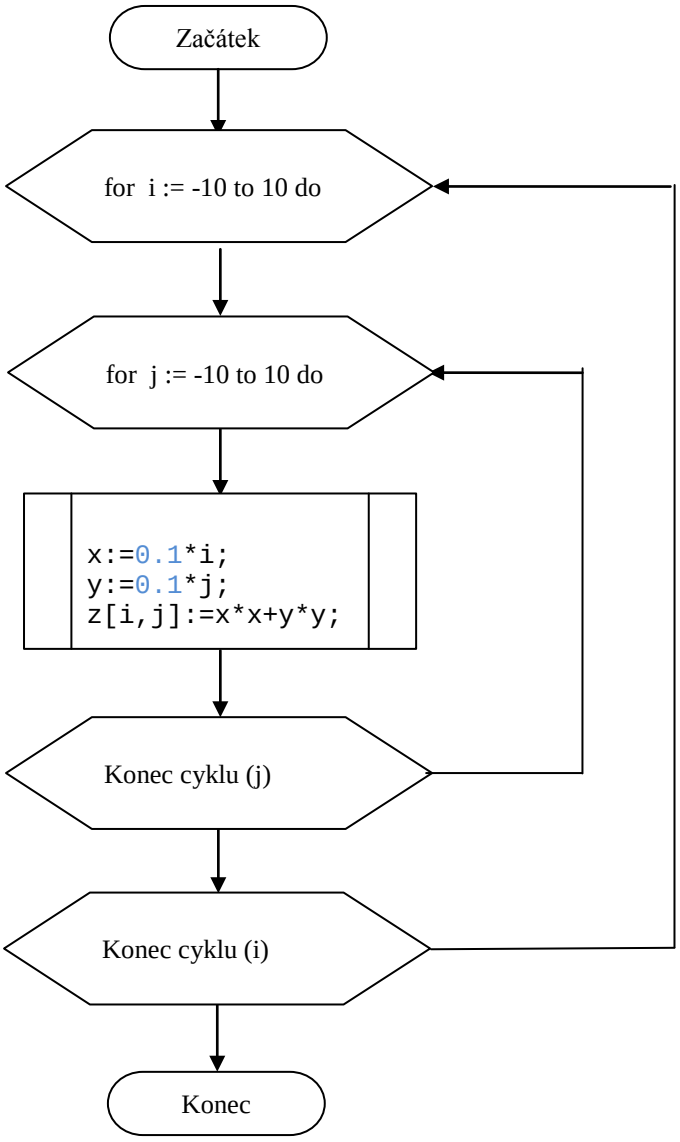
```
var
  Form1: TForm1;
  i,j: integer;
  x,y: real;
  z: array [-10..10,-10..10] of real;
```

Pro vlastní výpočet použijeme jeden cyklus, který bude postupně generovat hodnoty řídicí proměnné „i“ od -10 do 10, a pak už je snadné matematicky svázat hodnotu proměnné „x“ s okamžitou hodnotou proměnné „i“ tak, aby se měnila od -1 do 1. Převodním vztahem je vynásobení hodnoty proměnné „i“ desetinou. Druhý cyklus bude vnořen do prvního a bude zcela obdobně řešit výpočet vztahu mezi „j“ a „y“. Protože se systematicky se měnícími hodnotami řídicích proměnných cyklů „i“ a „j“ se zároveň vhodně mění i hodnoty proměnných „x“ a „y“, můžeme v každém bodě vypočítat funkční hodnotu a tu uložit do odpovídajícího prvku pole „z“. Více informací o problematice polí můžete získat spuštěním animovaného programu z CD pod názvem ***6d Pole promennych***.

Spuštění výpočtu můžeme inicializovat události kliknutí na tlačítko tak, jak je patrno z následujícího fragmentu zdrojového kódu.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  for i:=-10 to 10 do
    for j:=-10 to 10 do
      begin
        //výpočet nezávise proměnných
        x:=0.1*i;
        y:=0.1*j;
        //výpočet funkční hodnoty s uložením do pole "z"
        z[i,j]:=x*x+y*y;
      end;
end;
```

Názorná ukázka funkce programu ve formě vývojového diagramu je uvedena na obr. 6.2.



Obr. 6.2 Vývojový diagram pro výpočet funkce $z = (x^2+y^2)$

Shrnutí pojmů

Cyklem nazýváme takovou programovou konstrukci, která umožní automaticky provádět opakovaní určité části programu (sekvence cyklů). Cyklus je ukončen po splnění ukončovací podmínky.

a) cyklus s pevným počtem opakování (nepodmíněný) – jde o příkaz se syntaxí:

```
for Celociselna_promenna:= Dolni_mez to Horni_mez do prikaz;
    Opakované příkazy;
```

nebo

```
for Celociselna_promenna:= Horni_mez downto Dolni_mez do prikaz;
  Opakované příkazy;
```

b) cykly řízené podmínkou (podmíněné) jsou dvojího druhu:

- s testem podmínky na začátku cyklu

```
while logicka_podmika do
    prikaz;
```

- s testem podmínky na konci cyklu

```
repeat
   prikaz;
until logicka_podmika;
```

Otázky

Co je to cyklus?

Jaké druhy cyklu znáte?

Co to jsou vnořené cykly?

Úlohy k řešení

Naprogramujte úlohu pro řešení funkce $x^2 + y^2$ a pokuste se zjistit, jaké hodnoty jsou uloženy v poli „z“.

6.4. Vlastní procedury a funkce

Čas ke studiu: 1,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat a využívat procedury.
- definovat a využívat funkce.

Výklad

Už jsme se seznámili s různými funkcemi a procedurami, které jsou součástí knihovny programovacího jazyka. Vytvořili jsme si i několik procedur událostí. Řekli jsme si, že jde vlastně o nějaké podprogramy, které na požádání (volání funkce nebo procedury) řeší určité specifické zadání (vypočtou hodnotu sinus v požadovaném bodě apod.). Byly vytvořeny proto, aby nebylo třeba stále programovat běžné, často se opakující algoritmy i značné složitosti. Navíc každý složitější algoritmus je vhodné tvořit hierarchickým způsobem. Zjednodušeně řečeno - vyšší vrstvy pracují s obecnějšími informacemi (vlastní tělo programu), nižší vrstvy řeší detaily (procedury a funkce). Využití této myšlenky může podstatně zpřehlednit vlastní program a navíc umožní rozdělit komplexní úlohu do separátních celků, které mohou být řešeny samostatně a nezávisle na ostatních. Musí však být dopředu dána pravidla pro předávání údajů mezi nimi a hlavním programem.

Pokud se v našem programu budou opakovat některé výpočty vícekrát, zvláště pak v různých částech programu, je výhodné naučit se vytvářet vlastní funkce a procedury. Problematika tvorby funkcí a procedur není úplně jednoduchá, proto se zde seznámíme jen s nezbytně nutnými poznatky tak, abychom mohli procedury a funkce v praxi programování využít.

Procedura se skládá z:

- označení, kterým je vždy klíčové slovo **procedure**,
- identifikátoru, kterým je název procedury,
- formálních parametrů s udáním typu (nejsou povinné),
- a vlastního těla podprogramu, který řeší specifický algoritmus procedury.

```
procedure NazevProcedury(parametry:DatovyTyp);
  begin
   … tělo podprogramu (algoritmus procedury);
  end;
```

Jako příklad si zde uvedeme proceduru, která skryje komponentu Edit1 pokud byla viditelná a naopak.

```
procedure prepni;
  begin
   Form1.Edit1.Visible := not(Form1.Edit1.Visible);
  end;
```

Chceme-li proceduru využít pro výpočet, musíme ji z hlavního programu, případně z jiné procedury nebo funkce volat. Volání musí obsahovat označení, identifikátor, skutečné parametry shodných datových typů ve správném počtu a pořadí (jsou-li v deklaraci volané procedury).

Volání procedury pro přepínání viditelnosti komponenty Edit1 by vypadalo následovně:

```
prepni;
```

Funkce se skládá z:

- označení, kterým je vždy klíčové slovo **function**,
- identifikátoru, kterým je název funkce,
- formálních parametrů s udáním typu (nejsou povinné),
- definice datového typu funkcí vraceného výsledku,
- a vlastního těla podprogramu, který řeší specifický algoritmus funkce.

