VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA

ALGORITMY A DATOVÉ STRUKTURY

Břetislav Krček
Ivan Kolomazník

Vytvořeno v rámci projektu Operačního programu Rozvoje lidských zdrojů
CZ.04.1.03/3.2.15.1/0016
Studijní opory s převažujícími distančními prvky pro předměty teoretického základu studia.

Tento projekt je spolufinancován Evropským sociálním fondem
a státním rozpočtem České republiky

OBSAH

Předmluva 5

Pokyny ke studiu 6

Úvod 9

1. Algoritmizace 11
1.1. Algoritmus 11
1.2. Jazyk vývojových diagramů 17
1.3. Cyklus 27
1.3.1. Převod algoritmů z vývojových diagramů do Delphi 32
1.4. Strukturovaný údaj 41
1.4.1. Pole 42
1.5. Metody vytváření algoritmu 58
1.5.1. Metody návrhu algoritmu shora dolů a zdola nahoru 59
1.5.2. Strukturované programování 65
1.5.3. Modulární programování 69
1.6. Procedury 71

2. Delphi 77
2.1. Stavba programu 77
2.2. Abeceda a lexikální jednotky 80
2.3. Datové typy 83
2.4. Konstanty, proměnné, výrazy 90
2.5. Přiřazovací příkaz 93
2.6. Příkazy vstupu a výstupu - základní informace 94
2.7. Prázdný příkaz 98
2.8. Složený příkaz 99
2.9. Podmínkové příkazy - příkazy ***if*** a ***case*** 101
2.10. Příkazy cyklu - příkazy ***while***, ***repeat*** a ***for*** 107
Funkce *Random* 112
2.11. Typ pole (***array***) 116
2.12. Typ záznam (***record***) 125
Příkaz **with** 125
2.13. Typ množina (***set***) 131
2.14. Typ soubor (***file***) 134
2.15. Typ řetězec (***string***) 140
2.16. Procedury a funkce 145
2.17. Dynamická deklarace, typ ukazatel (typ *Pointer*) 169

2.18. Jednotka (***unit***) 173
2.19. Direktivy překladače 179
2.20. Objektově orientované programování (OOP) 187

3. Konzolové aplikace v Delphi 197

4. Ladění programů v Delphi 201
4.1. Položky menu určené pro ladění 202
4.2. Použití bodů zastavení 204
Nastavení a zrušení bodu zastavení 204
Okno bodů zastavení (*Breakpoint List*) 205
Kontextová menu okna bodů zastavení 206
Povolení a zakázání bodů zastaveni 207
Upravení bodu zastavení 207
Příkaz *Run to Cursor* 209
4.3. Prohlížení proměnných 210
Rychlé prohlížení hodnot proměnných 210
Okno *Watch List* 211
4.4. Krokování programu - příkazy *Step Over* a *Trace Into* 215
Příkaz *Step Over* 215
Příkaz *Trace Into* 215
4.5. Ostatní ladící nástroje 216
Dialogové okno *Evaluate/Modify* 216
Okno *Call Stack* 217
Okno *CPU* 217
Příkaz *Go to Address* 218

Literatura 219

STUDIJNÍ OPORY S PŘEVAŽUJÍCÍMI DISTANČNÍMI PRVKY PRO PŘEDMĚTY TEORETICKÉHO ZÁKLADU STUDIA

je název projektu, který uspěl v rámci první výzvy Operačního programu Rozvoj lidských zdrojů. Projekt je spolufinancován státním rozpočtem ČR a Evropským sociálním fondem. Partnery projektu jsou Regionální středisko výchovy a vzdělávání, s.r.o. v Mostě, Univerzita obrany v Brně a Technická univerzita v Liberci. Projekt byl zahájen 5.1.2006 a bude ukončen 4.1.2008.

Cílem projektu je zpracování studijních materiálů z matematiky, deskriptivní geometrie, fyziky a chemie tak, aby umožnily především samostatné studium a tím minimalizovaly počet kontaktních hodin s učitelem. Je zřejmé, že vytvořené texty jsou určeny studentům všech forem studia. Studenti kombinované a distanční formy studia je využijí k samostudiu, studenti v prezenční formě si mohou doplnit získané vědomosti. Všem studentům texty pomohou při procvičení a ověření získaných vědomostí. Nezanedbatelným cílem projektu je umožnit zvýšení kvalifikace širokému spektru osob, které nemohly ve studiu na vysoké škole z různých důvodů (sociálních, rodinných, politických) pokračovat bezprostředně po maturitě.

V rámci projektu jsou vytvořeny jednak standardní učební texty v tištěné podobě, koncipované pro samostatné studium, jednak e-learningové studijní materiály, přístupné prostřednictvím internetu. Součástí výstupů je rovněž banka testových úloh pro jednotlivé předměty, na níž si studenti ověří, do jaké míry zvládli prostudované učivo.

Bližší informace o projektu můžete najít na adrese http://www.studopory.vsb.cz/.

Přejeme vám mnoho úspěchů při studiu a budeme mít radost, pokud vám předložený text pomůže při studiu a bude se vám líbit. Protože nikdo není neomylný, mohou se i v tomto textu objevit nejasnosti a chyby. Předem se za ně omlouváme a budeme vám vděčni, pokud je na nás upozorníte.

ESF – ROVNÉ PŘÍLEŽITOSTI PRO VŠECHNY

POKYNY KE STUDIU

V úvodu si vysvětlíme jednotnou pevnou strukturu každé kapitoly textu, která by vám měla pomoci k rychlejší orientaci při studiu. Pro zvýraznění jednotlivých částí textu jsou používány ikony a barevné odlišení, jejichž význam nyní objasníme.

Průvodce studiem

vás stručně seznámí s obsahem dané kapitoly a s její motivací. Slouží také k instrukci, jak pokračovat dál po vyřešení kontrolních otázek nebo kontrolních textů.

Cíle

vás seznámí s učivem, které v dané kapitole poznáte a které byste po jejím prostudování měli umět.

Předpokládané znalosti

shrnují stručně učivo, které byste měli znát ještě dříve než kapitolu začnete studovat. Jsou nezbytným předpokladem pro úspěšné zvládnutí následující kapitoly.

Výklad

označuje samotný výklad učiva dané kapitoly, který je členěn způsobem obvyklým v matematice na definice, věty, případně důkazy.

Definice 1.1.1.
Zavádí důležité pojmy v dané kapitole.

Věta 1.1.1.
Uvádí základní vlastnosti pojmů zavedených v dané kapitole.

Důkaz: Vychází z předpokladů věty a dokazuje tvrzení uvedené ve větě.

Poznámka

neformálně komentuje vykládanou látku..

Řešené úlohy

označují vzorové příklady, které ilustrují probrané učivo.

Příklad Uvádí zadání příkladu.

Řešení: Uvádí podrobné řešení zadaného příkladu.

Úlohy k samostatnému řešení

obsahují zadání příkladů k procvičení probraného učiva. Úlohy označené ✖ patří k obtížnějším a jsou určeny zájemcům o hlubší pochopení tématu.

Výsledky úloh k samostatnému řešení

obsahují správné výsledky předchozích příkladů, slouží ke kontrole správnosti řešení.

Kontrolní otázky

obsahují soubor otázek k probranému učivu včetně několika odpovědí, z nichž je vždy alespoň jedna správná.

Odpovědi na kontrolní otázky

uvádějí správné odpovědi na kontrolní otázky.

Kontrolní test

obsahuje soubor příkladů k probranému učivu.

Výsledky testu

uvádějí správné odpovědi na příklady kontrolního testu.

Shrnutí lekce

obsahuje stručný přehled učiva, které by měl student po prostudování příslušné kapitoly zvládnout.

Literatura

obsahuje seznam knih, které byly použity při tvorbě příslušného textu a na které byly případně uvedeny odkazy k hlubšímu prostudování tématu.

Piktogram, který upozorňuje na důležité vztahy nebo vlastnosti, které je nezbytné si zapamatovat.

ÚVOD

K zápisu postupu pro řešení nějaké úlohy je třeba nejen znát tento postup (*algoritmus*), ale je třeba znát i vhodné výrazové prostředky pro jeho zápis. Velmi výhodné pro zápis různých algoritmů (nejen programů pro počítač, ale např. i pro zápis některých pracovních postupů) jsou často programovací jazyky (zejména univerzální, např. *Pascal*). Při vytváření a zápisu algoritmu formou programu na počítači je pak mj. možno využít i mnoha prostředků, které mohou vytváření algoritmu velmi usnadnit.

Předkládaná skripta mají právě nejen seznámit začátečníky s problematikou vytváření algoritmů, ale rovněž seznámit začátečníky i mírně pokročilé se zápisem algoritmů formou programů v Delphi.

Delphi je vývojové prostředí (vývojový nástroj) firmy Borland určené pro snadné vytváření programových aplikací pro Windows. Základem Delphi je programovací jazyk Object Pascal. Jednotlivé verze Delphi poskytují takovou šíři prostředků, že jsou vhodné jak pro soustavnou profesionální práci tak i pro výuku základů programování. V Delphi totiž začátečník může „odhodit“ značnou část prostředků vývojového prostředí, místo standardních aplikací vytváří jen konzolové aplikace (viz kap. 3. *Konzolové aplikace v Delphi*), které jsou pro svou jednoduchost velmi vhodné pro výuku základů algoritmizace a programování.

Ve vývojovém prostředí Delphi má uživatel k dispozici velké množství prostředků (včetně vizuálních komponent), které mu značně usnadňují vytváření různých programových aplikací. Jednoduše lze např. vytvořit menu aplikace, které je svým vzhledem, bohatostí a členěním do úrovní srovnatelné s menu takových aplikací, jako např. *Word* a *Excel*.

Zatímco samotné vytvoření položky menu je jednoduchá záležitost, je obvykle podstatně těžší napsat část programu zajišťující činnost, která se má po volbě položky provést. Často pak místo klikání na nabízené prostředky Delphi a na kopírování vhodných částí z bohatých a rozsáhlých částí nápovědy Delphi dochází na „klasické“ programování (vytváření algoritmu včetně návrhu struktury zpracovávaných údajů, psaní příkazů, vytváření podprogramů, …), se kterým by se měl programátor, zejména pak začátečník, přednostně seznámit.

Údaje (údaje = *data*), se kterými se v programu pracuje, včetně vstupních i výstupních údajů nemusí být jen jednoduché, často mají bohatší strukturu (např. vektory, matice), jedná se pak o *strukturované údaje*. Obecně jsou *datové struktury* uvedeny již v kap. 1., v kap. 2. jsou pak blíže probrány některé datové struktury z Delphi.

V kap. 1. *Algoritmizace* se objasňuje vytváření algoritmů přednostně pomocí vývojových diagramů. Při vytváření vývojových diagramů je však kladen důraz na používání takových řídicích konstrukcí, které odpovídají zásadám strukturovaného programování, a které lze tedy jednoduše „doslovně“ překládat z jazyka vývojových diagramů do Delphi. Vzhledem k orientaci skript na Delphi jsou i u vývojových diagramů důsledně uváděny deklarace, a to přímo formou převzatou z Delphi. Téměř od počátku jsou pak i v kap. 1. vývojové diagramy doplňovány příslušnými zápisy v Delphi (při prvních čteních lze tyto zápisy „přeskočit“).

Závěrečné části 2.17. až 2.20. kap. 2. *Delphi* již poněkud přesahují jeden z hlavních cílů těchto skript (zápis jednoduchých algoritmů), neboť představují „výlety“ do oblastí, které by bylo vhodné zvládnout pro vytváření složitějších standardních aplikací v Delphi.

Kap. 4. *Ladění programů v Delphi* obsahuje ve zhuštěné podobě základní informace i praktické rady vhodné nejen pro začátečníky ladící konzolové aplikace, ale i pro pokročilejší programátory vytvářející standardní aplikace v Delphi.

Konvence zápisu skript

Pro přehlednost a čitelnost skript je různou formou zápisu odlišen běžný text (písmo *Times Roman*) od obecných tvarů příkazů či jiných konstrukcí (základem písmo *Courier*). Souvisle podtržený text (písmo *Times Roman*, podtržená kurzíva) vždy představuje symbol, který zastupuje konkrétní konstrukci. Klíčová slova Delphi (blíže viz 2.2. *Abeceda a lexikální jednotky*) jsou v běžném textu i ukázkách psána tučně. Blíže o formě zápisu programů v Delphi viz 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi*. Tučné písmo se dále používá pro zvýraznění některých částí běžného textu, např. pro nové termíny. Nově zaváděné a tučně psané termíny jsou psány buď kurzívou, nebo obyčejným písmem, přičemž obyčejné (a tučné) písmo se používá jen při zavádění termínu v podbarveném rámečku, a to jen v případě, že je tu termín přímo definován nebo alespoň dostatečně objasněn. V běžném textu se dále používá kurzíva (netučná) pro identifikátory (např. názvy proměnných, procedur, …) a pro odkazy na části skript.

Z důvodu zajištění plné syntaktické správnosti obecných tvarů různých dílčích konstrukcí i částí programů nejsou za jejich zařazením do textu uváděna interpunkční znaménka.

Autoři jednotlivých kapitol:

RNDr. Břetislav Krček, CSc. - 1. a 2. kapitola,
Doc. Dr. Mgr. Ivan Kolomazník - 3. a 4. kapitola.

1. ALGORITMIZACE

1.1. Algoritmus

Výklad

Aby bylo možno zadat (předepsat) počítači, resp. i člověku, postup pro řešení nějaké úlohy (návod pro řešení úlohy), je třeba:

- znát nebo teprve nalézt příslušný postup (postup, který má určité, dále uvedené vlastnosti se nazývá ***algoritmus***),
- vyjádřit tento postup výrazovými prostředky srozumitelnými pro jeho vykonavatele - vytvořit ***program***.

Realizaci zadaného postupu řešení úlohy je možno obecně charakterizovat jako transformaci vstupu na výstup (transformaci konkrétního zadání na příslušný výsledek), např. na základě vstupních informací se má získat nová výstupní informace. Při řešení úloh na počítači jsou nositelem informací ***údaje***. Vstupním informacím tedy odpovídají ***vstupní údaje*** a výstupním informacím ***výstupní údaje*** (místo termínu údaje se rovněž používá termín ***data***). Realizaci zadaného postupu řešení úlohy na počítači je pak možno považovat za transformaci vstupních údajů na výstupní údaje.

Zadání úlohy, které vlastně představuje implicitní zadání výsledku, určuje podmínky, jež musí výstupní údaje splňovat. Soubor těchto podmínek se obvykle nazývá *výstupní podmínka úlohy*. Zadání úlohy rovněž určuje soubor podmínek pro vstupní údaje, který se opět souhrnně označuje jako *vstupní podmínka úlohy*.

Postup pro řešení úlohy musí být vždy navržen s ohledem na realizaci, a to tak, aby se celý postup skládal jen z činností, které je schopen provádět příslušný vykonavatel (např. počítač, člověk či stroj). Elementární činnosti, které je schopen realizátor vykonávat a ze kterých je možno sestavit složitější činnosti, se nazývají ***operace***, též ***kroky***. Nalézt postup, přesněji *algoritmus*, pro řešení zadané úlohy tedy znamená nalézt takovou sestavu operací, která zabezpečí požadovanou transformaci vstupních údajů na údaje výstupní.

Pojem algoritmus, který byl použit v předchozím textu, je z hlediska algoritmizace (***algoritmizace*** - nauka o vytváření algoritmů) pojmem základním.

Algoritmus je přesný popis definující konečný (např. výpočtový) proces, vedoucí od měnitelných vstupních údajů až k žádaným výsledkům (proces, při němž se transformují vstupní údaje na údaje výstupní), který má následující vlastnosti: ***determinovanost***, ***hromadnost*** a ***rezultativnost***.

Determinovanost (určenost) algoritmu znamená, že při realizaci algoritmu nesmí být v žádném případě pochyb o dalším postupu, tedy při realizaci algoritmu musí být vždy jednoznačně určena nejen činnost, která se právě provádí, ale i činnost, která má následovat. Přitom realizace algoritmu nesmí být podmíněna žádnými dalšími podmínkami než těmi, které jsou v něm uvedeny.

Hromadnost (masovost) algoritmu znamená, že algoritmus není použitelný jen pro určité vstupní údaje, ale pro libovolné vstupní údaje, které splňují příslušnou vstupní podmínku. Množina přípustných vstupních údajů tvoří oblast použití algoritmu (*definiční obor algoritmu*).

Rezultativnost algoritmu znamená, že pro vstupní údaje, které vyhovují příslušné vstupní podmínce, musí dát algoritmus výsledek po konečném počtu kroků.

Program je algoritmus pro určitou strukturu vstupních údajů, který je vyjádřen **programovacím jazykem**, tj. pomocí přesně definovaných výrazových prostředků.

Programátor - tvůrce programu.

Procesor - vykonavatel (realizátor) programu.

Programování (jako činnost) je proces, který má dvě složky:

- tvorbu algoritmu,
- vyjádření algoritmu programovacím jazykem.

Složky procesu programování se nemusí uskutečňovat odděleně, algoritmus se může vytvářet přímo pomocí programovacího jazyka.

Proměnná je veličina určitého typu, která může během realizace algoritmu nabývat různých hodnot příslušného typu, tj. může měnit svoji hodnotu, ne však svůj typ.

Např. *celočíselná proměnná* může mít jen celočíselnou hodnotu, *reálná proměnná* jen reálnou hodnotu. V dalším textu budou zpočátku užívány jen uvedené *číselné proměnné*. Později pak budou zavedeny proměnné více typů, např. *proměnná typu řetězec* (též *řetězcová proměnná*), jejíž hodnotou je nějaký text (posloupnost přípustných znaků). Při práci na počítači je možno si pod pojmem proměnná představit jméno paměťového místa.

Paměťové místo určitého typu je část paměti, do které může být uložena právě jedna hodnota tohoto typu (tj. v případě reálné proměnné právě jedno reálné číslo). K označení (pojmenování) proměnných dosud uvedených typů bude užíváno nejen jedno písmeno tak, jak je to obvyklé v matematice, ale **identifikátor**, což je posloupnost písmen a číslic začínající písmenem.

Další objasnění pojmu algoritmus bude provedeno na příkladech, jejichž hlavním cílem je však přiblížit čtenáři tvorbu algoritmů. O procesoru, který by tyto algoritmy realizoval, bude zatím známo jen to, že „zná význam“ příkazů popsaných v následujícím textu a že je schopen tyto příkazy vykonávat.

Zápis algoritmu formou očíslovaných kroků

Jedním z možných vyjádření algoritmu je zápis ve tvaru očíslovaných kroků, které se, pokud samy tyto příkazy nestanoví jinak, provádějí v přirozeném pořadí, tj. v pořadí daném rostoucími čísly příkazů. Každý příkaz musí mít přesně definovaný význam, což znamená, že musí být určeno, jakým způsobem má procesor tento příkaz realizovat. Pokud by programátor (tvůrce algoritmu) neznal přesně význam některého příkazu, kterého použije, mohla by se činnost procesoru lišit od programátorových představ.

K zápisu algoritmů formou očíslovaných kroků (příkazů) by postačovaly např. jen příkazy (viz např. [5]):

- *příkaz čtení*, umožňující vstup (čtení) vstupních údajů,
- *příkaz tisku*, umožňující výstup (tisk) výstupních údajů,
- *přiřazovací příkaz*, umožňující vypočítat hodnotu výrazu a přiřadit ji zadané proměnné,

- *příkaz skoku*, umožňující měnit přirozené pořadí příkazů,
- *příkaz podmíněného skoku*, umožňující měnit přirozené pořadí příkazů v případě, že je splněna podmínka, kterou tento příkaz obsahuje,
- *příkaz stop*, předepisující ukončení práce.

Z uvedených příkazů budou v následujícím textu blíže objasněny jen příkaz čtení, příkaz tisku (*příkazy vstupu a výstupu*) a přiřazovací příkaz, které se budou dále používat i ve vývojových diagramech.

Příkazy vstupu a výstupu, přiřazovací příkaz

Seznam je posloupnost prvků seznamu oddělených určenými oddělovači. Jako oddělovač bude zatím používána jen čárka.

Příkaz čtení je tvořen slovem *Čtení*, za nímž za dvojtečkou následuje vstupní seznam. Prvky vstupního seznamu jsou názvy proměnných, jimž mají být přiřazeny hodnoty prostřednictvím vstupního zařízení.

Např. příkaz čtení

```
Čtení: A,B
```

znamená, že na vstupním zařízení se mají vzít dvě nejbližší čísla, přičemž první se má stát hodnotou proměnné *A*, tj. má se uložit do paměťového místa *A* (krátce: první číslo se má uložit do proměnné *A*), druhé číslo se má uložit do proměnné *B*. Naproti tomu příkaz

```
Čtení: AB
```

znamená, že na vstupním zařízení se má vzít jen jedno (nejbližší další) číslo a má se uložit do proměnné označené identifikátorem *AB*,

Příkaz tisku je tvořen slovem *Tisk*, dvojtečkou a výstupním seznamem. Prvkem výstupního seznamu může být jméno proměnné, jejíž hodnota se má objevit na výstupním zařízení (např. na obrazovce, je-li výstupním zařízením monitor, nebo má být vytištěna na papíře, je-li výstupním zařízením tiskárna). Prvkem výstupního seznamu může být rovněž **řetězec**, což je libovolná posloupnost přípustných znaků, která je z obou stran ohraničena apostrofem. Znaky řetězce, tj. znaky, které jsou mezi apostrofy, představují text, který se má objevit na výstupním zařízení.

Má-li např. celočíselná proměnná *N* hodnotu 123, potom příkaz

```
Tisk: 'Celkem␣', N, '␣pokusů.'
```

zajistí, že se na výstupním zařízení objeví znaky:

```
Celkem␣123␣pokusů.
```

Znakem ␣ je obvykle v psaném textu označována mezera. Znak ␣ bude v dalším textu používán jen v případech, ve kterých bude třeba mezery zvýraznit.

Při zápisu obecných tvarů některých dalších příkazů bude použit podtržený text. ***Souvisle podtržený text*** v obecném tvaru příkazu bude představovat symbol, který zde zastupuje konkrétní konstrukci (konkrétní vyjadřovací prostředky). Obecný tvar příkazu tisku by tedy mohl mít např. tvar

Tisk: ***výstupní seznam***

Přiřazovací příkaz má obecný tvar

proměnná ← ***výraz***

kde

proměnná	je označení proměnné,
výraz	představuje výraz,
←	je znak pro přiřazení (přiřazovací znak).

Význam přiřazovacího příkazu: Nejprve se vypočte hodnota výrazu napravo od přiřazovacího znaku, a tato hodnota se potom přiřadí proměnné, jejíž označení je uvedeno vlevo od přiřazovacího znaku.

Např. příkaz

```
K ← K^2 + 1
```

(čti: *do K přiřaď* $K^2 + 1$) znamená, že se má nejprve pro stávající hodnotu proměnné *K* vypočítat hodnota výrazu $K^2 + 1$, a teprve potom se má původní hodnota proměnné *K* nahradit (přemazat) vypočtenou hodnotou. Náhradu staré hodnoty proměnné hodnotou novou je možno si představit jako přemazání staré video nahrávky nahrávkou novou.

Např. po provedení příkazů

```
K ← 5
K ← K^2 + 1
```

má tedy proměnná *K* hodnotu 26.

Řešené úlohy

Příklad 1.1.1. Vyjádřete algoritmus, podle kterého se pro zadaná reálná čísla *a*, *b*, *c* vypočtou kořeny rovnice $a \cdot x^2 + b \cdot x + c = 0$, je-li tato kvadratická.

Řešení:

```
1.      Čtení: A,B,C
2.      D  ← B^2 - 4*A*C
3.      X1 ← (-B+sqrt(D))/(2*A)
4.      X2 ← (-B-sqrt(D))/(2*A)
5.      Tisk: X1,X2
6.      Stop
```

V zápisu algoritmu jsou použity proměnné:

A, B, C	-	koeficienty rovnice,
D	-	diskriminant rovnice,
X1	-	první kořen,
X2	-	druhý kořen.

Navržený algoritmus je vzhledem k zadání špatný, neboť je použitelný jen v případě, že rovnice příslušející vstupním údajům *a*, *b*, *c* je kvadratická a nemá imaginární kořeny. Obdobně by bylo možno navrhnout částečné řešení např. jen pro případ, že kořeny příslušné rovnice by byly imaginární, přičemž by podle něho mohl pracovat i člověk, který zná jen reálná čísla, tj. nezná komplexní čísla (viz dále). Úplné řešení (viz první příklad z části 1.2.) pak musí obsahovat podmínky, které umožní výběr příslušného dílčího postupu (dílčího algoritmu).

Zápis algoritmu formou očíslovaných kroků, který obsahuje podmínky (např. v příkazech podmíněného skoku), však obvykle bývá pro začátečníka těžký a málo přehledný. Uvedené nevýhody může odstranit zápis algoritmu formou vývojového diagramu.

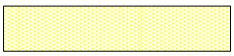
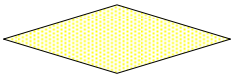
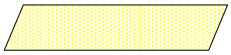
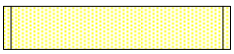
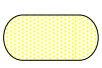
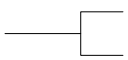
1.2. Jazyk vývojových diagramů

Výklad

Jazyk vývojových diagramů umožňuje nejen názorný a přehledný zápis hotových algoritmů, ale lze jej též velmi účinně využívat přímo při vytváření nových algoritmů.

Vývojový diagram je grafický způsob zápisu algoritmů, v němž jsou k zápisu jednotlivých příkazů použity ***značky***, které se spojují ***spojnicemi***.

Podle normy se spojnice kreslí vodorovně nebo svisle a mohou se označit šipkami (mohou být orientovány). Podle normy neorientovaná spojnice vyjadřuje implicitně směr shora dolů nebo zleva doprava. Na rozdíl od normy však v dalším textu bude zajištěna *implicitní orientace všech spojnic* tím, že vstup do všech značek bude kreslen jen shora a výstup ze všech značek, s výjimkou rozhodování, jen zdola, což současně přispěje i k větší přehlednosti vývojových diagramů. V dalším textu budou používány jen následující značky vybrané z normy pro vývojové diagramy (ČSN 36 9030):

Značka	**Název značky** ; význam, příp. příklad použití
	• **zpracování** ; pro zápis jednoduchých příkazů přiřazení
	• **rozhodování** (podle normy též stejná značka pro **větvení** nebo **přepínání**) ; pro zápis podmínky
	• **vstup** nebo **výstup** ; pro zápis příkazu čtení nebo tisku
	• **příprava** ; pro modifikaci činnosti, používá se pro zápis cyklů se známým počtem opakování nebo pro zápis záhlaví procedur
	• **předem definovaná činnost** ; představuje jinde rozpracovanou činnost, např. obecnou proceduru
	• **spojka** ;
	• **mezní značka** ; např. pro zahájení nebo ukončení činnosti
	• **poznámka** ; slouží k zápisu poznámky vedle značky, dále bude používána i pro zápis deklarací

Kosočtverec bude zpočátku používán jen jako značka *rozhodování*, tj. budou se do něj psát jen podmínky ve tvaru výrokové formy, která je v okamžiku plnění výrokem. Je-li výrok

pravdivý, postupuje se dále po větvi (spojnici) označené znakem + , v opačném případě po větvi označené znakem - .

Počátek vývojového diagramu je označen mezní značkou *Začátek* a odpovídá zahájení činnosti při realizaci algoritmu. Spojnice, která ze značky *Začátek* vychází, určuje další značku. Vychází-li ze značky jediná spojnice, je další postup zřejmý. Z některých značek však může vycházet více spojnic zvaných větve (např. ze značky *rozhodování* musí vycházet právě dvě větve). Podmínka, která je zapsaná uvnitř těchto značek, však opět umožní jednoznačně rozhodnout o dalším postupu. Realizace algoritmu vždy končí příchodem na některou z mezních značek *Konec*. Vývojový diagram se může rozpadat na více větví, a může tedy obsahovat více značek *Konec*. Pro převod vývojového diagramu do některých programovacích jazyků („např.“ do Delphi) je vhodné, aby vývojový diagram obsahoval právě jednu značku *Konec*, což však lze vždy lehce zajistit prodloužením příslušných spojnic na jedinou značku *Konec*.

Upozornění: Následující příklady budou orientovány především na vytváření a zápis algoritmů jazykem vývojových diagramů. U většiny vývojových diagramů však již budou pro pokročilejší studenty uváděny ukázky zápisu příslušného algoritmu nebo jeho části v Delphi. Pro naprosté začátečníky by bylo vhodné, aby se zápisy v Delphi začali blíže zabývat až po prostudování části 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi* a dále pak podle potřeby při studiu kap. 2. *Delphi*.

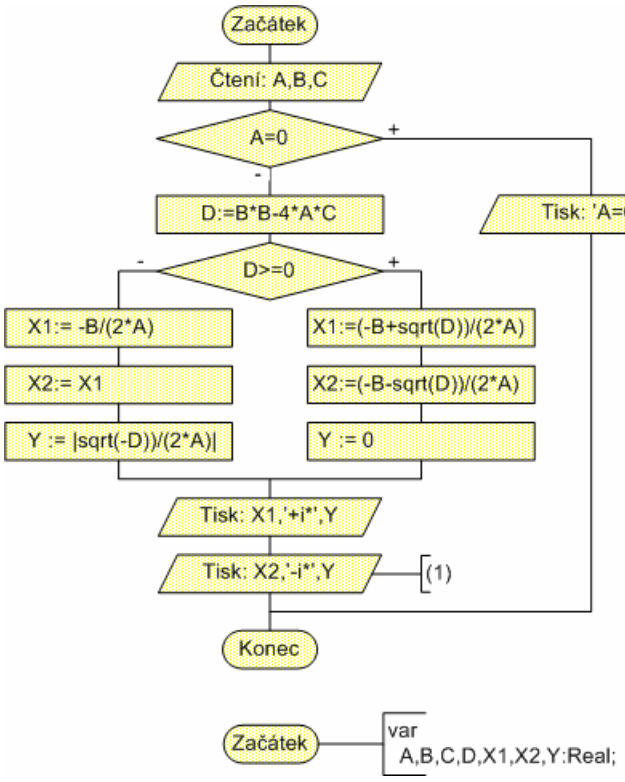
Zápis v Delphi nemusí být členěn do řádků. Pro zvětšení přehlednosti a názornosti však podle běžných pravidel (která budou uvedena později) budou zápisy v Delphi nejen členěny do řádků, ale bude používán i zápis do různých svislých úrovní (odsazení). Zápisem do úrovní se zvýrazní (ale nezmění !) vztahy mezi příkazy a vztahy mezi částmi některých příkazů. V Delphi se v přiřazovacím příkazu místo znaku ← používá znak (dvojznak) **:=** (viz dále).

Řešené úlohy

Příklad 1.2.1. Vyjádřete algoritmus, podle kterého se pro zadaná reálná čísla a, b, c vypočtou kořeny rovnice $a \cdot x^2 + b \cdot x + c = 0$, je-li tato kvadratická (viz př. 1.1.1.).

Řešení: Požadovaný algoritmus bude vyjádřen tak, aby podle něj mohl pracovat i člověk, který zná jen reálná čísla, tj. nezná komplexní čísla. Po načtení vstupních údajů se nejprve zjistí zda příslušná rovnice je či není kvadratická (viz podmínka:

A=0). V případě kvadratické rovnice se pak podle hodnoty diskriminantu (viz podmínka: *D≥0*) provedou příslušné dílčí výpočty reálné a imaginární části obou kořenů. Tisk výsledků je totiž navržen společně pro případ, že kořeny jsou reálné, i pro případ, že jsou imaginární (a tedy komplexně sdružené). První kořen má tvar *X1+i*Y* , druhý *X2-i*Y* (u imaginárních *X1 = X2*). I reálné kořeny se tedy tisknou jako komplexní čísla s nulovou imaginární částí, tj. *X1+i*0* a *X2-i*0*.




```
program Rovnice;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A,B,C,D,X1,X2,Y:Real;
begin
  ReadLn(A,B,C);
  if A=0 then
    WriteLn('A=0')
  else
  begin
    D:=sqr(B)-4*A*C;
    if D>=0 then
    begin
      X1:=(-B+sqrt(D))/(2*A);
      X2:=(-B-sqrt(D))/(2*A);
      Y :=0;
    end
    else
    begin
      X1:=-B/(2*A);
      X2:=X1;
      Y:=abs(sqrt(-D)/(2*A));
    end;
    WriteLn;
    WriteLn(X1,'+i*',Y);
    WriteLn(X2,'-i*',Y);
  end;
  ReadLn;
end.
```

Poznámka

Z uvedeného vývojového diagramu lze lehce zjistit, které proměnné jsou potřebné pro realizaci příslušného algoritmu. Z důvodu snadnější orientace čtenáře a pro usnadnění příp. převodu algoritmu z jazyka vývojových diagramů do Delphi bude u jednotlivých (celých)

vývojových diagramů v poznámce u značky *Začátek* uváděn přehled příslušných proměnných, přičemž bude přímo používána forma *deklarace proměnných* převzatá z Delphi (viz druhé uvedení značky *Začátek* pod vývojovým diagramem, resp. přímo deklarační část příslušného programu v Delphi).

Výklad

Deklarace proměnných v Delphi začíná klíčovým slovem **var** (což je zkratka ze slova *variable*), typ reálných proměnných je označován identifikátorem *Real*, typ celočíselných proměnných identifikátorem *Integer*.

Např. deklaraci celočíselných proměnných *M*, *N* a reálné proměnné *X* lze zapsat ve tvaru:

```
var
  M,N:Integer;
  X:Real;
```

Při zápisu programů v některých programovacích jazycích se deklarace číselných proměnných buď vůbec neuvádějí (některé verze jazyka Basic), nebo je není třeba uvádět (jazyk Fortran). Zápis deklarací v Delphi však není samoúčelný. Kdyby např. v uvedeném algoritmu při tisku druhého kořene rovnice bylo v příslušném příkazu tisku (ve vývojovém diagramu je u tohoto příkazu poznámka (1)) místo písmene *Y* omylem zapsáno písmeno *V*, poskytovaly by příslušné programy ve zmíněných formách jazyka Basic a Fortran pro některá vstupní data špatné výsledky (bez jakéhokoliv chybového nebo varovného hlášení). V Delphi, kde se proměnné deklarují, by však překladač ohlásil, že v uvedeném příkazu narazil na neznámý identifikátor. Toto chybové hlášení je pak možno „odstranit“ dvěma způsoby: zapsáním správného písmene v uvedeném příkazu tisku nebo (nesprávně !) dopsáním identifikátoru *V* do deklarací proměnných.

Při zápisu algoritmů jako nových konzolových aplikací v Delphi se některé řádky programu vytvářejí automaticky, např. řádky:

```
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
```

Mnohé řádky lze vytvořit jako prostý překlad z vývojového diagramu do Delphi, některé je však třeba upravit nebo doplnit. Před koncovým **end** programu je např. vhodné použít

příkaz *ReadLn*, který zajistí, že vstupně-výstupní okno aplikace se neuzavře ihned po výpisu výsledků, ale až po stisku klávesy *Enter*. Pro větší komfort uživatele by dále bylo vhodné např. místo řetězce *A=0* tisknout řetězec *Rovnice není kvadratická*, příkaz

```
ReadLn(A,B,C)
```

nahradit příkazy

```
Write('a = ');
ReadLn(A);
Write('b = ');
ReadLn(B);
Write('c = ');
ReadLn(C)
```

(tj. před vstupem každého údaje zajistit tisk příslušné ***vstupní výzvy***) a příkazy pro tisk kořenů doplnit na tvar:

```
WriteLn('1. koren = ',X1:7:3,' + i *',Y:7:3);
WriteLn('2. koren = ',X2:7:3,' - i *',Y:7:3);
```

Kromě úvodních řetězců jsou tu navíc požadavky na formátování vystupujících hodnot. Je-li konkrétně ve výstupním seznamu u jednotlivých reálných proměnných uvedeno *:7:3*, znamená to, že hodnota proměnných má být uvedena na 3 desetinná místa a pro její celkový zápis má být rezervováno 7 pozic. Potom např. kořen -1.2+3i by byl vytištěn ve tvaru:

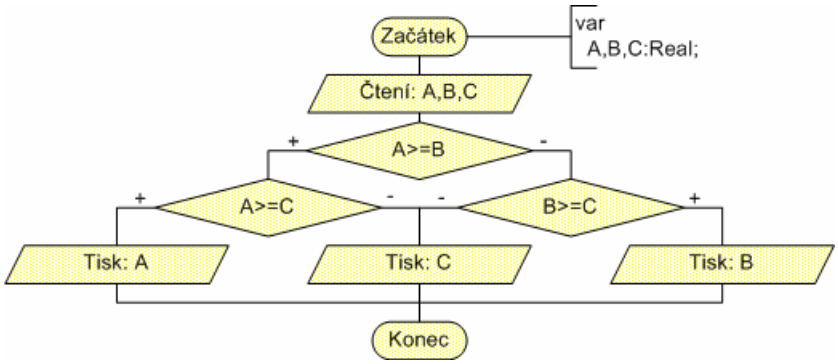
```
1. ␣koren␣=␣␣-1.200␣+␣i␣*␣␣3.000
```

Řešené úlohy

Příklad 1.2.2. Vyjádřete algoritmus pro nalezení největší hodnoty, která se vyskytuje mezi třemi zadanými čísly, jež jsou z intervalu (-1000, 1000).