```
function NazevFunkce(parametry:DatovyTyp):DatovyTypVysledkuFunkce;
  begin
   … tělo podprogramu (algoritmus funkce);
  end;
```

Funkce na rozdíl od procedury vrací výsledek řešení ve formě vypočtené hodnoty.

Jako příklad si zde uvedeme funkci, která vypočte objem válce.

```
function Valec(polomer,vyska:real):real;
  begin
   Valec := PI*sqr(polomer)*vyska;
  end;
```

Chceme-li funkci využít pro výpočet, musíme ji z hlavního programu, případně z jiné procedury nebo funkce volat. Volání je ve formě příkazu přiřazení, a proto musí obsahovat proměnnou, do které má být výsledek řešení přiřazen, symbol přiřazení, označení, identifikátor, skutečné parametry shodných datových typů ve správném počtu a pořadí (jsou-li v deklaraci volané funkce).

Volání funkce pro výpočet objemu válce pro poloměr r =10 m a výšku v = 2 m by vypadalo následovně:

```
ObjemValce := Valec(10,2);
```

Shrnutí pojmů

Procedura se skládá z:

- označení, kterým je vždy klíčové slovo **procedure**,
- identifikátoru, kterým je název procedury,
- formálních parametrů s udáním typu (nejsou povinné),
- a vlastního těla podprogramu, který řeší specifický algoritmus procedury.

```
procedure NazevProcedury(parametry:DatovyTyp);
  begin
   … tělo podprogramu (algoritmus procedury);
  end;
```

Funkce se skládá z:

- označení, kterým je vždy klíčové slovo **function**,
- identifikátoru, kterým je název funkce,
- formálních parametrů s udáním typu (nejsou povinné),
- definice datového typu funkcí vraceného výsledku,
- a vlastního těla podprogramu, který řeší specifický algoritmus funkce.

```
function NazevFunkce(parametry:DatovyTyp):DatovyTypVysledkuFunkce;
  begin
   … tělo podprogramu (algoritmus funkce);
  end;
```

Otázky

Co je to procedura a jak se používá?

Co je to funkce a jak se používá?

Úlohy k řešení

1. Naprogramujte funkce, které nejsou v základní knihovně Delphi. Jde například o funkce tangens, kotangens, logaritmus se základem deset, faktoriál apod.

2. Vyzkoušejte si naprogramovat nějaké složitější procedury a funkce a ověřte jejich funkčnost.

7. GRAFIKA

7.1. Základní grafické prostředky

Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- využívat komponentu Image.
- aplikovat základní grafické příkazy.

Výklad

Vše co doposud umíme naprogramovat, může být navenek prezentováno jen v textové formě. Pokud pracujeme přímo s textem nebo je-li výstupem několik málo čísel, pak je to v pořádku. Ale počítáme-li např. body grafu nějaké funkce, jsme výstupem omezení pouze na tabulkovou formu.

Jenže programovací jazyk Delphi nabízí i takové prostředky, abychom byli schopni naprogramovat nejen graficky průběh funkce, ale i nějaké tvary a popisy. Pokud budeme mluvit o grafice v Delphi, pak budeme mít na mysli operace s jednotlivými body (pixely) na obrazovce. Delphi nám nabízí několik možností využití grafiky, my se však zaměříme jen na jednu základní.

Komponenta Image.

Nejprve budeme potřebovat komponentu, která je ke grafickým operacím určena, tzn. která má takové metody, jež umožňují ovládat nastavení barvy jednotlivých pixelů. Touto komponentou, i když tyto možnosti nabízí i samotný formulář, je komponenta s názvem „Image“.

Ta na rozdíl od samotného formuláře zajišťuje automatické překreslování své plochy, je-li například překryta nějakým oknem jiné aplikace Windows. To formulář z principu práce OS MS Windows nedokáže.

Komponenta „Image“ se nachází v záložce přídavných (Additional) komponent. Po přetažení do formuláře a spuštění projektu se ovšem nikde v okně neukáže, dokud do ní nezačnete „kreslit“. Zbytečně ji proto nehledejte a rychle se naučte do ní něco „nakreslit“.

Nyní si popíšeme základní příkazy, které nám to umožní.

Každý příkaz, který budeme pro „kreslení“ používat musí začínat názvem komponenty, do které chceme kreslit. Tím je „Image“ s pořadovým číslem tak, jak se nám vytvořilo při vložení komponenty do formuláře. Pokud jsme název nezměnili, tak předpokládejme, že bude „Image1“. Před tento název můžeme předřadit název rodiče, jak jsme již dříve používali.

```
Form1.Image1.
```

Po zadání poslední tečky se nám objeví nabídka vlastností komponenty „Image“. Vidíme, zde již některé známé vlastnosti například pro nastavení rozměrů komponenty, její polohy a podobně. My však chceme kreslit, a proto nás v té chvíli bude nejvíce zajímat vlastnost „Canvas“ neboli plátno, na které již lze kreslit.

```
Form1.Image1.Canvas.
```

A to už máme před sebou nabídku všech vlastností a metod souvisejících s plátnem (Canvas). Ty nejvýznamnější si teď popíšeme.

1) Chceme-li nakreslit na obrazovce jeden elementární bod (pixel) určité barvy, použijeme na to metodu „pixels“. Ta má sice dva významy, ale bod na obrazovce v objektu „Image1“ vykreslí příkaz typu:

```
Form1.Image1.Canvas.Pixels[x,y] := barva;
```

kde x,y jsou celočíselné hodnoty souřadnice bodu v komponentě „Image1“. Barva je určena buď předdefinovanou konstantou jako např. `clred`, nebo může být využita funkce RGB(r,g,b).

Příklad použití:

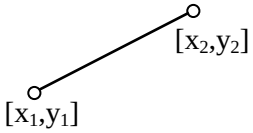
```
Form1.Image1.Canvas.Pixels[10,20] := clred;
```

nebo

```
Form1.Image1.Canvas.Pixels[25,45] := RGB(255,0,0);
```

Takto vytvořený jediný bod na obrazovce je sotva viditelný, ale pokud byste tento příkaz použili k vykreslení celého grafu funkce, bylo by to možné.

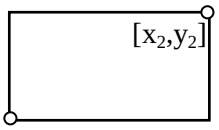
2) Často je třeba nakreslit přímku, která je určena svými krajními body v ploše komponenty. Na to ovšem budeme potřebovat dva příkazy. Prvním příkazem (MoveTo) umístíme kurzor na místo počátku přímky a druhým příkazem (LineTo) z tohoto bodu vykreslíme čáru (přímku) do koncového bodu, který je parametrem tohoto příkazu.



Příklad použití:

```
Form1.Image1.Canvas.MoveTo(x1,y1);
Form1.Image1.Canvas.LineTo(x2,y2);
```

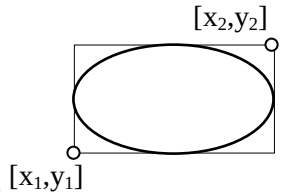
3) Obdélník je také jeden ze základních tvarů používaných při kreslení. Určení velikosti, tvaru a polohy obdélníka se provádí určením souřadnic dvou jeho protilehlých rohů.



Příklad použití:

```
Form1.Image1.Canvas.Rectangle(x1,y1,x2,y2);
```

4) Dalším často používaným grafickým prvkem je elipsa, která se využívá i pro kreslení kružnic. Určení velikosti, tvaru a polohy elipsy se provádí souřadnicemi dvou protilehlých rohů obdélníka, do kterého je elipsa vepsaná.



Příklad použití:

```
Form1.Image1.Canvas.Ellipse(x1,y1,x2,y2);
Form1.Image1.Canvas.Ellipse(x-r,y-r,x+r,y+r);
```

5) Na plátno komponenty Image musí jít umístit i textové informace

Za tímto účelem existuje metoda TextOut.

Příklad použití:

```
Form1.Image1.Canvas.TextOut(x,y,'text');
```

Tento příkaz umístí požadovaný text do místa určeného souřadnicemi x, y. Je samozřejmé, že text může být umístěn v proměnné typu „string“ (nebo i číselné s následnou konverzí do textu), která nahradí text psaný přímo do příkazu.

6) Barvu bodu v příkazu „pixels“ nebylo složité definovat, ale jak postupovat při změně barvy, případně dalších atributů nakreslených objektů? Pro určení parametrů obrysů použijeme jednu z vlastností plátna pod názvem Pen (pero).

```
Form1.Image1.Canvas.Pen.
```

Z nabídky vlastnosti pera můžeme vybrat například barvu (Color) nebo tloušťku (Width) a přiřadit jim požadovanou hodnotu:

Příklad použití:

```
Form1.Image1.Canvas.Pen.Color:=clblue;
Form1.Image1.Canvas.Pen.Width:=3;
```

Obrys objektu bude modrý v tloušťce 3 pixelů.

Barvu, případně styl výplně grafických objektů můžeme nastavit ve vlastnosti plátna s názvem Brush (štětec).

```
Form1.Image1.Canvas.Brush.Color:=clgreen;
```

Výplň objektu bude v tomto případě zelená.

To by mělo stačit k tomu, abychom dokázali dostatečně využít grafické možnosti při programování. Jistě jste si všimli velkého možností položek ve vlastnostech plátno (canvas). Pokud budete mít zájem, můžete si vyzkoušet, co které umožňují. Vodítkem vám může být animační výukový program z CDpod názvem ***7a Graficke funkce****, nebo helpy programu Delphi.*

TIP!

Někdy je výhodné programově zjišťovat velikost kreslící plochy, přesněji řečeno šířku (Width) a výšku (Height) komponenty „Image“ v pixelech. K tomu můžeme využít dvou vlastností komponenty Image:

```
xmax:= Form1.Image1.Width;
ymax:= Form1.Image1.Height;
```

To můžeme využít při mazání plátna, které provedeme překreslením celé plochy obdélníkem definované barvy:

```
Form1.Image1.Canvas.Pen.Color:=clwhite;
Form1.Image1.Canvas.Brush.Color:=clWhite;
Form1.Image1.Canvas.Rectangle(0,0,xmax,ymax);
```

Shrnutí pojmů

Komponenta **Image** je určena ke grafickým operacím, která má takové metody, jež umožňují ovládat nastavení barvy jednotlivých pixelů, kreslit přímky a jiné jednoduché geometrické tvary.

Základní kreslicí funkce a procedury

Vykreslení bodu na pozici x, y v definované barvě:

```
Form1.Image1.Canvas.Pixels[x,y] := barva;
```

Přemístění kurzoru na pozici x, y:

```
Form1.Image1.Canvas.MoveTo(x1,y1);
```

Vykreslení úsečky z místa kurzoru do bodu x_2, y_2:

```
Form1.Image1.Canvas.LineTo(x2,y2);
```

Vykreslení obdélníku určením souřadnic dvou jeho protilehlých rohů:

```
Form1.Image1.Canvas.Rectangle(x1,y1,x2,y2);
```

Vykreslení elipsy určením souřadnic dvou protilehlých rohů opsaného obdélníku:

```
Form1.Image1.Canvas.Ellipse(x1,y1,x2,y2);
```

Vypsání textu na pozici x, y:

```
Form1.Image1.Canvas.TextOut(x,y,'text');
```

Otázky

1. Jakým způsobem je možno nakreslit v komponentě Image bod?
2. Jakým způsobem je možno nakreslit v komponentě Image úsečku?
3. Jakým způsobem je možno nakreslit v komponentě Image obdélník?
4. Jakým způsobem je možno nakreslit v komponentě Image elipsu?
5. Jakým způsobem je možno nakreslit v komponentě Image kružnici?
6. Jak se nastavuje barva obrysu a výplně?

Úlohy k řešení

1. Vyzkoušejte si naprogramovat nějaké složitější obrazce v komponentě Image a ověřte jejich funkčnost programu.
2. Vytvořte proceduru pro vykreslení kružnice.
3. S pomocí animovaného programu z CD pod názvem ***7b Kresleni mysi*** se pokuste vytvořit grafický program ke kreslení.

7.2. Vykreslení grafu funkcí

Čas ke studiu: 2,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- využít grafické příkazy pro vykreslení grafů.
- využít grafické příkazy pro popisy os grafů.

Výklad

Velmi častou úlohou při programování je nutnost zobrazit grafy funkcí nebo jiných grafických závislostí. Ve vývojových prostředích programovacích jazyků sice mohou být nabízené již vytvořené komponenty pro realizaci grafu, ale my se zde naučíme využívat základní příkazy pro grafiku tak, abychom byli schopni jednodušší grafy funkcí vytvořit.

Protože jde o vytvoření grafické prezentace, bude vhodné použít komponentu „Image“, která na svém plátně „Canvas“ umožňuje vytvářet elementární tvary a texty, které lze snadno využít pro kreslení grafů.

I když by v zásadě nebyl problém vytvořit různé typy grafů, zde si ukážeme, jak vytvořit základ pro nejpoužívanější grafy, a to grafy spojnicové, nebo přesněji grafy x-y, podobně jak je známe na příklad z Excelu. Do formuláře vložíme komponentu „Image1“ a upravíme její velikost a polohu ve formuláři.

To můžeme provést i softwarově použitím následujících příkazů:

```
Form1.Image1.Top := Poloha_obrazku_odzhora;
Form1.Image1.Left := Poloha_obrazku_zleva;
Form1.Image1.Width := Sirka_obrazku;
Form1.Image1.Height := Vyska_obrazku;
```

Nyní bude třeba vytvořit osy x a y. Jejich poloha je závislá na konkrétní úloze a může být dána stabilně na nějaké hodnotě, uživatel si jí může volit nebo může být poloha os vypočtena přímo počítačem. Zde zvolíme nejjednodušší variantu s pevnou polohou os. Protože budeme vykreslovat funkce sinus a kosinus, povedeme osu x přesně středem komponenty „Image1“ a osu y u jejího levého okraje tak, aby zbylo místo na popisky os. Těmito požadavky jsou dány souřadnice počátku souřadnicového systému (deklarujeme dvě celočíselné proměnné X0 a Y0), ke kterému budeme následně vztahovat veškerá zobrazovaná data a určíme je použitím následujících příkazů:

```
X0 := 40;
Y0 := Form1.Image1.Height div 2;
```

Vhodně si zvolíme délky os s ohledem na velikost obrázku a očekávané hodnoty závislé i nezávislé proměnné. Námi zvolené goniometrické funkce budeme zobrazovat pro x od 0° do 360°, v měřítku 1:1, a protože hodnoty závislé proměnné y jsou v rozmezí od -1 do 1, zvolíme měřítko 200:1.

Z toho vyplývá, že délky os x a y (v pixelech) můžeme určit použitím následujících příkazů:

```
delka_x := 360;
delka_y := 200*2;
```

Pokud bychom chtěli vykreslit složitější funkce, u kterých nejsme schopni ani odhadnout průběh, museli bychom měřítka nastavit softwarově po zjištění extrémů zobrazované funkce v požadovaném pro sledované hodnoty nezávislé proměnné x.
Když máme délky os a souřadnice počátku souřadnicového systému, už nám nic nebrání osy vykreslit.

Pro vykreslení osy x použijeme následující příkazy:

```
form1.Image1.Canvas.MoveTo(X0-2,Y0);
form1.Image1.Canvas.LineTo(X0+delka_x,Y0);
```

a pro vykreslení osy y použijeme následující příkazy:

```
form1.Image1.Canvas.MoveTo(X0,10);
form1.Image1.Canvas.LineTo(X0,Form1.Image1.Height -10);
```

Nyní bude zapotřebí na jednotlivé osy umístit ekvidistantní (stejně vzdálené od sebe) značky, odpovídající významným hodnotám pro zobrazení.

Podmínka stejné vzdálenosti dává možnost provést kreslení značek v cyklu, což program značně zjednoduší. Jako značky bývá zvykem používat krátké čárky kolmé na osu a jejich vzdálenosti od sebe (v pixelech) si pro jednotlivé osy definujeme do celočíselných proměnných například takto:

```
pocet_dilku_x := 360;
pocet_dilku_y := 20;
```

Z délky osy a požadované vzdálenosti mezi značkami vypočteme počet úseků na ose. Jelikož počet úseků musí být celé číslo, které po přičtení jedničky udává počet značek a tím i maximální hodnotu řídicí proměnné cyklu, pomocí kterého budeme značky vykreslovat, musíme zde použit celočíselné dělení.