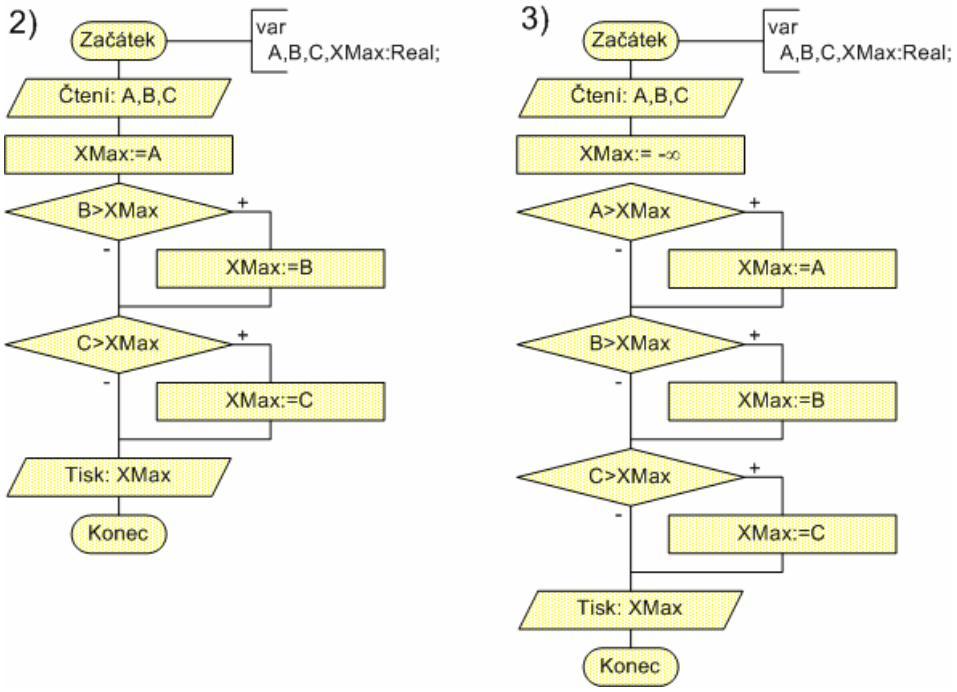
Řešení 1.:

Zadaná čísla budou načtena do paměťových míst *A*, *B*, *C*. Řešením tedy bude tisk obsahu jednoho z těchto paměťových míst. Kdyby všechna zadaná čísla byla stejná, bylo by jedno, které z nich by se tisklo. Pokud tomu tak není, je třeba nejprve provést příslušná zjištění, což vyjadřuje např. následující vývojový diagram:



Základní myšlenka navrženého algoritmu je z vývojového diagramu sice dobře patrná a bylo by ji možno použít i v případě, že by se místo tří čísel měla zpracovávat čísla čtyři. S dalším zvětšováním počtu vstupujících čísel by však naprosto neúměrně narůstaly problémy se zakreslováním vývojového diagramu založeného na stejném principu. Vývojový diagram pro tři čísla obsahuje sice jen 3 rozhodování, pro čtyři čísla by však obsahoval již 6 rozhodování a např. pro deset čísel již 45 rozhodování. Úměrně počtu vstupujících čísel by rovněž narůstaly požadavky na počet paměťových míst. Kromě toho navržený algoritmus také není příliš vhodný pro přepis do Delphi.

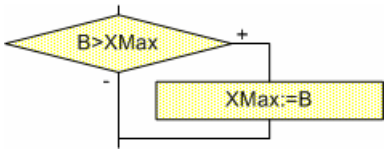
Řešení 2. a 3.:



Obě řešení jsou založena na užívání proměnné *XMax*, jejíž hodnota se postupně mění tak, aby to stále byla největší hodnota, která se vyskytuje mezi dosud zpracovanými čísly. Algoritmy *(2)* a *(3)* se navzájem liší jen způsobem zajištění

počáteční hodnoty proměnné *XMax*. V algoritmu *(2)* je nejprve do *XMax* dosazena hodnota načtená do proměnné *A* (krátce jen hodnota *A*), kterou je možno na počátku považovat za dosud největší. V algoritmu *(3)* je jako počáteční hodnota proměnné *XMax* použito *mínus nekonečno*. V programovacích jazycích, ve kterých nejsou symboly pro ±∞, lze použít příslušné strojové ±∞ (jako ***strojové plus nekonečno*** se na počítači označuje největší zobrazitelné číslo příslušného typu). V dané úloze však funkci -∞ může převzít libovolné číslo, které je menší než číslo -1000, a které je tedy určitě menší než *A* (výsledek prvního rozhodování je pak kladný a počáteční náhradní hodnota proměnné *XMax* je nahrazena hodnotou *A*).

Algoritmus *(3)* předepisuje pro všechna tři čísla *A*, *B*, *C* stejný způsob zpracování. V algoritmu *(2)* je tentýž způsob zpracování předepsán jen pro čísla *B*, *C*, číslo *A* se tu zpracovává jiným způsobem. Např. část vývojových diagramů *(2)* a *(3)* pro zpracování čísla *B* má shodný tvar:



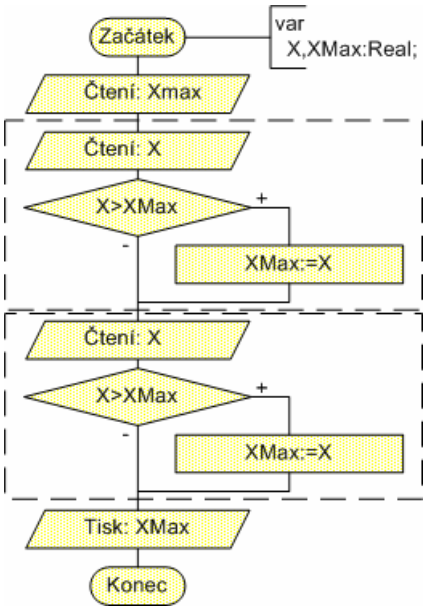
Ve vývojovém diagramu *(2)* se tedy dvakrát a ve vývojovém diagramu *(3)* se třikrát opakují podobné, nikoliv však stejné části, a není tedy možno mechanicky pomocí cyklu (viz část *1.3. Cyklus*) předepsat opakování jedné z nich.

Řešení 4.:

Další řešení je již navrženo tak, že pro zpracování druhého a třetího čísla je předepsán naprosto shodný dílčí algoritmus (jeho opakování tedy bude možno později lehce předepsat pomocí cyklu, ve vývojovém diagramu je zvýrazněn čárkovaným zarámováním). V navrženém algoritmu se zadaná čísla s výjimkou prvního načítají po jednom do proměnné *X*, tj. postupně se střídají v příslušném paměťovém místě *X*. Po načtení každého dalšího čísla do proměnné *X* se zjišťuje, zda není třeba hodnotu proměnné *XMax* opravit, tj. nahradit ji (přemazat ji) právě přečteným číslem tak, aby v proměnné *XMax* stále byla největší hodnota, která se dosud mezi přečtenými čísly vyskytla. Po zpracování posledního čísla je pak „dosud největší hodnota“ hodnotou největší, tj. výsledkem. Na počátku zpracování, pokud je známo jen první číslo, je jeho hodnota dosud největší. Algoritmus proto předepisuje načtení prvního čísla přímo do proměnné *XMax*.

Na rozdíl od předchozích algoritmů je možno na základě právě popsaného algoritmu poměrně snadno vytvořit algoritmus i pro nalezení největší hodnoty

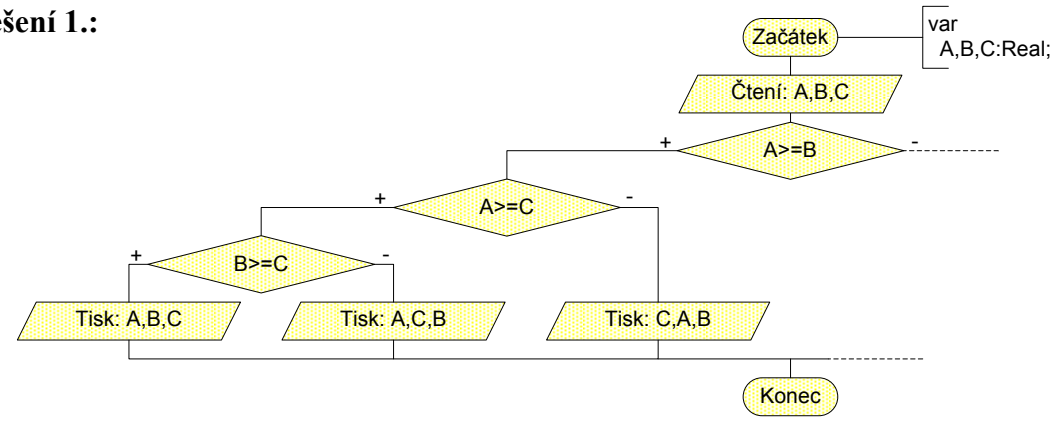
z posloupnosti více čísel (viz příklad 1.3.3.), přičemž i při rostoucím počtu prvků posloupnosti se nebudou zvyšovat nároky na počet paměťových míst.




```
program Maximum;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  X,XMax:Real;
begin
  ReadLn(XMax);
  ReadLn(X);
  if X>XMax then
    XMax:=X;
  ReadLn(X);
  if X>XMax then
    XMax:=X;
  WriteLn(XMax);
  ReadLn;
end.
```

Příklad 1.2.3. Vyjádřete algoritmus pro seřazení tří zadaných čísel do nerostoucí posloupnosti, tj. podle velikosti od „největšího“ po „nejmenší“.

Řešení 1.:

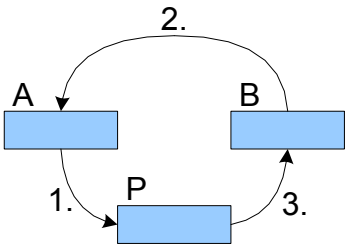


Tři čísla mohou být seřazena celkem 3!, tj. 6 způsoby. Řešení je tedy možno založit na zjištění, které pořadí ze šesti možných má požadované vlastnosti (uvedená část vývojového diagramu obsahuje jen tři z těchto pořadí). Na stejném principu by sice bylo možno vyjádřit i algoritmus pro seřazení n čísel ($n \geq 4$) do nerostoucí posloupnosti, ale vyjádření tohoto algoritmu by bylo prakticky neúnosné, neboť každému z $n!$ možných pořadí by odpovídala zvláštní větev vývojového diagramu.

Řešení 2.:

Další řešení je možno založit na principu přerovnání zadaných čísel do požadovaného pořadí, přičemž k uložení výsledné posloupnosti budou užita táž paměťová místa *A*, *B*, *C*, do kterých byla načtena původní posloupnost zadaných čísel. V příslušném algoritmu je obsah jednotlivých paměťových míst postupně porovnáván s obsahem následujících paměťových míst. Nevyhovují-li čísla uložená v porovnávané dvojici paměťových míst požadovanému uspořádání, je pořadí těchto čísel změněno, tj. navzájem se vymění v příslušných paměťových místech.

Vlastní výměnu (záměnu) hodnot uložených ve dvou paměťových místech, např. v místech *A*, *B*, je možno provést podle naznačeného schématu s využitím jednoho pomocného paměťového místa *P* ve třech krocích:



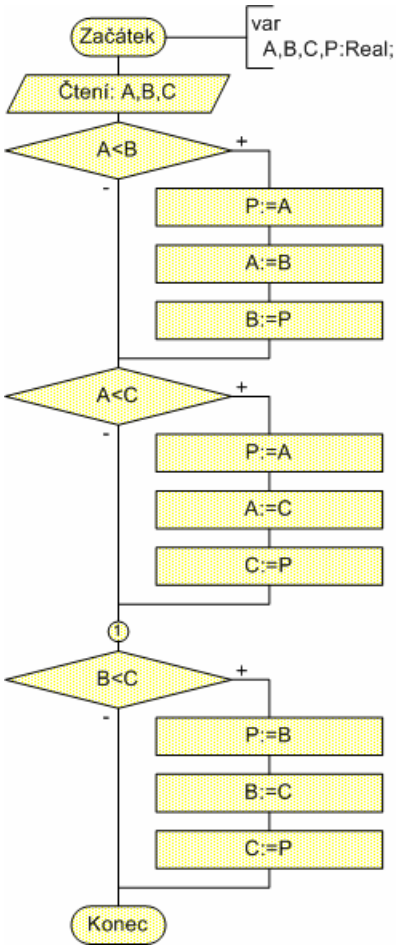

```
1.   P := A
2.   A := B
3.   B := P
```

Algoritmus, založený na přerovnání zadaných čísel do požadovaného pořadí pomocí vzájemných výměn hodnot ve dvojicích paměťových míst, je na rozdíl od algoritmu z prvního řešení vhodný i pro seřazení (setřídění) početnější posloupnosti čísel, která jsou však uložena v poli (viz část 1.4.1. *Pole*, př. 1.4.4.). Algoritmy pro seřazení množiny prvků do nerostoucí nebo do neklesající posloupnosti se nazývají ***algoritmy třídění***.

Realizaci následujícího algoritmu třídění pro tři čísla je možno rozdělit na dvě fáze. V první fázi největší číslo, resp. jedno z největších čísel, obsadí první paměťové místo, tj. paměťové místo *A*, v druhé fázi se obdobně hledá číslo, které obsadí druhé paměťové místo. Prvek, který při případných výměnách neobsadí žádné z uvedených míst, pak bude v posledním z paměťových míst.

Poznámka **pro programy v Delphi:**

Podobně, jak bylo naznačeno u programu z př. 1.2.1., bylo by vhodné doplnit a upravit „holé“ verze programů uváděné i u dalších příkladů (tj. zajistit čtení vstupních údajů po jednom, doplnění čtení o vstupní výzvy, doplnit a vhodně upravit všechny výstupy z programu).




```
program Trideni;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A,B,C,P:Real;
begin
  ReadLn(A,B,C);
  if A<B then
  begin
    P:=A;
    A:=B;
    B:=P;
  end;
  if A<C then
  begin
    P:=A;
    A:=C;
    C:=P;
  end;
  if B<C then
  begin
    P:=B;
    B:=C;
    C:=P;
  end;
  WriteLn(A,B,C);
  ReadLn;
end.
```

Úlohy k samostatnému řešení

Příklad 1.2.4. Vyjádřete algoritmus pro zjištění počtu záporných čísel, která se vyskytují mezi třemi zadanými čísly.

Příklad 1.2.5. Vyjádřete algoritmus pro nalezení největší hodnoty a dále nejmenší hodnoty, které se vyskytují mezi třemi zadanými čísly.

Příklad 1.2.6. Vyjádřete algoritmus pro výpočet aritmetického průměru z kladných prvků dvouprvkové posloupnosti čísel, přičemž pokud posloupnost neobsahuje kladná čísla, poskytne algoritmus číslo nula.

Příklad 1.2.7. Vyjádřete algoritmus, podle kterého se zjistí, kolik ze zadaných tří čísel je větších než aritmetický průměr těchto čísel.

1.3. Cyklus

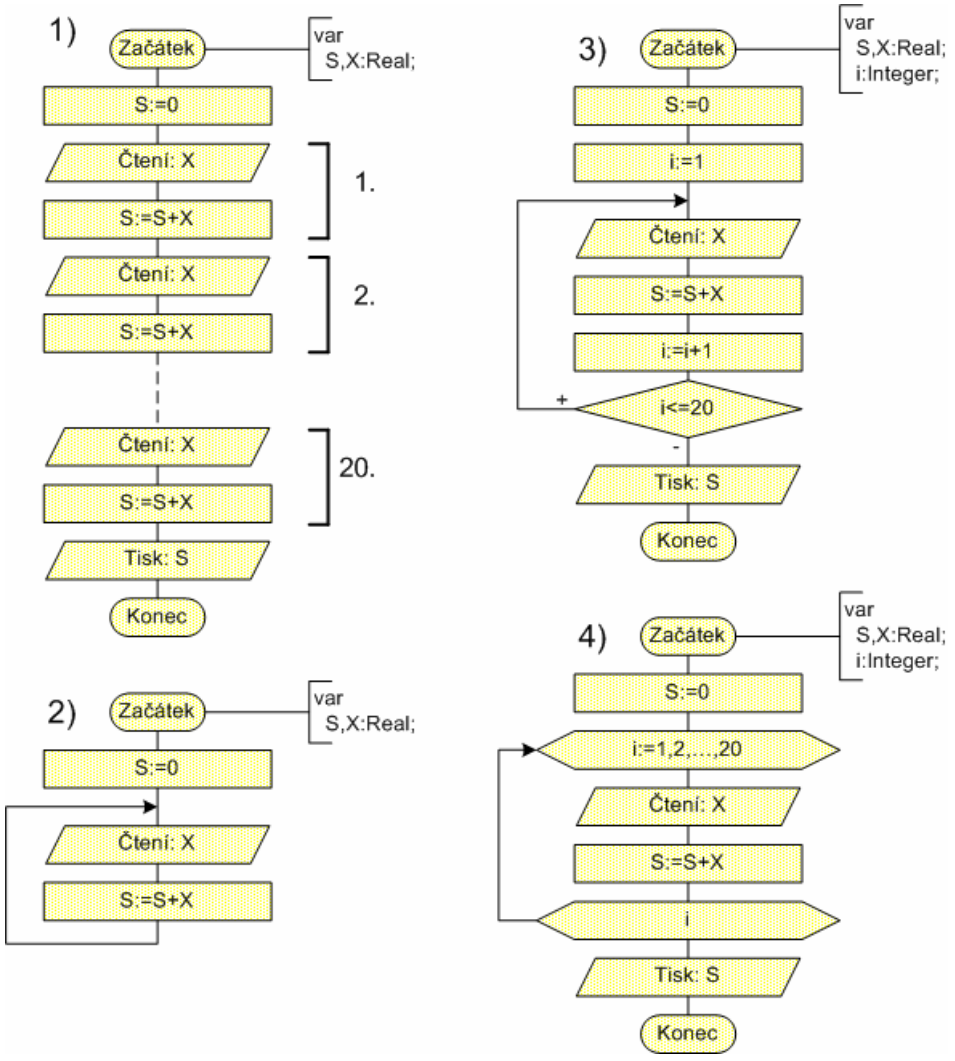
Při vytváření algoritmů vhodných pro realizaci na počítači je často třeba předepsat opakování nějaké činnosti (dílčího algoritmu), k čemuž se s velkou výhodou využívají cykly.

Cyklus je část algoritmu, která předepisuje (několikanásobné) opakování nějakého dílčího algoritmu. Dílčí algoritmus, jehož opakování cyklus předepisuje, se nazývá **tělo cyklu**. Počet opakování těla konečného cyklu je buď předem explicitně (výslovně) znám, nebo závisí na nějaké jiné podmínce. Každému konečnému cyklu přísluší proměnná, jejíž hodnota se při provádění cyklu obecně mění a je určující pro to, zda se v cyklu bude či nebude pokračovat, tato proměnná se nazývá **parametr cyklu**.

Řešené úlohy

Příklad 1.3.1. Vyjádřete algoritmus pro výpočet součtu 20 reálných čísel dodávaných ze vstupního zařízení.

Řešení:



Jako *(1)* je označen jen náznak vývojového diagramu, v němž je pro sčítání 20 čísel užito metody postupného přičítání.

Výklad

Metoda postupného přičítání, zvaná též ***kumulace součtu***, spočívá v tom, že sčítání více čísel se neprovádí najednou, ale že se jedno číslo po druhém postupně přičítá do paměťového místa, v němž musí být na počátku uloženo číslo nula (musí být vynulováno). Analogicky se postupuje i při výpočtu *vícenásobného součinu*, přičemž jako počáteční hodnota paměťového místa pro součin se použije jednička (pokud by se použil přímo první činitel, potom by se první činitel nezpracovával stejně jako ostatní).

Pokračování řešení př. 1.3.1.:

V *(1)* se naznačuje, že označená dvojice příkazů by se musela dvacetkrát opakovat. V *(2)* je ukázáno, jak by šlo předepsat nekonečně mnoho opakování zmíněné dvojice příkazů, tj. jak by šlo vytvořit nekonečný cyklus opakování. Podle zadání je však třeba vytvořit cyklus, v němž by se tělo cyklu, tj. zmíněná dvojice příkazů opakovala právě dvacetkrát. Jednotlivá opakování těla cyklu je tedy třeba počítat a po dosažení požadovaného počtu opakování tuto činnost ukončit.

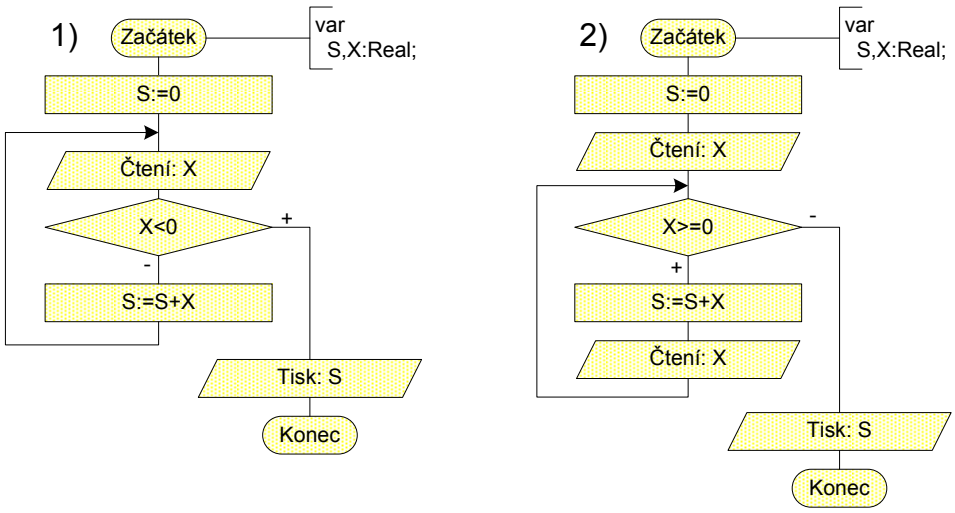
Jednu z mnoha variant, jak lze vytvořit cyklus, který předepisuje určitý počet opakování těla cyklu, vyjadřuje vývojový diagram *(3)*. K počítání počtu průchodů přes požadované tělo cyklu (tj. zmíněnou dvojici příkazů) tu slouží proměnná *i* („počítadlo“ *i*). Hodnota proměnná *i* se po každém zopakování těla cyklu zvětší o 1, a tato nová hodnota je pak použita ke zjištění, zda se má či nemá v cyklu pokračovat. Proměnná *i* je tedy parametrem zakresleného cyklu (smyčky).

Má-li člověk provést určitý počet opakování nějaké činnosti, obvykle si počítá (uchovává v paměti), kolikrát již činnost provedl. Při zápisu algoritmů se však jeví obecně jako výhodnější počítat, po kolikáté se činnost „právě nyní“ provádí. Z uvedených dvou možností významu parametru cyklu je v *(3)* použita druhá možnost, tj. při provádění těla cyklu má parametr cyklu *i* hodnotu, která udává, kolikáté číslo posloupnosti se právě zpracovává.

Vzhledem k tomu, že v algoritmizaci je *cyklus se známým počtem opakování* poměrně častým konstrukčním prvkem (častou *řídicí strukturou*), má pro jeho zápis většina programovacích jazyků (i jazyk vývojových diagramů) speciální prostředky (viz řešení *(4)* a dále).

Příklad 1.3.2. Vstupní údaje tvoří posloupnost kladných čísel a dále číslo záporné (které tedy již nepatří mezi prvky posloupnosti). Vyjádřete algoritmus pro zjištění součtu prvků posloupnosti.

Řešení:



Stejně jako v předchozím příkladě je i tentokrát třeba opakovaně sčítat vstupující prvky posloupnosti. Na rozdíl od předchozího příkladu však tentokrát není počet prvků posloupnosti znám, opakovaná činnost má být ukončena po načtení záporného čísla. Příslušný algoritmus *(1)* opět obsahuje cyklus. Vlastnosti parametru cyklu tu má proměnná *X*. Při jednotlivých průchodech tělem cyklu se její hodnota při čtení obecně mění, a tato nová hodnota slouží ke zjištění, zda se má již cyklus ukončit. Hodnota parametru cyklu však přímo nesouvisí s počtem opakování (na rozdíl od hodnoty parametru cyklu v předchozím příkladě).

Algoritmus *(2)* vznikl z algoritmu *(1)* úpravami, které mají usnadnit jeho převod do Delphi. Řídicí struktury použité v *(2)* lze totiž přímo přepisovat („doslovně“ překládat) do Delphi. Programy v Delphi k vývojovým diagramům z tohoto a předchozího příkladu jsou uvedeny až za částí 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi*.

Výklad

Cykly z řešení příkladů 1.3.1. a 1.3.2. jsou ukázkami dvou základních typů konečných cyklů. Při práci na počítači se též někdy používají *nekonečné cykly*, které se pak ale nějakým „násilným“ způsobem přerušují. V algoritmech se však mohou vyskytovat jen cykly konečné, proto i další text se bude týkat jen konečných cyklů.

Základní typy (konečných) cyklů :

- **cyklus se známým počtem opakování** - cyklus s explicitně vyjádřeným počtem opakování,
- **cyklus s neznámým počtem opakování** - cyklus s implicitně vyjádřeným počtem opakování (zvaný též ***cyklus řízený podmínkou***).

Rozhodující skutečností pro to, zda se jedná o cyklus se známým počtem opakování, je znalost počtu opakování na počátku provádění cyklu, tj. při realizaci cyklu, nikoliv již při jeho programování (zápisu). Počet opakování u cyklu se známým počtem opakování může být např. jedním ze vstupních údajů programu (viz př. 1.3.3.).

Části (konečného) cyklu :

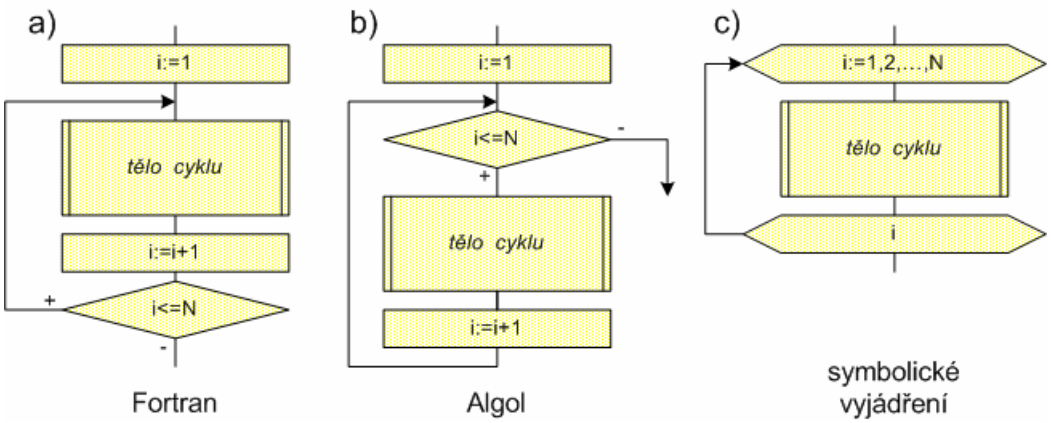
- ***tělo cyklu*****,**
- ***příprava dalšího průchodu*****,**
- ***test na ukončení cyklu*****.**

Ve vývojovém diagramu *(3)* k př. 1.3.1. je přípravou dalšího průchodu příkaz *i:= i+1*, přípravou prvního průchodu je příkaz *i:= 1*. Přípravu dalšího průchodu ve vývojových diagramech k př. 1.3.2. představuje příkaz *čtení: X* , který současně rovněž náleží do těla cyklu.

Tři části vnitřku cyklu je možno při vyjadřování cyklu seřadit 3!, tj. 6 způsoby, které jsou u cyklu se známým počtem opakování prakticky rovnocenné. Na základě požadavků na jednoduchost, jednotnost a přehlednost zápisů se v jednotlivých programovacích jazycích zavádějí pro vyjadřování cyklů speciální prostředky, a v souvislosti s tím je obvykle jedno ze šesti zmíněných vyjádření cyklu se známým počtem opakování považováno za základní vyjádření cyklu.

Základní vyjádření cyklu se známým počtem opakování

Následující tři části vývojových diagramů představují **tři vyjádření cyklu se známým počtem opakování, jejichž parametr *i* nabývá hodnot 1, 2, ..., *N*, kde *N* je počet opakování** (za předpokladu, že uvnitř těla cyklu nejsou příkazy měnící hodnotu parametru cyklu *i* nebo hodnotu proměnné *N*, potom by totiž uvedená vyjádření nemusela mít požadovaný význam):



Z uvedených vyjádření cyklů se známým počtem opakování pokládá např. programovací jazyk Fortran za základní vyjádření *(a)*, dále nazývané *vyjádření typu Fortran*, programovací jazyk Algol pak vyjádření *(b)*, dále nazývané *vyjádření typu Algol*.

Pro zápis cyklu se známým počtem opakování se často užívá symbolické vyjádření *(c)*, které může odpovídat jak vyjádření *(a)*, tak i *(b)*, podle toho, který význam se mu přisoudí. Ze samotného symbolického vyjádření *(c)* tedy nelze poznat, zda představuje činnost popsanou v *(a)* nebo v *(b)*. Pro potřebu této kapitoly bude symbolickému vyjádření *(c)* přisouzen nejprve význam *(a)*, tj. význam typu *Fortran*. Význam, který symbolickému vyjádření *(c)* přisuzuje Delphi, je poněkud složitější a bude uveden až v části 2.10. *Příkazy cyklu*.

Při realizaci vyjádření typu *Fortran*, tj. *(a),* se tělo cyklu provede vždy alespoň jednou, a to i v případě, že proměnná *N* má při vstupu do cyklu hodnotu nula. Má-li však proměnná *N* hodnotu nula při vstupu do cyklu s vyjádřením typu *Algol*, tj. *(b)*, je cyklus ihned ukončen, tzn. tělo cyklu se neprovede ani jednou. Je-li hodnota proměnné *N* větší nebo rovna jedné, jsou výsledky realizace *(a)* a *(b)* stejné.

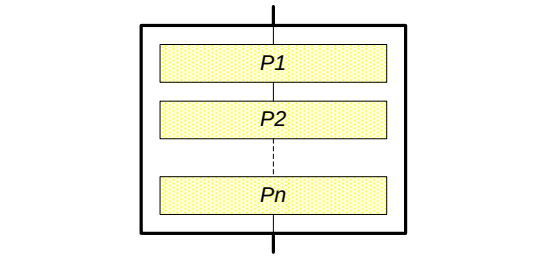
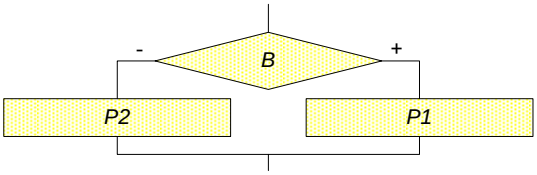
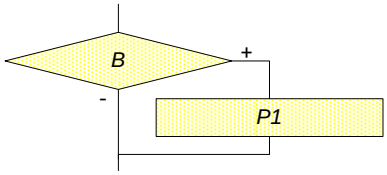
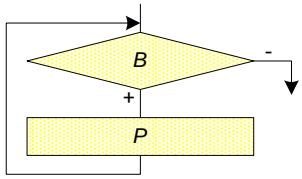
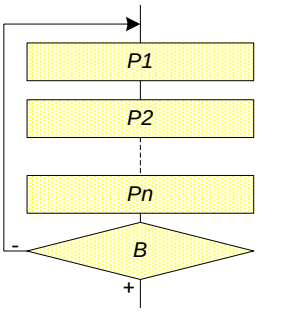
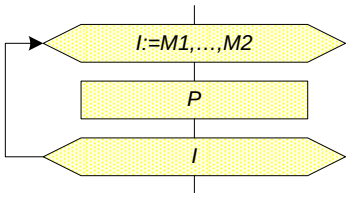
Pro některé řídicí struktury použité v předchozích vývojových diagramech existuje jednoduchý, takřka „doslovný“ překlad do Delphi. Pokud tedy programátor navrhne vývojový diagram jen pomocí těchto řídicích struktur, bude pak možno příslušný algoritmus poměrně lehce do Delphi převést.

Každá z následujících řídicích struktur je vztažena alespoň na jeden ***operační blok***, (vyjádřený značkou zpracování), který však nemusí být jednoduchým operačním blokem, tj. nemusí představovat jen jeden jednoduchý příkaz (např. příkaz přiřazení), ale který může být opět tvořen libovolnou z uvedených řídicích struktur (viz též část 1.5.2. *Strukturované programování*).

1.3.1. Převod algoritmů z vývojových diagramů do Delphi

Výklad

Každý programovací jazyk umožňuje snadný zápis některých řídicích struktur. Konkrétně např. řídicí struktury z následující tabulky lze v Delphi vyjádřit jediným (strukturovaným) příkazem a v dalším textu proto budou uvedené řídicí struktury přednostně používány.

část vývojového diagramu	**zápis v Delphi /** ***název*** – vysvětlivky
	**`begin`** *P1* **`;`** *P2* **`;`** ... **`;`** *Pn* **`end`** ***složený příkaz*** - z hlediska dalšího užití se jedná o jediný příkaz
	**`if`** *B* **`then`** *P1* **`else`** *P2* ***podmíněný příkaz úplný (příkaz if úplný)***
	**`if`** *B* **`then`** *P1* ***podmíněný příkaz neúplný (příkaz if neúplný)***
	**`while`** *B* **`do`** *P* ***příkaz cyklu while*** - pro vyjádření cyklu s testem před operačním blokem, který je tvořen jediným příkazem
	**`repeat`** *P1* **`;`** *P2* **`;`** ... **`;`** *Pn* **`until`** *B* ***příkaz cyklu repeat*** - pro vyjádření cyklu s testem za operačním blokem, který je tvořen posloupností příkazů
	**`for`** *I* **`:=`** *M1* **`to`** *M2* **`do`** *P* ***příkaz cyklu for*** - pro vyjádření cyklu se známým počtem opakování

Význam symbolů použitých v tabulce:

B	-	logický výraz zapsaný podle syntaxe Delphi, tj. výraz typu *Boolean* (viz 2.3. *Datové typy*),
P, *P1*, *P2*, ... , *Pn*	-	libovolné příkazy,
I	-	proměnná ordinálního typu, např. celočíselná (viz též 2.3.),
M1, *M2*	-	výrazy stejného typu jako proměnná *I*.

Řešené úlohy

Programy k vývojovým diagramům z př. 1.3.1.

```
program Cyklus3;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  S,X:Real;
  I:Integer;
begin
  S:=0;
  I:=1;
  repeat
    ReadLn(X);
    S:=S+X;
    I:=I+1;
  until not(I<=20);
  WriteLn(S);
  ReadLn;
end.
```

```
program Cyklus4;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  S,X:Real;
  I:Integer;
begin
  S:=0;
  I:=1;
  for I:=1 to 20 do
  begin
    ReadLn(X);
    S:=S+X;
  end;
  WriteLn(S);
  ReadLn;
end.
```

Forma zápisu programů v Delphi:

Program v Delphi je vhodné (nikoliv nutné) členit ustáleným (obvyklým) způsobem do řádků. Ke zvýšení přehlednosti a názornosti je dále vhodné při zápisu programu využívat i ***odsazení*** (***zápis do úrovní***). V editoru Delphi, který používá neproporcionální písmo, odpovídá optimální ***modul odsazení*** (vyjádření odsazení o jednu úrovň) dvěma znakům.

V uvedených programech k př. 1.3.1. a 1.3.2. jsou obvyklým způsobem zapsány složené příkazy a příkazy cyklu **for**, **repeat** a **while**.

Každý příkaz obvykle začíná na novém řádku, příkazy vnořené do strukturovaného příkazu se vzhledem k němu zapisují o úroveň vpravo s výjimkou vnoření složeného příkazu.

Řešené úlohy

Program k vývojovému diagramu z př. 1.3.2.

```
program Cyklus2;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  S,X:Real;
begin
  S:=0;
  ReadLn(X);
  while X>=0 do
  begin
    S:=S+X;
    ReadLn(X);
  end;
  WriteLn(S);
  ReadLn;
end.
```

Příklad. 1.3.3. Vstupní údaje tvoří nejprve přirozené číslo *N* z intervalu <2, 1000> a dále posloupnost *N* reálných čísel. Vyjádřete algoritmus:

a) pro nalezení největší hodnoty z hodnot prvků posloupnosti (v algoritmizaci se největší hodnota obvykle nazývá *maximum*, což neodpovídá termínu *maximum množiny* v matematice),

b) pro nalezení indexů (pořadových čísel) míst výskytu těch prvků posloupnosti, které nabývají největší hodnoty, tj. maxima,

c) pro nalezení indexu prvního místa výskytu maxima prvků posloupnosti,

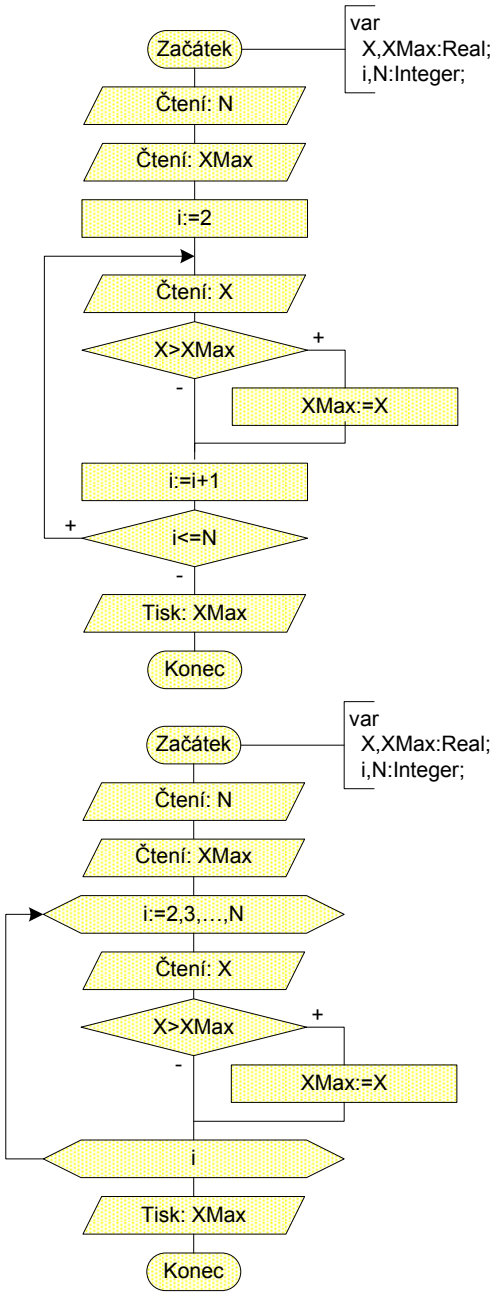
d) pro nalezení indexu posledního místa výskytu maxima prvků posloupnosti,

e) pro nalezení počtu všech výskytů maxima prvků posloupnosti.

Řešení (a):

Některé principy algoritmů pro nalezení největší hodnoty byly popsány při řešení příkladu 1.2.2., nyní bude použit poslední z nich, neboť podle něj lze pomocí cyklu předepsat pro všechny prvky posloupnosti, s výjimkou prvního, zcela stejný dílčí algoritmus. V následujícím řešení je použit cyklus, který předepisuje *N*-1 opakování. Celá posloupnost má sice N prvků, ale první z nich je zpracován odlišným způsobem, je totiž přímo načten do proměnné *XMax*. Parametr cyklu *i* by tedy mohl nabývat např. hodnot 1, 2, ..., *N*-1. Ve vývojovém diagramu však parametr cyklu

nabývá při provádění cyklu hodnot 2, 3, ..., *N*. Při prvním průchodu tělem cyklu, tj. při zpracování 2. prvku posloupnosti, má tedy parametr cyklu *i* hodnotu 2, obecně při zpracování *k*-tého prvku posloupnosti má parametr cyklu *i* hodnotu *k*.