```
pocet_useku_x := delka_x div pocet_dilku_x ;
pocet_useku_y := delka_y div pocet_dilku_y ;
```

Vykreslení značek na osu x pak můžeme následujícím algoritmem:

```
for i:= 1 to pocet_useku_x do
  begin
    form1.Image1.Canvas.MoveTo(X0+pocet_dilku_x*i,Y0-2);
    form1.Image1.Canvas.LineTo(X0+pocet_dilku_x*i,Y0+2);
  end;
```

a podobně pro osu y:

```
for i:= 0 to pocet_useku_y do
  begin
    form1.Image1.Canvas.MoveTo(X0-2,Y0-pocet_dilku_y
          *(i-(pocet_useku_y) div 2));
    form1.Image1.Canvas.LineTo(X0+2,Y0-pocet_dilku_y
          *(i-(pocet_useku_y) div 2));
  end;
```

Umístění značek na osy nám graf sice podstatně zpřehlednilo, ale k úplné čitelnosti grafu budeme muset v místě značek vypsat skutečné hodnoty jak nezávislé proměnné (osa x), tak proměnné závislé (osa y). Z hlediska vlastního vykreslení číselných údajů bude postup obdobný jako při kreslení značek, jen pro určení hodnot čísel budeme muset využít hodnoty řídicí proměnné cyklu matematicky upravené příslušným měřítkem.

Vykreslení jednotlivých hodnot na osu x pak můžeme použít následující programové konstrukce:

```
for i:= 0 to (pocet_useku_x) do
  begin
    form1.Image1.Canvas.TextOut(X0+pocet_dilku_x*i,Y0+5,
         IntToStr(pocet_dilku_x*i));
  end;
```

a podobně pro osu y:

```
for i:= 0 to (pocet_useku_y) do
  begin
    form1.Image1.Canvas.TextOut(X0-25,Y0-8-pocet_dilku_y
           *(i-(pocet_useku_y) div 2),
         FloatToStr((pocet_dilku_y*i-delka_y div 2)/
         delka_y div 2));
  end;
```

kde: `delka_y div 2` je měřítko v ose y.

Osy jsou v tomto okamžiku připraveny a před námi je nyní informačně nejdůležitější část, a tou je vykreslení vlastní grafické podoby zadané funkce. Nejprve červenou barvou vykreslíme graf funkce y = sin(x) a pak jen příkazy mírně upravíme a vykreslíme modře funkci y = cos(x).

Zde je nutno si uvědomit, že v komponentě „Image“ (ale nejen tam) se s rostoucí hodnotou souřadnice y (svislé), pohybujeme na obrazovce směrem dolů, kdežto rostoucí hodnoty vykreslované funkce bývá zvykem kreslit opravdu směrem nahoru. Toho nejlogičtěji dosáhneme výrazem Y0 – y, čímž se nám zároveň graf funkce vztáhne k počátku souřadnicového systému (samozřejmě nesmíme zapomenout na Y0).

Protože vyžadujeme, aby byl graf vykreslen v každém bodě obrazovky ve směru osy x, použijeme i zde cyklus s konečným počtem opakování, kde právě proměnna delka_x určuje maximální hodnotu řídicí proměnné cyklu. Jelikož budeme chtít graf vykreslit již pro x = 0, bude počáteční hodnota řídicí proměnné cyklu také rovna nule. V našem případě tedy získáme 361 průchodů v cyklu a tím i 361 bodů na obrazovce (delka_x je 360 a jeden bod v nule).

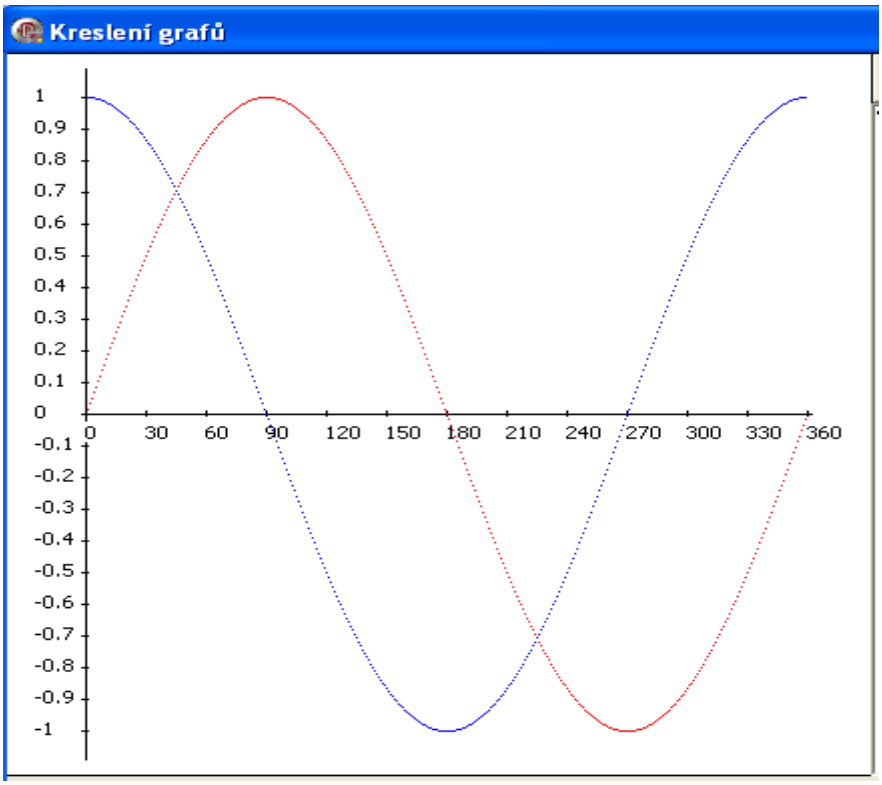
Pro funkci vykreslení křivky sin(x) můžeme vytvořit tento program:

```
for i:= 0 to (pocet_useku_y) do
    begin
      x:=i*pi/180;
      y:=(delka_y div 2)*sin(x);
      form1.Image1.Canvas.Pixels[X0+i,Round(Y0-y)]
                                        :=clred;
    end;
```

Chceme-li vykreslit do stejného obrázku ještě funkci cos(x), stačí poslední fragment programu zkopírovat a přepsat jen funkci a barvu. Bude to vypadat následovně:

```
for i:= 0 to (pocet_useku_y) do
    begin
      x:=i*pi/180;
      y:=(delka_y div 2)*cos(x);
      form1.Image1.Canvas.Pixels[X0+i,Round(Y0-y)]
                                        :=clblue;
    end;
```

Grafický výstup po spuštění vytvořeného programu můžeme vidět na obr. 7.1. Ještě by bylo vhodné jednotlivé osy popsat a doplnit titulek grafu. Již víme, že nám stačí vhodně aplikovat příkaz `form1.Image1.Canvas.TextOut(x,y,'text');`. Zkuste si to již sami.



Obr. 7.1 Využití grafických příkazů pro vykreslení grafu funkcí

Shrnutí pojmů

Pro kreslení grafu funkcí musíme umět použít následující příkazy:

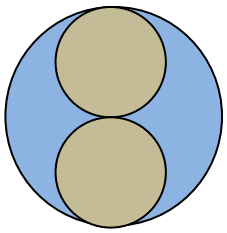
```
Form1.Image1.Top := Poloha_obrazku_odzhora;
Form1.Image1.Left := Poloha_obrazku_zleva;
Form1.Image1.Width := Sirka_obrazku;
Form1.Image1.Height := Vyska_obrazku;
Form1.Image1.Canvas.MoveTo(X0,Y0);
Form1.Image1.Canvas.LineTo(X1,Y1);
Form1.Image1.Canvas.Pixels[X0,Y0]:=Barva;
Form1.Image1.Canvas.TextOut(X0,Y0);
```

Otázky

1. Které grafické funkce a procedury se využívají při vykreslování grafů?
2. Které grafické funkce a procedury byste využili pro kreslení sloupcových grafů?

Úlohy k řešení

1. Naprogramujte úlohu vykreslování grafu funkcí dle předchozího textu a graf doplňte o popis jednotlivých os a legendu. Pokuste se na místo příkazu „Pixels“ použít příkazy „MoveTo“ a „LineTo“. Křivka grafu pak nebude přerušovaná. Jako vzor Vám může posloužit animovaný program z CD pod názvem ***7c Grafy funkci***.
2. Pokuste se vytvořit program, jehož výstupem by byl následující obrazec. Ale pozor, změnou jediného parametru „r“ by měl snadno měnit svůj rozměr.



8. LADĚNÍ PROGRAMŮ

8.1. Chyby

Čas ke studiu: 1,0 hodin

Cíl: Po prostudování tohoto odstavce budete umět

- definovat jednotlivé typy chyb.
- identifikovat a odstraňovat chyby v programu.

Výklad

Ladění programů (nalezení a oprava chyb) není snadnou záležitostí. Mnoho chyb vznikne nepozorností programátora, kdy formální zápis zdrojového kódu není v souladu s definicemi programovacího jazyka (porušují pravidla jazyka). Takovým chybám říkáme ***syntaktické chyby.*** Se syntaktickými chybami nemůže být zdrojový kód přeložen a spuštěn. Zde je však situace poměrně jednoduchá, neboť počítač při snaze o překlad chyby tohoto typu snadno odhalí, podá o nich chybová hlášení a je na programátorovi, aby je uvedl na pravou míru v duchu programovacího jazyka. Teprve po odstranění všech syntaktických chyb lze program spustit.

POZOR!
Může se stát, že překládač identifikuje chybu v řádku, o kterém bychom s jistotou řekli, že je v pořádku. Poměrně často se stává, že chyba vznikne někde jinde a až v některém z následujících řádků se projeví. Typicky jde o opomenutí středníku za předchozím příkazem, anebo neshoda v mezích begin – end.

Příklad nejčastěji se vyskytujících syntaktických chyb

```
řádek      příkazy
26         procedure TForm1.FormCreate(Sender: TObject);
27         begin
28           i:=1;
29           j:=2
30           k:=6;
31           begin
32             l:=3
33         end;
34
35         end.
```

Při snaze o překlad indikuje překládač několik syntaktických chyb a podá o nich hlášení se slovním označením specifikace chyby. Hlášení pro výše uvedený program by vypadal následujícím způsobem:

[Error] Unit1.pas(28): E2003 Undeclared identifier: 'i'
[Error] Unit1.pas(29): E2003 Undeclared identifier: 'j'
[Error] Unit1.pas(30): E2066 Missing operator or semicolon
[Error] Unit1.pas(32): E2003 Undeclared identifier: 'l'
[Error] Unit1.pas(35): E2029 ';' expected but '.' found
[Error] Unit1.pas(37): E2029 Declaration expected but end of file found
[Fatal Error] Project1.dpr(5): F2063 Could not compile used unit 'Unit1.pas'

V řádku 28, 29 a 32 byla nalezena stejná chyba, která v překladu znamená, že nebyl deklarován datový typ proměnných „i“, „j“ a „l“. Aby překládač již tuto chybu nehlásil, je nutno tyto tři proměnné deklarovat v oddíle **var** programu – například:
var i,j,l : integer;
Proč ale v řádku 30, kde také není deklarovaná proměnná „k“ není nalezena stejná chyba? To proto, že v předchozím řádku chybí středník a překládač sice zaregistroval chybu, ale není jednoznačně schopen říct, o jakou ze dvou možných chyb jde. Z překladu hlášení o chybě v řádku 30 vyplývá, že buďto programátor zapomněl třeba znak nějaké matematické operace a řádek 30 je pokračováním řádku 29, anebo zapomněl na řádku 29 středník. Po opravě středníku by již překládač našel chybu i v nedefinované deklaraci proměnné „k“, kterou bychom snadno opravili připsáním „k“ mezi ostatní v oddíle **var**.
Další chyba není přímo v řádku 35, jak ji identifikoval překládač, tam se teprve projevila. V překladu by hlášení mohlo znít, že je očekáván středník, ale byla nalezena tečka. Přepsáním by sice chyba zanikla, ale chybělo by nám ukončení celého programu (**end.**). Kdybychom dopsali **end.** Bylo by sice po formální stránce vše v pořádku (v tomto programu dokonce zcela v pořádku), ale obecně jsme si tímto postupem mohli vyrobit chybu logickou – mohlo

dojít ke změně původní struktury programu. Program by šlo spustit, ale mohl by dávat špatné výsledky. Takové chyby se již hledají daleko hůře.

Proč se to stalo? Nehledali jsme opravdovou příčinu chyby, která se nachází někde před řádkem 35, překládač ji ale ještě není schopen identifikovat. Před řádkem 33 chybí napsat párový příkaz „end“ se středníkem k příkazu „begin“ z řádku 31 (v této konstrukci by bylo jednodušší „begin“ vymazat). Odstraněním této chyby bychom zároveň odstranili chybové hlášení o chybě na řádku 37.

Poslední hlášení jen potvrzuje to, v tomto případě nemůže být náš program úspěšně přeložen. To se podaří, až budou odstraněny všechny zde uvedené chyby.

Dalším, horším typem chyb jsou (jak již bylo zmíněno) ***chyby logické***. Ty není schopen počítač odhalit, a pokud je neodhalí během ladění programátor, mohou se projevit až při užívání programu nějakou chybou. Logická chyba může způsobit zkreslení výsledku řešení nebo mohou být dokonce generovány výsledky zcela nesmyslné.

I zde nám může být integrované vývojové prostředí podstatně zjednodušit práci při hledání a opravě logických chyb. Velmi častým zdrojem těchto chyb bývají formálně špatně převedené matematické vztahy syntaxe programovacího jazyka. Jako příklad můžeme uvést jednoduchý matematický vztah:

$$y = \frac{a}{b \cdot x}$$

I když jde o vztah velice jednoduchý, častou chybou při přepisu do programovacího jazyka je nerespektování pořadí provádění matematických operací. Pak bývá tento vztah reprezentován v programu takto:

```
y := a/b*x;
```

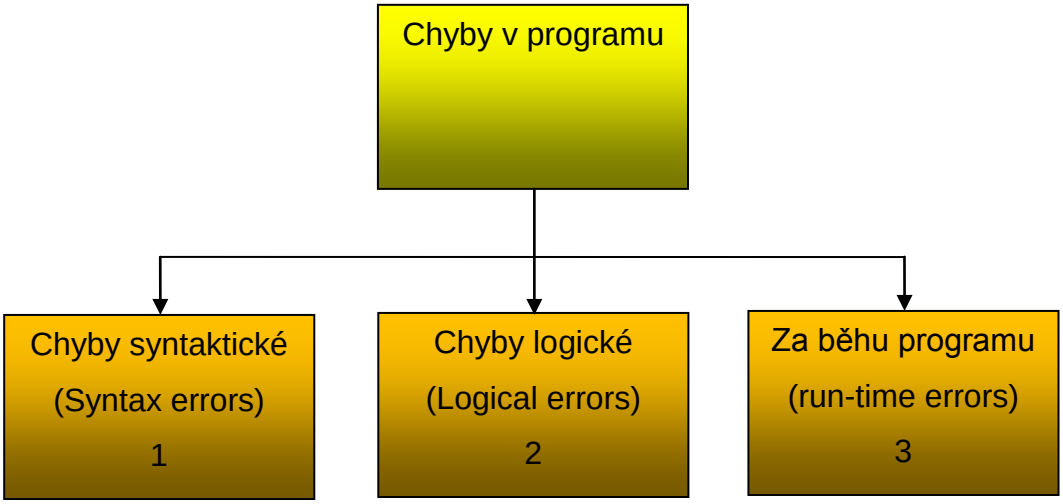
Program pak provádí operace o stejné prioritě zleva doprava, čímž vlastně vypočítá úplně jiný matematický vztah:

$$y = \frac{a}{b} \cdot x$$

Pokud se navíc stane, že hodnota x se pohybuje kolem nuly, je odhalení takové chyby i pomocí integrovaných prostředků pro ladění programu velmi obtížné.

Posledním typem jsou ***chyby za běhu programu,*** kdy program pracuje s nedovolenými hodnotami dat. To může způsobit, že se chod programu neočekávaně zcela zastaví.

Shrnutí pojmů



1. Program obsahuje chyby, které překračují pravidla jazyka Delphi.
2. Program pracuje podle špatného algoritmu.
3. Program pracuje s nedovolenými hodnotami dat.

Nejčastější chyby:

Program očekává na vstupu jiný typ dat.
Hodnota je mimo dovolený rozsah.
Hodnota dělitele je nulová.
Hodnota v odmocnině není nezáporná, v logaritmech není kladná.
Program se snaží otevřít neexitující datový soubor.
Program se snaží otevřít datový soubor, který je již otevřen jinou aplikací.
apod.

Otázky

1. Co jsou to syntaktické chyby, jak se projevují a co je jejich nejčastější příčinou?
2. Co jsou to logické chyby, jak se projevují a co je jejich nejčastější příčinou?
3. Co jsou to chyby za běhu programu, jak se projevují a co je jejich nejčastější příčinou?

Úlohy k řešení

1. Vytvořte program dle výše uvedeného příkladu a snažte se najít chyby a pomocí hlášení překládače identifikujte typ chyby.

8.2. Debugger

Čas ke studiu: 1,5 hodin

Cíl: Po prostudování tohoto odstavce budete umět

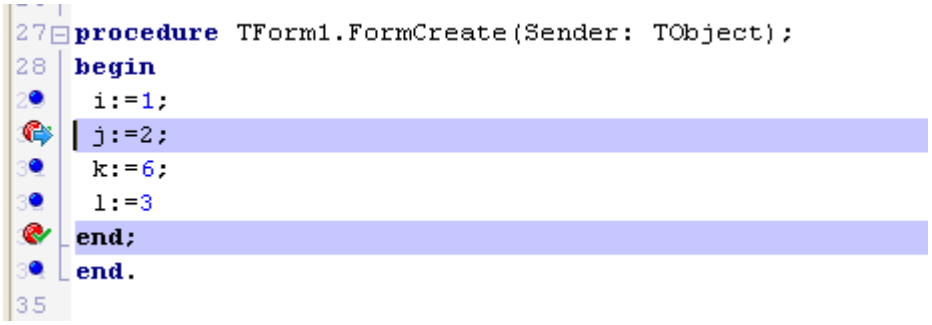
- zastavit běh programu v libovolném místě.
- vyhledávat chyby v programu pomocí krokování.
- chyby z programu odstranit.

Výklad

Při základním popisu grafického integrovaného vývojového prostředí Delphi zde již byla zmínka o tzv. debuggru. Byť jsou jeho možnosti širší, uvedeme si zde jen ty pro nás nedůležitější, které se naučíme v praxi při ladění používat. Debugger hlavně umožňuje zastavit program v námi určeném místě, postupně krok po kroku procházet jednotlivé příkazy ve sledu jako by je dělal program za normálního běhu a přitom je schopen sledovat a ukazovat aktuální hodnoty proměnných, které máme v úmyslu sledovat. I když to může být velmi pracné je to jediná cesta jak ověřit každý příkaz, jestli skutečně vykonává tu činnost, kterou jsme od něj očekávali.

Práce s debuggrem:

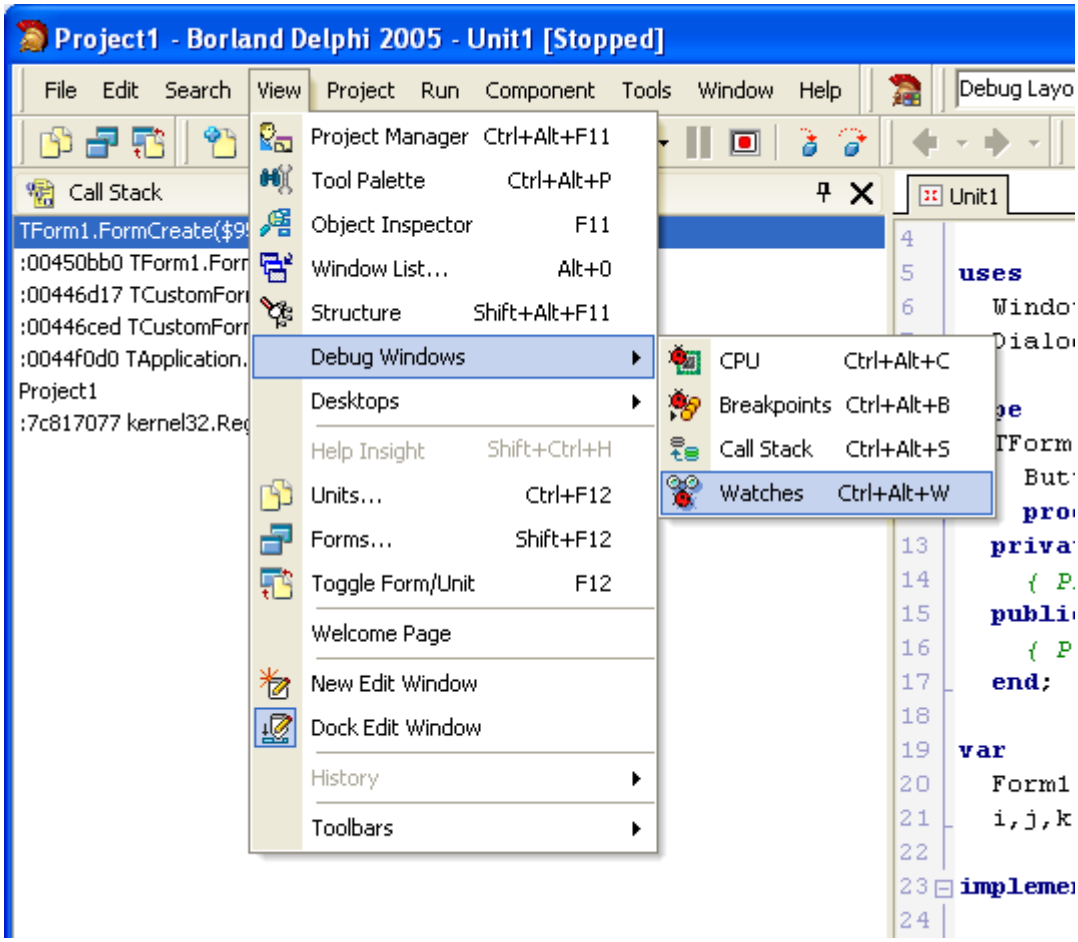
- Zastavení běhu programu v požadovaném místě (místo, kde je podezření, že zde mohla vzniknout logická chyba) můžeme zajistit tak, že v editačním režimu umístíme pomocí myši kurzor na řádek programu, kde chceme běh programu zastavit. Program tentokrát nebudeme spouštět stiskem funkční klávesy F9, ale použijeme klávesu F4. Program se spustí a až doběhne na označené místo, zastaví se. Řádek, na kterém se běh programu zastavil, je označen výrazným pruhem, ale příkaz se ještě neprovede.



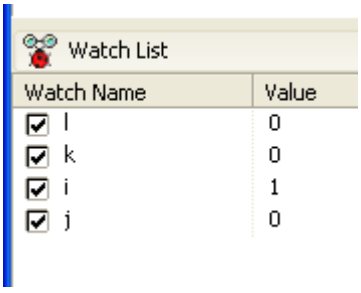
Obr. 8.1 Postup pro zobrazení Watch List

Je zde však i jiný způsob, který umožňuje v editačním režimu označit třeba i více bodů zastavení chodu programu (bodu přerušení - breakpoint) a to kliknutím myši na číslo řádku příkazu, na kterém chceme program zastavit (obr. 8.1). V místě kliku se objeví větší červený bod, který můžeme zrušit opětným kliknutím. Pak můžeme překlad a chod programu spustit klávesou F9. Běh programu se zastaví na prvním bodu přerušení. Opakovaným stiskem klávesy F9 pokračuje běh programu dál až na další bod přerušení.

- Krokování je užitečná a velmi praktická funkce debuggeru. Umožňuje postupně procházet jednotlivé příkazy programu. Krokování můžeme nejsnadněji provádět stiskem funkční klávesy F7. Každé stisknutí F7 vede k provedené činnosti definované označeným příkazem a k přechodu na příkaz následující.
- Sledování a zobrazení aktuálních hodnot vybraných proměnných je nutnou podmínkou aplikace debuggeru, jinak by nám nebylo k ničemu, že můžeme program dočasně zastavit, případně jej krokovat. Efektivně lze v přerušeném režimu zobrazit hodnotu té proměnné, na kterou na několik sekund umístíme kurzor myši. Debugger rovněž umožňuje sledovat a zobrazovat aktuální hodnoty i více proměnných a to nepřetržitě. Je ovšem nutno zobrazit panel Watch List. Do kterého se dá následně umístit sledování vybraných proměnných. Postup na zobrazení okna Watch List je patrný z obr. 8.2, kde z menu vybereme položku View, v roletkových nabídkách pokračujeme ve výběru Debug Windows a následně Watches. Nyní se otevře okno s názvem Watch List, do kterého můžeme přímo z editoru programu přetáhnout ty proměnné, jejichž hodnoty nás budou zajímat (obr. 8.3).