Řešení formou vývojového diagramu je zapsáno dvakrát. V prvním zápisu je zmíněný cyklus se známým počtem opakování vyjádřen konkrétně, a to způsobem *Fortran*, ve druhém zápisu je tentýž cyklus vyjádřen symbolicky.

Program v Delphi odpovídá druhému vývojovému diagramu (je tu použit cyklus **for**), první vývojový diagram je uveden nejen pro srovnání, ale bude využit i při následující simulaci.

```
program MaxA;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  X,XMax:Real;
  I,N:Integer;
begin
  ReadLn(N);
  ReadLn(XMax);
  for I:=2 to N do
  begin
    ReadLn(X);
    if X>XMax then
      XMax :=X;
  end;
  WriteLn(XMax);
  ReadLn;
end.
```

Výklad

Při ověřování správnosti algoritmu, resp. přímo programu, a tedy i vývojového diagramu, je možno použít metodu zvanou ***simulace*** (simulace = předstírání činnosti). Programátor při této metodě napodobuje (předstírá) činnost vykonavatele algoritmu, obvykle tedy činnost

počítače. O prováděné simulaci je vhodné pořizovat zápis např. na tabuli nebo na papír. Zápis na tabuli by mohl více odpovídat způsobu práce počítače, neboť starý zápis v určitém paměťovém místě (tj. na vyhrazeném a pojmenovaném kousku tabule) je možno smazat a nahradit zápisem novým. Zápis s mazáním však neumožňuje případnou zpětnou kontrolu, a proto je výhodnější zápis bez mazání ve formě tabulky. Při tomto způsobu zápisu se vždy místo mazání začne psát další řádek záznamu o výpočtu. Místo paměťového místa tedy v tabulce přísluší každé použité proměnné jeden sloupec této tabulky.

Činnost programátora při simulaci bude popsána na právě řešeném příkladu:

1. Programátor si připraví a pojmenuje paměťová místa, která odpovídají jednotlivým proměnným použitým v programu, resp. připraví záhlaví odpovídající tabulky. V ověřovaném vývojovém diagramu jsou použity proměnné:

   XMAX, X, i, N

2. Zvolí si určitá data v souladu se zadáním úlohy (tj. data vyhovující vstupní podmínce), zde např.:

9	3.0	5.0	8.0	2.0	8.0	8.0	3.0	2.0	5.0
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
N	**1.**	**2.**	**3.**	**4.**	**5.**	**6.**	**7.**	**8.**	**9.**

   (bylo-li jako první zvoleno číslo 9, znamená to, že za ním musí následovat posloupnost 9 čísel).

3) Programátor začne pracovat podle programu, přečtená data postupně odškrtává a do připravených paměťových míst, resp. do příslušných míst tabulky, zapisuje načtené nebo vypočtené hodnoty. Předepsané tisky realizuje na vyhrazenou část papíru nebo tabule.

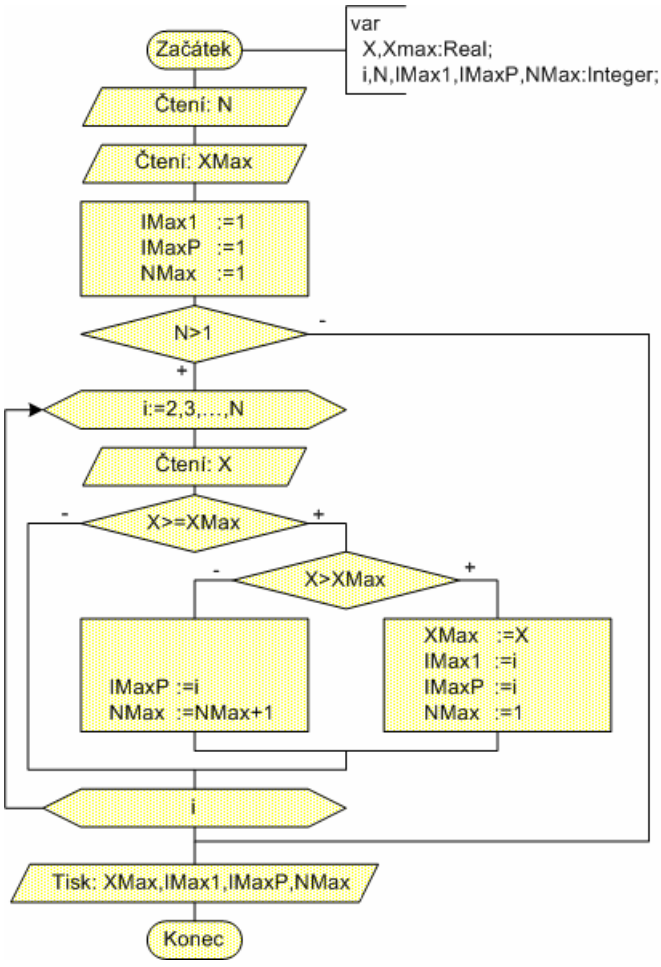
X	XMax	I	**N**
5.0	3.0	2	9
8.0	5.0	3	-//-
2.0	8.0	4	-//-
8.0	-//-	5	-//-
8.0	-//-	6	-//-
3.0	-//-	7	-//-
2.0	-//-	8	-//-
5.0	-//-	9	-//-
5.0	8.0	10	9

Prostor vyhrazený pro tisk: 8

Řešení (b): Za předpokladu, že na vstupním zařízení jsou uvedené údaje pouze jedenkrát a že se na vstupním zařízení nelze vracet k dříve přečteným údajům, je úloha dosud popsanými prostředky prakticky neřešitelná. K řešení je třeba použít pole (viz 1.4. *Pole*, př. 1.4.1.).

Řešení (a, c, d, e): Následující vývojový diagram i příslušný zápis v Delphi představují programy pro současné plnění požadavků (a), (c), (d), (e). V obou programech jsou shodně užity proměnné:

i	-	parametr cyklu,
N	-	počet prvků posloupnosti,
X	-	proměnná, do níž jsou postupně načítány prvky posloupnosti,
XMax	-	dosud největší hodnota ze zpracovaných prvků posloupnosti,
IMax1	-	první místo výskytu dosud největší hodnoty,
IMaxP	-	zatím poslední místo výskytu dosud největší hodnoty,
NMax	-	prozatímní počet výskytů dosud největší hodnoty.



Po ukončení cyklu se význam proměnných *XMax, IMax1, IMaxP* a *NMax* změní. Prozatímní dosavadní hodnoty se stanou hodnotami výslednými. Uvedené programy je možno použít nejen pro hodnoty N dané zadáním, ale i pro $N > 1000$. Program v Delphi je dále možno použít i pro $N = 1$. Pokud by se cyklus z vývojového diagramu realizoval způsobem Algol, byl by příslušný algoritmus rovněž použitelný i pro $N = 1$.

Programy pro jednotlivé požadavky (a), (c), (d), (e) by bylo možno získat např. jen vhodným seškrtáním uvedeného programu pro současné plnění požadavků (a), (c), (d), (e).

```
program MaxACDE;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  X,XMax:Real;
  I,N,IMax1,IMaxP,NMax:Integer;
begin
  ReadLn(N);
  ReadLn(XMax);
  IMax1:=1;
  IMaxP:=1;
  NMax :=1;
  if N>1 then
  begin
    for I:=2 to N do
    begin
      ReadLn(X);
      if X>=XMax then
        if X>XMax then
        begin
          XMax :=X;
          IMax1:=I;
          IMaxP:=I;
          NMax :=1;
        end
        else
        begin
          IMaxP:=I;
          NMax :=NMax+1;
        end;
    end;
  end;
  WriteLn(XMax7:1,IMax1:5,ImaxP:5,NMax:5);
  ReadLn;
end.
```

Příklad. 1.3.4. Vstupní data tvoří přirozené číslo $N > 2$ a posloupnost N celých čísel. Vyjádřete algoritmus, podle kterého se zjistí, zda jsou v posloupnosti bezprostředně po sobě dvě stejná čísla. V případě, že ano, zajistěte tisk jejich společné hodnoty a tisk příslušné dvojice indexů.

Řešení: Základem programu bude cyklus, v jehož těle se načte vždy jeden další (nový) prvek posloupnosti, porovná se s předchozím (starým) prvkem posloupnosti a v případě shody dojde k požadovaným tiskům. V následujícím vývojovém diagramu a v příslušném programu v Delphi jsou užity proměnné:

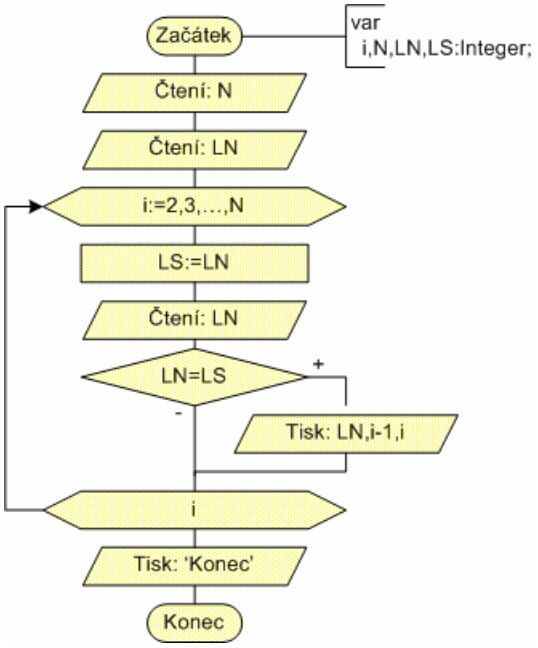
N - počet prvků posloupnosti,

i - parametr cyklu,

LN - nový (další) prvek posloupnosti,

LS - starý (předcházející) prvek posloupnosti.

Před vstupem do cyklu bude v paměťovém místě *LN* 1. prvek posloupnosti, pak se tu v průběhu cyklu postupně vystřídají 2., 3., ... , *N*-tý prvek posloupnosti. V průběhu cyklu se vždy před načtením nové hodnoty do *LN* stávající hodnota z *LN* překopíruje do *LS*. Základní myšlenku algoritmu by bylo možno vyjádřit slovy: „Po přečtení a zpracování se nové číslo stává starým, přičemž zpracováním se rozumí porovnání starého a nového čísla a případný tisk požadovaných hodnot“. Dříve tedy, než je do paměťového místa *LN* načteno další číslo, je třeba dosavadní hodnotu tohoto paměťového místa uschovat v paměťovém místě *LS* (viz příkaz *LS* := *LN*).




```
program DvaStejne;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  I,N,LN,LS:Integer;
begin
  ReadLn(N);
  ReadLn(LN);
  for I:=2 to N do
  begin
    LS:=LN;
    ReadLn(LN);
    if LN=LS then
      WriteLn(LN:5,I-1:5,I:5);
  end;
  WriteLN('Konec');
  ReadLn;
end.
```

Úlohy k samostatnému řešení

Příklad 1.3.4. Vyjádřete algoritmus pro zjištění počtu záporných čísel, která se vyskytují mezi třemi zadanými čísly.

Příklad 1.3.5. Vstupní data tvoří posloupnost reálných čísel z intervalu (-100, 100). Za touto posloupností je dále zařazeno číslo větší než 1000. Vytvořte program pro nalezení:

a) aritmetického průměru prvků posloupnosti,

b) aritmetického průměru čtverců prvků posloupnosti,

c) aritmetického průměru těch prvků posloupnosti, které jsou z intervalu <-30, 70),

d) počtu těch prvků posloupnosti, které jsou větší než aritmetický průměr všech prvků posloupnosti (předpokládejte, že posloupnost má méně než 600 prvků).

Návod:

a) Zadání připouští, že posloupnost může být i prázdná. V tomto případě by prvním vstupním údajem bylo číslo větší než 1000 a požadovaný průměr by nebyl definován.

b), c) Navrhněte pouze nezbytné úpravy v řešení úlohy (a).

d) Za předpokladu, že vstupní údaj lze číst jen jednou (sekvenční čtení), je úloha dosud uvedenými prostředky neřešitelná (je třeba užít pole, viz př. 1.4.2.).

Příklad 1.3.6. Vytvořte jediný program pro současné plnění požadavků (a), (b) i (c) z př. 1.3.5.

Návod: Za předpokladu, že vstupní údaje lze číst jen jednou (sekvenční čtení), není možno vytvořit příkazovou část požadovaného programu jen pouhým postupným napojením příkazových částí z programů pro samostatné požadavky (a), (b) a (c). Program však lze vytvořit pomocí jediného cyklu, v jehož těle se vždy zpracuje jeden další prvek posloupnosti s ohledem na všechny tři požadavky.

Příklad 1.3.7. Vstupní data tvoří přirozené číslo N a za ním následuje posloupnost N reálných čísel. Vytvořte program pro nalezení:

a) aritmetického průměru všech prvků posloupnosti,

b) aritmetického průměru kladných prvků posloupnosti,

c) počtu kladných, počtu záporných a počtu nulových prvků posloupnosti.

Příklad 1.3.8. Vytvořte program pro nalezení maxima i nalezení minima hodnot prvků posloupnosti reálných čísel, jestliže víte, že:

a) posloupnost tvoří N čísel, přičemž přirozené číslo N je prvním vstupním údajem (řešte nejprve jen pro $N \geq 2$),

b) posloupnost je tvořena jen kladnými čísly, přičemž za posledním prvkem posloupnosti je zařazeno záporné číslo,

c) posloupnost tvoří N čísel z intervalu (-100, 100), přičemž přirozené číslo N je prvním vstupním údajem.

1.4. Strukturovaný údaj

Výklad

Při řešení mnoha problémů se pracuje s vektory nebo s maticemi, případně i s jinak uspořádanými skupinami údajů, které představují zvláštní případy strukturovaného údaje, tj. skupiny (souboru) údajů s určitou charakteristickou strukturou (stavbou). Složitějším strukturovaným údajem je např. skupina údajů o žákovi, která se nachází na jedné dvojstránce třídního katalogu základní školy. Uvedený strukturovaný údaj je pak možno považovat za prvek ještě složitějšího strukturovaného údaje - třídního katalogu.

Strukturovaný údaj je skupina údajů s určitou charakteristickou skladbou (s určitým uspořádáním). Z hlediska řešeného problému je možno považovat strukturovaný údaj za údajovou jednotku nějakého vyššího abstraktního typu. Každý ze strukturovaných údajů určitého typu (tj. typu představujícího určitou strukturu, která je dána počtem složek, typem jednotlivých složek a případně i vztahy mezi složkami), pak představuje hodnotu tohoto strukturovaného typu a může se stát hodnotou proměnné příslušného typu, tj. strukturované proměnné, jejíž hodnotou může být jen strukturovaný údaj uvažovaného typu.

Příkladem matematického strukturovaného údaje je komplexní číslo, jakožto uspořádaná množina dvou reálných čísel. Každé komplexní číslo představuje hodnotu komplexního typu (komplexní hodnotu). Komplexní proměnná je pak proměnná, která může nabývat jen komplexních hodnot. Jako další příklad strukturovaného údaje je možno uvést datum, jakožto uspořádanou trojici pořadového čísla dne v měsíci, názvu měsíce a čísla roku. Potom např. *14. leden 2003* je hodnota typu *datum.*

Strukturované údaje je možno dělit ze dvou hledisek:

1. **Strukturovaný údaj:**
   - **homogenní** (všechny jeho složky jsou stejného typu),
   - **nehomogenní**.
2. **Strukturovaný údaj:**
   - **statický** (nemění počet složek během své existence),
   - **dynamický**.

Složky, které tvoří strukturovaný údaj, jsou opět údaje určitého typu. Strukturovaný údaj, jehož všechny složky jsou stejného typu, se nazývá homogenní strukturovaný údaj. V opačném případě se pak jedná o nehomogenní strukturovaný údaj. Z uvedených příkladů strukturovaných údajů tedy např. komplexní číslo představuje homogenní strukturovaný údaj, neboť obě jeho složky jsou údaje stejného typu. Zmíněné datum je zase příkladem nehomogenního strukturovaného údaje, neboť typy jeho složek nejsou stejné. V obou případech se však jedná o statické strukturované údaje, tj. o strukturované údaje, které na rozdíl od dynamických strukturovaných údajů nemění počet složek během své existence.

1.4.1. Pole

Výklad

Vektory (jako uspořádané *n*-tice čísel) a matice jsou zvláštními případy homogenních statických strukturovaných údajů zvaných *pole*.

Pole je homogenní statický strukturovaný údaj, jehož složky zvané ***prvky pole*** se rozlišují jen indexy, přičemž všechny prvky jednoho pole mají stejný počet indexů a jednotlivé indexy, které obvykle nabývají jen celočíselných hodnot, mají pevný rozsah, tj. u každého indexu je nezávisle na ostatních indexech určena jeho nejmenší hodnota (dolní mez indexu) a největší hodnota (horní mez indexu). Z rozsahu jednotlivých indexů pole lze tedy zjistit počet všech prvků tohoto pole.

Pole se označuje jediným jménem, kterým je obvykle identifikátor. Prvky pole jsou pak označovány pomocí jména pole a indexů. Jména všech prvků jednoho pole jsou tedy tvořena jménem tohoto pole a uspořádanou množinou indexů, přičemž počet indexů je u všech prvků pole stejný.

Rozměr (dimenze) pole: Počet indexů, který je u všech prvků jednoho pole stejný, se nazývá rozměrem (dimenzí) pole. Vektor je tedy jednodimenzionální pole, neboť jeho prvky mají jediný index, matice je dvojdimenzionální pole, její prvky mají indexy dva. Na matici lze z hlediska algoritmizace rovněž pohlížet jako na „vektor“, jehož složkami jsou opět vektory, tj. jako na nejjednodušší případ pole polí.

Pozor! Při volnějším vyjadřování může být pouhý termín *pole* použit ve smyslu:

- pole jako konstanta (tj. pro konstantu typu pole),
- pole jako proměnná (tj. pro proměnnou typu pole),
- pole jako typ (tj. pro typ pole).

Např. v matematice se termín *vektor* někdy používá pro vektorovou konstantu, někdy pro vektorovou proměnnou a někdy jen pro vyjádření typu vektoru.

Výklad

Prvky proměnné typu pole se v některých programovacích jazycích nazývají indexované proměnné. ***Indexovaná proměnná*** je tedy prvkem pole (přesněji prvkem proměnné typu pole) a při práci na počítači indexovaná proměnná představuje, stejně jako jednoduchá proměnná, jméno paměťového místa. Mají-li být pro pole, tj. pro všechny prvky pole, vymezena a pojmenována paměťová místa, musí být kromě jména pole zadán též rozsah pole, tj. počet indexů a rozsah jednotlivých indexů, což se provádí deklarací (popisem) pole.

I kdyby u předchozích vývojových diagramů, v nichž se indexovaná proměnná nevyskytuje, nebyly uváděny deklarace používaných proměnných, bylo by možno z těchto diagramů zjistit (a to i bez znalosti zadání úlohy a bez znalosti vstupních údajů) jména všech jednoduchých proměnných a tedy i konkrétní počet všech paměťových míst nutných pro realizaci příslušného algoritmu.

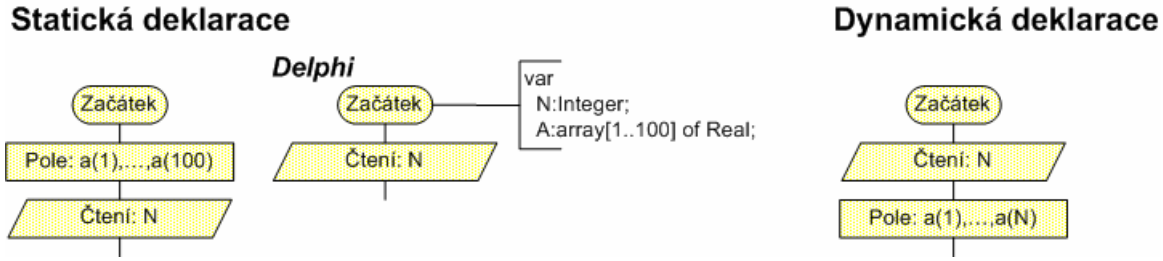
Z pouhého zápisu indexované proměnné v algoritmu (např. ze zápisu a_i) však nevyplývá, kolik prvků příslušné pole má a jakých hodnot indexy nabývají. Aby tedy bylo možno připravit konkrétní počet všech paměťových míst nutných pro realizaci příslušného algoritmu, je třeba vždy (ve všech programovacích jazycích) pole deklarovat. V případě jednorozměrného pole stačí uvést jeho název a hodnotu nejnižšího a nejvyššího indexu (v některých jazycích je nejnižším indexem implicitně 0 nebo 1).

Z hlediska algoritmizace i některých programovacích jazyků je možno deklaraci polí provádět dvěma způsoby. Jestliže rozsah pole nezávisí na konkrétních hodnotách vstupních údajů, může být použita deklarace pole, která se označuje jako statická.

Statická deklarace pole je deklarace, v níž jsou pro popis mezí jednotlivých indexů užity konstanty. Statická deklarace pole představuje konkrétní popis pole, který umožňuje vymezení tohoto pole ještě před zahájením vlastní práce podle programu.

Dynamická deklarace pole (z hlediska algoritmizace) používá, na rozdíl od statické deklarace pole, pro popis mezí indexů pole jedné nebo více proměnných. Konkrétní rozsah takto popsaného pole závisí na hodnotě použité proměnné, resp. proměnných, v tom okamžiku, ve kterém se provádí vymezení pole, což na rozdíl od statické deklarace pole musí být vždy až po zahájení práce podle programu. Pro dynamickou deklaraci pole musí být v příslušném programovacím jazyku k dispozici *příkaz pole*.

Mají-li být při realizaci algoritmu užity reálné proměnné a_1, a_2, ... , a_n , kde n je z intervalu <2, 100>, je třeba deklarovat pole, které má při statické deklaraci prvky a_1 až a_{100} . Následující části vývojových diagramů naznačují základní rozdíl mezi statickou a dynamickou deklarací:



Rozlišování statické a dynamické deklarace má značný význam zejména proto, že některé programovací jazyky (např. Fortran) připouštějí jen statickou deklaraci pole. Programovací jazyky, které umožňují dynamickou deklaraci pole (např. Algol, Basic), připouštějí rovněž i statickou deklaraci pole. Přitom není podstatné, zda pole deklarovaná staticky jsou vytvářena jinak než pole deklarovaná dynamicky (např. v mnoha verzích jazyka Basic jsou i staticky popsaná pole vytvářena dynamicky).

Konkrétně Delphi má jen statickou deklaraci pole, která, na rozdíl od mnoha jiných programovacích jazyků, nemá formu příkazu. Podobně, jako se v Delphi deklarují jednotlivé jednoduché proměnné, se tu deklaruje i pole (přesněji proměnná typu pole), které v některých jiných programovacích jazycích představuje jen množinu indexovaných proměnných.

Vzhledem k orientaci těchto skript na Delphi bude proto v následujících vývojových diagramech používána jen statická deklarace pole, a to přímo formou převzatou z Delphi (viz příslušná část vývojového diagramu), která obsahuje i informaci o typu prvků pole.

Příslušná přímá deklarace požadovaného pole *A* a proměnné *N* má tedy v Delphi např. tvar:

```
var
  A:array[1..100] of Real;
  N:Integer;
```

(na pořadí deklarace proměnné *A* typu pole a celočíselné proměnné *N* tu nezáleží).

Řešené úlohy

Příklad 1.4.1. Vstupní údaje tvoří přirozené číslo *N* z intervalu <2, 100> a dále posloupnost *N* celých čísel. Vyjádřete algoritmus pro nalezení všech míst výskytu maxima posloupnosti, tj. největší hodnoty, které nabývá některé z čísel posloupnosti (viz př. 1.3.3.b).

Řešení: Požadovaných míst výskytu maxima posloupnosti může být více. Např. pro vstupní údaje $N = 11$ a posloupnost čísel

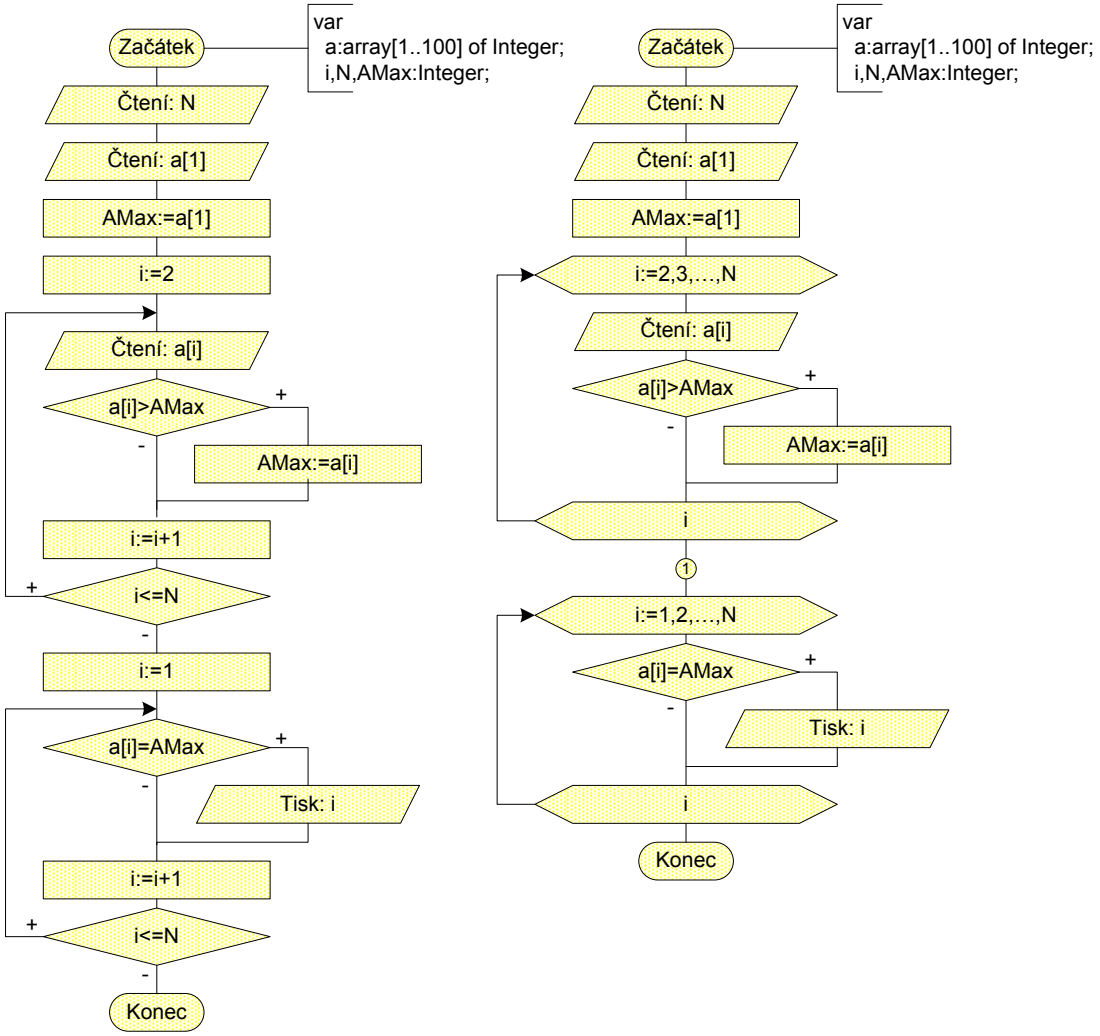
3, 5, 5, 8, 1, 8, 2, 5, 8, 1, 5

jsou řešením pořadová čísla

4, 6, 9.

Místo výskytu maxima posloupnosti je však možno hledat teprve po nalezení tohoto maxima, což znamená, že se nejprve musí najít maximum posloupnosti (viz př. 1.3.3.a), a teprve potom se musí celá posloupnost znovu projít, aby mohla být provedena příslušná zjištění. To však znamená, že všechna čísla posloupnosti musí být znovu k dispozici, např. mohou být uložena v paměťových místech počítače. Pro prvky posloupnosti může být potřeba až 100 paměťových míst, která je výhodné označit pomocí indexů, např. $a_1, a_2, \ldots, a_{100}$.

Celé řešení, které úzce navazuje na řešení př. 1.3.3.a, je formou vývojového diagramu uvedeno dvakrát, přičemž druhý vývojový diagram se od prvního liší jen tím, že je v něm pro cykly se známým počtem opakování použito jejich symbolické vyjádření.

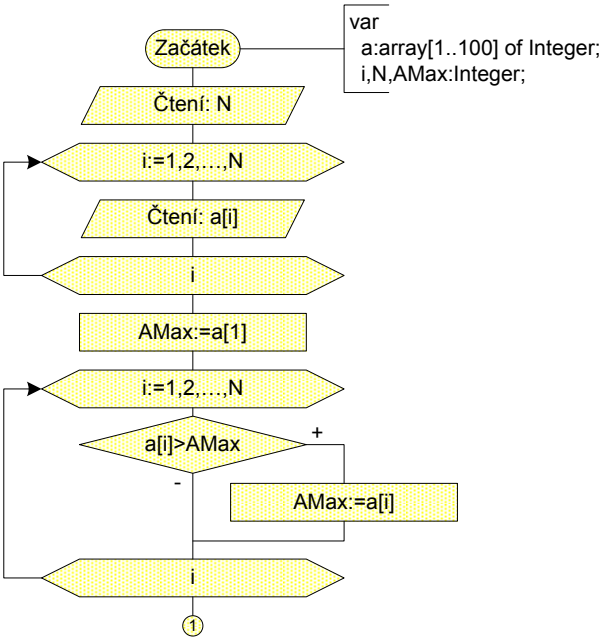


Vzhledem k tomu, že při řešení př. 1.3.3.a (hledání maximální hodnoty z posloupnosti *n* čísel) nebylo užito pole, bylo třeba v jednom cyklu zajistit jak čtení, tak i vlastní hledání maximální hodnoty. V nyní řešeném příkladu však pole musí být užito, a je tedy možno oddělit čtení prvků posloupnosti od vlastního hledání maximální hodnoty, což v jistém smyslu celé řešení zpřehlední a zjednoduší. Řešení pak více odpovídá základní myšlence modulárního programování (viz část 1.5.3. *Modulární programování*), tj. myšlence rozložit řešený problém na pokud možno izolované podproblémy, jejichž jednotlivé řešení je podstatně dostupnější než souhrnné řešení celého problému.

Hrubá osnova následujícího řešení by pak mohla být vyjádřena ve třech bodech:

1) Čtení vstupních údajů.
2) Hledání maximální hodnoty.
3) Hledání indexů prvků, které mají nalezenou maximální hodnotu.

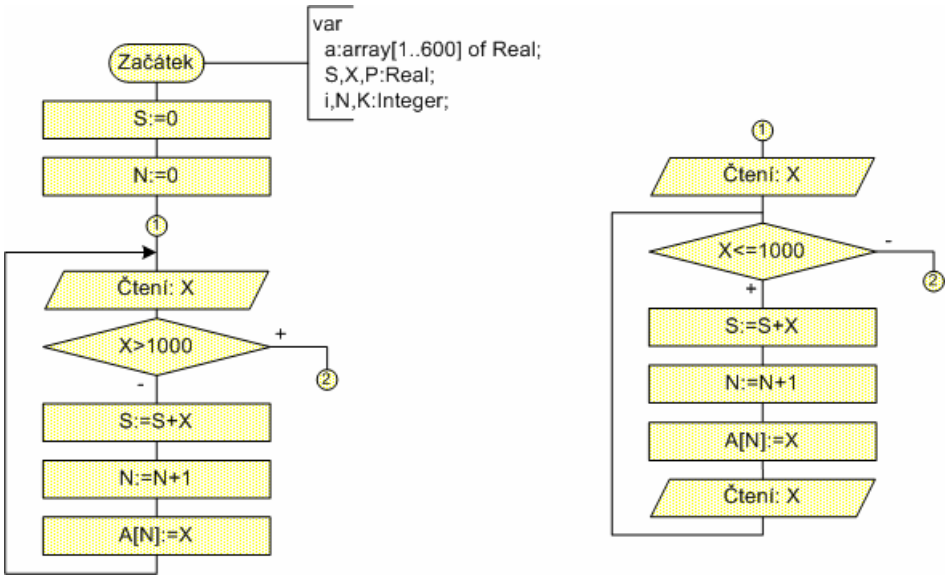
V cyklu pro hledání maxima pak může parametr *I* nabývat hodnot 1, 2 ,… ,*N*, i když by stačily jen hodnoty 2, 3 ,…, *N* (pro všechny prvky posloupnosti je tak „ zde sice zbytečně“ předepsána stejná činnost). Vývojový diagram je uveden jen po spojku *1*, dále viz předchozí řešení. Zápis v Delphi je uveden celý a pro zpřehlednění tisku výsledků tisku je doplněn tiskem textu „*Indexy:* “.




```
program Indexy;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A:array[1..100]of Integer;
  I,N,AMax:Integer;
begin
  ReadLn(N);
  for I:=1 to N do
    ReadLn(A[I]);
  AMax:=A[1];
  for I:=1 to N do
    if A[I]>AMax then
      AMax:=A[I];
  Write('Indexy: ');
  for I:=1 to N do
    if A[I]=AMax then
      Write(I:4);
  ReadLn;
end.
```

Příklad 1.4.2. Vstupní údaje tvoří posloupnost reálných čísel, která jsou z intervalu (-100, 100), a dále pak číslo větší než 1000. Vyjádřete algoritmus pro nalezení počtu těch prvků posloupnosti, které jsou větší než aritmetický průměr všech prvků posloupnosti. Řešte za předpokladu, že posloupnost má méně než 600 prvků.

Řešení: Jedná se vlastně o př. 1.3.5.d, který úzce navazuje na př. 1.3.5.a. Jednotlivé prvky posloupnosti totiž mohou být porovnány s aritmetickým průměrem posloupnosti teprve až po výpočtu tohoto průměru. Vzhledem k tomu, že po výpočtu průměru se bude třeba vrátit k jednotlivým prvkům posloupnosti, je výhodné (a prakticky téměř nutné) uložit je do pole. Již při načítání prvků posloupnosti do pole lze provádět jejich postupné sčítání a počítání pro potřebu následujícího výpočtu aritmetického průměru (první řešení). Obdobně jako v předchozím příkladě 1.4.1. však lze zahájit výpočty až po načtení všech vstupních údajů (viz dále druhé řešení).



Z prvního řešení je uvedena jen část vývojového diagramu před spojkou *2* (přičemž část mezi spojkami *1* a *2* je vyjádřena dvěma způsoby), druhé řešení je uvedeno celé (včetně zápisu v Delphi).

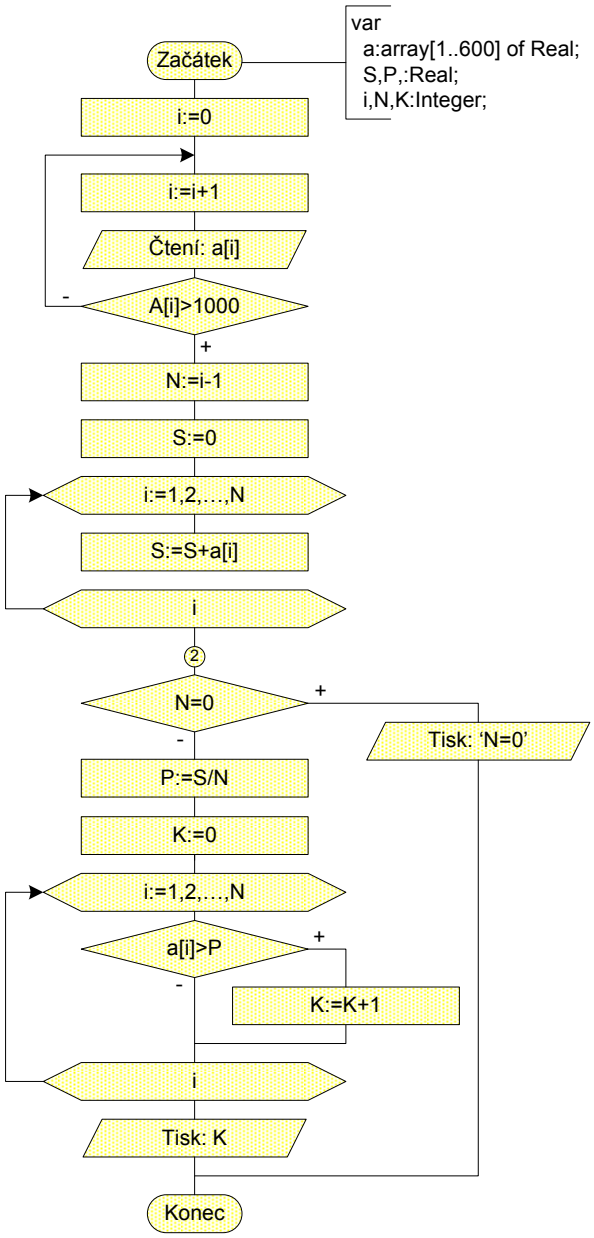
Význam použitých proměnných:

- *a* - pole pro prvky posloupnosti,
- *S* - součet (dosud zpracovaných) prvků posloupnosti,
- *N* - počet (dosud zpracovaných) prvků posloupnosti,
- *X* - pomocná proměnná pro načítání prvků posloupnosti,
- *P* - aritmetický průměr prvků posloupnosti,
- *K* - proměnná pro počet hledaných prvků,
- *i* - parametr cyklů se známým počtem opakování.

V prvním řešení je mj. ukázán postup, při němž nejsou vstupní údaje načítány přímo do prvků pole, ale do pomocné (pracovní) proměnné *X*, a teprve po částečném zpracování jsou uloženy do příslušného prvku pole ***a***, přičemž druhé vyjádření mezi spojkami *1* a *2* by bylo, na rozdíl od prvního, vhodnější pro převod do Delphi.

V následujícím druhém řešení (též v Delphi) je v prvním cyklu jen přímé načtení prvků posloupnosti do pole ***a***.

Prvním cyklem obou řešení je cyklus s neznámým počtem opakování. Po jeho ukončení jsou vždy (alespoň) načteny všechny vstupní údaje a je znám počet prvků posloupnosti. Další cyklus nebo cykly jsou již tedy cykly se známým počtem opakování.




```
program Pocet;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A:array[1..600] of Real;
  S,P:Real;
  I,N,K:Integer;
begin
  I:=0;
  repeat
    I:=I+1;
    ReadLn(A[I]);
  until A[I]>1000;
  N:=I-1;
  S:=0;
  for I:=1 to N do
    S:=S+A[I];
  if N=0 then
    WriteLn('N=0')
  else
  begin
    P:=S/N;
    K:=0;
    for I:=1 to N do
      if A[I]>P then
        K:=K+1;
    WriteLn(K);
  end;
  ReadLn;
end.
```

Příklad. 1.4.3. Vstupní údaje tvoří přirozené číslo N a dále posloupnost N celých kladných čísel, přičemž:

a) $N = 780$ a prvky posloupnosti jsou čísla nejvýše osmiciferná,

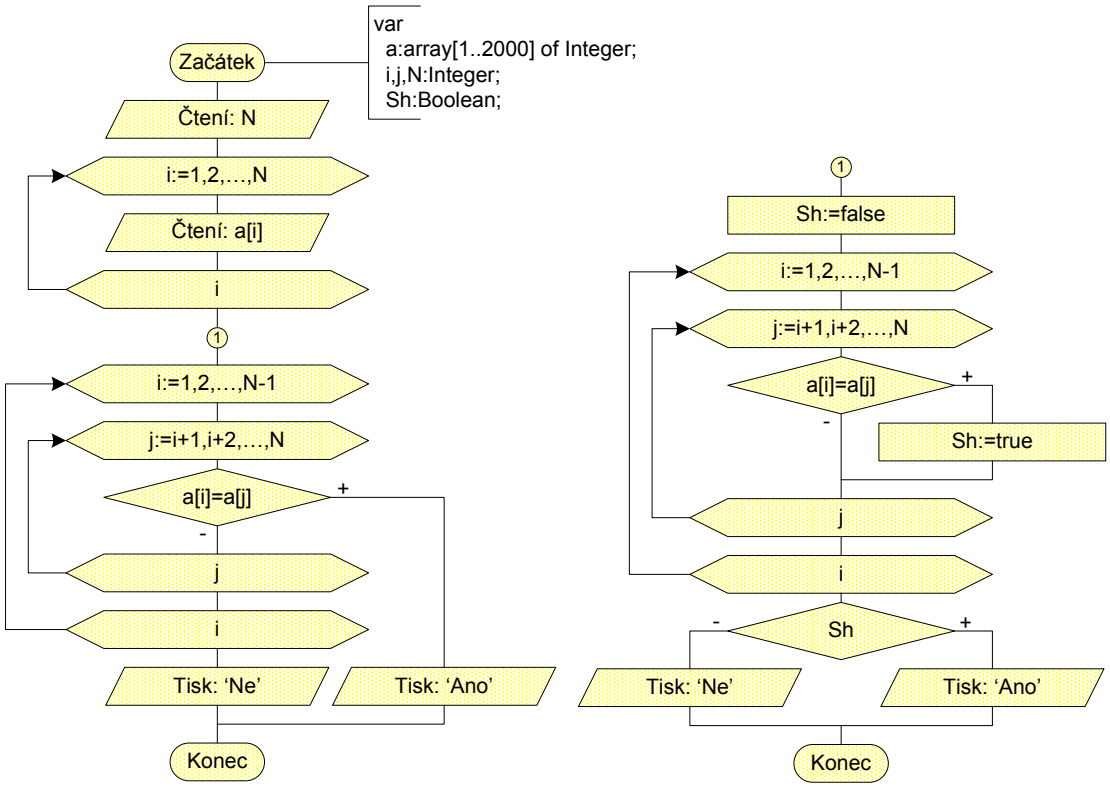
b) $N = 780$ a prvky posloupnosti jsou čísla nejvýše trojciferná,,

b) $N = 1510$ a prvky posloupnosti jsou čísla nejvýše trojciferná.

Vyjádřete algoritmus, podle kterého se zjistí, zda jsou v posloupnosti alespoň dva prvky stejné.

Řešení: **a)** V navrženém řešení jsou prvky posloupnosti uloženy do pole ***a***. Vlastní algoritmus pro požadované zjištění začíná u spojky *1* a je založen na řízeném porovnávání (srovnávání) prvků posloupnosti. Nastane-li při některém srovnání

shoda, je další porovnávání prvků zbytečné a následuje tisk kladné odpovědi a ukončení práce. Nenastává-li shoda, pokračuje se v dalším porovnávání.



Porovnávání prvků posloupnosti jsou ve vývojovém diagramu za spojkou *1* a jsou předepsána v následujícím pořadí: Nejprve se srovnává první prvek se všemi následujícími, pak druhý prvek se všemi následujícími, tj. se třetím, čtvrtým atd., pak třetí se všemi následujícími atd., což je vyjádřeno dvěma do sebe *vloženými cykly* (*vřazenými cykly*), v těle vnějšího cyklu s parametrem *i* je vnitřní cyklus s parametrem *j*. Tato část vývojového diagramu, dosud uvedená dvakrát, bude uvedena ještě jednou.

Vzhledem k tomu, že maximální možný počet opakování v jednotlivých cyklech je znám, je v prvním vyjádření (které bezprostředně navazuje na úvodní část vývojového diagramu) formálně využito cyklů se známým počtem opakování. V případě shody však není předepsaný maximální možný počet opakování využit, neboť je tu předepsáno předčasné opuštění cyklu (*skok z cyklu*). Příslušná řídicí struktura však není v souladu se zásadami strukturovaného programování (viz 1.5.2. *Strukturované programování*), a i když v Delphi *příkaz skoku* existuje, nebude dále používán.

Uvedený problém by bylo možno vyřešit např. tak, že místo skoku z cyklu by se informace o shodě uložila do vhodné proměnné a po provedení všech předepsaných opakování (tj. i „zbytečných“) by se hodnota této proměnné otestovala. Pokud by byla použita celočíselná proměnná, stačilo by, kdyby nabývala jen dvou hodnot:

0 - shoda (dosud) nenastala,

1 - shoda (již) nastala.

Výklad

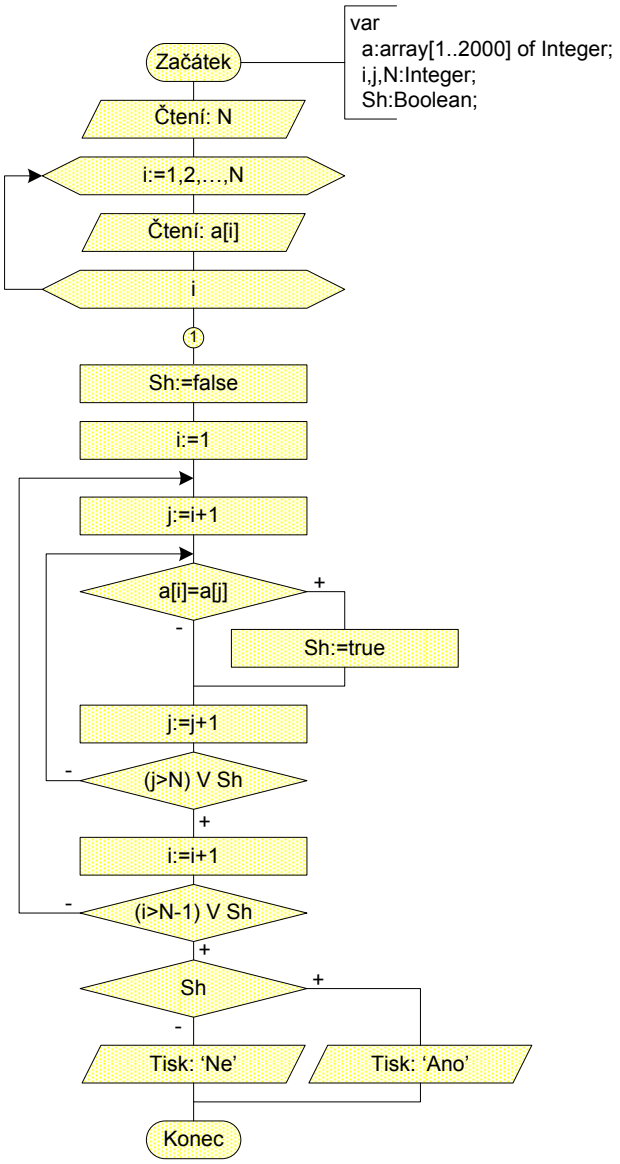
Pro vyjadřování a pro práci s logickými hodnotami 0 a 1 mají mnohé programovací jazyky speciální prostředky. V Delphi je možno používat ***logické proměnné***, jejichž typ je v Delphi označen identifikátorem *Boolean* a které mohou nabývat (jen) hodnot *False* (logická 0) a *True* (logická 1).

Pokračování řešení př. 1.4.3.a:

Užití logické proměnné, která je nazvána *Shoda* (ve vývojovém diagramu jen *Sh*), je ukázáno v druhém a třetím řešení. Nevýhodou druhého řešení je skutečnost, že i v případě nalezení shodných prvků probíhají zbývající, nyní již zbytečná, srovnávání.

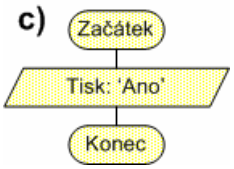
```
{Část druhého řešení}
  Shoda:=False;
  for I:=1 to N-1 do
    for J:=I+1 to N do
      if A[I]=A[J] then
        Shoda:=True;
  if Shoda then
    WriteLn('Ano')
  else
    WriteLn('Ne');
  ~~~
```

Na rozdíl od druhého řešení jsou ve třetím řešení místo symbolického vyjádření cyklů se známým počtem opakování použita nejen vyjádření cyklu typu *Fortan*, ale v testu na ukončení cyklu je testována i proměnná *Shoda*. Nabude-li proměnná *Shoda* hodnotu *True*, je provádění cyklů „řádně“ ukončeno. Tvar obou cyklů pak přímo odpovídá struktuře příkazu **repeat** z Delphi.




```
program ShodaA3;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A:array[1..2000] of
    Integer;
  I,J,N:Integer;
  Shoda:Boolean;
begin
  ReadLn(N);
  for I:=1 to N do
    ReadLn(A[I]);
  Shoda:=False;
  I:=1;
  repeat
    J:=I+1;
    repeat
      if A[I]=A[J] then
        Shoda:=True;
      J:=J+1;
    until (J>N) or Shoda;
    I:=I+1;
  until (I>N-1) or Shoda;
  if Shoda then
    WriteLn('Ano')
  else
    WriteLn('Ne');
  ReadLn;
end.
```

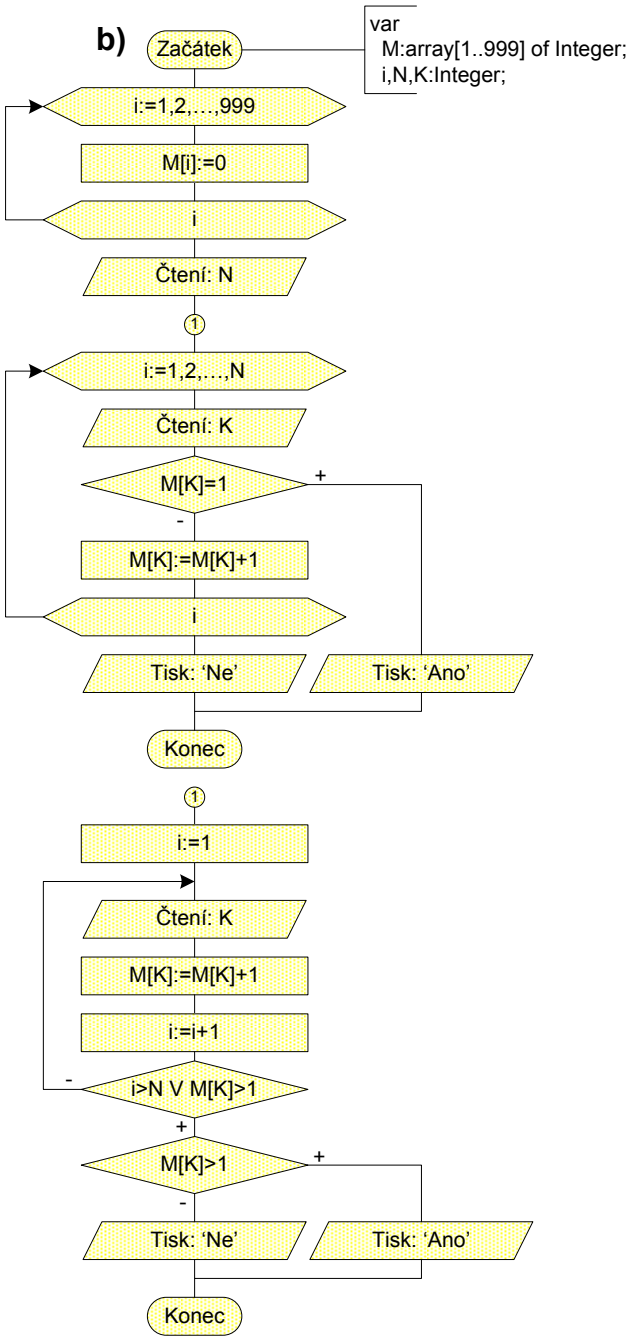
Řešení b), c): Uvedená řešení úlohy (a) nejsou vázána na počet cifer jednotlivých prvků posloupnosti, a proto každé z nich může být užito i pro posloupnost trojciferných čísel. V případě (c) je však předem znám výsledek, neboť zadaná posloupnost má 1510 prvků a přitom posloupnost nejvýše trojciferných čísel (celých kladných), v níž by každá dvě čísla byla různá, by mohla mít nejvýše 999 prvků.



Pro případ (c) tedy stačí řešení *c*. V případě (b) není, na rozdíl od (c), výsledek předem znám, práce podle řešení navržených pro případ (a) by však mohla představovat až 303 810 srovnání, což je důvodem k hledání méně pracného algoritmu.

Méně pracný algoritmus pro zadání (b) vyjadřuje následující vývojový diagram *b*. Každé možné hodnotě nejvýše trojciferného přirozeného čísla tu odpovídá jeden prvek pole *M*. V každém prvku pole *M* je nejprve nula, po prvním přečtení kteréhokoliv čísla z intervalu <1, 999> se hodnota příslušného prvku pole *M* změní na jedna. U každého načítaného prvku posloupnosti je tedy možno lehce zjistit, zda se jeho hodnota mezi předchozími čísly už vyskytla nebo nevyskytla.

Podle prvního návrhu části programu (b) za spojkou *1* by tedy v každém prvku pole *M* byla buď nula, nebo jednička, a místo celočíselného by tedy mohlo být použito booleovské pole.



Celočíselné pole zde však slouží jako ukázka *pole počítadel*, které by bylo výhodné pro mnoho příbuzných příkladů (viz př. 1.4.9.).

```
program ShodaB;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  M:array[1..999]of Integer;
  I,N,K:Integer;
begin
  for I:=1 to 999 do
    M[I]:=0;
  ReadLn(N);
  I:=1;
  repeat
    ReadLn(K);
    M[K]:=M[K]+1;
    I:=I+1;
  until (I>N) or (M[K]>1);
  if M[K]>1 then
    WriteLn('Ano')
  else
    WriteLn('Ne');
  ReadLn;
end.
```

První návrh části programu (b) za spojkou *1* však opět obsahuje skok z cyklu. Pro přepis do Delphi byla proto tato část vývojového diagramu navržena ještě jednou tak, aby bylo možno využít příkaz **repeat**.

Vhodným námětem pro samostatnou práci by bylo přepracování této části algoritmu tak, aby byla vhodná pro použití příkazu **while**.

Další možnost jak vyjádřit tuto část programu bude uvedena přímo v části 1.5.2. *Strukturované programování*.

Příklad. 1.4.4. Vstupní údaje tvoří přirozené číslo N z intervalu <2, 100> a dále posloupnost N reálných čísel. Vyjádřete algoritmus pro seřazení prvků posloupnosti do nerostoucí posloupnosti.

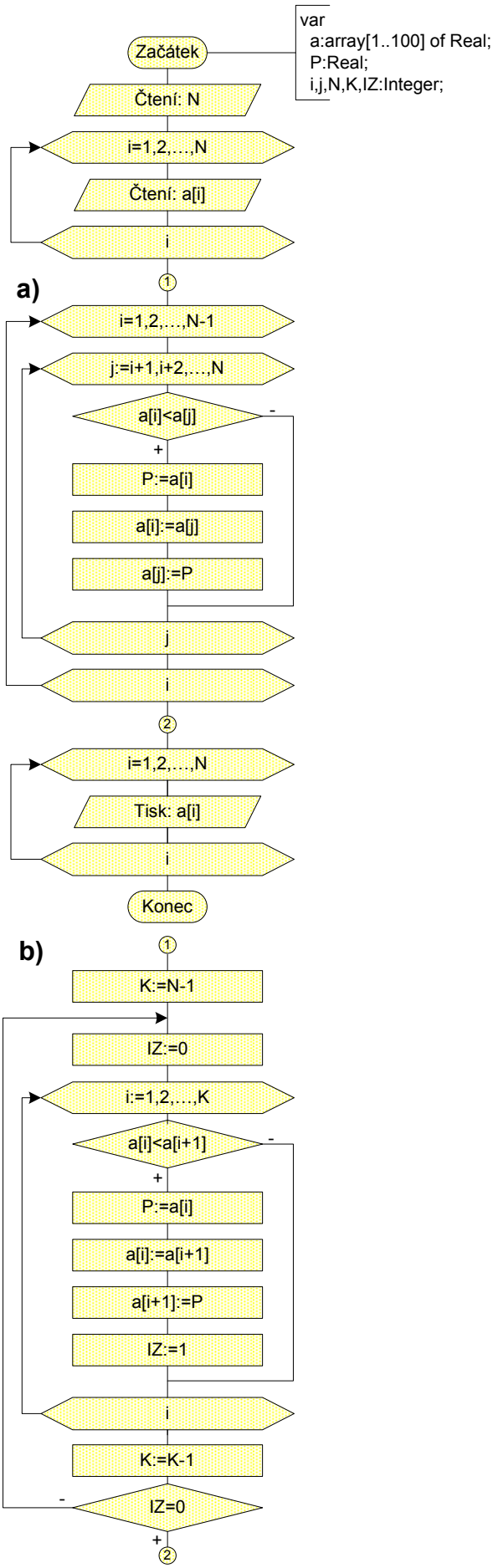
Řešení: Značná část algoritmů pro seřazení prvků posloupnosti do nerostoucí nebo do neklesající posloupnosti (*algoritmy třídění*) je založena na stejném principu, kterým je přerovnání prvků posloupnosti do požadovaného pořadí v rámci paměťových míst, v nichž je posloupnost uložena. Základem třídění (seřazování) je řízená záměna hodnot vybraných dvojic paměťových míst (viz řešení příkladu 1.2.3.).

Z mnoha známých algoritmů třídění budou uvedeny jen dva. Následující vývojový diagram je spojkami *1* a *2* rozdělen na tři části. První část vývojového diagramu, tj. část před spojkou *1*, odpovídá čtení vstupních údajů a uložení prvků posloupnosti do paměťových míst $a_1, a_2, \ldots, a_N$ pole $\boldsymbol{a}$. Střední část vývojového diagramu, která vyjadřuje vlastní třídění, je navržena dvěma způsoby *(a)* a *(b)*. Třetí část představuje tisk výsledku, kterým je nová posloupnost čísel v paměťových místech $a_1, a_2, \ldots, a_N$.

Třídění *(a)* je založeno na výběru „největšího“ prvku z dosud neuspořádané části posloupnosti. Nejprve se tedy vybírá největší ze všech prvků, resp. jeden z největších, a přemístí se na první pozici, pak se z prvků na dalších pozicích vybírá opět „největší“ a přemístí se na druhou pozici atd.

V návrhu *(b)* je k třídění použita *bublinová metoda* (nejjednodušší varianta). Třídění je zde zajišťováno opakovaným prováděním činnosti zvané *průchod*, při níž se prochází posloupnost prvků od počátku do konce a postupně se kontroluje a případně i zajišťuje uspořádání dvojic (jen) sousedních prvků.

Průchody se přitom opakují tak dlouho, až se provede průchod bez výměn. Proměnná *IZ - indikace změn* má po průchodu bez výměn hodnotu 0, jinak 1 (místo celočíselné proměnné *IZ* by tedy opět bylo možno použít proměnnou typu *Boolean*).




```
program Trid1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A:array[1..100] of Real;
  P:Real;
  I,J,N:Integer;
begin
  ReadLn(N);
  for I:=1 to N do
    ReadLn(A[I]);
  {Začátek třídění (A)}
  for I:=1 to N-1 do
    for J:=I+1 to N do
      if A[I]<A[J] then
      begin
        P   :=A[I];
        A[I]:=A[J];
        A[J]:=P;
      end;
  {Konec třídění}
  for I:=1 to N do
    WriteLn(A[I]);
  ReadLn;
end.
```

```
  {Začátek třídění (B)}
  K:=N-1;
  repeat
    IZ:=0;
    for I:=1 to K do
      if A[I]<A[I+1] then
      begin
        P     :=A[I];
        A[I]  :=A[I+1];
        A[I+1]:=P;
        IZ    :=1;
      end;
    K:=K-1;
  until IZ=0;
  {Konec třídění}
```

Oba návrhy jsou zakresleny tak, aby především jednoduše a názorně vyjadřovaly základní myšlenku příslušného třídění, není tedy např. brán ohled na pracnost, kterou oba návrhy pro třídění předepisují.

Úlohy k samostatnému řešení

Příklad 1.4.5. Vstupní údaje tvoří přirozené číslo N z intervalu <2, 100> a dále souřadnice dvou N-složkových vektorů ***a***, ***b*** v pořadí:

1) $a_1, b_1, a_2, b_2, \ldots, a_N, b_N,$
2) $a_1, a_2, \ldots, a_N, b_1, b_2, \ldots, b_N,$
3) $a_N, b_N, a_{N-1}, b_{N-1}, \ldots, a_1, b_1,$
4) $a_1, a_2, \ldots, a_N, b_N, b_{N-1}, \ldots, b_1,$
5) $a_1, b_N, a_2, b_{N-1}, \ldots, a_N, b_1$.

Vyjádřete algoritmus pro výpočet skalárního součinu vektorů ***a*** a ***b*** , tj. pro výpočet hodnoty $\boldsymbol{a}\cdot\boldsymbol{b}$, kde

$\mathbf{a}\cdot\mathbf{b} = a_1\cdot b_1 + a_2\cdot b_2 + \ldots + a_N\cdot b_N$,

dvěma způsoby:

a) tak, aby pro souřadnice obou vektorů bylo třeba co nejméně paměťových míst, tj. pokud možno bez užití pole pro vektor ***a***, resp. pro vektor ***b***,

b) tak, aby vlastní výpočet skalárního součinu začínal až po přečtení všech složek obou vektorů.

Příklad 1.4.6. Vstupní údaje tvoří přirozené číslo $N \leq 1000$ a dále posloupnost N celých čísel. Vyjádřete algoritmus:

a) pro nalezení počtu kladných, počtu záporných a počtu nulových prvků posloupnosti,

b) podle kterého se nejprve vytisknou všechny kladné prvky posloupnosti, potom všechny záporné prvky posloupnosti a na závěr se vytiskne informace o počtu nulových prvků posloupnosti,

c) podle kterého se vytisknou buď jen kladné prvky posloupnosti, nebo jen záporné prvky posloupnosti, přičemž kladné se tisknou v případě, že je jich více než záporných,

d) pro výpočet aritmetického průměru kladných prvků posloupnosti a aritmetického průměru záporných prvků posloupnosti.

Příklad 1.4.7. Vyjádřete algoritmus pro výpočet odhadu střední hodnoty (aritmetického průměru) a směrodatné chyby tohoto odhadu (střední kvadratické chyby) pro posloupnost n měření skalární veličiny x, tj. pro hodnoty $x_1, x_2, \ldots, x_n$, kde $n \leq 50$ je první vstupní údaj.

Návod: Střední kvadratická chyba má hodnotu $\sqrt{\frac{\Sigma(x_i - \bar{x})^2}{n \cdot (n-1)}}$, kde $\bar{x}$ je aritmetický průměr z x_i.

Příklad 1.4.8. Vstupní údaje tvoří přirozené číslo N z intervalu <10, 100> a dále pak posloupnost N celých čísel. Vyjádřete algoritmus:

a) podle kterého se zjistí, zda se v posloupnosti vyskytují tři stejná čísla bezprostředně po sobě, pokud ano, nechť je vždy tištěn index prvního z nich,

b) podle kterého se zjistí, zda se v posloupnosti vyskytují čtyři záporná čísla bezprostředně po sobě, pokud ano, nechť je vždy tištěn index prvního z nich,

c) podle kterého se vytisknou z posloupnosti jen záporná čísla, a to v opačném pořadí než jsou v zadané posloupnosti.

Příklad 1.4.9. Vstupní údaje tvoří posloupnost přirozených nejvýše dvojciferných čísel a dále pak číslo -1. Vyjádřete algoritmus:

a) pro zjištění počtu výskytů jednotlivých čísel od 1 do 99 ,

b) pro zjištění těch čísel, která se v posloupnosti vyskytují právě jednou,

c) pro zjištění těch čísel, která se v posloupnosti vyskytují alespoň pětkrát,

d) pro zjištění počtu těch čísel od 1 do 99 , která se v posloupnosti nevyskytují,

e) podle kterého se vytisknou počty čísel z jednotlivých desítek (např. 2. desítka - čísla od 11 do 20).

1.5. Metody vytváření algoritmu

Výklad

Vytváření algoritmu, resp. programu, je tvůrčí proces, a proto pro něj neexistuje žádný jednoznačný návod. Při vytváření algoritmu, tj. při analýze problému, návrhu a rozpracování algoritmu, a zejména pak při vyjadřování algoritmu pomocí programovacího jazyka, je však možno dodržovat některé zásady a z nich vyplývající přístupy (příp. přímo postupy či metody), které celý proces programování značně usnadní.

Metody, příp. jen postupy, se kterými lze vystačit při vytváření menších programů, se mohou stát nepřijatelnými při tvorbě velkých programů nebo programových systémů. Při řešení složitějších problémů již prakticky nelze postupovat jen nahodile a intuitivně, ale k vytváření příslušného programu je třeba přistupovat racionálně, systematicky a s jistou znalostí metod programování.

Mezi běžné požadavky na program patří, aby byl přehledný a aby pokud možno jednoduchým a názorným způsobem vyjadřoval řešení problému. Požadavky na jednoduchost a názornost však nelze zajistit až při vlastním zápisu algoritmu programovacím jazykem, ale musí být zabezpečovány již během celého vytváření algoritmu, tj. již při analýze problému a při návrhu rozpracování algoritmu, tak, aby struktura získaného programu byla jasným odrazem struktury řešeného problému a aby přehledně a názorně vyjadřovala způsob řešení zvolený programátorem.

Přístupy k analýze problému, k návrhu a rozpracování algoritmu je možno rozdělit z více hledisek:

- **Přístup k návrhu algoritmu** - k návrhu algoritmu je možno přistupovat dvěma základními způsoby, kterým odpovídají dvě metody návrhu algoritmu:
  metoda návrhu algoritmu shora dolů,
  metoda návrhu algoritmu zdola nahoru.
- **Přístup k vytváření řídicí struktury algoritmu** - systematický a metodický přístup k vytváření řídicí struktury algoritmu jen z několika málo „vhodných“ základních řídicích struktur rozpracovává metoda zvaná ***strukturované programování***.
- **Přístup k členění problému na izolované podproblémy** - myšlenka rozložit řešený problém na izolované podproblémy, jejichž řešení je dostupnější než souhrnné řešení celého problému, je základem metody zvané ***modulární programování***.

1.5.1. Metody návrhu algoritmu shora dolů a zdola nahoru

Výklad

Metoda návrhu algoritmu shora dolů je založena na analýze problému a jeho postupném rozkladu na dílčí problémy (podproblémy), které jsou pak podle potřeby dále rozkládány. Tomuto stupňovitému rozkladu (dekompozici) problému na podproblémy pak odpovídá i návrh algoritmu, který je nejprve hrubý a potom se postupně zjemňuje až do konečného tvaru, za nějž je možno považovat tvar, který se již dá plně vyjádřit příslušným programovacím jazykem. Návrh programu tedy vlastně začíná jen jakousi osnovou, pro jejíž zápis je obvykle možno využít programovací jazyk jen v omezené míře, neboť v počátečních příkazech se běžně vyskytují abstraktní příkazy (využívající často výrazy z přirozeného jazyka) pro vyjádření některých dosud nerozpracovaných dílčích algoritmů, resp. vztahujících se na některé zatím blíže nespecifikované strukturované údaje nebo vztahy mezi těmito strukturovanými údaji.

Metoda návrhu algoritmu zdola nahoru je obvykle od počátku vázána na konkrétní programovací jazyk (nebo alespoň na určité výrazové prostředky) a je pro ni charakteristický postup od jednoduchých příkazů, resp. jednoduchých řídicích struktur, přes složitější řídicí struktury až po celkové řešení. Je zřejmé, že celý tento postup se neobejde bez určitého minimálního počátečního nástinu (hrubého návrhu) řešení.

U obou metod návrhu algoritmu se tedy začíná analýzou problému, jejímž výsledkem je u metody návrhu algoritmu shora dolů přesný, ale jen hrubý počáteční návrh řešení, kdežto u metody návrhu algoritmu zdola nahoru je výsledkem počáteční analýzy jen přibližný, orientační hrubý návrh řešení. U metody návrhu algoritmu shora dolů se pak počáteční návrh dále rozpracovává (zjemňuje) na základě postupné stupňovité dekompozice řešeného problému, zatímco u metody návrhu algoritmu zdola nahoru slouží počáteční návrh jen k vymezení nebo přímo vytvoření dílčích algoritmů, z nichž se pak postupnou syntézou vytváří řešení celého problému.

Každá z uvedených metod návrhu algoritmu má svoje přednosti, jejichž uplatnění závisí nejen na složitosti a povaze řešeného problému a na charakteru používaných výrazových prostředků, ale rovněž i na schopnostech programátora.

Metoda návrhu algoritmu shora dolů je obvykle spojována s dalšími progresivními metodami, zejména se strukturovaným programováním, a je vhodná především pro řešení přesně definovaných problémů. Přednosti metody návrhu algoritmu zdola nahoru se uplatní zejména při řešení více příbuzných problémů (má pak větší smysl vytvářet dílčí algoritmy, které se použijí ve více řešeních) a dále též při řešení problémů, u nichž má programátor potíže s vytvořením přesného počátečního návrhu, přičemž tyto potíže mohou mít kromě subjektivních i objektivní příčiny, např. příliš volné zadání nebo neúplné zadání úlohy.

K ukázkám návrhu algoritmu oběma metodami bude použit následující příklad, jehož úplné řešení již není zcela jednoduché, a který je tedy vhodný pro systematický a metodický postup řešení.

Řešené úlohy

Příklad 1.5.1. Vstupní údaje tvoří přirozená čísla N, L, obě z intervalu <2, 9>, a dále v pořadí *po řádcích* prvky matice A typu (N,N). Sestavte vývojový diagram pro výpočet a tisk matic $A^1, A^2, \ldots, A^L$.

Řešení: Za výsledek počáteční analýzy úlohy je možno pro obě metody návrhu algoritmu považovat zjištění, že všechny matice jsou typu (N,N) a že je lze získat opakovaným násobením, tj.:

$$A^2 = A \cdot A ,$$
$$A^3 = A^2 \cdot A ,$$
$$\ldots$$
$$A^L = A^{L-1} \cdot A ,$$

přičemž nezáleží na tom, zda se počítá $A^i \cdot A$ nebo $A \cdot A^i$ (násobení matic sice není komutativní, uvedené tvrzení však vyplývá z platnosti asociativního zákona pro násobení matic).

První ukázka postupu řešení se bude týkat návrhu algoritmu metodou zdola nahoru. Po ukázce postupu při návrhu algoritmu metodou shora dolů by totiž ukázka postupu návrhu algoritmu metodou zdola nahoru, provedená pro stejnou úlohu, ztratila téměř smysl, neboť čtenář by již měl k dispozici přesný počáteční návrh algoritmu.

Metoda návrhu algoritmu zdola nahoru

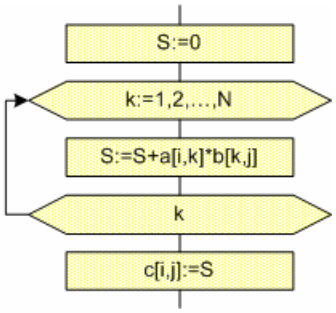
Při řešení úlohy bude zřejmě třeba provádět takové činnosti jako čtení matice, tisk matice a násobení matic. Je tedy možné si nejprve připravit příslušné dílčí algoritmy. Tato příprava je však spojena s rizikem, že některé části dílčího algoritmu budou zpočátku vytvořeny ve tvaru, který bude třeba později při vytváření celkového řešení částečně upravit nebo i podstatnějším způsobem změnit.

Při vytváření dílčích algoritmů je vhodné se nejprve soustředit na ty algoritmy, které se z hlediska řešeného problému jeví jako nejdůležitější. Pro řešení zadaného příkladu bude zřejmě klíčovým algoritmem algoritmus pro násobení matic.

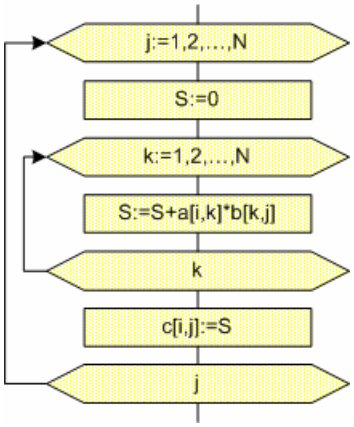
Např. při výpočtu matice $C = A \cdot B$ je prvek matice C na pozici (i,j), tj. prvek $c_{i,j}$, definován vztahem

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

což lze při výpočtu realizovat podle části vývojového diagramu:



Pomocí této části vývojového diagramu lze pak navrhnout větší část vývojového diagramu pro výpočet *i*-tého řádku matice *C*:



Uvedený dílčí algoritmus by pak mohl být využit při návrhu ještě větší části vývojového diagramu pro výpočet celé matice *C*, tj. části vývojového diagramu, která již představuje řešení uvažovaného dílčího algoritmu, tj. algoritmu pro násobení matic.

Obdobně by bylo možno navrhnout dílčí algoritmus pro čtení matice a dílčí algoritmus pro tisk matice. Při užití metody návrhu algoritmu zdola nahoru se tak postupně vytvářejí a současně i vyjadřují dílčí algoritmy, jejichž syntézou (po případné úpravě) lze postupně získat celé řešení zadané úlohy.

Metoda návrhu algoritmu shora dolů

Osnova (nejhrubšího) návrhu celého řešení:

1. čtení vstupních údajů,
2. výpočet a tisk požadovaných matic.

Dále pak následuje postupná analýza a rozpracování (zjemnění) jednotlivých bodů osnovy.

Návrh zjemnění 1. bodu osnovy:

1.1. čtení řídicích údajů N, L,

1.2. čtení matice A.

Analýza a zjemnění 2. bodu osnovy:

Při výpočtu a tisku matic A^1, A^2, ... , A^L se s výjimkou matice A^1 bude opakovaně provádět vždy výpočet a tisk další mocniny matice A. Pro vyjádření 2. bodu osnovy pomocí cyklu se známým počtem opakování (např. s parametrem k = 2, 3, ... , L) je třeba navrhnout takový způsob výpočtu a tisku jednotlivých mocnin matice A, aby výpočet a tisk jedné (další) mocniny matice A představoval tělo zamýšleného cyklu. K výpočtu a tisku požadovaných mocnin matice A není třeba mít k dispozici 10 polí ($L <= 10$), ale lze kromě pole A opakovaně využívat jen dvě další pole, např. B a C.

Předběžný návrh rozpracování 2. bodu osnovy:

2.1. tisk A,

2.2. $B := A \cdot A$, tj. $B = A^2$,

2.3. tisk B,

2.4. $C := B \cdot A$, tj. $C = A^3$,

2.5. tisk C,

2.6. $B := C \cdot A$, tj. $B = A^4$,

2.7. tisk B,

~~~

V tomto návrhu se opět vyskytují jen abstraktní příkazy. Např. příkaz *tisk A* představuje požadavek na tisk matice A, příkaz $B := A \cdot A$ zase požadavek na výpočet součinu matic $A \cdot A$ a jeho uložení do pole B, tj. jedná se o příkazy, k jejichž plnému vyjádření ve výsledném vývojovém diagramu bude potřeba více značek.
~~~

V předběžném návrhu rozpracování 2. bodu osnovy se již sice některé body opakují, ale opakovaná činnost se neváže k výpočtu a tisku jediné (další) mocniny matice A, ale k výpočtu a tisku vždy dvou následujících mocnin matice A. Tento nedostatek je možno odstranit např. tím, že jedno z polí B a C se bude používat výlučně pro matice, které mají být tištěny, a druhé jen pro výpočet dalších mocnin matice A. Tzn., že každá vypočtená matice se před tiskem zkopíruje do tiskového pole, např. do pole B. Při výpočtu další mocniny matice A se pak vždy bude počítat součin $B \cdot A$ a bude se ukládat do pole C. Na počátku naznačeného postupu je však třeba rovněž i zadanou matici A zkopírovat do pole B.

Opravený návrh rozpracování 2. bodu osnovy:

2.1. $B := A$,

2.2. tisk B, tj. tisk matice A^1,

2.3. $C := B \cdot A$, tj. $C = A^2$,

2.4. $B := C$,

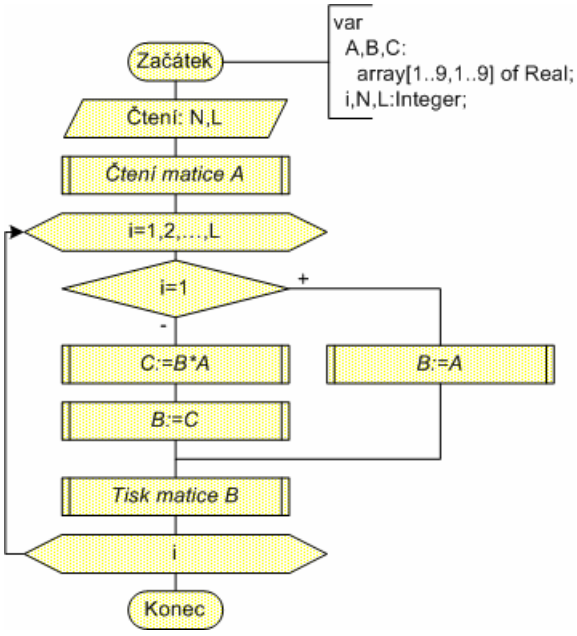
2.5. tisk B,

2.6. $C := B \cdot A$, tj. $C = A^3$,

2.7. $B := C$,

2.8. tisk B,

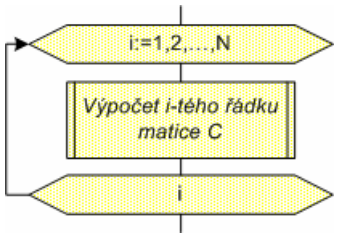
2.9. $C := B \cdot A$, tj. $C = A^4$,

~~~

V opraveném návrhu jsou již pro vyjádření výpočtu a tisku matic A^2, A^3, ... užity zcela stejné abstraktní příkazy, a proto lze tento návrh jednoduše vyjádřit pomocí cyklu. Pomocí rozhodování je možno do uvažovaného cyklu zahrnout i kopírování matice A do pole B, a tedy i tisk této matice.