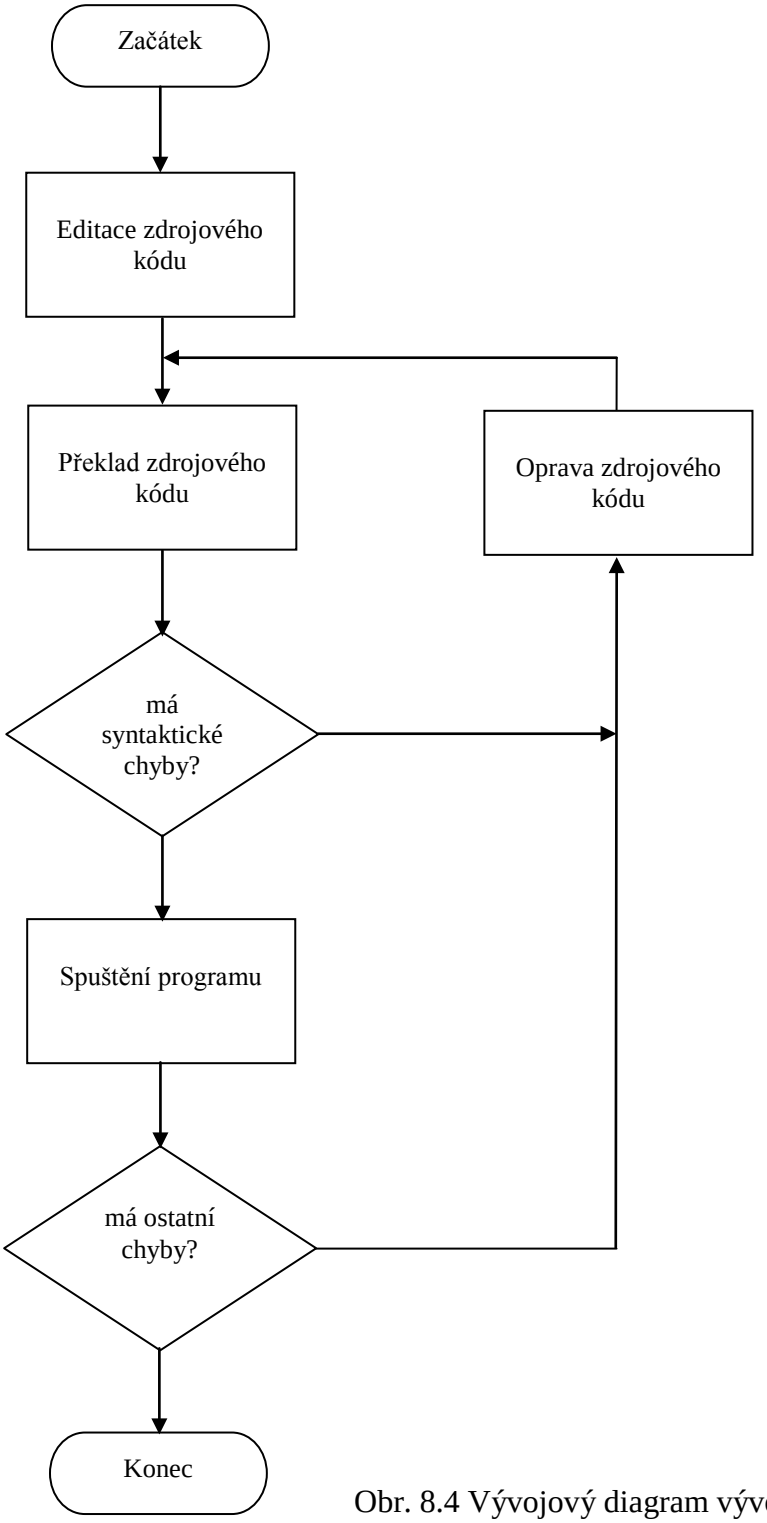


Obr. 8.2 Postup pro zobrazení Watch List



Obr. 8.3 Watch List s ukázkou zobrazovaných proměnných

Postup při vývoji počítačového programu je možno znázornit vývojovým diagramem (Obr. 8.4). Je z něj patrno, že obsahuje cykly s neznámým počtem opakování. My autoři vám v této chvíli přejeme, aby se vám tyto počty opakování co nejrychleji minimalizovaly na únosnou mez.



Obr. 8.4 Vývojový diagram vývoje programu

Shrnutí pojmů

Debugger hlavně umožňuje zastavit program v námi určeném místě, postupně krok po kroku procházet jednotlivé příkazy ve sledu jako by je dělal program za normálního běhu a přitom je schopen sledovat a ukazovat aktuální hodnoty proměnných, které máme v úmyslu sledovat.

Zastavení běhu programu v požadovaném místě můžeme zajistit tak, klikneme na řádek programu, kde chceme běh programu zastavit a program spustíme stiskem funkční klávesy F4. Nebo kliknutím myši na číslo řádku příkazu, na kterém chceme program zastavit, umístíme tzv. bodu přerušení (breakpoint).

Krokování programu umožňuje postupně procházet jednotlivé příkazy programu. Provádíme stiskem funkční klávesy F7.

Sledování a zobrazení aktuálních hodnot vybraných proměnných se provádí v panelu „Watch List“, do kterého se dá umístit vybrané proměnné přetažením z programu.

Otázky

1. Co představuje pojem ladění programu?
2. Jak lze zastavit běh programu v libovolném místě?
3. K čemu se provádí krokování?
4. Jak se dají zjistit aktuální hodnoty proměnných během ladění programu?

Úlohy k řešení

1. Vytvořte složitější program, který by řešil libovolné téma vašeho zájmu a využijte funkci debuggeru k jeho odladění.
2. Pro zdokonalení Vašich vědomostí bylo vytvořeno ještě několik animovaných výukových programů. Ty by Vám měly ulehčit přípravu na závěrečné zkoušky. Programy najdete na CD. Pokuste se vytvořit obdobné programy, případně se snažte jejich funkci dále zlepšit.

9. SAMOSTATNÁ PROGRAMÁTORSKÁ ČINNOST

Této části učebního textu věnujte zvýšenou pozornost. Je zde uveden seznam animovaných výukových programů, které jsou určeny k vaší samostatné programátorské činnosti. Programy jsou koncipovány tak, abyste v naprosté většině případu vystačili s aplikací znalostí nabytých v tomto učebním textu. Jen malé procento řešení vyžaduje znalosti získané nad rámec opory. V tom případě vám dobře poslouží nápovědy ve vlastních ukázkových animovaných programech, kde většinou můžete využít volby zobrazení fragmentů programu a následné kliknutí do místa formuláře, který nějak souvisí s vámi řešenou problematikou.

Úlohy k řešení

Seznam animovaných výukových programů pro samostatnou činnost:

9a Nahodna cisla

Program je určen pro výuku použití náhodných čísel. Vedle vysvětlení pojmu a ukázky použití graficky prezentuje náhodnost čísel.

9b Dialog

Program je určen pro výuku práce s textovými soubory, jejich čtením a ukládáním na záznamové médium. Pro tyto operace jsou využívány dialogové komponenty programovacího jazyka.

9c Minutka

Program je určen pro výuku práce se systémovým datem a časem, pro svou práci využívá časovač „Timer“.

9d Matice

Program je určen pro výuku práce s vícerozměrnými poli. Jako praktická aplikace byla vybrána práce s maticemi. Program je určen pro provádění základních operací s maticemi.

9e Cisla anglicky

Program je určen pro výuku práce s textovými řetězci. Program je možno také použít pro základní výuku čísel v anglickém jazyce.

9f Hledani maxima

Program je určen pro osvětlení principu hledání maxima počítačovým programem. Je zde vysvětlen zdánlivý rozdíl přístupu člověka a počítače. Tento program je určen pro využití na přednáškách (nutný komentář).

9g Algoritmus hledani maxima

Program je určen pro výuku tvorby jednoho ze základních algoritmu využívaných v práci počítačů. Animace využívá názorné vysvětlení na vývojovém diagramu, tak jako paralelně přímo v programu.

9h Algoritmus trideni

Program je určen pro výuku tvorby dalšího ze základních algoritmu využívaných v práci počítačů. Animace využívá názorné vysvětlení na vývojovém diagramu, tak jako paralelně přímo v programu. Jako metoda je použit princip probublávání.

9i RGB

Program je určen pro výuku počítačového skládání barev. Vedle vysvětlení pojmu RGB ukazuje praktické použití barev v počítačové grafice.

9j Mzdy

Program je určen pro výuku oborových dovedností. Patří mezi soubor programů z oblasti ekonomiky. Je zaměřen na výpočty mezd.

9k Uroceni

Program je určen pro výuku oborových dovedností. Patří mezi soubor programů z oblasti ekonomiky. Je zaměřen na výpočty úroků při finančních operacích.

9l Bilance

Program je určen pro výuku oborových dovedností. Patří mezi soubor programů z oblasti ekonomiky. Je zaměřen na výpočty bilancí rentability výroby a určení bodu zvratu.

9m Statistika

Program je určen pro výuku oborových dovedností. Patří mezi soubor programů z oblasti kvality řízení. Je zaměřen na výpočty základních statistických charakteristik.

9n Regrese

Program je určen pro výuku oborových dovedností. Patří mezi soubor programů z oblasti ekonomiky i kvality řízení. Je zaměřen na výpočty lineární regrese naměřených dat.

ZÁVĚR

Shrnutí

Vážení čtenáři,

pokud jste při studiu tohoto učebního textu došli až sem a v každé kapitole jste si úspěšně vyzkoušeli všechny úkoly na počítači, měli byste dokázat i s omezenými prostředky řešit většinu úloh technické praxe. Hlavně byste však měli mít dobrý základ pro další prohloubení znalostí v oblasti programování.

Většina doporučené literatury je psána na vysoké odborné úrovni s daleko širším záběrem. Ty Vám mohou být zdrojem dalších informací při řešení problémů, které přesahují rámec tohoto učebního textu.

Většina z Vás byla doposud schopna vyjadřovat se pouze v přirozeném jazyku. Cílem této učební opory bylo naučit vás „myslet“ v jednom z jazyků programovacích.

Doporučení

Nabyli jste svou poctivou prací novou dovednost. Bylo to spojeno nejen se studiem teorie programování, ale hlavně s vlastním vývojem programů na počítači. Pokud Vás programování alespoň trochu nadchlo nebo jen bavilo, neustávejte. Snažte se vymýšlet další a další problémy, které byste na počítači chtěli řešit. Bude to sice spojeno s nutností osvojení si dalších teoretických znalostí, ale stojí to za to. Budete moci efektivně řešit mnoho úkolů, které před vámi stojí při studiu jiných vyučovaných předmětů.

LITERATURA

[1] ELLER, F. *Delphi 6 příručka programátora*, 1.vyd. Praha: Grada Publishing, a.s. 2002. 272s. ISBN 80-247-0303-3.

[2] KADLEC, V. *Učíme se programovat v Delphi a jazyce Object Pascal*. 1.vyd. Brno: CP Books a.s. 2001. 306 s. ISBN 80-722-6245-9.

[3] CANTÚ, M. Myslíme v jazyku *Delphi 7 knihovna zkušeného programátora*, 1.vyd. Praha: Grada Publishing, a.s. 2003. 580s. ISBN 80-247-0694-6.

[4] PÍSEK, S. *Delphi začínáme programovat, podrobný průvodce začínajícího uživatele*. 2.vyd. Praha: Grada Publishing, a.s. 2002. 325 s. ISBN 80-247-0547-8.

[5] KADLEC, V. *Delphi hotová řešení*. 1.vyd. Brno: CP Books a.s. 2003. 312 s. ISBN 80-251-0017-0.

[6] HOLAN, T. *Delphi v příkladech*. 2.vyd. Praha: Technická literatura BEN, 2001. 207 s. ISBN 80-7300-033-4.

[7] SVOBODA, L. *1001 tipů a triků pro Delphi*. 2.vyd. Brno: CP Books a.s. 2003. 552 s. ISBN 80-722-6488-5.

[8] HÁLA, T. *Pascal Učebnice pro střední školy*. 1.vyd. Brno: CP Books a.s. 2002. 304 s. ISBN 80-722-6733-7.

[9] WRÓBLEWSKI, P. *Algoritmy, datové struktury a programovací techniky*. Brno CP Books a.s. 2004. 352. ISBN 80-251-0343-9.

[10] KVOCH, M. *Programování v Turbo Pascalu 7.0*. České Budějovice: KOPP nakladatelství. 240 s. ISBN 80-901342-5-4.

[11] PUTZ, K. *Pascal učebnice základu programování*, 1.vyd. Praha: Grada Publishing, a.s. 2007. 268s. ISBN 978-80-247-1255-0.

[12] PUTZ, K. *Pascal pokročilélejší programátorské techniky*, 1.vyd. Praha: Grada Publishing, a.s. 2007. 264s. ISBN 978-80-247-1266-6.

[13] *Komunitní portál po programátory v Delphi*. c2009 [cit. 12. 3. 2010]. Dostupné z < http://delphi.cz/>.

[14] *Umíme to s Delphi, seriál článků*. Autor: KADLEC, V. c2005 [cit. 12. 3. 2010]. Dostupné z < http://www.zive.cz/clanky/>.

[15] *Diskusní forum: Delphi*. c2003 [cit. 12. 3. 2010]. Dostupné z < http://forum.builder.cz/ >.

[16] *Delphi programming*. c2000 [cit. 12. 3. 2010]. Dostupné z < http://forums.devshed.com/delphi-programming-90/>.

[17] MARKOV A. A.: *Těorija algorifmov*. Trudy Matěmatičeskogo instituta AN SSSR *38*,176 (1951).

[18] ČSN 36 9001-12 (369001), Počítače a systémy zpracování údajů. Názvosloví. Nosiče, paměti a připojené prostředky, 1985

[19] ČSN ISO 5807 "Zpracování informací. Dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy programu a diagramy zdrojů systému" 1996

PŘÍLOHA - SEZNAM ANIMOVANÝCH PROGRAMŮ