K přehlednému a názornému vyjádření takto upraveného návrhu je již vhodné použít schematický vývojový diagram, tj. vývojový diagram, ve kterém ještě nejsou všechny činnosti zcela rozpracovány:


~~~

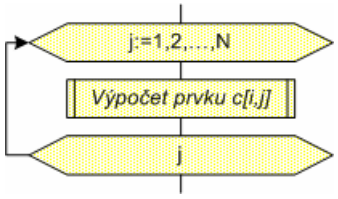
Dále by měl následovat rozbor všech dosud jen symbolicky vyjádřených činností, tj. rozbor abstraktních příkazů, který by vedl až k jejich detailnímu vyjádření pomocí konkrétních příkazů.

Na ukázku bude proveden jen počáteční rozbor a příslušné částečné vyjádření činnosti, která je ve schematickém vývojovém diagramu vyjádřena značkou , tj. výpočtu matice *C* jako součinu matic *B* a *A* .

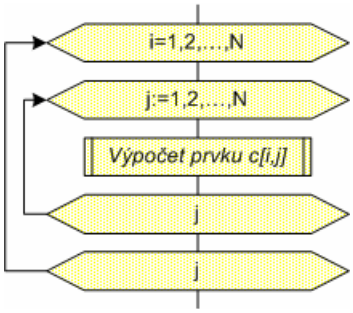
Výpočet matice *C* může být proveden po řádcích, tj. mohou se nejprve počítat prvky 1. řádku, potom prvky 2. řádku atd., což by bylo možno vyjádřit opět schematicky částí vývojového diagramu:



Výpočet prvků *i*-tého řádku matice *C*, tj. výpočet prvků c_{i1}, c_{i2}, ... , c_{iN}, je rovněž možno předepsat pomocí cyklu se známým počtem opakování. Abstraktní příkaz, který je v předchozí části vývojového diagramu označen jako *Výpočet i-tého řádku matice C*, lze tedy opět schematicky vyjádřit další částí vývojového diagramu:



Dosazením druhé části vývojového diagramu na příslušné místo v první části vývojového diagramu se získá rozpracovanější návrh výpočtu matice *C* :



Dále by bylo třeba analyzovat a rozpracovat činnost označenou jako *výpočet prvku* c_{ij}, což již v rámci ukázky návrhu algoritmu metodou shora dolů nebude prováděno, neboť tento problém byl již řešen při ukázce návrhu algoritmu metodou zdola nahoru. Při sestavování výsledného vývojového diagramu by pak mj. bylo ještě třeba doplnit příslušné deklarace a změnit identifikátor parametru cyklu ze schematického vývojového diagramu, např. místo *i* tu použít identifikátor *ii*, aby uvnitř cyklu s parametrem *i* nebyl opět cyklus s parametrem *i* (viz první a třetí dílčí část vývojového diagramu).

Poznámka

Celkové řešení př. 1.5.1. viz př. 2.11.1.

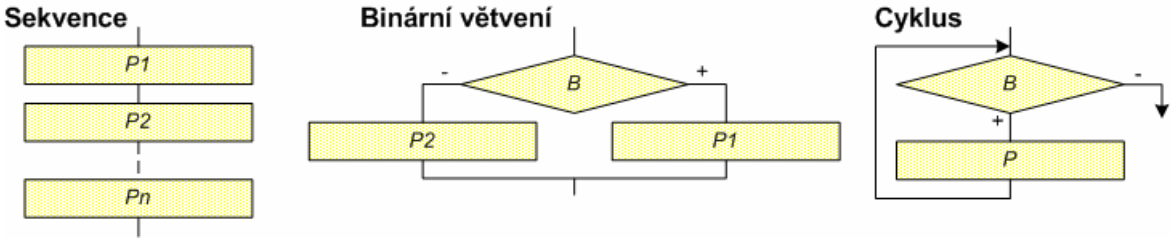
1.5.2. Strukturované programování

Výklad

Strukturované programování je metoda systematického a metodického přístupu k vytváření takové řídicí struktury algoritmu, která by byla jasným odrazem řešeného problému a zvolené metody řešení a která by byla tvořena jen z několika málo typů jednoduchých stavebních konstrukcí (stavebních obratů), zvaných ***základní řídicí struktury***, z nichž každá bude mít právě jeden vstup a právě jeden výstup.

Spojení strukturovaného programování s návrhem algoritmu metodou shora dolů je obecně považováno za progresivní a vysoce efektivní metodu programování. Vytváření algoritmů jen pomocí několika málo typů vhodných základních řídicích struktur rovněž zlepšuje (až na nepatrné výjimky) názornost a přehlednost zápisu algoritmů. Lepší orientace v zápisu algoritmu pak přispívá nejen k snadnějšímu pochopení algoritmu, ale může rovněž výrazně usnadnit úpravy i vlastní vytváření algoritmu.

Základní řídicí struktury (sekvence, binární větvení, cyklus) již byly uvedeny v části 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi*.



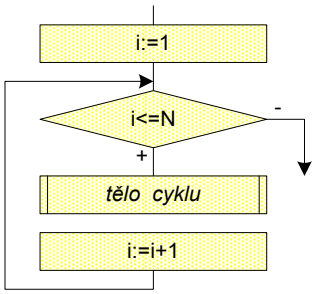
Každá ze základních řídicích struktur je vztažena alespoň na jeden ***operační blok***, (vyjádřený značkou zpracování), který však nemusí být jednoduchým operačním blokem, tj. nemusí představovat jen jeden jednoduchý příkaz (např. příkaz přiřazení), ale který může být opět tvořen libovolnou řídicí strukturou s jediným vstupem a jediným výstupem. V tomto smyslu je tedy možno chápat každou ze základních řídicích struktur, která je použita pro konkrétní operační blok, resp. bloky, rovněž jen jako jediný (strukturovaný) operační blok.

Kromě základní řídicí struktury cyklus (cyklus s testem před operačním blokem) byly v části 1.3.1. uvedeny další dvě řídicí struktury vyjádření cyklu (cyklus s testem po operačním bloku, cyklus se známým počtem opakování). Pomocí struktury cyklus s testem před

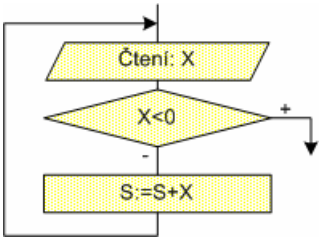
operačním blokem lze sice vyjádřit každý cyklus, ale k jednoduššímu a přirozenějšímu vyjádření některých cyklů jsou vhodnější i uvedené další řídicí struktury, které opět mají jediný vstup a jediný výstup. V části 1.3.1. jsou dále rovněž uvedeny příkazy Delphi, které přímo vyjadřují jednotlivé uvedené řídicí struktury.

Základním řídicím strukturám ve vyjádření algoritmu odpovídají příslušné základní typy strukturovaných operačních bloků. Nahrazováním jednoduchých operačních bloků v základních řídicích strukturách strukturovanými operačními bloky lze vytvářet libovolné řídicí struktury, které mají opět právě jeden vstup a právě jeden výstup. Přitom úroveň strukturování není omezena, tj. není omezen stupeň (hloubka) nahrazování (viz např. popis části vývojového diagramu v následujícím odstavci).

Je třeba zdůraznit, že *operační blok cyklu* obecně není totéž co *tělo cyklu*. Jako tělo cyklu se označuje dílčí algoritmus, jehož opakování je třeba nějakým způsobem zajistit. Ze způsobu tohoto zajištění, tj. z volby řídicí struktury cyklu, však mohou mj. vyplynout další činnosti, kromě vlastního testu na ukončení cyklu, které se pak budou s tělem cyklu opakovaně provádět, a které tedy budou s tělem cyklu tvořit operační blok cyklu. Např. uvedené vyjádření cyklu se známým počtem opakování pomocí cyklu s testem před operačním blokem představuje řídicí strukturu sekvence příkazu *i := 1* a zmíněného cyklu s testem před operačním blokem, přičemž tento operační blok je tvořen sekvencí těla cyklu a příkazu *i := i + 1*.



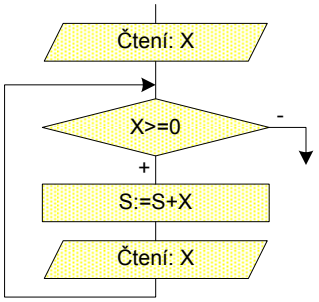
V prvním řešení příkladu 1.3.2. je použita pro vyjádření cyklu řídicí struktura, která neodpovídá žádné ze tří v této části uvedených řídicích struktur pro vyjádření cyklu. Zobrazená řídicí struktura má však opět právě jeden vstup a právě jeden výstup a lze ji poměrně jednoduše převést (transformovat) na řídicí strukturu vytvořenou ze základních řídicích struktur sekvence a cyklus s testem před operačním blokem.



Začátečníci často považují uvedený dílčí algoritmus za „logický“, neboť v těle cyklu se vždy zpracovává to číslo, které bylo rovněž přečteno v témže průchodu tělem cyklu. V druhém řešení téhož příkladu se nejprve, ještě před vstupem do cyklu, přečte „do zásoby“ první číslo a při jednotlivých průchodech tělem cyklu se pak vždy zpracovává číslo přečtené

před tímto průchodem a dále se opět do zásoby načítá číslo, které se bude zpracovávat až při případném dalším průchodu tělem cyklu.

Uvedený dílčí algoritmus z druhého řešení je sekvencí příkazu čtení a cyklu s testem před operační částí (která je sekvencí dvou jednoduchých příkazů). Zápis tohoto dílčího algoritmu v Delphi pak má tvar:



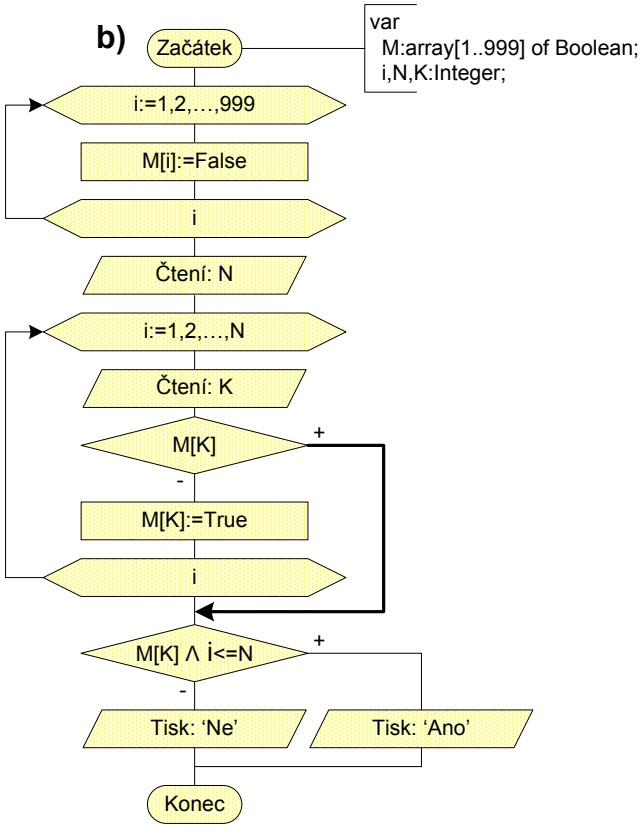

```
~~~
ReadLn(X);
while X>=0 do
begin
  S:=S+X;
  ReadLn(X);
end;
~~~
```

Analogii k uvedenému problému představují i dva návrhy prvního řešení př. 1.4.2. Vzhledem ke skutečnosti, že k řešení tohoto příkladu je nutné užít pole, bylo možno (viz druhé řešení) oddělit zpracování prvků posloupnosti od jejich načítání (vhodná zásada pro vytváření algoritmů). Místo jednoho cyklu se složitějším tělem jsou ve druhém řešení dva cykly s jednodušším tělem, což mj. činí zápis algoritmu přehlednější a pochopitelnější.

U prvního řešení př. 1.4.3.a i př. 1.4.3.b byly použity skoky z cyklů, a vznikla tak řídicí struktura, která má dva výstupy, což sice není v souladu se základní zásadou strukturovaného programování, ale zato tato řešení poměrně velmi názorně vyjadřují skutečnost, že připravený (známý) maximální počet opakování nemusí být pro konkrétní vstupní údaje vyčerpán a že cyklus může skončit dříve, tj. že může dojít k jeho předčasnému ukončení.

I v Delphi je sice (obecný) příkaz skoku, ale jeho používání obvykle odporuje uvedené základní zásadě strukturovaného programování. V souladu s touto zásadou jsou však některé specializované skoky (v Delphi jsou pro ně speciální příkazy) – např. ***skok na konec cyklu*** (*Break*), což je vlastně skok bezprostředně za konec cyklu.

Skok na konec cyklu bude demonstrován na dalším řešení př. 1.4.3.b (zjišťuje se, zda v posloupnosti celých kladných trojciferných čísel jsou alespoň dvě stejná). V řešení bude dále místo původně použitého celočíselného pole použito již dříve avizované booleovské pole.




```
program ShodaBB;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  M:array[1..999]of Boolean;
  I,N,K:Integer;
begin
  for I:=1 to 999 do
    M[I]:=False;
  ReadLn(N);
  for I:=1 to N do
  begin
    ReadLn(K);
    if M[K] then
      Break;
    M[K]:=True;
  end;
  if M[K] and (I<=N) then
    WriteLn('Ano')
  else
    WriteLn('Ne');
  ReadLn;
end.
```

1.5.3. Modulární programování

Výklad

Modulární programování je metoda vytváření programů, která je založena na myšlence rozložit řešený problém na pokud možno izolované podproblémy, jejichž jednotlivé řešení je podstatně dostupnější než souhrnné řešení celého problému. Řešení celého problému se pak vytváří z řešení dílčích problémů, přičemž tato řešení jsou obvykle zpracována ve formě modulů.

Modul je dílčí algoritmus zpracovaný jako samostatný program nebo jako určitým způsobem upravená, relativně uzavřená část programu, která je schopna samostatného zpracování a používání, tj. se kterou je možno nakládat jako s určitým celkem. Moduly tedy představují určité konkrétní stavební bloky, které mohou být využity nejen při řešení jednoho, ale i více problémů. Modulární programování se běžně používá při vytváření různých, zejména velkých programových systémů, např. operačních systémů počítače.

Modulární programování však nesouvisí jen s tvorbou programových systémů, ale jeho výhod lze též využít při vytváření jediného samostatného programu pro řešení většího nebo i jen běžného problému, pokud ovšem příslušný programovací jazyk obsahuje výrazové prostředky, které buď přímo umožňují v rámci jednoho programu vytvářet a užívat moduly, nebo pomocí kterých lze činnost vytváření a užívání modulů dostatečně napodobit.

Ve vyšších programovacích jazycích, tj. v nichž uživatel může vytvářet procedury (viz část 1.6. *Procedury*), umožňují právě procedury vhodnou formou zajistit vytváření a užívání modulů, které vystupují jen jako součást jednoho programu.

Základní myšlenku modulárního programování lze dobře realizovat zejména v těch programovacích jazycích, v nichž procedury představují relativně samostatné programové jednotky.

Např. v programovacím jazyku Fortran může být program tvořen více programovými jednotkami (*Pozor* – termín *jednotka* je tu užíván v jiném významu než v Delphi), z nichž právě jedna je hlavní a ostatní vedlejší. Kromě jediné výjimky (programová jednotka *block data*) jsou ostatní vedlejší programové jednotky formálně rovnocenné a každá z nich představuje vyjádření jediné procedury (obecné nebo funkční). Každá z programových

jednotek se zpracovává (překládá) nezávisle na ostatních programových jednotkách. V programovacím jazyku Fortran jsou tedy vlastně pojmy modul a programová jednotka totožné a každý program s procedurami, který se skládá z více programových jednotek, má modulární stavbu.

V Delphi lze činnost vytváření a užívání modulů lehce napodobit, jestliže procedury deklarované přímo na základní úrovni bloku programu nemají nelokální proměnné (viz část 2.16. *Procedury a funkce*). V Delphi je dále možno, na rozdíl např. od referenční verze jazyka Pascal, vytvářet program z více programových jednotek (viz část 2.18. *Jednotka (unit)*), a je tu tedy možno metodu modulárního programování plně realizovat.

Bez znalosti alespoň procedur (viz část 1.6. *Procedury* a část 2.16. *Procedury a funkce*) nemá téměř smysl uvádět příklad konkrétního zápisu programu s modulovou stavbou.

I začínající programátor však v jistém smyslu moduly využívá. Běžně např. používá zápisy některých matematických funkcí (např. *sin*, *ln*) nebo používá některé prostředky pro práci se soubory (např. *ChDir*, *Rename*) a vcelku ho ani nezajímá kdo a v jaké formě (zda ve formě programu či podprogramu) příslušný výpočet nebo příslušnou činnost naprogramoval.

1.6. Procedury

Cíle

Při programování se často v jedné nebo více úlohách opakují tytéž dílčí algoritmy. K tomu, aby nebylo třeba takové části algoritmů opakovaně programovat, slouží ve vyšších programovacích jazycích procedury.

Výklad

Procedura je dílčí algoritmus vyjádřený programovacím jazykem v takové formě, že tvoří relativně samostatnou algoritmickou jednotku, která má ***jméno*** a kterou lze velmi jednoduše vícenásobně používat v různých místech jednoho algoritmu nebo i v různých místech různých algoritmů. Požadavek na realizaci (na provedení) dílčího algoritmu, který je vyjádřen procedurou, se nazývá **volání procedury** a rovněž musí mít formu předepsanou příslušným programovacím jazykem.

Procedura může tvořit buď zcela uzavřenou algoritmickou jednotku, nebo může určitými formami komunikovat s okolím. Základní formou této komunikace je komunikace pomocí parametrů. Procedura, která žádným způsobem nekomunikuje s okolím, bude nazývána ***uzavřená procedura***. Podle toho, zda procedura má či nemá parametry, se označuje jako ***procedura s parametry*** nebo ***procedura bez parametrů***.

Při práci na počítači, který má tónový generátor, může být uzavřenou procedurou např. dílčí algoritmus pro realizaci nějaké melodie (znělky), která by mohla zaznít (být volána) na začátku a na konci provádění programu, celý text příslušného dílčího algoritmu by však byl v programu formou procedury uveden jen jednou. Pokud by však tato znělka měla být hrána např. na počátku provádění programu a na konci provádění programu různou rychlostí, bylo by třeba mít možnost předat tento požadavek do procedury např. prostřednictvím nějakého parametru. K realizaci zmíněné znělky by mohla být vytvořena procedura s parametrem, přičemž tímto parametrem by např. mohla být základní délka tónu vyjádřená v tisícinách sekundy a označená T. Při volání procedury by pak bylo třeba dodat konkrétní hodnotu (skutečnou hodnotu) základní délky tónu, která by nahradila pomyslnou délku T a pro kterou by tedy byl proveden algoritmus procedury.

Matematické funkce, které má programátor v daném jazyce k dispozici, mají obvykle rovněž formu procedury (přesněji funkční procedury). Chce-li programátor některou z těchto funkcí užít, musí se seznámit s jejím jménem (identifikátorem) a dále počtem a pořadím jejich

parametrů. Např. při výpočtu obecné mocniny (v Delphi funkce *Power* se dvěma parametry) není jedno, zda se zadá napřed báze a potom exponent, či naopak.

Volání procedury, jež informuje procesor o tom, kterou proceduru má realizovat, musí v případě procedury s parametry obsahovat nejen jméno procedury, ale i ***skutečné parametry***, tj. parametry, kterými mají být při realizaci algoritmu nahrazeny ***formální parametry*** užité při popisu procedury, tj. při vyjádření procedury (popis procedury = ***deklarace procedury***).

Deklarace procedury s parametry tedy pro procesor představuje jen návod k práci s nějakými fiktivními objekty (pomyslnými objekty), které jsou popsány pomocí formálních parametrů. Volání procedury s parametry pak pro procesor představuje nejprve požadavek na náhradu formálních parametrů parametry skutečnými, a dále pak požadavek na provedení algoritmu procedury pro tyto aktuální skutečné parametry.

Procesy deklarace a volání procedury je možno přirovnat k některým procesům ve školní výuce. Deklarace procedury odpovídá některými svými znaky procesu, při němž je žák učitelem seznamován s postupem řešení obecného problému, např. s postupem při násobení „nějakých“ dvou matic *A* a *B*. Volání procedury je pak srovnatelné s požadavkem učitele na to, aby žák použil postup, s nímž byl seznámen, pro určité konkrétní zadání, např. pro násobení určitých dvou matic *C* a *A*.

Podle formy (způsobu) vyjádření a užívání se obvykle rozlišují dva základní druhy procedur. Vzhledem k nejednotné terminologii budou v následujícím dělení za termíny obvyklými v algoritmizaci uvedeny v závorkách též termíny používané v Delphi.

Procedury (podprogramy) se dělí na:

- ***funkční procedury*** (funkce),
- ***obecné procedury*** (procedury).

Funkční procedura je procedura, která je i svou formou vyjádření speciálně určena k získání (k výpočtu) jediné hodnoty (obvykle jednoduché, tj. nikoliv strukturované).

Obecná procedura je procedura, jejíž forma je vhodná pro jakýkoliv (tj. obecný) algoritmus, tedy i algoritmus, jehož výsledkem může být i více hodnot (jednoduchých nebo strukturovaných). Obecná procedura však může sloužit též k získání jen jediné jednoduché hodnoty (stejně jako funkční procedura) nebo nemusí jako výsledek poskytovat žádné hodnoty (např. procedura k tisku nějakého obrázku).

Příkladem funkční procedury, kterou však uživatel programovacího jazyka nevytváří, ale jen užívá, je kterákoliv matematická jednoargumentová funkce, jež je v tomto jazyku definovaná. Funkce, které jsou v programovacím jazyku definovány, tj. které jsou uživateli v příslušném programovacím jazyku k disposici, se nazývají ***standardní funkce***. K standardním funkcím v mnoha programovacích jazycích obvykle patří některé goniometrické funkce (sinus, kosinus), logaritmické funkce, exponenciální funkce a další. V popisu prostředků programovacího jazyka musí být obsaženy nejen informace o tom, jaké jméno mají jednotlivé standardní funkce, tj. jakým identifikátorem jsou označeny, ale též informace o tom, jaké mají jednotlivé funkce parametry, tj. jaký je počet parametrů funkce, jaký je jejich význam a jak jsou uspořádány.

Elementární matematické funkce mají většinou jediný parametr. Volání funkce, které kromě jména funkce obsahuje skutečný parametr, tj. argument funkce, pak představuje požadavek na výpočet příslušné funkční hodnoty, tj. na provedení algoritmu příslušné procedury, jehož výsledkem je požadovaná hodnota. Z popisu prostředků programovacího jazyka se programátor např. dozví, že pro výpočet kosinu má k dispozici funkci *cos*(*x*), kde argument (parametr) *x* je v radiánech. Pak např. zápis *cos*(1.5) představuje volání uvedené funkce. Současně však tento zápis představuje (určuje) i místo, na které má být výsledek volání dosazen.

Programovací jazyk, v němž je možno vytvářet procedury, poskytuje uživateli mj. možnost užívat kromě standardních funkcí i další funkce, jejichž popis sám vytvoří ve formě funkční procedury.

V některých programovacích jazycích (též např. v Delphi) může být součástí deklarace procedury deklarace další, tj. vnitřní procedury (***vnořené procedury***), která je pak lokální vzhledem k proceduře, v níž je deklarována, tj. mimo proceduru, v níž je deklarována, ji nelze užít (příslušný identifikátor je však možno použít k jiným účelům). V těchto programovacích jazycích je pak možno vytvářet *programy s hierarchickou strukturou*.

Hierarchická struktura programu úzce souvisí s postupným návrhem programu metodou shora dolů, v němž se zpočátku používají abstraktní příkazy. Programovací jazyk, v němž je možno vytvářet procedury, poskytuje totiž programátorovi možnost používat v určité formě abstraktní příkazy, neboť volání procedury je vlastně příkaz k provedení dílčího algoritmu, který tato procedura vyjadřuje.

V mnohých programovacích jazycích s procedurami, tj. jazycích, ve kterých je možno nejen volat, ale i vytvářet (deklarovat) procedury, jsou však možnosti deklarace nebo

možnosti volání značně omezeny. Často např. není dovoleno *rekurzivní volání procedur*, tj. není dovoleno, aby procedura volala sama sebe, ať již přímo nebo prostřednictvím jiných procedur (o rekurzi viz blíže *Výklad* za řešením př. 2.16.3.).

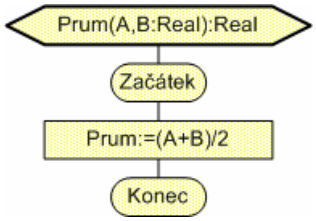
Vyjádření procedury musí mít formu předepsanou příslušným programovacím jazykem. Samotný jazyk vývojových diagramů však nemá pevná pravidla pro vyjadřování procedur, obvykle se do něj tato pravidla „přebírají“ z cílového programovacího jazyka (zde tedy z Delphi), takže pak nejsou problémy s příslušnými překlady (převody).

Dále proto bude zatím jen formou komentovaných příkladů a odkazů ukázáno, jak se deklarují a užívají jednoduché procedury, blíže viz část 2.16. *Procedury a funkce*.

Řešené úlohy

Příklad 1.6.1. Deklarujte funkci pro výpočet aritmetického průměru dvou reálných čísel.

Řešení: Má-li někdo (člověk nebo stroj) počítat aritmetický průměr dvou reálných čísel, je třeba mu tato dvě čísla dodat, odebírat se pak od něj bude jedno číslo. Příslušná funkce proto bude mít dva vstupní parametry, např. *A*, *B*. Aritmetický průměr z nějakých dvou reálných čísel *A*, *B* je *(A+B)/2*.




```
function Prum(A,B:Real):Real;
  begin
    Prum:=(A+B)/2;
  end;
```

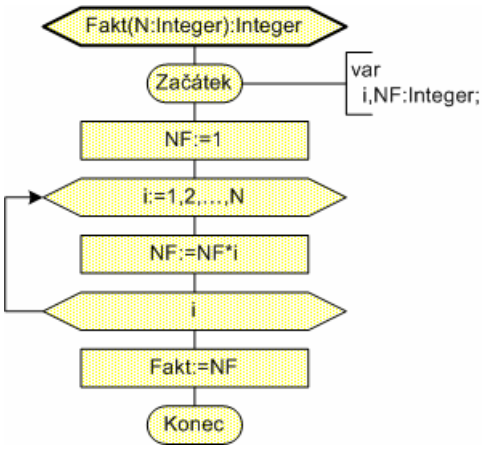
Vývojový diagram procedury má formu programu, před nímž je záhlaví procedury (značka příprava, ve které je formou převzatou z Delphi uveden identifikátor procedury, seznam formálních parametrů a u funkční procedury i typ výsledku).

V případě funkční procedury se výsledek nepředává prostřednictvím parametru, ale prostřednictvím názvu funkce. Proto je třeba zajistit, aby získaná hodnota byla přiřazena k identifikátoru funkční procedury (v uvedeném řešení viz příkaz *Prum:=(A+B)/2*).

Příklad 1.6.2. Deklarujte funkci pro výpočet *n!*.

Řešení: Dle definice *n!* představuje vícenásobný součin *1·2·…·n*, který lze realizovat postupným násobením v cyklu se známým počtem opakování, přičemž před vstupem

do cyklu je třeba zajistit, aby proměnná pro průběžné výsledky (zde proměnná *NF*) měla počáteční hodnotu 1.



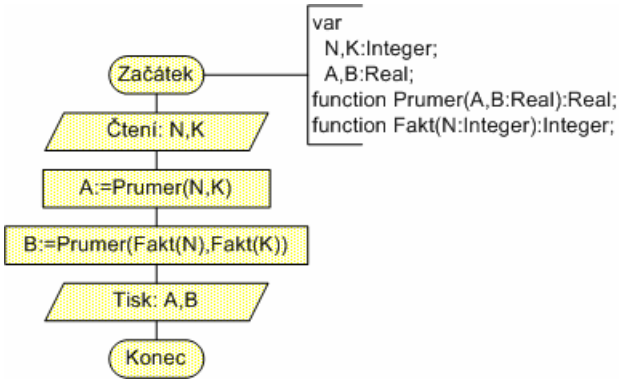

```
function Fakt(N:Integer):Integer;
  var
    I,NF:Integer;
  begin
    NF:=1;
    for I:=1 to N do
      NF:=NF*I;
    Fakt:=NF;
  end;
```

Při uvedeném výpočtu faktoriálu pomocí cyklu se v dílčím algoritmu kromě vstupního parametru *N* používají dále dvě proměnné (zde *I* a *NF*), které tu jsou deklarované lokálně (viz deklarace u značky *Začátek*). Výsledná hodnota je opět přiřazena k identifikátoru funkční procedury.

Příklad 1.6.3. Vytvořte program, který zajistí výpočet aritmetického průměru dvou zadaných přirozených čísel a dále výpočet aritmetického průměru jejich faktoriálů. K řešení využijte funkce z předchozích dvou příkladů.

Řešení: Jedná se o jednoduchý program, který má jen demonstrovat, jak se ve vývojovém diagramu funkce používají a jak se tu v deklarační části na ně odkazuje.

V deklarační části vývojového diagramu je vhodné, na rozdíl od deklarační části odpovídajícího programu v Delphi, uvádět procedury (obecné i funkční) jen pomocí jejich záhlaví, tj. formou, která se používá v interferenční části jednotek Delphi (viz část 2.18. *Jednotka (unit)*).



Ukázka byla dále zvolena tak, aby bylo zřejmé, že identifikátor formálního parametru a identifikátor příslušného skutečného parametru se mohou a nemusí shodovat.

Výklad

V uvedených procedurách zatím byly použity jen vstupní parametry, výstupní hodnota byla předávána prostřednictvím identifikátoru (funkční) procedury. U procedur (obecných i funkčních) se však i výstupní hodnoty mohou předávat formou parametrů. Z hlediska algoritmizace se pak parametry procedur dělí následujícím způsobem (blíže viz část 2.16. *Procedury a funkce*):

Rozdělení parametrů procedur z hlediska algoritmizace (tj. podle směru předávaných hodnot):

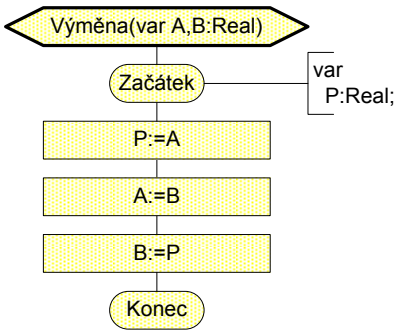
- ***parametry vstupní***,
- ***parametry výstupní***,
- ***parametry vstupně-výstupní***.

Při deklaraci procedury parametrům výstupním a vstupně-výstupním předchází v záhlaví procedury klíčové slovo **var** (viz následující příklad).

Řešené úlohy

Příklad 1.6.4. Deklarujte proceduru pro výměnu hodnot dvou paměťových míst typu *Real*.

Řešení: Algoritmus výměny viz př. 1.2.3., pomocná proměnná *P* musí být stejného typu jako parametry *A*, *B*. V proměnných *A*, *B* jsou nejprve dodány vstupní hodnoty, po provedení procedury jsou tu hodnoty výstupní, tj. parametry *A*, *B* jsou příkladem parametrů vstupně-výstupních.




```
procedure Vymena(var A,B:Real);
  var
    P:Real;
  begin
    P:=A;
    A:=B;
    B:=P;
  end;
```

2. DELPHI

Výklad

Delphi je vývojové prostředí (vývojový nástroj) firmy Borland určené pro snadné vytváření programových aplikací. Základem Delphi je programovací jazyk Object Pascal, což je moderní jazyk, který si zachoval dobrou čitelnost a elegantní zápis původního jazyka Pascal. Programovací jazyk Pascal navrhl počátkem sedmdesátých let prof. Niclaus Wirth. Autor sledoval návrhem jazyka dva cíle:

- vytvořit jazyk vhodný pro systematickou výuku programování, založený na omezeném počtu jednoduchých a srozumitelných konstrukcí,
- navrhnout strukturu jazyka tak, aby jej bylo snadné implementovat na většině tehdejších počítačů.

Na rozšíření a popularitě jazyka Pascal se značnou mírou podílely i některé jeho velmi zdařilé implementace. V souvislosti s vytvářením kompilátorů vznikly další verze jazyka Pascal. Pro počítače kompatibilní s PC IBM navrhla firma Borland jazyk Turbo-Pascal. V současné době již firma Borland jako nadstavbu jazyka Turbo-Pascal rozšiřuje několikátou verzi vývojového prostředí Delphi.

Jednotlivé verze Delphi poskytují takovou šíři prostředků, že jsou vhodné jak pro soustavnou profesionální práci tak i pro výuku základů programování (i u naprostých začátečníků se však předpokládá jistá počítačová gramotnost, konkrétně alespoň znalost práce s operačním systémem a vhodným editorem). V Delphi totiž začátečník může „odhodit“ značnou část prostředků vývojového prostředí, a pracuje pak v prostředí, které pro programování algoritmů v mnoha směrech předčí dřívější poměrně zdařilé integrované prostředí jazyka Turbo-Pascal (zkratka TP). Vzhledem k úzké návaznosti Delphi na TP bude v dalším textu na některé odchylky Delphi od TP upozorňováno.

2.1. Stavba programu

Části programu:

1. ***záhlaví***,
2. ***deklarační část***,
3. ***příkazová část***.

Záhlaví programu začíná klíčovým slovem **program** a obsahuje jméno programu, kterým je identifikátor.

Deklarační část má předepsaný tvar a definují se v ní významy identifikátorů nedefinovaných jazykem, které se v programu používají.

Např. při deklaraci proměnné se uvádí nejen její jméno, ale i typ, přičemž typ proměnné blíže vymezuje její vlastnosti, tj. množinu přípustných hodnot a množinu přípustných operací. Při deklaraci podprogramu se zase uvádí nejen identifikátor podprogramu, ale celé záhlaví podprogramu i tělo podprogramu.

Složení deklarační části (v závorce klíčová slova pro příslušný úsek)**:**

0. úsek seznamu jednotek (**uses**),
1. úsek deklarací návěští (**label**),
2. úsek deklarací konstant (**const**),
3. úsek deklarací typů (**type**),
4. úsek deklarací proměnných (**var**),
5. úsek deklarací procedur a funkcí (**procedure**, **function**).

Poznámka

V Delphi nemusí být pořadí úseků 1 až 5 dodrženo, stačí jen, je-li zajištěna logická návaznost.

Příkazová část je vždy tvořena ***složeným příkazem***, za kterým následuje tečka.

Složený příkaz je tvořen posloupností příkazů, která je uzavřena do příkazových závorek **begin** a **end**, přičemž jednotlivé příkazy této posloupnosti se oddělují středníkem (blíže viz část 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi* a část 2.8. *Složený příkaz*).

Složený příkaz patří mezi ***strukturované příkazy***, což jsou příkazy, jejichž částí je opět nějaký příkaz (pro něj se pak užívá termín ***vnořený příkaz***).

Složený příkaz vytváří z vnořené posloupnosti příkazů (ať již víceprvkové, jednoprvkové nebo i prázdné) jediný příkaz, což je možno využít též při zápisu dalších strukturovaných příkazů, neboť vnořeným příkazem každého strukturovaného příkazu vždy může být příkaz složený (v některých konkrétních případech však použití příkazových závorek **begin** a **end** není nutné).

Rozdělení příkazů

Příkaz:

- jednoduchý příkaz (žádnou jeho dílčí částí již není příkaz):
  - přiřazovací příkaz,
  - příkaz procedury (příkazem procedury je i příkaz vstupu, resp. výstupu),
  - příkaz skoku,
  - prázdný příkaz,
- strukturovaný příkaz:
  - složený příkaz,
  - příkaz podmínkový:
    - příkaz **if** úplný,
    - příkaz **if** neúplný,
    - příkaz **case**,
  - příkaz cyklu:
    - příkaz **while**,
    - příkaz **repeat**,
    - příkaz **for**,
  - příkaz **with**.

Strukturované příkazy popisují postupné, podmíněné, opakované nebo vázané provádění vnořených dílčích příkazů. Tvary a významy některých strukturovaných příkazů byly již uvedeny v 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi*.

2.2. Abeceda a lexikální jednotky

Výklad

Všechny konstrukce jazyka jsou tvořeny z přípustných symbolů, které tvoří abecedu jazyka. V jazyku Pascal (základu Delphi) a v některých jiných programovacích jazycích však pojem symbol není totožný s pojmem znak. Kromě jednoznakových symbolů (písmena, číslice, některé speciální symboly) a dvojznakových symbolů (např. <=) tu existují i víceznakové symboly, které jsou tvořeny vyhrazenými slovy. ***Vyhrazená slova***, též ***klíčová slova***, jsou v textu a v programech psána tučně (některá vyhrazená „slova“ jsou ve skutečnosti složeninami ze dvou slov, např. **downto** = *down to*).

Ze symbolů abecedy se vytvářejí vyšší jednotky jazyka, a to jak na úrovni lexikální (lexikon = slovník), tak i syntaktické.

Abeceda (jazyka Pascal):

- písmena latinské abecedy (nerozlišují se malá a velká písmena, doporučuje se však ustálené používání velkých a malých písmen, které značně přispívá k čitelnosti programu),
- číslice desítkové soustavy,
- speciální symboly (jejich tvar a význam je definován jazykem, používají se jako operátory, oddělovače či omezovače různých konstrukcí):
  - jednoznakové (např. + | - | ;),
  - dvojznakové (např. := | <=),
  - klíčová slova (např. **begin** | **if** | **else** | **for**).

Lexikální jednotky:

- speciální symboly,
- identifikátory,
- čísla,
- návěští,
- znakové řetězce,
- komentáře.

Oddělovače lexikálních jednotek:

- mezera,
- oddělovač řádků,
- komentář (současně se jedná i o lexikální jednotku).

Program v Delphi je posloupností lexikálních jednotek. Mezi některými lexikálními jednotkami musí být alespoň jeden oddělovač. Pokud by se totiž posloupnost některých dvou sousedních lexikálních jednotek jevila opět jako lexikální jednotka, je třeba zmíněné dvě jednotky oddělit.

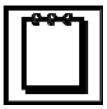
Jedním z oddělovačů lexikálních jednotek je mezera. Mezeru je možno použít i mezi jednotkami, které oddělovač nepotřebují, nebo mezi jednotkami, mezi kterými se již příslušný oddělovač nachází, obvykle se tak zlepšuje čitelnost programu.

Identifikátor - posloupnost písmen a číslic, která začíná písmenem a která se neshoduje s klíčovým slovem. Identifikátory se používají jako jména proměnných, konstant, typů, procedur apod. V Delphi může být „jako písmeno“ použito i podtržítko, tj. znak _ .

Významy některých identifikátorů, např.

Real, *Write*, *Sqrt*,

jsou definovány jazykem (nazývají se ***standardní identifikátory***). Všechny ostatní identifikátory, s výjimkou identifikátoru v deklaraci typu typového ukazatele (viz část 2.17. *Dynamická deklarace, typ ukazatel*), musí být nejprve definovány a teprve potom mohou být odpovídajícím způsobem užity.



Poznámka

S výjimkou výpisu programů a jejich částí, tj. v běžném textu, jsou identifikátory psány *nepodtrženou kurzívou* (*podtržená kurzíva* je vyhrazena pro zápis symbolů, **tučné písmo** se používá pro klíčová slova a pro zvýraznění některých částí textu).

Číslo může být v programu zapsáno jako:

- číslo celé (např. 27 | +027 | -11),
- číslo reálné:
  - v desetinném tvaru (např. 0.91 | +00.9100 | 27.0),
  - v semilogaritmickém tvaru (např. +0.49E+03 | 0.49E3).

Např. zápis 27.0 není zápisem celého čísla 27, ale je zápisem příslušného reálného čísla v desetinném tvaru.

Návěští - posloupnost číslic v rozsahu od 0 do 9999, která může být použita k předznačení příkazů pro potřebu příkazů skoku.

Řetězec - posloupnost přípustných znaků, která je uzavřena v apostrofech. Zápis apostrofu do řetězce lze zajistit zápisem obrazu apostrofu, což je v Delphi zdvojený apostrof.

Příklady řetězců:

```
'Datum:␣␣'
'X''␣=␣'
```

Komentář - posloupnost znaků kromě znaku *}*, která je uzavřena ve složených závorkách, tj. mezi znaky *{ }*, a která se obvykle používá pro zápis různých vysvětlivek přímo do programu. Za komentář se v Delphi též považuje vše, co se nachází v řádku od dvojice lomítek *//* do konce tohoto řádku. V Delphi je možno dále pro uzavření komentáře místo složených závorek použít dvojznakové symboly *(* *)* .

Dvojznakové závorky *(* *)* lze využít např. při ladění programu, neboť je pak možno i větší část programu s běžnými komentáři označit jako jeden „větší“ komentář a vyřadit ji tak přechodně z překladu. Např.

```
  Read(N);        {čtení počtu prvků}
(*
  if N>5 then
    N:=5;         {omezení N shora}
*)
  ~~~
```

Pozor! Začíná-li posloupnost znaků komentáře znakem ***$***, nejde o pouhý komentář, ale je to ***direktiva překladače*** (*direktiva kompilátoru*). Pomocí direktiv překladače (viz 2.19.) lze ovlivňovat některé vlastnosti překladače. Např. direktiva *{$APPTYPE CONSOLE}* informuje překladač, že vytvářený program není běžnou aplikací, ale jen aplikací pro konzolu.

2.3. Datové typy

Výklad

Z hlediska efektivního využívání paměti počítače i strojového času a z hlediska přesnosti výsledků aritmetických operací je výhodné ukládat a zpracovávat celá čísla jinak než čísla reálná. Mnohé programovací jazyky proto např. umožňují předepsat skutečnost, že konkrétní číselná proměnná bude nabývat jen celočíselných hodnot, tj. že bude celočíselného typu. V Delphi má uživatel kromě základního celočíselného typu (typ *Integer*) a základního reálného typu (typ *Real*) ještě další standardní typy, a mimo to též má možnost definovat další typy.

Každý datový typ specifikuje nejen určitou množinu, jejíž prvky se nazývají hodnotami daného typu, ale i množinu operací nad těmito hodnotami. Jednotlivé datové typy se označují (obdobně jako proměnné) identifikátorem.

Např. místo deklarace

```
var
  M,N:Integer;
  K,L:1..100;
```

která představuje přímou deklaraci proměnných *M*, *N* standardního celočíselného typu *Integer* a celočíselných proměnných *K*, *L*, jež mohou nabývat jen hodnot od 1 do 100, je možno nejprve deklarovat celočíselný typ vázaný na příslušný interval (označen bude např. identifikátorem *TypSto*), a teprve potom pomocí identifikátoru typu deklarovat proměnné *K*, *L*, tj.

```
type
  TypSto=1..100;
var
  M,N:Integer;
  K,L:TypSto;
```

Aby se identifikátory typů zřetelněji odlišovaly od jiných identifikátorů (není to však nutné), budou v dalším textu identifikátory uživatelem definovaných typů začínat buď trojicí písmen *Typ*, nebo alespoň jen písmenem *T*.

Dělení datových typů lze obecně provádět z různých hledisek, např.:

1) Datové typy:

- **standardní**, tj. definované jazykem, např. *Integer*, *Real*, *Boolean*,
- **definované uživatelem**, např. v poslední ukázce *TypSto*.

2) Datové typy:

- **jednoduché**,
- **strukturované**,
- **ukazatelové**.

Jednoduchý typ je typ, který se z hlediska operací použitelných v programu jeví jako dále nedělitelný.

Strukturovaný typ (např. typ pole) specifikuje množinu strukturovaných hodnot, které se skládají z dílčích složek. Pomocí strukturovaných typů lze v programu deklarovat strukturované proměnné nebo konstanty či další strukturované typy.

Typ ukazatel umožňuje práci s adresami. Pomocí adres se např. udává, kde v paměti počítače začíná uložení jednotlivých proměnných (blíže viz část 2.17. *Dynamická deklarace, typ ukazatel (typ pointer)*).

Příklad: Mají-li např. *X*, *Y* představovat body v třírozměrném prostoru, jejichž souřadnice se mohou v průběhu programu měnit, lze příslušná proměnná třísložková pole deklarovat přímo zápisem

```
var
  X,Y:array[1..3] of Real;
```

nebo nejprve deklarovat příslušný typ (bude označen identifikátorem *TypBod*), tj. psát

```
type
  TypBod=array[1..3] of Real;
var
  X,Y:TypBod;
```

Rozdělení typů

Datový typ:

- **jednoduchý**:
  - **reálný** (typ *Real*, kromě základního typu *Real* v Delphi též např. typ *Extended*),
  - **ordinální**:
    - **celočíselný** (typ *Integer*, v Delphi též např. typ *Byte*) a příslušné intervaly,
    - logický (typ *Boolean*) a příslušné intervaly,
    - znakový (typ *Char*) a příslušné intervaly,
    - výčtový a příslušné intervaly,
- **strukturovaný**:
  - **pole** (klíčové slovo **array**),
  - **záznam** (klíčové slovo **record**),
  - **množina** (klíčové slovo **set**),
  - **soubor** (klíčové slovo **file** nebo identifikátor *TextFile*),
  - **řetězec** (klíčové slovo **string**),
- **ukazatel**.

Standardní funkce definované pro všechny ordinální typy:

Pro každý ordinální typ (tj. jednoduchý typ kromě typu *Real*), jehož množinou hodnot je $\{t_0, t_1, ..., t_{n-1}\}$, kde $t_0<t_1<...<t_{n-1}$, jsou definovány funkce:

Succ následník, tj. *Succ*(t_i) = t_{i+1} ,

Před předchůdce, tj. *Pred*(t_i) = t_{i-1} ,

Ord pořadové číslo, tj. *Ord* (t_i) = i .

Např. v ASCII (kód používaný mj. na PC IBM), v němž jsou velká písmena seřazena podle abecedy a znak *A* má kód 65, tedy platí

Succ('B') = 'C',

Pred('B') = 'A',

Ord ('B') = 66 .

Základní standardní typy

Pro všechny standardní typy jsou definovány všechny ***relace uspořádání***, tj.

< , <= , = , <> , >= , > .

Operace, které jsou definovány pro jednotlivé typy, budou v následujícím přehledu uváděny jen pomocí příslušných operátorů.

Pro zápis množin hodnot jednotlivých standardních typů bude používána běžná matematická symbolika, tj.

{ } množina zadaná výčtem prvků,

< > uzavřený interval na množině reálných nebo jen celých čísel.

Typ *Boolean*

- množina hodnot = {*False*, *True*}, přičemž *False* < *True*, tj. jedná se o typ definovaný nad množinou pravdivostních hodnot *False* (nepravda = logická 0) a *True* (pravda = logická 1),
- relace uspořádání: všechny,
- operace: **and** , **or** , **not** .

Typ *Integer*

- množina hodnot = interval <- *MaxInt* -1, *MaxInt* > na množině celých čísel, kde *MaxInt* je standardní identifikátor maximálního celého čísla typu *Integer* zobrazitelného na konkrétním počítači, v Delphi *MaxInt* = 2147483647, v TP *MaxInt* = 32767 (v Delpi dále celočíselné typy *Cardinal*, *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word*, *Longword*),
- relace uspořádání: všechny,
- operace: + , - , * , **div** , **mod** , / ,

  kde operátor **div** poskytuje výsledek po celočíselném dělení, operátor **mod** zbytek po celočíselném dělení a operátor / poskytuje výsledek obyčejného dělení (tj. výsledek typu *Real*), např.

```
14 div 4 = 3
14 mod 4 = 2
14 / 4   = 2.5
```

- příklady dalších standardních funkcí (tj. kromě *Succ*, *Pred*, *Ord*):

  Abs *Abs*(x) = |x| ,

  Sqr *Sqr*(x) = x^2 ,

  Odd *Odd*(x)= *True* pro *x* liché a *False* pro *x* sudé.

Typ *Real*

- množina hodnot = podmnožina množiny reálných (racionálních) čísel (v Delphi též typy *Real48*, *Single*, *Double*, *Extended*, *Comp*, *Currency*),
- relace uspořádání: všechny,
- operace: + , - , * , / ,
- příklady standardních funkcí:

  Abs , *Sqr* , *Sqrt* , *Ln* , *Exp* ,

  Sin , *Cos* - argument v radiánech,

  Arctan - výsledek v radiánech,

  Trunc *Trunc(x)* = celá část hodnoty *x*,

  Round *Round(x)* = *Trunc(x+0.5)* pro $x \geq 0$ a *Trunc(x-0.5)* pro $x < 0$,

  Frac *Frac (x)* = desetinná část hodnoty *x*,

  Pi - výsledkem Ludolfovo číslo,

  Random - výsledkem pseudonáhodné číslo (blíže viz poznámka u př. 2.10.1.).

Typ *Char*

- množina hodnot = množina všech znaků vnější abecedy (patří sem všechny znaky ASCII), hodnotu typu *Char* lze zapsat pomocí příslušného jednoznakového řetězce,
- relace uspořádání: všechny (např. zápis *'A' < 'B'* lze číst: „*znak A předchází před znakem B*“, relace se využívají např. k řazení podle abecedy),
- příklady dalších standardních funkcí (tj. kromě *Succ*, *Pred*, *Ord*):

  Chr *Chr(i)* = znak, jehož pořadové číslo je *i*
  (v ASCII např. *Chr*(66) = 'B'), tedy

```
Chr(Ord(x))  = x ,
Ord(Chr(i))  = i .
```

Typ *string* (typ řetězec)

V Delphi je možno pracovat s řetězcovými konstantami i řetězcovými proměnnými dvou typů, první typ je převzatý z jazyka Turbo-Pascal (blíže viz část 2.15. *Typ řetězec (string)*), druhý z jazyka C. Pro první typ, pro který je možno v Delphi používat též identifikátor *ShortString*, je charakteristická pevná maximální délka jeho uložení v paměti počítače (implicitně a současně maximálně 255 znaků), druhý typ (označovaný též identifikátorem *AnsiString*) má „téměř" neomezenou délku (používá se pro něj termín *dlouhé řetězce*).

Typ string převzatý z jazyka Turbo-Pascal (v Delphi je označován též identifikátorem *ShortString*):

- množina hodnot = množina všech posloupností 0 až 255 znaků vnější abecedy,
- relace uspořádání: všechny (relace se využívají např. k řazení podle abecedy),
- operace: + (nazývá se zřetězení),
- standardní funkce: budou uvedeny až v části 2.15. *Typ řetězec (string)*.

Příklad:

```
var
  S,T:string;
begin
  S:='aha';
  T:='Pr'+S+' !';
  WriteLn(T);
  ~~~
```

V uvedené části programu se hodnota řetězcové proměnné *T* vytvoří („poskládá") z řetězců *'Pr'*, *'aha'* a *' !'*, a program tedy zajistí tisk řádku s textem: *Praha !* .

Jednoduché typy definované uživatelem:

- výčtové typy,
- typy definované pomocí intervalu.

Výčtový typ (vyjmenovaný typ)

- zadává se výčtem (vyjmenováním) hodnot typu,
- má obecný tvar

type *jméno typu* = (h_0, h_1, ~~~, h_n)

kde *jméno typu* je identifikátor zaváděného typu,

h_0, h_1, ~~~, h_n jsou identifikátory hodnot zaváděného typu, přičemž $h_0 < h_1 < ~~~ < h_n$.

Typ definovaný pomocí intervalu

- má obecný tvar:

type *jméno typu* = *min* .. *max*

kde *jméno typu* je identifikátor zaváděného typu,

min, *max* jsou hodnoty již definovaného ordinálního typu, který se vzhledem k zaváděnému typu nazývá ***typ spojený s definovaným typem*** (též ***hostitelský typ***).

Příklad: Týdenní rozvrh vyučování se často zobrazuje jako tabulka (matice), v níž pozice (3,2) obvykle odpovídá 2. hodině ve středu, při jiném zobrazení by však tato pozice mohla odpovídat 3. hodině v úterý. V Delphi je možno předejít podobným nejasnostem nahrazením číselných indexů indexy vhodného typu.

Viz např. následující ukázka:

```
program Rozvrh;
  ~~~
type
  TDen     =(Po,Ut,St,Ct,Pa,So,Ne);
  TPracDen = Po..Pa;
  THodina  = 1..8;
var
  Den:TDen;
  ~~~
begin
  for Den:=Po to Ne do ~~~
  ~~~
```

Poznámka

Výčtový typ patří mezi ordinální typy a lze tedy pro něj užívat funkce *Succ*, *Pred* a *Ord*, přičemž výsledkem funkce *Ord* je pořadové číslo příslušné hodnoty z deklaračního výčtu, přičemž se začíná od nuly (viz indexy v obecném tvaru výčtového typu).

Např. vzhledem k deklaraci typu *TDen* z předchozí ukázky:

```
Succ(Ut) = St
Pred(Ut) = Po
Ord (Po) = 0
```

2.4. Konstanty, proměnné, výrazy

Výklad

Konstanta může být v programu zapsaná dvěma způsoby, a to jako:

- **literál**, tj. přímý zápis hodnoty v programu,
- **pojmenovaná konstanta**, tj. pomocí identifikátoru, který je pro příslušnou hodnotu použit v úseku deklarací konstant (úsek deklarací konstant začíná klíčovým slovem **const**).

Příklad: Následují výňatky z programu, v němž se provádí převody teplot vyjádřených v kelvinech a ve stupních Celsia a který se dále týká analytické geometrie v rovině, v níž jsou body zadávány dvěma souřadnicemi.

```
~~~
const
  T0=273.15;
  Dimenze=2;
var
  TK,TC,V,S:Real;
  A,B,C:array[1..Dimenze] of Real;
  I:Integer;
begin
  ~~~
  TK:=T0+TC;
  ~~~
  S:=0;  {začátek výpočtu vzdálenosti bodu od počátku}
  for I:=1 to Dimenze do
    S:=S+Sqr(A[I]);
  V:=Sqrt(S);
  ~~~
```

Výklad

Na rozdíl od proměnné hodnotu pojmenované konstanty nelze změnit v příkazové části programu. Používání pojmenovaných konstant místo literálů zvyšuje modifikovatelnost programu a obvykle i srozumitelnost programu (v mnemotechnice identifikátoru konstanty se může projevit i význam konstanty). Pokud by se např. v předchozí ukázce mělo „přejít“ z roviny do prostoru, stačí provést jen příslušnou změnu v deklaracích (tj. *Dimense = 3*).

Proměnná (viz 1.1.) je datový objekt, který může během realizace programu nabývat různých hodnot, ale jen jednoho typu, a to toho, který je předepsán deklarací této proměnné.

Výraz je předpis pro získání hodnoty, který je tvořen z operandů, operátorů a kulatých závorek. Nestanoví-li závorky jinak, je výraz vyhodnocován podle následujících priorit.

Priority:

1. výraz v závorce, volání funkce,
2. not ,
3. * , / , div , mod , and ,
4. + , - , or ,
5. < , <= , = , <> , >= , > .

Pozor! Priorita některých operátorů je jiná než v matematice a než např. v programovacím jazyku Basic.

V Delphi, na rozdíl např. od jazyka Basic, není definován aritmetický operátor pro umocňování. Při zápisu mocnin s malým přirozeným exponentem je možno využít funkce *Sqr* a opakovaného násobení (viz následující ukázky).

Pro zápis jiných mocnin, tj. i pro zápis mocnin s reálným exponentem, a tedy též pro zápis odmocnin lze funkci využít *Power* nebo matematický vztah

$$A^B = e^{B*\ln A} \quad \text{(pro } A > 0\text{)},$$

tzn. případ A=0 je třeba v programu zvlášť ošetřit (např. pomocí příkazu **if**).

Funkce *Power* je z jednotky *Math* a má dva reálné argumenty *Base* a *Exponent*. Má-li být v programu užita funkce *Power* , je třeba uvést identifikátor *Math* v seznamu jednotek programu, tj. v seznamu za klíčovým slovem **uses**.

Příklad:

matematický zápis	zápis v Delphi	číslo ukázky
$\frac{a \cdot b}{c \cdot d}$	`((A)*(B))/((C)*(D))`	1
	`(A*B)/(C*D)`	2
	`A*B/(C*D)`	3
	`A*B/C/D`	4
$\frac{a \cdot b}{c} \cdot d$	`A*B/C*D`	5
$\sin^4\left(\frac{1+x}{1+e^{-x}}\right)$	`sqr(sqr(sin((1+X)/(1+exp(-X)))))`	6
$3^{2 \cdot n-1}$	`exp((2*N-1)*ln(3))`	7
	`Power(3, 2*N-1)`	8
$x \in <2, 3 \cdot y)$	`(X>=2) and (X<3*Y)`	9

V ukázkách 1 až 4 jsou zápisy stejného matematického výrazu. V ukázce 1 je výraz přepsán tak, že každý operand je uzavřen do závorek. V dalších dvou ukázkách je pak počet závorek postupně snižován. Zápis v ukázce 4 je již sice bez závorek, ale na rozdíl od předchozích ukázek tu musel být změněn jeden aritmetický operátor. Za jednoduché, názorné a dostatečně blízké běžnému matematickému zápisu je možno považovat zápisy z ukázek 2 a 3. V případě, že by v některém ze zápisů z ukázek 1 až 3 byly vypuštěny všechny závorky, vznikl by výraz s jiným významem (viz ukázka 5).

V ukázce 9, na rozdíl od odpovídajícího matematického zápisu, jsou použité závorky vzhledem k uvedeným prioritám nezbytné!

2.5. Přiřazovací příkaz

Předpokládané znalosti

Z hlediska algoritmizace byl přiřazovací příkaz zaveden již v části 1.1. *Algoritmus*, v této části budou uvedeny jen příslušné konkretizace pro Delphi.

Výklad

Obecný tvar přiřazovacího příkazu:

proměnná := *výraz*

kde

proměnná je označení proměnné, jíž se má přiřadit hodnota,

výraz je výraz, jehož hodnota je ***kompatibilní vzhledem k přiřazení*** s proměnnou *proměnná*.

Význam přiřazovacího příkazu, zvaného též *příkaz přiřazení*: Nejprve se vypočte hodnota výrazu napravo od přiřazovacího znaku, a tato hodnota se potom (je-li to možné) přiřadí proměnné, jejíž označení je uvedeno vlevo od přiřazovacího znaku.

Kompatibilita vzhledem k přiřazení: V přiřazovacím příkazu buď hodnota výrazu *výraz* musí patřit do množiny specifikované typem proměnné *proměnná*, nebo *výraz* je celočíselný a *proměnná* je reálná.

Příklad: Je-li *I* proměnná typu *Integer* a *X* proměnná typu *Real*, pak nejsou přípustné např. příkazy:

```
I:=I+1.8;
I:=X;
```

Některé programovací jazyky však obdobné příkazy připouštějí. Pro tuto skutečnost se používá obvykle termín ***ztráta přesnosti***, neboť pak implicitně dochází k zaokrouhlování. V Delphi musí být v uvedených případech zaokrouhlování explicitně předepsáno např. pomocí standardních funkcí *Tunc* nebo *Round*, tj. musí být předepsáno jak se má zaokrouhlovat (zda se má např. odříznout desetinná část výsledku, či zda se má zaokrouhlovat běžným matematickým způsobem).

2.6. Příkazy vstupu a výstupu - základní informace

Výklad

Nositeli vstupních a výstupních údajů jsou soubory dat. V Delphi se každý soubor dat (též *datový soubor*) chápe jako posloupnost složek stejného typu, pro něž jsou definovány určité operace, které umožňují sekvenční zpracování souboru (tj. jeho postupné čtení či zápis). Speciálním případem souborů jsou soubory textové (blíže viz část 2.14. *Typ soubor (file)*).

Textový soubor je soubor, jehož prvky jsou znaky vnější abecedy (např. znaky ASCII) a který je členěn na řádky. Řádky jsou odděleny znakem ***eoln*** (*end of lin*e), přičemž délka řádku není pevná, tj. jednotlivé řádky mohou obsahovat různý počet znaků.

Standardní textové soubory: V Delphi jsou k dispozici dva standardní textové soubory:

- soubor ***Input*** pro vstup dat (obvykle z klávesnice),
- soubor ***Output*** pro výstup dat (obvykle na obrazovku).

Výstup do souboru *Output*

Výstup do souboru *Output*, tj. obvykle výstup na obrazovku, se provádí pomocí procedury *Write*, resp. *WriteLn* (zkratka ze slov *write line*).

Obecný tvar základního příkazu výstupu do souboru *Output* je

```
Write(parametr)
```

kde *parametr* musí mít jeden z tvarů:

vystupující údaj

vystupující údaj **:** *počet znaků*

vystupující údaj **:** *počet znaků* **:** *počet desetinných míst*

kde

vystupující údaj	je výraz reprezentující výstupní hodnotu některého z typů *Boolean*, *Char*, *Integer*, *Real* a **string** a příbuzných typů,
počet znaků	je výraz reprezentující počet znaků výstupní hodnoty,
počet desetinných míst	je výraz reprezentující počet desetinných míst výstupní hodnoty.

Význam základního příkazu výstupu: Parametrem příkazu *Write* se udává především hodnota některého z uvedených typů, jejíž vnější reprezentace vystoupí na obrazovku. Pomocí parametru příkazu *Write* je možno dále explicitně předepsat celkový počet znaků vnější reprezentace a v případě výstupu hodnoty reálného typu také počet desetinných míst.

Má-li parametr příkazu *Write* tvar jen *vystupující údaj*, vystupují hodnoty typů *Char* a ***string*** ve skutečné délce, formát výstupu u ostatních typů je určen implementací.

Obsahuje-li parametr příkazu *Write počet znaků*, závisí formát výstupu též na minimálním počtu znaků potřebném pro *vystupující údaj* (dále bude označován symbolem *potřebný počet znaků*):

- je-li *počet znaků* > *potřebný počet znaků* , doplní se výstupní hodnota zleva mezerami,
- je-li *počet znaků* < *potřebný počet znaků* , pak hodnoty typů *Integer*, *Real* a i hodnoty příbuzných typů vystoupí v minimálním potřebném počtu znaků.

Příkaz *WriteLn* zapisuje do výstupního souboru oddělovač řádků, což se např. při výstupu na obrazovku projeví přechodem na další řádek.

Pozor! - Při výstupu do konzolového okna se po zaplnění celého řádku automaticky přechází na další řádek, a proto, dojde-li současně k provádění příkazu *WriteLn*, budou realizovány dvě řádkové změny.

Obecně je možno v příkazu výstupu *Write* uvádět seznam parametrů (tj. nejen jeden parametr), oddělovačem parametrů je pak čárka. Stejný seznam může být použit i v příkazu *WriteLn*.

Místo posloupnosti

```
Write(P1); Write(P2);~~~ ; Write(Pn) ; WriteLn
```

lze psát

```
Write(P1,P2,~~~,Pn); WriteLn
```

nebo jen

```
WriteLn(P1,P2,~~~,Pn);
```

Řešené úlohy

Příklad Účinky některých příkazů *Write* umístěných za následující částí programu v Delphi zachycuje dále uvedená tabulka:

```
var
  N:Integer;
  X:Real;
  L:Boolean;
  Z:Char;
begin
  N:=-117;
  X:=Pi;   {Pi - standardní funkce, Pi=3.141592~~~}
  L:=true;
  A:='*';
```

Tabulka:

příkaz	účinek
`Write (N:4)`	`-117`
`Write (N:6)`	`␣␣-117`
`Write (N:2)`	`-117`
`Write (X:9:5)`	`␣␣3.14159`
`Write (X:8:4)`	`␣␣3.1416`
`Write (X:11)`	`␣3.14E+0000`
`Write (L:6)`	`␣␣TRUE`
`Write (Z:3)`	`␣␣*`
`Write ('Ano':6)`	`␣␣␣Ano`

Čtení ze souboru *Input*

Výklad

Soubor *Input* (tj. obvykle data z klávesnice) se čte pomocí procedury *Read*, resp. *ReadLn* (zkratka ze slov *read line*).

Základní tvar příkazu vstupu pro čtení ze souboru *Input* je

`Read(`*proměnná*`)`

kde *proměnná* je označení proměnné, která je některého z typů *Real*, **string**, *Integer*, *Char*, *Boolean* (včetně příbuzných typů).

Význam příkazu vstupu (v základním tvaru): Ze souboru *Input* se přečte jedna (další) lexikální jednotka a její hodnota se přiřadí proměnné *proměnná* (je-li to možné).

Např. je-li *proměnná* typu *Char*, je lexikální jednotkou jeden znak, je-li *proměnná* typu *Integer*, je lexikální jednotkou libovolný počet mezer a příp. znaků *eoln*, příp. znaménko a neprázdná posloupnost cifer.

Procedura *ReadLn* umožňuje přechod na čtení nového řádku i bez přečtení aktuálního řádku až do konce. Je-li při zadávání číselných hodnot za každou hodnotou použita klávesa *Enter*, nezáleží prakticky na tom, zda se použije procedura *Read* nebo *ReadLn*. Načítají-li se však z klávesnice též hodnoty řetězcových či znakových proměnných, je třeba respektovat skutečnost, že po načtení jakékoliv hodnoty procedurou *Read* zůstává k dispozici pro další čtení znak, který za touto hodnotou následuje (přesněji řečeno následoval), tj. obvykle *Enter*. Pokračuje-li se pak např. načítáním do řetězcové proměnné, je „ihned“ (tj. bez zásahu uživatele) načten prázdný řetězec. Uvedený problém lze tedy odstranit načtením předchozí hodnoty pomocí procedury *ReadLn*.

Následuje-li bezprostředně po sobě více příkazů čtení, je možno je sloučit obdobným způsobem jako posloupnost dříve uvedených příkazů výstupu.

2.7. Prázdný příkaz

Výklad

Prázdný příkaz nepředepisuje žádnou akci a je definován jako prázdný řetězec znaků abecedy jazyka. Prázdný příkaz se obvykle využívá jako kompoziční prvek ve strukturovaných příkazech.

Příklad:

```
if X<5 then  else Write('*')
```

V uvedeném úplném příkazu **if** mezi klíčovými slovy **then** a **else** není zapsáno nic, tj. je tu prázdný příkaz. Příslušný dílčí algoritmus však lze vyjádřit (a zřejmě přehledněji) neúplným příkazem **if**:

```
if X>=5 then Write('*')
```

tj. nahrazením původní podmínky *X<5* její negací.

Prázdný příkaz se často používá např. i před **end** ve složených příkazech, viz doprovodný text ke komentářům *{2}* a *{3}* z řešení následujícího příkladu z části 2.8. *Složený příkaz*.

2.8. Složený příkaz

Výklad

Složený příkaz vytváří z posloupnosti příkazů jediný (strukturovaný) příkaz.

Obecný tvar složeného příkazu:

begin *příkaz*$_1$; *příkaz*$_2$; ~~~ ; *příkaz*$_n$ **end**

kde *příkaz*$_i$ (pro i=1, 2, ..., n) je libovolný příkaz, přičemž n = 0, 1, 2, ...

Význam složeného příkazu: Složený příkaz je ve strukturovaných příkazech považován za (jediný) příkaz, což znamená, že pomocí složeného příkazu lze ve strukturovaných příkazech předepsat plnění i více příkazů na místech, kde lze (formálně) zapsat jen jeden příkaz.
V tomto smyslu tedy klíčová slova **begin** a **end** představují ***příkazové závorky***.

Řešené úlohy

Příklad 2.8.1. Vstupní údaje: přirozené číslo N z intervalu <1, 50> a dále posloupnost N reálných čísel. Vytvořte program, podle kterého se zjistí, kolik prvků posloupnosti je větších než aritmetický průměr prvků posloupnosti. Před každým čtením zajistěte tisk vhodné vstupní výzvy.

Řešení:

```
program Pocet;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A:array[1..50]of Real;
  I,N,NV:Integer;
  S,P:Real;
begin
  Write('N = ');
  ReadLn(N);
  S:=0;
  for I:=1 to N do
  begin
    Write('A[',I:2,'] = ');                    {1}
    ReadLn(A[I]);
    S:=S+A[I];                                 {2}
  end;
```

```
  P :=S/N;
  NV:=0;
  for I:=1 to N do
    if A[I]>P then
      NV:=NV+1;
  WriteLn('Pocet = ',NV);
  ReadLn;                                          {3}
end.
```

V navrženém programu jsou dva cykly **for**. První slouží nejen ke čtení, ale i ke sčítání prvků posloupnosti. Je-li tělo cyklu **for** tvořeno více než jedním příkazem, je třeba je uzavřít do příkazových závorek **begin** a **end**, a vytvořit tak z něj jediný (složený) příkaz, neboť cyklus **for** se vztahuje jen na první příkaz, který následuje za klíčovým slovem **do**. Z hlediska překladače jazyka je tělo uvedeného cyklu tvořeno 4 příkazy. Příkazu čtení totiž předchází příkaz (viz komentář *{1}*) zajišťující tisk vstupní výzvy, např. před čtením prvního prvku posloupnosti se objeví na obrazovce text:

```
A[ 1] =
```

Čtvrtým příkazem těla zmíněného cyklu je prázdný příkaz. Za příkazem *S:=S+A[I]*, tj. před **end**, je totiž středník, což „znamená“, že se tu ještě nachází prázdný příkaz.

Středník za posledním neprázdným příkazem ve složeném příkazu (tj. v poslední ukázce viz nejen příkaz s komentářem *{2}*, ale i *{3}*) je zbytečný, často se tu však z praktických důvodů píše.

2.9. Podmínkové příkazy - příkazy *if* a *case*

Výklad

Podmínkové příkazy umožňují zápis binárního nebo *n*-árního větvení algoritmu v závislosti na splnění podmínky, která je jejich explicitní nebo implicitní součástí.

Podmínkový příkaz:

- příkaz **if** (podmíněný příkaz)
  - příkaz **if** úplný (podmíněný příkaz úplný),
  - příkaz **if** neúplný (podmíněný příkaz neúplný),
- příkaz **case** (příkaz přepínače, přepínač).

Obecný tvar příkazu if úplného:

`if` *podmínka* **`then`** $příkaz_1$ **`else`** $příkaz_2$

kde

podmínka je výraz typu *Boolean*,

$příkaz_i$ (pro i=1, 2) je libovolný příkaz.

Obecný tvar příkazu if neúplného:

`if` *podmínka* **`then`** $příkaz_1$

kde význam symbolů je stejný jako u příkazu **if** úplného.

Předpokládané znalosti

Význam příkazů if byl již částečně uveden v části 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi*. Vzhledem k tomu, že příkazy vnořené do podmíněného příkazu mohou být libovolné, a tedy i podmíněné, je třeba pro úplnost doplnit, že význam příkazu

`if` $podmínka_A$ **`then if`** $podmínka_B$ **`then`** $příkaz_1$ **`else`** $příkaz_2$

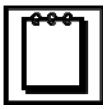
je shodný s významem příkazu

`if` $podmínka_A$ **`then begin if`** $podmínka_B$ **`then`** $příkaz_1$ **`else`** $příkaz_2$ **`end`**

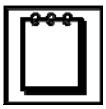
tj., že část **else** příkazu **if** patří vždy k nejbližšímu předcházejícímu „volnému“ **then**. Na rozdíl od předchozího příkazu tedy např. v příkazu

if *podmínka*$_A$ **then begin if** *podmínka*$_B$ **then** *příkaz*$_1$ **end else** *příkaz*$_2$

vypuštění dvojice příkazových závorek **begin** a **end** znamená změnu významu příkazu.



Poznámka



Přehlednost příkazu **if** je možno někdy výrazně zvýšit pomocí vhodného zápisu do úrovní a použitím případně i zbytečných příkazových závorek. Obvyklé způsoby zápisu příkazu **if** do úrovní zachycují následující ukázky:

```
if podmínka then
begin
  ~~~
end
else
begin
  ~~~
end
```

Nejsou-li příkazy vnořené do příkazu **if** složené, užívá se obvykle následující způsob zápisu do úrovní:

```
if podmínka then
  příkaz1
else
  příkaz2
```

Analogicky se zapisují do úrovní rovněž i příkazy **if** neúplné.

Příkaz case slouží k zápisu *n*-árního větvení, při němž se vybírá k realizaci jeden z *n* příkazů podle hodnoty výrazu (zvaného ***výběrový výraz***), který je součástí implicitně předepsané podmínky.

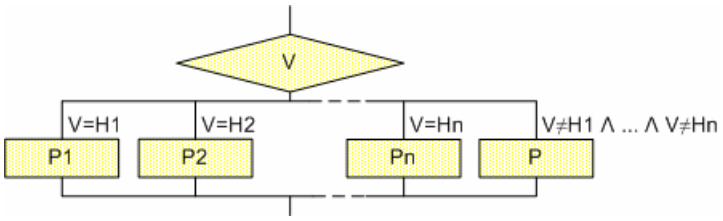
Obecný tvar základního rozšířeného příkazu case:

```
case výraz of
  hodnota1 : příkaz1;
  hodnota2 : příkaz2;
  ~~~
  hodnotan : příkazn;
  else příkaz
end
```

kde

výraz	je ***výběrový výraz***, který musí být ordinálního typu,
$hodnota_i$ pro i=1,2,...,n	je konstanta stejného typu jako je výběrový výraz,
$příkaz_i$	pro i=1,2,...,n je libovolný příkaz,
příkaz	je libovolný příkaz.

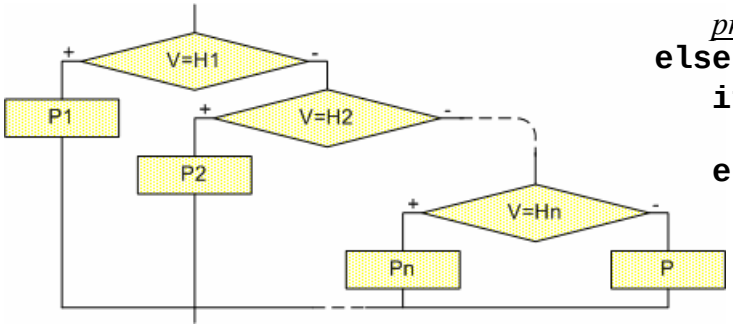
Symbolické vyjádření obecného tvaru základního rozšířeného příkazu case ve vývojovém diagramu:



Základní nerozšířený příkaz case se od rozšířeného liší jen tím, že neobsahuje část

```
else příkaz
```

Význam „správně užitého“ uvedeného základního rozšířeného příkazu **case** naznačuje následující složený příkaz (nejednoznačnost předznačení příkazů vnořených do příkazu **case** se projeví již při překladu):




```
begin
  if výraz = hodnota1 then
    příkaz1
  else
    if výraz = hodnota2 then
      příkaz2
    else
      if výraz = hodnota3 then
        příkaz3
      else
        ~~~
          if výraz = hodnotan then
            příkazn
          else
            příkaz
end
```

Na rozdíl od příkazu **if** může být v příkazu **case** před klíčovým slovem **else** středník.

Obecný příkaz case (nerozšířený i rozšířený) se od základního liší jen tím, že před každým z vnořených příkazů *příkaz*$_i$ může být formou seznamu uvedeno více konstant *hodnota*$_k$ (nejen tedy jediná konstanta), konstanty mohou být v seznamu zadány i pomocí intervalů (viz další příklad).

Řešené úlohy

Příklad 2.9.1. V proměnných *Den*, *Měsíc*, *Rok* je uloženo datum. Napište část programu, která zajistí, že do proměnné *Počet* bude uložen počet dnů v příslušném měsíci.

Řešení: Řešení je navrženo tak, aby obsahovalo různé způsoby předznačení v příkazu **case**. Např. pro měsíce červenec „až“ srpen je použit interval. K získání počtu dnů měsíce února (i pro přestupný rok) by mohly být využity např. vnořené příkazy **if** nebo přímo jeden příkaz **if** s poměrně složitou podmínkou. V ukázce je však formou složeného příkazu použit postup, při němž se nejprve vždy do proměnné *Počet* přiřadí číslo 28. Pak, je-li číslo roku dělitelné 4, se tato hodnota proměnné *Počet* přemaže číslem 29. Obdobně se na závěr provádí případná oprava na konce těch století, v nichž má únor zase jen 28 dnů.

Část **else** není v navrženém řešení použita, mohla by být např. využita v případě zadání nesmyslného čísla měsíce, nebo by mohla být využita pro jednu ze

tří „skupin“ měsíců (nejvýhodnější by bylo použít ji pro nejpočetnější skupinu měsíců).

```
~~~
var
  Den,Mesic,Rok,Pocet:Integer;
begin
  ~~~
  case Mesic of
    1,3,5,7..8,10,12: Pocet := 31;
    4,6,9,11        : Pocet := 30;
    2: begin
         Pocet:=28;
         if Rok mod 4 = 0 then
           Pocet := 29;
         if (Rok div 100) mod 4 <> 0 then
           Pocet := 28;
       end;
  end;
  ~~~
```

Příklad 2.9.2. V proměnných *Den*, *Měsíc*, *Rok* je uloženo datum. Napište část programu, která zajistí, že do proměnné *Období* bude uložen název příslušného ročního období.

Řešení: První řešení je založeno na využití příkazu **case** s výběrovým výrazem *Měsíc*100+Den* . Výběrový výraz „připomíná“ část základu rodného čísla, jeho hodnotou je čtyřciferné číslo, první dvě cifry udávají číslo měsíce, poslední dvě číslo dne v měsíci. Např. prvnímu dnu léta přísluší číslo 0621.

```
~~~
var
  Den,Mesic,Rok,MD:Integer;
  Obdobi:string;
begin
  ~~~
  case Mesic*100+Den of
    0321..0620: Obdobi := 'Jaro';
    0621..0922: Obdobi := 'Leto';
    0923..1220: Obdobi := 'Podzim';
    else        Obdobi := 'Zima';
  end
~~~
```

Uvedený příkaz **case** mohou různé kompilátory realizovat různě pracným způsobem. Jeden z nejméně pracných postupů by bylo možno získat nahrazením uvedeného příkazu **case** následujícími příkazy:

```
MD := Mesic*100+Den;
if MD<0321 then
  Obdobi := 'Zima'
else
  if MD<0621 then
    Obdobi := 'Jaro'
  else
    if MD<0923 then
      Obdobi := 'Leto'
    else
      if MD<1221 then
        Obdobi := 'Podzim'
      else
        Obdobi := 'Zima';
```

Poznámka

V uvedených ukázkách s příkazem **case** byla před běžnou formou při zápisu do úrovní dána přednost názornosti.

2.10. Příkazy cyklu - příkazy *while*, *repeat* a *for*

Předpokládané znalosti

Pomocí cyklu (viz 1.3. *Cyklus*) je možno jednoduše a přehledně předepsat opakování nějaké činnosti. Činnost, jejíž opakování má cyklus zajistit, se nazývá *tělo cyklu*.

Z hlediska algoritmizace se (konečné) cykly rozdělují podle toho, zda je, či není znám v explicitní formě počet opakování těla cyklu, na:

- ***cykly se známým počtem opakování*** (též *cykly s explicitním počtem opakování*),
- ***cykly s neznámým počtem opakování*** (též *cykly řízené podmínkou*).

Jelikož u cyklu s neznámým počtem opakování nelze explicitně předepsat počet opakování, je třeba jeho ukončení vázat na splnění, resp. nesplnění, určité podmínky. Je však zřejmé, že i na cyklus se známým počtem opakování lze pohlížet jako na cyklus řízený podmínkou, která pak přímo souvisí s počtem opakování těla cyklu (viz *Základní vyjádření cyklu se známým počtem opakování* v části 1.3. *Cyklus*).

Výklad

V části 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi* byly uvedeny dva zvláštní, ale velmi významné případy cyklu řízeného podmínkou: *cyklus s testem před operačním blokem* a *cyklus s testem po operačním bloku*. Delphi obsahuje prostředky, které umožňují přímo vyjadřovat jak cyklus se známým počtem opakování (příkaz **for**), tak i oba uvedené případy cyklů řízených podmínkou (příkaz **while** a příkaz **repeat**). V příkazu **for** je počet opakování předepsán explicitně pomocí počáteční a koncové hodnoty parametru cyklu, v příkazech **while** a **repeat** je počet opakování předepsán implicitně ve tvaru podmínky.

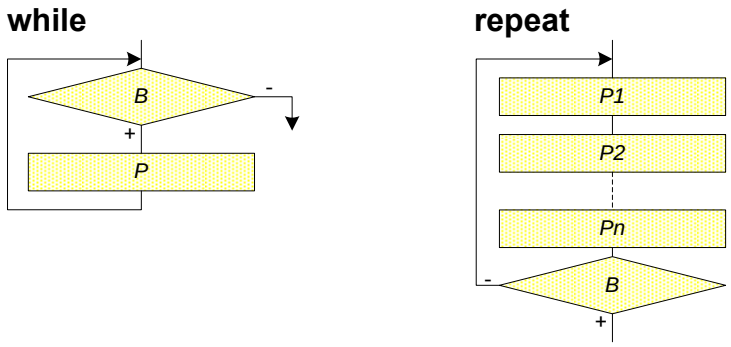
Obecný tvar příkazů while a repeat:

```
while podmínka do příkaz
repeat příkaz1 ; příkaz2 ; ~~~ ; příkazn until podmínka
```

kde

podmínka	je výraz typu *Boolean*,
příkaz , $příkaz_i$	pro i=1,2,...,n jsou libovolné příkazy.

Význam příkazů while a repeat byl vyjádřen již v 1.3.1. *Převod algoritmů z vývojových diagramů do Delphi* následujícími částmi vývojových diagramů:



Z uvedeného vyjádření vyplývá:

- příkaz **while** slouží k vyjádření cyklu s testem před operačním blokem, tvořeným jediným příkazem (může to však být i příkaz složený),
- příkaz **repeat** slouží k vyjádření cyklu s testem za operačním blokem, tvořeným posloupností příkazů (jejich oddělovačem jsou středníky),
- není-li na počátku provádění příkazu **while** splněna požadovaná podmínka, vnořený příkaz se neprovede ani jednou,
- při provádění příkazu **repeat** se vnořená posloupnost příkazů provede vždy alespoň jednou.

Obecný tvar příkazu for (dvě možnosti):

for *parametr cyklu* := $výraz_1$ **do** $výraz_2$ **do** *příkaz*

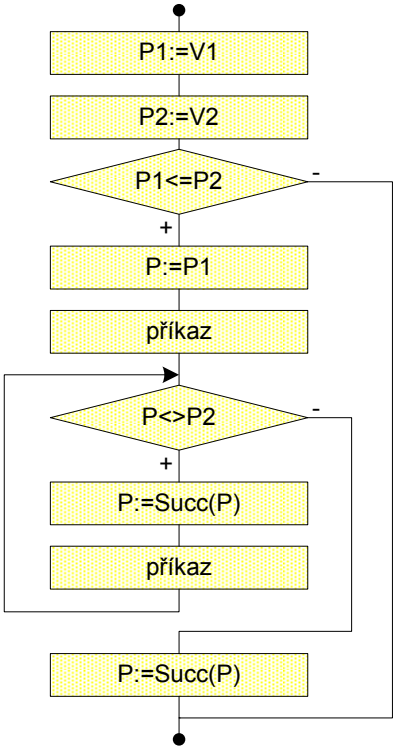
for *parametr cyklu* := $výraz_1$ **downto** $výraz_2$ **do** *příkaz*

kde

parametr cyklu	je identifikátor proměnné ordinálního typu, která je parametrem cyklu (parametr cyklu tedy nemůže být např. typu *Real*),
$výraz_1$, $výraz_2$	jsou výrazy, jejichž hodnotu lze přiřadit parametru cyklu, přičemž $výraz_1$ určuje (často přímo udává) počáteční a $výraz_2$ koncovou hodnotu parametru cyklu,
příkaz	je libovolný příkaz, který představuje tělo cyklu (často to bývá příkaz složený).

Význam příkazu for: Příkaz **for** s klíčovým slovem **to** [**downto**] se provádí tak, že hodnota parametru cyklu se postupně zvětšuje [zmenšuje] v rámci daného typu od hodnoty *výraz*$_1$ do hodnoty *výraz*$_2$ a pro každou hodnotu parametru cyklu se provede tělo cyklu. Je-li však *výraz*$_1$ > *výraz*$_2$ [*výraz*$_1$ < *výraz*$_2$], tělo cyklu se neprovede ani jednou.

Přesný význam (způsob realizace) příkazu **for** je poměrně složitý, při standardním užití jej vyjadřuje následující složený příkaz (*Pozor při převodu programů z jazyka Turbo-Pascal!* V Delphi má totiž parametr cyklu **for** po ukončení cyklu obvykle jinou hodnotu než v TP):




```
begin
  pom1:=výraz1;
  pom2:=výraz2;
  if pom1<=pom2 then
  begin
    parametr cyklu:=pom1;
    příkaz;
    while parametr cyklu <> pom2 do
    begin
      parametr cyklu:=succ(parametr cyklu);
      příkaz
    end
    parametr cyklu:=succ(parametr cyklu);
  end
end
```

Poznámka

- V příslušné části vývojového diagramu jsou opět použity „kratší“ identifikátory.
- Význam příkazu **for** s klíčovým slovem **downto** by se z uvedeného složeného příkazu získal analogicky ({ ≤→ ≥ },{succ → pred}).

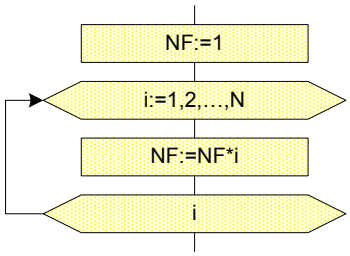
Proměnné *pom*$_1$, *pom*$_2$ v uvedeném složeném příkazu jsou pomocné vnitřní proměnné vytvářené překladačem (tj. programátorovi nedostupné), do kterých se na počátku provádění příkazu **for** uloží hodnoty výrazů *výraz*$_1$, *výraz*$_2$ a v další části provádění příkazu **for** se již tyto výrazy nepoužívají. Znamená to, že mezní hodnoty parametru cyklu nemohou být během provádění cyklu změněny.

V Delphi je dále zakázáno měnit hodnotu parametru cyklu příkazy těla cyklu, porušení uvedeného zákazu vyvolá chybové hlášení (na rozdíl od TP).

Řešené úlohy

Příklad : Napište část programu pro výpočet *n!*, je-li hodnota *n* v proměnné *N* a pro výsledek (tj. pro *n!*) je určena proměnná *NF*.

Řešení:




```
var
  I,N,NF:Integer;
begin
  ~~~
  NF:=1;
  for I:=1 to N do
    NF:=NF*I;
  ~~~
```

Příklad : Napište část programu, která zajistí provedení příkazu *příkaz* pro *n*+1 hodnot reálné proměnné *X*, jež nabude hodnot $x_0, x_1, \ldots, x_n$, přičemž hodnoty $x_0, x_1, \ldots, x_n$ vzniknou rozdělením intervalu <*a*, *b*> na *n* stejných dílů (*n* přirozené číslo, *n*>1), tj. $x_0 = a$, $x_i = a+i \cdot h$, pro $i=1,2,\ldots,n$, kde $h=(b-a)/n$. Požadovanou část napište:

a) pomocí cyklu **for** s parametrem *i* (*i* - pomocná celočíselná proměnná),

b) bez užití pomocné celočíselné proměnné (a tedy bez užití příkazu **for)**.

Řešení: V řešení *(A)* se jednotlivé hodnoty proměnné *X* počítají podle vztahu ze zadání v cyklu **for**, jehož parametrem je celočíselná proměnná *I*. V řešení *(B)* je parametrem cyklu přímo reálná proměnná *X*, ta však nemůže být parametrem cyklu **for**. V ukázce byl proto konkrétně použit cyklus **while**, v němž se hodnota parametru cyklu *X* postupně zvyšuje přičítáním hodnoty zvané ***krok cyklu***.

Požadovaný *příkaz* se má poprvé provést pro $X = A$ a naposled pro $X = B$. V příkazu **while** však není použita podmínka $X <= B$, ale podmínka $X <= XB+H/2$, neboť zpracování hodnot typu *Real* probíhá vždy jen s určitou přesností a použití podmínky $X <= B$ by mohlo vést k tomu, že by se *příkaz* pro zadáním požadovanou poslední (ale poněkud nepřesně spočítanou) hodnotu *X* již neprovedl. K požadované koncové hodnotě *B* se proto obvykle přidává dostatečně malá část kroku *H*, např. *H/2*, resp. *H/10.*

```
{varianta A}
var
  X,A,B,H:Real;
  I,N:Integer;
begin
  ~~~
  H:=(B-A)/N;
  for I:=0 to N do
  begin
    X:=A+I*H;
    příkaz;
  end;
  ~~~
```

```
{varianta B}
var
  X,A,B,H:Real;
  I:Integer;
begin
  ~~~
  H:=(B-A)/N;
  X:=A;
  while X <= B+H/2 do
  begin
    příkaz;
    X:=X+H;
  end;
  ~~~
```

V jazyku Basic by mohla mít požadovaná část programu tvar:

```
H=(B-A)/N
for X=A to B+H/10 step H
  příkaz
next X
```

(v jazyku Basic může být parametrem cyklu **for** i reálná proměnná, příkaz **for** může dále obsahovat část **step**, pomocí níž lze lehce předepsat požadovanou změnu parametru cyklu, i zde je však třeba s ohledem na numerické nepřesnosti při práci v typu *Real* vhodně volit horní mez cyklu).

Výklad

Vztah příkazů while a repeat:

Zápis

while *podmínka* **do** *příkaz*

je ekvivalentní zápisu

```
begin
  if podmínka then repeat příkaz until not (podmínka)
end
```

Zápis

repeat $příkaz_1$; $příkaz_2$; ~~~ ; $příkaz_n$ **until** *podmínka*

je ekvivalentní zápisu

```
begin
  příkaz1; příkaz2; ~~~ ; příkazn;
  while not (podmínka) do
  begin
    příkaz1; příkaz2; ~~~ ; příkazn;
  end
end
```

Příklad 2.10.1: Napište část programu, v níž se generuje posloupnost výsledků hodu kostkou, tj. posloupnost čísel 1 až 6 s rovnoměrným rozdělením, tak dlouho:

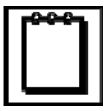
a) až „padne“ šestka,

b) až padnou celkem dvě šestky (tj. nemusí padnout bezprostředně po sobě),

c) až padnou bezprostředně po sobě dvě šestky,

d) až padne celkem *n* šestek, kde *n* je vstupní údaj,

e) až padne bezprostředně po sobě *n* šestek, kde *n* je vstupní údaj.



Poznámka

V Delphi slouží ke generování náhodných čísel s rovnoměrným rozdělením funkce *Random*.

Funkce *Random*

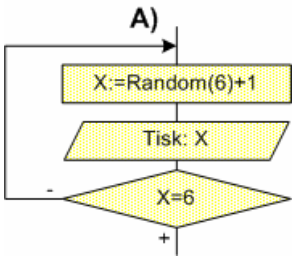
Funkce *Random* bez parametru poskytuje (pseudonáhodné) reálné číslo z intervalu <0, 1) s rovnoměrným rozložením, funkce *Random* s parametrem typu *Integer* obdobně poskytuje celé číslo z intervalu <0, *Parametr*), např. *Random(6)* poskytne některé z čísel 0, 1, …, 5. Před prvním voláním funkce *Random* v programu je obvykle vhodné volat proceduru *Randomize*.

Procedura *Randomize* inicializuje generátor náhodných čísel náhodnou hodnotou odvozenou z údaje systémových hodin (program, v němž není procedura *Randomize* použita, by při každém spuštění poskytoval stejnou posloupnost náhodných čísel).

Řešení př. 2.10.1:

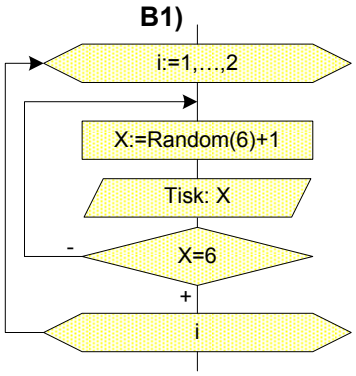
S výjimkou varianty (a) budou uváděny jen požadované části programů, všechny zde používané proměnné jsou typu *Integer*. U varianty (a) bude pro potřebu začátečníků kromě příslušné části vývojového diagramu uveden i celý program v Delphi.

Formulace požadavku (a) jednoznačně odpovídá cyklu **repeat**, v němž kromě generování je i tisk požadovaného náhodného čísla:

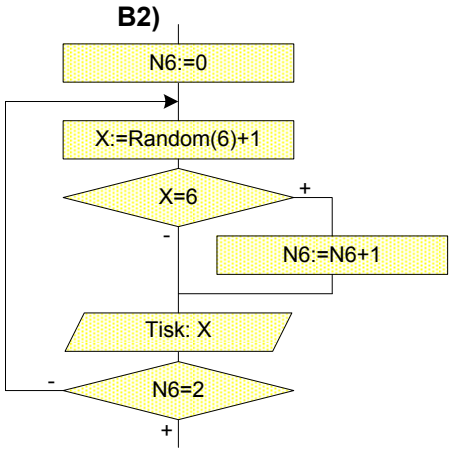



```
program KostkaA;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  X:Integer;
begin
  Randomize;
  repeat
    X:=Random(6)+1;
    Write(X:2);
  until X=6;
  ReadLn;
end.
```

Část programu pro požadavek (b) je navržena dvakrát. Ve variantě *B1* je pomocí cyklu **for** předepsáno dvojí opakování dílčího algoritmu pro požadavek (a), tj. pro hod jedné šestky, ve variantě *B2* je použito počítadlo padlých šestek (proměnná *N6*) a generování čísel pomocí cyklu **repeat** končí, jakmile proměnná *N6* nabude hodnotu 2. Před vstupem do uvedeného cyklu musí mít proměnná *N6* nulovou hodnotu.

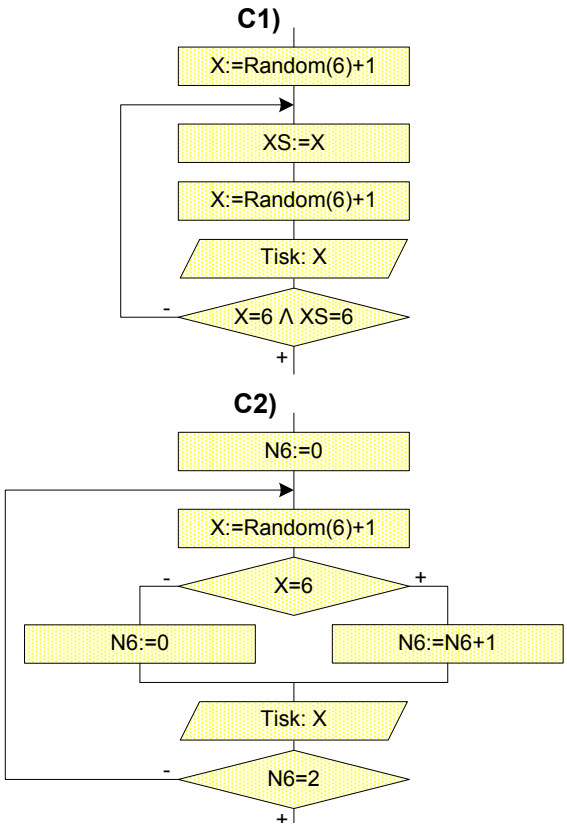



```
{Varianta B1}
for I:=1 to 2 do
  repeat
    X:=Random(6)+1;
    Write(X:2);
  until X=6;
```




```
{Varianta B2}
N6:=0;
repeat
  X:=Random(6)+1;
  if X=6 then
    N6:=N6+1;
  Write(X:2);
until N6=2;
```

Část programu pro požadavek (c) je rovněž navržena dvakrát. Ve variantě *C1* je generování čísel ukončeno, jakmile nové číslo *X* i předchozí „staré“ číslo *XS* je šestka. Před generováním nové hodnoty do proměnné *X* je proto její hodnota (tj. předchozí generované číslo) uložena do proměnné *XS*. Ve variantě *C2* je opět použito počítadlo padlých šestek (proměnná *N6*). Na rozdíl od dílčího algoritmu *B2* však musí být zajištěna nulová hodnota proměnné *N6* i tehdy, když je generováno jiné číslo než šestka. Pro požadavek (d) je možno použít princip (základní myšlenku) obou řešení požadavku (b), pro požadavek (e) je však vhodný jen princip z řešení *C2*.




```
{Varianta C1}
X:=Random(6)+1;
repeat
  XS:=X;
  X :=Random(6)+1;
  Write(X:2);
until (X=6) and (XS=6);
```

```
{Varianta C2}
N6:=0;
repeat
  X:=Random(6)+1;
  if X=6 then
    N6:=N6+1
  else
    N6:=0;
  Write(X:2);
until N6=2;
```

Úlohy k samostatnému řešení

Příklad 2.10.2. Programy (vývojové diagramy) požadované v příkladech 1.3.5. až 1.3.8. vyjádřete v Delphi. Případnou konkretizaci zadání pro potřebu Delphi, např. na úpravu tisků, proveďte v souladu se zadáním úlohy sami.

Příklad 2.10.3. Sestavte program, podle kterého se vytiskne tabulka se záhlavím

x sin(sqrt(x)) sqrt(sin(x))

pro čísla *x* od *a* do *b* s krokem *h*. Není-li funkční hodnota definována, tiskněte místo ní tři pomlčky.

Příklad 2.10.4. Sestavte program, podle kterého se nalezne nejmenší a dále největší netriviální dělitel zadaného přirozeného čísla, resp. se vytiskne informace o tom, že se jedná o prvočíslo.

Příklad 2.10.5. Vstupní údaje tvoří:

1) přirozené číslo *N* a dále posloupnost *N* přirozených čísel,
2) posloupnost přirozených čísel a dále číslo záporné.

Sestavte program, podle kterého se zjistí, kolik je v zadané posloupnosti prvků:

a) sudých,
b) lichých a současně dělitelných sedmi,
c) které nejsou dělitelné čísly 2 až 20.

Příklad 2.10.6. Mezi přirozenými čísly existují čtyři čísla, která se rovnají součtu třetích mocnin svých cifer. První z nich je 153 ($153 = 1^3 + 5^3 + 3^3$). Sestavte program pro nalezení zbylých tří čísel. *Pomůcka:* Hledaná čísla jsou menší než 410.

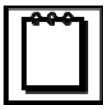
2.11. Typ pole (*array*)

Předpokládané znalosti

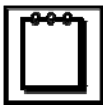
Mezi strukturovanými údaji, se kterými je možno v Delphi přímo pracovat, jsou i údaje typu pole (viz též část 1.4.1. *Pole*). Údaji typu pole jsou např. vektory a matice známé z matematiky.

Výklad

Pole je homogenní statický datový objekt, jehož složky zvané **prvky pole** jsou rozlišovány jen pomocí indexů, přičemž všechny prvky pole mají stejný počet indexů a jednotlivé indexy nabývají všech hodnot určitého ordinálního typu (často se jedná o typ definovaný pomocí intervalu).



Poznámka



- Pole je homogenní datový objekt, složky pole jsou tedy stejného typu.
- Typ prvků pole může být opět strukturovaný, tj. může to být opět pole, jedná se pak o pole polí. Pro pole, jehož prvky mají n indexů, se též užívá název **n-rozměrné pole**.
- Při volnějším vyjadřování se někdy používá pouze jen termín pole, přičemž se jedná buď o typ pole (pole jako typ), nebo konstantu typu pole (pole jako konstanta), či o proměnnou typu pole (pole jako proměnná, tj. proměnné pole).

Výklad

Popis typu jednorozměrného pole má tvar

`array` [*typ indexu*] `of` *typ složek*

kde

typ indexu je libovolný ordinální typ (často to bývá interval z typu *Integer*),

typ složek je libovolný typ.

Řešené úlohy

Příklad : Matematické vektory ***a*** a ***b*** s reálnými složkami $a_1, a_2, \ldots, a_{100}, b_1, b_2, \ldots, b_{100}$ a dále pole *DC* pro české názvy a pole *DA* pro anglické názvy dnů v týdnu lze deklarovat více způsoby. Např.

```
type
  TypDen=(Po,Ut,St,Ct,Pa,So,Ne);
var
  A,B:array[1..100] of Real;
  DC,DA:array[Po..Ne] of string;
```

nebo

```
type
  TypDen=(Po,Ut,St,Ct,Pa,So,Ne);
  TypVektor=array[1..100] of Real;
var
  A,B:TypVektor;
  DC,DA:array[TypDen] of string;
```

nebo

```
type
  TypDen=(Po,Ut,St,Ct,Pa,So,Ne);
  TypVektor=array[1..100] of Real;
  TypNazvy =array[TypDen] of string;
var
  A,B:TypVektor;
  DC,DA:TypNazvy;
```

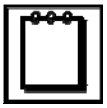
Výklad

Zpřístupnění složek proměnné typu pole (***selektor pole***) se vyjadřuje pomocí identifikátoru pole a indexu, resp. indexů, a u jednorozměrného pole má tvar

identifikátor pole [*indexový výraz*]

kde *indexový výraz* je výraz, jehož hodnota musí být kompatibilní s typem indexu pole.

Poznámka



Složka (položka, prvek) proměnné typu pole se v některých programovacích jazycích též nazývá **indexovaná proměnná**.

V Delphi je možno deklarovat vícerozměrné pole jako jednorozměrné pole, jehož složky jsou opět typu pole (musí se však jednat o již definovaný typ). Např. matici s reálnými prvky

je možno deklarovat jako jednorozměrné pole, jehož prvky jsou řádky, tj. vektory s reálnými složkami.

Řešené úlohy

Příklad : Deklarace reálné matice A typu (m,n), kde m,n jsou přirozená čísla, $m \leq 10$, $n \leq 5$.

Řešení:

1) Deklarace přímá

```
var
  A:array[1..10,1..5] of Real;
```

2) Deklarace prostřednictvím typu

```
type
  TMat=array[1..10,1..5] of Real;
var
  A:TMat;
```

nebo

```
type
  TVektor=array[1.. 5] of Real;
  TMatice=array[1..10] of TVektor;
var
  A:TMatice;
```

Poznámka

Při kterékoliv z uvedených deklarací matice A jsou následující dvě zpřístupnění např. prvku $a_{2,3}$ této matice zcela rovnocenná:

A[2,3] , A[2][3]

Současně zápis

A[2]

představuje označení 2. řádku matice **A**, což je možno využívat v některých příkazech, např. jediným příkazem

A[5]:=A[2]

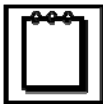
lze předepsat zkopírování 2. řádku matice A do 5. řádku této matice.

Řešené úlohy

Příklad : Napište část programu pro tisk (výstup) matice *A*, která má *m* řádků a *n* sloupců ($m \leq 10$, $n \leq 5$) a jejímiž prvky jsou reálná čísla z intervalu (-100, 100), která mají být uvedena na 3 desetinná místa. Zajistěte rovněž tisk řádkových indexů před jednotlivými řádky.

Řešení:

```
var
  A:array[1..10,1..5] of Real;
  I,J,M,N:Integer;
begin
  ~~~
  for I:=1 to M do
  begin                          {začátek příkazu pro tisk i-tého řádku}
    Write(I:2,'':3);             {příkaz pro tisk indexu řádku}
    for J:=1 to N do
      Write(A[I,J]:10:3);        {příkaz pro tisk prvku na pozici [i,j]}
    WriteLn                      {příkaz pro odřádkování na konci řádku}
  end;                           {konec příkazu pro tisk i-tého řádku}
  ~~~
```



Poznámka

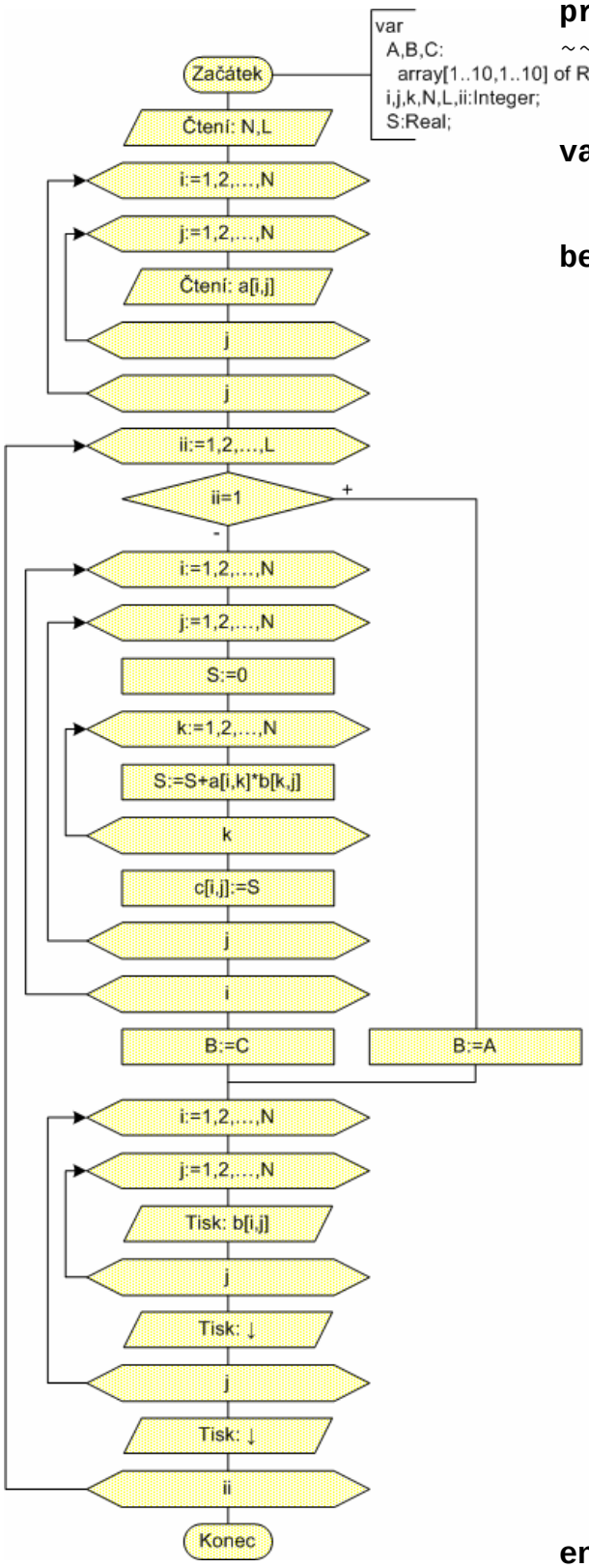
Řešené příklady na užití „obyčejných“ jednorozměrných polí budou v této části uvedeny jen omezeně, neboť u řešených příkladů 1.4.1. až 1.4.4. v části 1.4.1. *Pole* jsou nejen vývojové diagramy, ale i příslušné programy v Delphi.

Řešené úlohy

Příklad. 2.11.1. Vstupní údaje tvoří přirozená čísla *N*, *L*, obě z intervalu <2, 10>, a dále v pořadí po řádcích prvky reálné matice *A* typu (*N*,*N*). Vytvořte program pro výpočet a tisk matic $A^1, A^2, \ldots, A^L$.

Řešení: Jedná se vlastně o ilustrační př. 1.5.1. z části 1.5.1. *Metody návrhu algoritmu shora dolů a zdola nahoru*. V řešení tohoto příkladu je nejprve provedena analýza úlohy, a pak, nejprve při objasňování metody návrhu algoritmu zdola nahoru, je postupně formou vývojového diagramu částečně navržen dílčí algoritmus pro násobení matic, později, při objasňování metody návrhu algoritmu shora dolů, je opět formou vývojového diagramu navrženo hrubé řešení celé úlohy a dále je tu dokončen

návrh dílčího algoritmu pro násobení matic. Pro detailní zápis řešení celé úlohy tedy zbývá navrhnout jen dílčí algoritmy (v řešení př. 1.5.1. pouze zmíněné) pro čtení a tisk matice (tisk matice viz předchozí příklad, čtení matice je algoritmicky podobné).




```
program Mocniny;
~~~

var
  {viz vývojový diagram}
  ~~~
begin
  Write('N = ');
  ReadLn(N);
  Write('L = ');
  ReadLn(L);
  for I:=1 to N do
    for J:=1 to N do
    begin
      Write('A[',I,',',J,'] = ');
      ReadLn(A[I,J]);
    end {for J};
  WriteLn;
  for II:=1 to L do
  begin
    if II=1 then
      B:=A
    else
    begin
      for I:=1 to N do
        for J:=1 to N do
        begin
          S:=0;
          for K:=1 to N do
            S:=S+A[I,K]*B[K,J];
          C[I,J]:=S;
        end {for J};
      B:=C;
    end {if II=1};
    for I:=1 to N do
    begin
      for J:=1 to N do
        Write(B[I,J]:8:2);
      WriteLn;
    end {for I};
    WriteLn;
  end {for II};
  ReadLn;
end.
```

Pro úpravu požadovaných výstupů jsou důležité příkazy pro odřádkování, které jsou proto tentokrát uvedeny i ve vývojovém diagramu.

Řešené úlohy

Příklad. 2.11.2. Vstupní údaje tvoří nejprve přirozené číslo L z intervalu <1, 5>, pak přirozené číslo N z intervalu <2, 100> a dále pak souřadnice dvou N-složkových vektorů ***a***, ***b*** v pořadí, které je určeno číslem L, přičemž jednotlivým hodnotám čísla L přísluší následující pořadí:

1) $a_1, b_1, a_2, b_2, \ldots, a_N, b_N,$
2) $a_1, a_2, \ldots, a_N, b_1, b_2, \ldots, b_N,$
3) $a_N, b_N, a_{N-1}, b_{N-1}, \ldots, a_1, b_1,$
4) $a_1, a_2, \ldots, a_N, b_N, b_{N-1}, \ldots, b_1,$
5) $a_1, b_N, a_2, b_{N-1}, \ldots, a_N, b_1$.

Vyjádřete algoritmus pro výpočet skalárního součinu vektorů ***a*** a ***b*** , tj. hodnoty $\boldsymbol{a}\cdot\boldsymbol{b}$, kde

$$\mathbf{a}\cdot\mathbf{b} = a_1\cdot b_1 + a_2\cdot b_2 + \ldots + a_N\cdot b_N \ ,$$

tak, aby vlastní výpočet skalárního součinu začínal až po přečtení všech složek obou vektorů.

Řešení: Má-li být výpočet skalárního součinu zahájen až po načtení všech souřadnic obou vektorů, je třeba tyto souřadnice uložit do pole, resp. polí. Pokud budou souřadnice vektorů ***a***, ***b*** uloženy např. do jednorozměrných polí A, B přirozeným způsobem (tj. první souřadnice v první složce atd.), bude pak další část programu shodná pro všechna uvedená pořadí – viz následující řešení, ve kterém každému z pěti uvedených pořadí odpovídá jeden dílčí příkaz příkazu **case**.

```
program SoucinB;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  A,B:array[1..100] of Real;
  S:Real;
  I,N,L,K:Integer;
begin
  Write('Varianta = ');
  ReadLn(L);
  Write('N = ');
  ReadLn(N);
```

```
  {Čtení souřadnic obou vektorů}
  case L of
    1: begin {Varianta čtení 1}
         for I:=1 to N do
           ReadLn(A[I],B[I]);
       end;
    2: begin {Varianta čtení 2}
         for I:=1 to N do
           ReadLn(A[I]);
         for I:=1 to N do
           ReadLn(B[I]);
       end;
    3: begin {Varianta čtení 3}
         for I:=N downto 1 do
           ReadLn(A[I],B[I]);
       end;
    4: begin {Varianta čtení 4}
         for I:=1 to N do
           ReadLn(A[I]);
         for I:=N downto N do
           ReadLn(B[I]);
       end;
    5: begin {Varianta čtení 5}
(*
         for I:=1 to N do
           ReadLn(A[I],B[N+1-I]);
*)
         for I:=1 to N do
         begin
           Write('A[',I,'] = ');
           ReadLn(A[I]);
           K:=N+1-I;
           Write('B[',K,'] = ');
           ReadLn(B[K]);
         end;
       end;
  end {case L};
  {Začátek vlastního výpočtu}
  S:=0;
  for I:=1 to N do
    S:=S+A[I]*B[I];
  {Tisk výsledku}
  WriteLn('Součin =',S:10:4);
  ReadLn;
end.
```

V uvedeném programu je užito opět více komentářů. Komentáře mohou nejen zpřehlednit zápis programu (předchozí program), ale mohou i přispět k pochopení zapsaného algoritmu.

V Delphi je dále možno pomocí komentářů „dočasně“ vyřadit část příkazů – v uvedeném programu viz *{Varianta čtení 5}*. Varianty čtení 1 až 4 jsou „zatím“ bez vstupních výzev. U varianty 5 jsou již vstupní výzvy, původní zápis bez vstupních výzev tu byl zatím ponechán ve formě komentáře, pro který je vhodné místo závorek *{ }* použít závorky *(* *)*. V uvedeném případě to sice není nutné, ale v Delphi není možné, aby uvnitř komentáře v závorkách byly stejné závorky. Proto je vhodné vyhradit např. závorky *{ }* pro běžné komentáře a závorky *(* *)* používat na dočasné vyřazení některých částí programu.

Úlohy k samostatnému řešení

Příklad 2.11.3. Místo vývojových diagramů požadovaných v neřešených příkladech části 1.4.1. *Pole* sestavte programy v Delphi. Případnou konkretizaci zadání pro potřebu Delphi a pro úpravu tisku (výstupů) proveďte v souladu se zadáním úlohy sami.

Příklad 2.11.4. Vstupní údaje tvoří přirozená čísla *M*, *N*, obě z intervalu <1, 5>, a dále prvky reálné matice *A* typu (*M*,*N*) v pořadí

a) po řádcích,

b) po sloupcích.

Sestavte program, podle kterého se zajistí nejprve tisk zadané matice, pak tisk matice, která se od zadané liší jen tím, že záporné prvky jsou nahrazeny nulami, a na závěr se opět vytiskne zadaná matice, přičemž za každým řádkem bude uveden řádkový průměr.

Příklad 2.11.5. Vstupní údaje tvoří přirozená čísla *M*, *N*, obě z intervalu <1, 5>, a dále prvky reálné matice *A* typu (*M,N*) v pořadí po řádcích. Sestavte program, podle kterého se zajistí tisk zadané matice a tisk matice k ní transponované, přičemž:

a) transponování se provede jen při tisku,

b) transponovaná matice se nejprve uloží do pole *B*,

c) před tiskem transponované matice se nejprve matice *A* transponuje v poli *A*.

Příklad 2.11.6. Vstupní údaje tvoří přirozená čísla *M*, *N*, obě z intervalu <1, 5>, a dále prvky celočíselné matice *A* typu (*M,N*) v pořadí po řádcích. Sestavte program, podle kterého se zajistí tisk zadané matice a tisk matice, která vznikne ze zadané přerovnáním řádků do opačného pořadí, přičemž:

a) přerovnání se provede jen při tisku,

b) nová matice se nejprve uloží do pole *B*,

c) před tiskem nové matice se přerovnání řádků provede přímo v poli *A*.

Příklad 2.11.7. Vstupní údaje tvoří přirozená čísla *M*, *N*, obě z intervalu <2, 8>, a dále prvky reálné matice *A* typu (*M,N*) v pořadí po sloupcích. Sestavte program, podle kterého se nejprve zajistí tisk zadané matice a dále se vypočte a vytiskne:

a) aritmetický průměr všech prvků matice,

b) aritmetický průměr kladných prvků matice,

c) aritmetický průměr prvků hlavní diagonály,

d) aritmetický průměr kladných prvků hlavní diagonály,

e) aritmetický průměr prvků pod hlavní diagonálou a dále aritmetický průměr prvků nad hlavní diagonálou,

f) maximum z hodnot prvků hlavní diagonály,

g) index výskytu maximálního prvku, resp. jednoho z maximálních prvků hlavní diagonály,

h) *minimax matice*, tj. minimum z řádkových maxim matice,

i) dvojice indexů výskytu (jednoho z výskytů) minimaxu matice,

j) minimum z absolutních hodnot všech prvků hlavní diagonály,

k) minimum z hodnot kladných prvků hlavní diagonály.

Příklad 2.11.8. Napište část programu, která zajistí, že se do (vzestupně nebo sestupně) setříděné posloupnosti celých čísel zařadí další celé číslo tak, aby výsledná posloupnost byla opět setříděná.

Poznámka

Aby bylo možno požadovanou část programu ladit, je třeba ji zařadit do nějakého vhodného programu, který zajistí, že na počátku příslušného pole, resp. polí, bude setříděná posloupnost prvků. Pro počáteční ladění lze výchozí setříděnou posloupnost vytvořit např. jen „natvrdo“, tj. např. zápisem několika vhodných přiřazovacích příkazů nebo vhodným cyklem. Pro vytvoření výchozí setříděné posloupnosti je rovněž možno použít např. některý z algoritmů z př. 1.4.4., přičemž prvky nesetříděné posloupnosti je možno načítat nebo generovat pomocí generátoru náhodných čísel (viz funkce *Random*, poznámka u příkladu 2.10.1.).

Příklad 2.11.9. Napište část programu, která zajistí, že se do setříděné posloupnosti celých čísel zatřídí další setříděná posloupnosti celých čísel (vytvoření jedné setříděné posloupnosti ze dvou setříděných posloupností se obvykle označuje termínem *aktualizace*).

2.12. Typ záznam (*record*)

Předpokládané znalosti

V části 1.4. *Strukturovaný údaj* byly strukturované údaje rozděleny podle dvou kritérií (1. *homogenní/nehomogenní*, 2. *statické/dynamické*). U pole (homogenní a statický údaj, viz část 2.11. *Typ pole (array)*) se jednotlivé prvky rozlišují jen pomocí indexů, často je však výhodné mít možnost označit (alespoň některé) položky strukturovaného údaje jménem. Neméně výhodná je i možnost seskupit do jednoho strukturovaného údaje i položky různých typů. Obě uvedené výhody poskytují v Delphi záznamy.

Výklad

Záznam (***record***) je strukturovaný datový objekt skládající se z určitého počtu pojmenovaných složek, které se nazývají **položky záznamu** a které mohou být různého typu, tzn., že záznam je statický (obecně) nehomogenní datový objekt.

Popis typu záznam má obecný tvar

`record` *seznam položek* **`end`**

kde

seznam položek může být tvořen dvojicemi:

identifikátor deklarované položky : *typ položky*

přičemž oddělovačem dvojic je středník.

Zpřístupnění položky proměnné typu záznam (***selektor záznamu***) má tvar

proměnná typu záznam . *identifikátor položky*

Používá-li se v nějakém úseku programu několik položek jednoho záznamu nebo jedna položka vícekrát, lze zápis tohoto úseku programu zkrátit užitím příkazu **with**.

Příkaz with

Příkaz with je strukturovaný příkaz, který umožňuje zkrácené označování položek proměnné typu záznam.

Základní tvar příkazu with je

`with` *proměnná typu záznam* **`do`** *příkaz*

Význam příkazu with: Uvnitř příkazu *příkaz*, který je vnořen do příkazu **with**, je možno označovat položky proměnné *proměnná typu záznam* pouze pomocí příslušného identifikátoru položky.

Řešené úlohy

Příklad 2.12.1. V následujících ukázkách z programu pro organizaci sportovní akce je nejprve deklarován typ *TypDatum*, který je vzápětí použit při deklaraci typu *TypOsoba* (jeho položka *Narození* je typu *TypDatum*). Pomocí těchto typů jsou dále deklarovány proměnné typu záznam a jednorozměrné pole záznamů.

Řešení:

```
~~~
type
  TDatum = record
    Den  : 1..31;
    Mesic: 1..12;
    Rok  : Integer
  end;
  TOsoba = record
    Prijmeni : string[20];
    Jmeno    : string[15];
    Nar      : TDatum;
    Kategorie: 'A'..'H';
  end;
var
  Termin      : TDatum;
  ZavodnikNovy: TOsoba;
  Zavodnik    : array[1..200] of TOsoba;
  I,N         : Integer;
begin
  ~~~
  Termin.Mesic:= 2;
  ~~~
    Zavodnik[I] := ZavodnikNovy;
    Zavodnik[I].Kategorie := 'A';
```

Měl-li by zmíněný program mj. zajišťovat výpis závodníků, kteří mají v den pořádání závodu narozeniny, mohl by obsahovat příkaz

```
for I:=1 to N do
  if (Zavodnik[I].Nar.Mesic = Termin.Mesic) and
     (Zavodnik[I].Nar.Den = Termin.Den) then
    WriteLn(Zavodnik[I].Kategorie,' ',Zavodnik[I].Prijmeni,
            ' ',Zavodnik[I].Jmeno)
```

nebo s využitím příkazu **with** příkaz

```
for I:=1 to N do
  with Zavodnik[I] do
    if (Nar.Mesic=Termin.Mesic) and (Nar.Den=Termin.Den) then
      WriteLn(Kategorie,' ',Prijmeni,' ',Jmeno)
```

Příklad 2.12.2. Vstupní údaje tvoří přirozené číslo *N* z intervalu <1, 100> a dále posloupnost *N* dvojic reálných čísel, které představují souřadnice bodů v rovině. Vytvořte program, podle kterého se nejprve vytisknou počty bodů v jednotlivých kvadrantech a dále se vytisknou souřadnice a číslo kvadrantu těch bodů, jejichž vzdálenost od počátku je větší než průměrná.

Řešení: Mají-li se tisknout souřadnice bodů, jejichž vzdálenost od počátku je větší než vzdálenost průměrná, je třeba nejprve tuto průměrnou vzdálenost ze souřadnic všech bodů spočítat, teprve potom je možno o tisku rozhodovat. Souřadnice všech bodů tedy musí být (po jistou dobu) uloženy v paměti např. ve vhodném poli, resp. ve vhodných polích.

S čísly kvadrantů a se vzdálenostmi jednotlivých bodů se během provádění programu rovněž pracuje dvakrát, tyto údaje jsou však ze souřadnic bodů dostupné, a není je proto třeba do paměti ukládat, aby se však uvedené údaje nemusely počítat dvakrát, budou i ony při prvním výpočtu rovněž uloženy do paměti.

V programu by např. mohla být deklarována pole:

```
var
  X,Y,V0:array[1..100] of Real;
  Q:array[1..100] of Integer;
```

Údaje o jednom bodu by se pak nacházely ve čtyřech různých polích a společný by měly jen index. V navržených programech *Body1* a *Body2* však bude použito pole záznamů *B*, v němž každému bodu odpovídá jediná (strukturovaná) položka typu *TBod*:

```
type
  TBod=record
    X,Y:Real;   {X,Y – souřadnice bodu}
    V0:Real;    {V0 – vzdálenost bodu od počátku}
    Q:Integer; {číslo kvadrantu}
  end;
var
  B:array[1..100] of TBod;
```

Pokud by v nějakém dalším programu (který by nebyl jen konzolovou aplikací) měly být zmíněné body vykreslovány, bylo by možno deklaraci typu *TBod* doplnit např. o položky *Barva*, *Značka*, *Popis*:

```
  TBod=record
    ~~~
    Q:Integer;        {číslo kvadrantu}
    Barva:TColor;     {kódy barev z jednotky Graphics}
    Znacka: Char;
    Popis:string[3]   {popis bodu (max. 3 znaky)};
  end;
```

Oba následující programy *Body1* a *Body2* vyjadřují stejný algoritmus a liší se jen formou zápisu. Program *Body1* je napsán bez užití příkazu **with**. V programu *Body2* je příkaz **with** použit dvakrát a výhody, které tento příkaz poskytuje, pak názorně demonstruje zejména jeho první užití.

Pro počty bodů v jednotlivých kvadrantech je v obou programech použito pole počítadel, kterým zde je celočíselné pole *NQ*.

Námět pro samostatnou práci: Vytvořte požadovaný program bez užití pole záznamů, tj. např. s původně navrženými čtyřmi poli, a porovnejte s následujícími programy *Body1* a *Body2*, zejména pak s programem *Body2*.

```
program Body1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  TBod=record
    X,Y,V0:Real;
    Q:Integer;
  end;
var
  B:array[1..100] of TBod;
  I,N:Integer;
  S,V0P:Real;
  NQ:array[1..4] of Integer;
begin
  for I:=1 to 4 do
    NQ[I]:=0;
  S:=0;
  ReadLn(N);
  for I:=1 to N do
  begin
    ReadLn(B[I].X,B[I].Y);
    if B[I].Y>=0 then
      if B[I].X>=0 then
        B[I].Q:=1
      else
        B[I].Q:=2
    else
      if B[I].X<=0 then
        B[I].Q:=3
      else
        B[I].Q:=4;
    NQ[B[I].Q]:=NQ[B[I].Q]+1;
    B[I].V0:=sqrt(sqr(B[I].X)
             +sqr(B[I].Y));
    S:=S+B[I].V0;
  end;
  V0P:=S/N;
  for I:=1 to 4 do
    WriteLn(NQ[I]);
  WriteLn;
  for I:=1 to N do
    if B[I].V0>V0P then
      WriteLn(B[I].X,B[I].Y,
              ' ',B[I].Q);
  ReadLn;
end.
```

```
program Body2;
~~~
  ~~~
  ReadLn(N);
  for I:=1 to N do
    with B[I] do
    begin
      ReadLn(X,Y);
      if Y>=0 then
        if X>=0 then
          Q:=1
        else
          Q:=2
      else
        if X<=0 then
          Q:=3
        else
          Q:=4;
      NQ[Q]:=NQ[Q]+1;
      V0:=sqrt(sqr(X)+sqr(Y));
      S:=S+V0;
    end;
  V0P:=S/N;
  for I:=1 to 4 do
    WriteLn(NQ[I]);
  WriteLn;
  for I:=1 to N do
    with B[I] do
      if V0>V0P then
        WriteLn(X,Y,'  ',Q);
  ReadLn;
end.
```

Úlohy k samostatnému řešení

Příklad 2.12.3. Napište část programu, která zajistí třídění prvků pole *Závodník* z př. 2.12.1. tak:

a) aby závodníci byli seřazeni podle abecedy,

b) aby závodníci byli seřazeni podle data narození.

Příklad 2.12.4. Vstupní údaje tvoří neprázdná posloupnost dvojic reálných čísel, které představují souřadnice bodů v rovině, jež jsou různé od počátku. Za posloupností je zařazena dvojice souřadnic počátku. Vytvořte program, podle kterého se nejprve vytisknou počty bodů v jednotlivých kvadrantech a dále se vytisknou souřadnice a číslo kvadrantu těch bodů posloupnosti, které se nacházejí v minimální vzdálenosti od počátku. Řešte za předpokladu, že počet bodů posloupnosti je z intervalu <1, 100).

2.13. Typ množina (*set*)

Výklad

Typ množina (typ *set*) je strukturovaný homogenní statický typ, jehož hodnotami jsou všechny podmnožiny určité ***báze*** (*báze = základní množina*), která musí být definovaná pomocí ordinálního typu zvaného ***bázový typ množiny***.

Bázovým typem množiny může tedy být např. výčtový typ (vyjmenovaný typ) nebo vhodný interval z ordinálního typu (blíže viz nejbližší následující poznámka).

Popis typu set má obecný tvar

```
set of bázový typ
```

kde

bázový typ je ordinální typ.

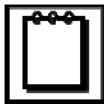
Má-li množina hodnot bázového typu *n* prvků, pak množina hodnot příslušného typu **set** má 2^n prvků, neboť se jedná o potenční množinu báze (***potenční množina báze*** = množina všech podmnožin množiny báze, mezi prvky potenční množiny se počítá i prázdná podmnožina).

Např. všemi podmnožinami tříprvkové množiny [A,B,C] (na rozdíl od matematiky se v Delphi neužívají pro zápis množiny závorky složené, ale hranaté) je následujících 2^3 podmnožin:

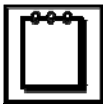
[ ], [A], [B], [C], [A,B], [A,C], [B,C], [A,B,C].

Deklaraci proměnné *P*, která by mohla nabývat (jen) 8 uvedených hodnot, a práci s uvedenými podmnožinami by v Delphi umožnily např. deklarace

```
type
  TBaze = (A,B,C);
  TSet  = set of TBaze;
var
  P:TSet;
```



Poznámka V Delphi jako *bázový typ* nesmí být použit libovolný ordinální typ, *bázový typ* nesmí mít více než 256 možných hodnot a ordinální hodnoty nejnižší a nejvyšší hodnoty typu musí být v rozsahu 0..255.



Řešené úlohy

Příklad: Jsou-li v programu deklarace

```
type
  TDen      = (Po,Ut,St,Ct,Pa,So,Ne);
  TDny      = set of TDen;
  TDnySlevy = set of So..Ne;
var
  Den:TDen;
  DnyOdjezdu,DnyPrijezdu:TDny;
  DnySlevy:TDnySlevy;
```

pak proměnná *DnySlevy* může nabýt právě jedné ze čtyř hodnot: [], [*So*], [*Ne*], [*So,Ne*] (na rozdíl od matematiky se v Delphi neužívají pro zápis množiny závorky složené, ale hranaté).

Mezi 2^7 možných hodnot proměnné *DnyOdjezdu* patří např. hodnoty [*Po*,*Pa*,*So*], [*Po*..*Pa*].

Pro typy **set** s kompatibilními bázovými typy jsou definovány obvyklé množinové a relační operace, označené operátory:

+	-	sjednocení,
*	-	průnik,
-	-	rozdíl,
=	-	rovnost,
<>	-	nerovnost,
<=	-	inkluze „být podmnožinou“,
>=	-	„být nadmnožinou“,
in	-	„být prvkem množiny“ (bázový typ množiny musí být kompatibilní s typem prvku).

V programu, který by obsahoval deklarace z předchozího příkladu, by pak bylo možno pracovat např. s množinami

DnyOdjezdu + *DnyPrijezdu*

DnyOdjezdu * *DnyPrijezdu*

či s relacemi

DnyOdjezdu = DnyPrijezdu

*Po **in** DnyPrijezdu*

a použít např. příkaz:

```
for Den:=Ut to So do
  if Den in DnyPrijezdu then
    ~~~
```

Úlohy k samostatnému řešení

Příklad 2.13.1. Napište program, který zajistí, že po načtení hodnoty do proměnné typu *Char* se zobrazí právě jeden z textů:

a) malé písmeno, velké písmeno, číslice, jiný znak,

b) písmeno (samohláska), písmeno (souhláska), jiný znak.

Příklad 2.13.2. Napište program, který zajistí, že se po načtení znaku do proměnné typu *Char* zobrazí buď načtený znak, nebo tečka, přičemž tečka se zobrazí jen v případě, že načteným znakem je jeden ze znaků čárka, tečka, dvojtečka, středník, lomítko.

Poznámka

Dílčí algoritmy z předchozích příkladů je možno využít např. v programech pro zpracování textů. K vytváření těchto programů jsou třeba informace z části 2.15. *Typ řetězec (string)*, a příp. i z části 2.14. *Typ soubor (file)*. Dílčí algoritmus z předchozího příkladu je možno dále využít např. v programech, v nichž se načítá časový údaj v hodinách a minutách formou řetězce (opět viz část 2.15. *Typ řetězec (string)*).

2.14. Typ soubor (*file*)

Výklad

Soubor (**vzhledem k operačnímu systému**) je základní logickou jednotkou dat. Obsahem souboru může být např. program nebo libovolný text či data pro aplikační program. Každý soubor je nějakým způsobem fyzicky realizován (proto bude dále označován termínem ***fyzický soubor***), může být např. uložen na pevném disku a jeho identifikace (plné určení) je dána nejen jeho jménem, ale i označením disku a cesty do příslušného adresáře.

Soubor (**vzhledem k Delphi**), dále jen ***soubor*** je homogenní dynamický strukturovaný údaj (viz 1.4. *Strukturovaný údaj*), který je tvořen posloupností složek stejného typu a pro nějž jsou definovány operace, které umožňují jeho **sekvenční zpracování**, tj. postupné vytváření nebo čtení.

Všechny soubory je tedy možno zpracovávat sekvenčně, u některých souborů je však možno pracovat s jejich složkami v libovolném pořadí, rozlišují se proto ***soubory sekvenční*** a ***soubory s přímým přístupem***. Soubory se obvykle používají pro vstup a výstup rozsáhlejších dat při realizaci programu. Na rozdíl od jednorozměrného pole, které je rovněž posloupností složek stejného typu, však počet složek souboru není omezen.

V programu se soubory, podobně jako ostatní strukturované údaje, označují identifikátorem, který musí být deklarován jako ***proměnná typu soubor*** (***souborová proměnná***). Na proměnnou typu soubor je možno zjednodušeně pohlížet jako na krátkou přezdívku, kterou se v programu nahrazuje rozsáhlá identifikace diskového souboru. Deklarace proměnné typu soubor však rovněž obsahuje i informaci o typu složek souboru, často např. bývají složky souboru určitého typu záznam (soubor je pak tvořen jen posloupností záznamů uvedeného typu). Časté jsou rovněž soubory, jejichž složky jsou typu *Char*, tj. soubory, které jsou tvořeny posloupností znaků vnější abecedy.

Pro deklaraci souborové proměnné souboru (krátce *při deklaraci souboru*) tvořeného znaky je v Delphi k dispozici standardní identifikátor *TextFile*. Speciálními znakovými soubory typu *TextFile* jsou v Delphi i vstupní soubor znaků dodávaných obvykle z klávesnice (soubor s identifikátorem *Input*) a výstupní soubor znaků vystupujících obvykle na monitor (identifikátor *Output*). I tyto speciální soubory (vstup z klávesnice, výstup na monitor) mají fyzickou realizaci a patří tedy mezi fyzické soubory.

Při realizaci programu musí být každé proměnné typu soubor přiřazen fyzický soubor, tj. např. soubor na disku nebo některé z uvedených periferních zařízení.

Rozdělení souborů

Soubor:

typový - soubor s explicitním udáním typu složek, při deklaraci typu typového souboru se používají klíčová slova **file of** ,

textový - soubor standardního typu *TextFile* , jenž je určen pro soubory, které mají složky typu *Char* a které se dělí na řádky (řádky mohou mít nestejnou délku),

netypový - soubor bez udání typu složek (při deklaraci typu souboru jen klíčové slovo **file**).

Typové soubory

Popis typu typového souboru má tvar

`file of` *typ složky*

kde

typ složky je libovolný typ s výjimkou takového typu, který sám je opět typu soubor nebo který obsahuje (na libovolné úrovni strukturalizace) složku typu soubor.

Řešené úlohy

Příklad : Deklarujte proměnnou typu soubor, jehož složky budou záznamy o studentech, přičemž tyto záznamy budou obsahovat mj. jméno studenta a ročník studia (k využití požadované proměnné v programu jsou třeba prostředky pro práci se soubory – viz dále v této části).

Řešení:

```
type
  TStudent = record
    Jmeno : string[40];
    Rocnik: Integer;
    ~~~
  end;
var
  S: file of Student;
  ~~~
```

Textový soubor

Výklad

Textový soubor je soubor deklarovaný pomocí standardního identifikátoru *TextFile*, textový soubor je tvořen znaky a je členěn do řádků, jež mohou mít nestejnou délku.

Soubory typu *TextFile* mají mnoho společných vlastností se soubory typu ***file of*** *Char*, každý z uvedených typů má však i své specifické vlastnosti. Typ ***file of*** *Char* je typem souboru s přímým přístupem, typ *TextFile* je sice jen typem sekvenčním, má však zase k dispozici některé prostředky, které využívají členění souboru na řádky.

Textový soubor se od všech ostatních souborů liší podstatně nejen v tom, že může být zpracováván jen sekvenčně (tj. bez přímého přístupu), ale i v tom, že nemůže být používán současně jako vstupní i výstupní, tj. nemůže být současně otevřen pro vstup i výstup.

Mezi textové soubory patří již dříve zmíněné soubory *Input* a *Output* (viz 2.6. *Příkazy vstupu a výstupu*), které se od ostatních textových souborů liší nejen tím, že jsou již deklarovány a přiřazeny určitým fyzickým souborům (např. soubor *Input* je obvykle přiřazen klávesnici), ale i tím, že jsou při realizaci programu automaticky otevírány a zavírány a že je není třeba výslovně uvádět např. v příkazech vstupu a výstupu (blíže viz poznámka před neřešenými příklady).

Některé standardní procedury a funkce pro práci se soubory

Značení: *soubor* - proměnná typu soubor,
soubor-T - proměnná typu *TextFile*,
soubor-F - proměnná typu typový soubor,
specifikace - výraz řetězcového typu, který udává přípustnou specifikaci diskového souboru.

Assign (*soubor*, *specifikace*) - procedura, která přiřadí proměnné typu soubor specifikaci diskového souboru.

Rewrite (*soubor*) - procedura pro založení a otevření nového souboru (přiřazeného proměnné *soubor*) pro zapisování (bezprostředně po otevření je soubor vždy prázdný).

Reset (*soubor*) - procedura pro otevření existujícího souboru (textový soubor se otevře jen pro čtení).

Read (*soubor-T* , *seznam*) ,

ReadLn (*soubor-T* , *seznam*) - procedury pro čtení z textového souboru, *seznam* je posloupnost proměnných, do nichž má být načtena hodnota (blíže viz v 2.6. *Příkazy vstupu a výstupu* procedury *Read* a *ReadLn*).

Read (*soubor-F*, *seznam*) - procedura pro čtení ze souboru *soubor-F* , *seznam* je posloupnost proměnných, které jsou stejného typu jako prvky souboru *soubor-F* .

Write (*soubor-T*, *seznam*) ,

WriteLn(*soubor-T*, *seznam*) - procedury pro zápis do textového souboru, *seznam* je posloupnost parametrů (blíže viz 2.6. procedury *Write* a *WriteLn*).

Write (*soubor-F*, *seznam*) - procedura pro zápis do souboru *soubor-F* , *seznam* je posloupnost proměnných, které jsou stejného typu jako prvky souboru *soubor-F* .

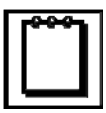
CloseFile (*soubor*) -procedura pro uzavření otevřeného souboru.

Eof (*soubor*) - funkce pro rozpoznání stavu konce souboru, na konci souboru vrací hodnotu *True* , jinak *False* .

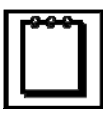
Eoln (*soubor-T*) - funkce pro rozpoznání stavu konce řádku textového souboru, na konci řádku vrací hodnotu *True* , jinak *False* .

V Delphi je mnoho dalších procedur a funkcí pro práci se soubory, pro jejich hledání v Helpu je vhodné znát názvy některých z nich, např.:

- příklady procedur pro všechny typy souborů: *AssignFile*, *ChDir*, *CloseFile*, *GetDir*, *MkDir*, *Read*, *Rename*, *Reset*, *Rewrite*, *RmDir*, *Write*,
- příklady funkcí pro všechny typy souborů: *Eof*, *IOResult*,
- příklady procedur jen pro textové soubory: *Append*, *ReadLn*, *WriteLn*,
- příklady funkcí jen pro textové soubory: *Eoln*, *SeekEof*, *SeekEoln*,
- příklady procedur jen pro soubory, které nejsou textové: *Seek*, *Truncate*,
- funkce jen pro soubory, které nejsou textové: *FilePos*, *FileSize*.



Poznámka



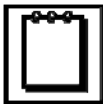
V TP měly některé odpovídající procedury a funkce pro práci se soubory „kratší“ identifikátor (např. *Close* místo *CloseFile*). I když je někdy možno i v Delphi tyto kratší identifikátory použít, není to vhodné, neboť může dojít ke kolizi při použití dalších programových jednotek.

Řešené úlohy

Příklad 2.14.1. V textovém souboru *Cisla.txt*, který se nachází v hlavním adresáři disku *D*, se na počátku každého řádku nachází číslo typu *Real*. Napište program pro zjištění součtu a počtu uvedených čísel a zápis těchto výsledků do souboru *Vysledky.txt*, který bude umístěn v hlavním adresáři disku *A*.

Řešení: V následujícím programu jsou oba soubory (až téměř přehnaně) otevírány co nejpozději a uzavírány co nejdříve.

```
program Soucet;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  X,S:Real;
  N:Integer;
  CS,TS:TextFile;
begin
  AssignFile(CS,'D:\Cisal.txt');
  AssignFile(TS,'A:\Soucet.txt');
  S:=0;
  N:=0;
  Reset(CS);
  while not Eof(CS) do
     {místo Eof(CS) vhodnější SeekEof(CS)}
  begin
    ReadLn(CS,X);
    S:=S+X;
    N:=N+1;
  end;
  CloseFile(CS);
  Rewrite(TS);
  WriteLn(TS,'Soucet = ',S);
  WriteLn(TS,'počet  = ',N);
  CloseFile(TS);
end.
```



Poznámka

Otevírat soubory co nejpozději a uzavírat je co nejdříve je základní a velmi praktická programátorská zásada při práci se soubory.

Textový soubor, např. soubor *Cisla.txt*, nutný pro ladění předchozího programu, je možno vytvořit přímo v prostředí Delphi volbou *File/New.../Text* nebo prostřednictvím kteréhokoliv

textového editoru, který do souboru ukládá jen psané znaky, ve Windows je to např. *Poznámkový blok*, ne již textový editor *Word*.

V následující ukázce možného tvaru souboru *Cisla.txt* jsou použity znaky · a ¶, které používá Word, první pro znázornění mezery, druhý pro ukončení odstavce (znak ¶ odpovídá stisku klávesy Enter):

```
1.2····¶
3.4··B··90··¶
```

Pokud by soubor *Cisla.txt* měl přesně uvedený tvar, bylo by při užití funkce *Eof* první přečtené číslo 1.2, druhé 3.4 a (***pozor na jistou záludnost***) třetí 0.0. Příkazem *ReadLn(CS,X)* se sice z druhého řádku přečte číslo 3.4, přeskočí se všechny následující znaky až do konce řádku (včetně), ale za znakem pro odřádkování se nachází ještě další neviditelný znak pro konec souboru. Pokud by druhý řádek nekončil odřádkováním nebo by byla použita funkce *SeekEof*, byly by výsledky správné, blíže viz Help (viz též Help k funkcím *Eoln* a *SeekEoln*).

Poznámka

Následující dva řádky mají stejný význam

```
Write(Output,'x = '); ReadLn(Input,X); WriteLn(Output);
Write('x = ');        ReadLn(X);       WriteLn;
```

neboť v procedurách *Read*, *ReadLn*, *Write* a *WriteLn* není třeba standardní textové soubory *Input* a *Output* uvádět.

Úlohy k samostatnému řešení

U všech příkladů z předchozích částí je možno vstup z klávesnice předělat na vstup ze souboru, vhodné je to zejména v případě, že vstupní údaje jsou rozsáhlejší.

Příklad 2.14.2. V každém řádku textového souboru *Matice1.txt* jsou uvedeny prvky jednoho řádku reálné matice A typu (M,N), kde M, N jsou z intervalu <2, 5>. Vytvořte program pro výpočet matice $A \cdot A^T$ a její uložení do textového souboru *Matice2.txt*.

Řešte nejprve obdobnou (jednodušší) úlohu pro případ, že matice A je jen čtvercová (počtem prvků v prvním řádku souboru *Matice1.txt* je totiž určen i druhý rozměr matice).

2.15. Typ řetězec (*string*)

Předpokládané znalosti

Typ řetězec je v Delphi standardním strukturovaným homogenním typem (viz část 1.4. *Strukturovaný údaj*), jehož prvky jsou znaky, tj. jsou typu *Char* (informace o typu *Char* a část úvodních informací o typu řetězec je již v části 2.3. *Datové typy* v oddílech *Typ Char* a *Typ string (typ řetězec)*).

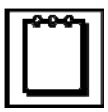
Výklad

V Delphi je možno pracovat s řetězcovými konstantami i řetězcovými proměnnými dvou typů, první typ je převzatý z jazyka Turbo-Pascal, druhý z jazyka C. Pro první typ, pro nějž je možno v Delphi používat též identifikátor *ShortString*, je charakteristická pevná délka části paměti pro jeho uložení (implicitně a současně maximálně 255 znaků, tj. z hlediska uložení jde o statický typ), druhý typ (dynamicky uložený, v Delphi též identifikátor *AnsiString*) může mít „téměř“ neomezenou délku (používá se pro něj termín *dlouhé řetězce*).

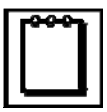
Typ řetězec je v Delphi jedním ze dvou standardních strukturovaných homogenních typů *ShortString* a *AnsiString*, jejichž prvky jsou znaky, přičemž u typu *ShortString* může mít řetězec maximálně 255 znaků, zatímco u typu *AnsiString* může mít řetězec „téměř“ neomezený počet znaků.

Počet znaků řetězce se nazývá **délka řetězce**. K uložení každého znaku řetězce je třeba jeden ***byte***.

Byte (čti *bajt*, značka *B*) je nejmenší přímo adresovatelná jednotka paměti. Obsah jednoho *byte* může nabývat právě jednoho z 2^8 stavů ($2^8 = 256$), tj. může tu být uloženo např. právě jedno z celých čísel 0 až 255, které představuje kód znaku ASCII.



Poznámka



Další text bude orientován jen na řetězcový typ převzatý z jazyka Turbo-Pascal, tj. na typ označovaný v Delphi též *ShortString*. Má-li mít v programu klíčové slovo **string** zcela stejný význam jako identifikátor *ShortString*, je třeba použít direktivu *{$H-}* (viz 2.19. *Direktivy překladače*).

V jazyku Turbo-Pascal může být řetězec tvořen posloupností 0 až 255 znaků, které se v paměti ukládají na sousedních *byte*. Je-li hodnotou řetězcové proměnné řetězec délky *n*, je uložen na *n*+1 sousedních *byte* (označovaných jako 0-tý až *n*-tý *byte*), přičemž v 0-tém *byte* uložení se nachází údaj o aktuální délce řetězce. Údaj o aktuální délce řetězce poskytuje rovněž funkce *Length* (viz dále).

Řetězec délky 0 se nazývá **prázdný řetězec**.

Vzhledem k tomu, že často bývají hodnotami některé řetězcové proměnné jen řetězce délky podstatně menší než 255, je možno (a je vhodné) deklarovat maximální možnou délku řetězcové proměnné v rozsahu od 1 znaku do 255 znaků a hospodárněji tak využívat paměť počítače.

Popis typu řetězec má jeden ze dvou tvarů

```
string
string [maximální délka řetězce]
```

kde

maximální délka řetězce je přirozené číslo od 1 do 255, přičemž tvar **string** má stejný význam jako

```
string [255]
```

Jednotlivé znaky řetězcové proměnné jsou přístupné pomocí indexu obdobně jako složky jednorozměrného pole. Aktuální délka řetězce (typu *ShortString*) se při běžné práci s řetězcovou proměnnou „automaticky“ zapisuje do 0-tého *byte* uložení formou znaku, jehož kód se rovná aktuální délce řetězce. K získání aktuální délky řetězce (jakéhokoliv typu) je však vhodnější používat funkci *Length*.

Některé procedury a funkce pro práci s řetězci

Copy (*S*, *I*, *K*) **function** Copy (*S*: **string**; *I*, *K*: Integer): **string**

Funkce *Copy* vrací *K* znaků z řetězce *S* , a to od *I*-tého znaku řetězce *S*.

Delete (*S*, *I*, *K*) **procedure** Delete (**var** *S*: **string**; *I*, *K*: Integer)

Procedura *Delete* odstraní *K* znaků z řetězcové proměnné *S* , a to od *I*-tého znaku proměnné *S*.

Insert (S_1, S, I) **procedure** Insert (S_1: **string**; **var** S: **string**; I: Integer)

Procedura *Insert* vkládá řetězec S_1 do řetězcové proměnné S na pozici I-tou (od pozice I-té) .

Length (S) **function** Length (S: **string**): Integer

Výsledkem funkce *Length* je skutečná délka řetězce S.

Pos (S_1, S) **function** Pos (S_1, S: **string**): Integer

Funkce *Pos* hledá podřetězec S_1 v řetězci S a jako výsledek vrací buď pozici prvního znaku řetězce S_1 v S, nebo nulu (není-li řetězec S_1 v S obsažen).

Str (P, S) **procedure** Str (P; **var** S: **string**)

Procedura *Str* vytváří řetězcové zobrazení S číselné hodnoty P (význam symbolu P viz příkaz *Write* - část 2.6. *Příkazy vstupu a výstupu – základní informace*).

Val (S, P, K) **procedure** Val (S:**string**; **var** P; **var** K: Integer)

Procedura *Val* převádí řetězec S na číselnou hodnotu a ukládá ji do proměnné P celočíselného nebo reálného typu (lze-li převod realizovat, pak K=0, jinak K≠0).

Řešené úlohy

Příklad Napište část programu pro zkrácení řetězce *S* na 40 (prvních) znaků, resp. pro jeho prodloužení na 40 znaků pomocí mezer.

Řešení: V následujícím řešení se na základě zjištěné výchozí délky řetězce provádí buď jeho prodloužení, nebo zkrácení:

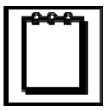
```
var
  S:string;
  I,K:Integer;
begin
  ~~~
  K:=Length(S);
  if K <= 40 then
    for I:=K+1 to 40 do
      S:=S+' '
  else
    S:=Copy(S,1,40);
  ~~~
```

Vzhledem k významu (realizaci) funkce *Copy* a příkazu **for**, je možno uvedený, poměrně názorný zápis příkazů nahradit následujícím kratším zápisem:

```
~~~
S:=Copy(S,1,40);
for I:=Length(S)+1 to 40 do
  S:=S+' ';
~~~
```

Nebo jen zápisem:

```
~~~
S:=S+'                                        ';
S:=Copy(S,1,40);
~~~
```



Poznámka

Vzhledem k terminologii z části 1.4. *Strukturovaný údaj* je řetězec typu *AnsiString* údajem ryze dynamickým (a samozřejmě homogenním), zatímco řetězec typu *ShortString* je vzhledem ke způsobu uložení v paměti počítače údajem statickým (a rovněž samozřejmě homogenním), i když práce s ním (pokud postačí deklarovaná délka jeho uložení) má z hlediska algoritmizace často charakter práce s údajem dynamickým.

Úlohy k samostatnému řešení

Za *obvyklý zápis jména a příjmení* do řetězcové proměnné bude v následujících příkladech považován zápis, který začíná prvním znakem jména (ne mezerou), mezi jménem a příjmením je jediná mezera a zápis končí posledním znakem příjmení, přičemž jen první písmena jména a příjmení jsou velká a (zatím) všechna písmena s diakritikou (národní akcentované znaky) jsou nahrazena příslušnými písmeny anglické abecedy (tj. např. místo *Č* písmeno *C*), zápis „českého“ písmena *ch* se nemění.

Za *běžný zápis časového údaje*, tj. časového údaje z intervalu <0, 24) hodin, který by měl být v celých hodinách a celých minutách, bude v následujících příkladech považován řetězec, v němž hodiny i minuty mohou být zapsány jednou nebo dvěma ciframi, přičemž může být použit kterýkoliv z běžných oddělovačů hodin a minut (dvojtečka, tečka, čárka, středník, lomítko), za příslušný *obvyklý zápis* pak bude považován zápis ve formátu *hh:mm* .

U běžného zápisu časového údaje se bude v následujících příkladech rozlišovat, zda je s mezerami nebo bez mezer, tj., zda před nebo za zápisem hodin či minut jsou či nejsou (zbytečné) mezery.

Příklad 2.15.1. Vytvořte program, který přečte jméno a příjmení zapsané obvyklým způsobem do jedné řetězcové proměnné a vytiskne:

a) příslušný monogram,

b) délku (počet znaků) jména a délku příjmení,

c) nejprve příjmení a potom jméno,

d) nejprve příjmení a potom jméno, přičemž celé příjmení bude vypsáno velkými písmeny,

e) kolikrát se v příjmení a jméně vyskytuje české písmeno ch.

Příklad 2.15.2. V řetězcové proměnné je jméno a příjmení, přičemž forma zápisu se od obvyklého zápisu liší jen tím, že všechna písmena jsou velká. Napište část programu, která zajistí převod jména a příjmení do obvyklého zápisu.

Poznámka

V Delphi je k dispozici funkce *UpCase* pro převod malého písmena (anglické abecedy) na příslušné velké písmeno, která má záhlaví:

```
function UpCase(Zn:Char):Char
```

Pokud je znak *Zn* z rozsahu *a* až *z* je výsledkem odpovídající velké písmeno, jinak výsledkem zůstává nezměněný znak *Zn*. Pro opačný převod v Delphi zatím funkce není, tento převod lze však lehce zajistit výměnou původního znaku za znak s příslušným kódem pomocí funkcí *Ord* a *Chr* (viz část 2.3. *Datové typy / Typ Char*). Malé písmeno z rozsahu *a* až *z* má kód o 32 (tj. o 2^5) větší než odpovídající velké písmeno, např. tedy

```
Chr(Ord('A')+32) = 'a'
```

Příklad 2.15.3. Vytvořte program, podle kterého načte řetězec, a je-li řetězcem běžný časový údaj bez mezer, vytiskne se v obvyklém formátu, jinak se vytiskne vhodné varovné hlášení, např. řetězec *hh:mm* (využijte případného řešení př. 2.13.2.).

Příklad 2.15.4. Vytvořte program, podle kterého načte řetězec, a je-li řetězcem běžný časový údaj s mezerami, vytiskne se v obvyklém formátu, jinak se vytiskne vhodné varovné hlášení.

2.16. Procedury a funkce

Předpokládané znalosti

V Delphi může být jakýkoliv dílčí algoritmus deklarován z hlediska algoritmizace jako *procedura* (viz část 1.6. *Procedury*). Při *deklaraci procedury* je příslušný dílčí algoritmus pojmenován, jsou mu přiděleny případné *formální parametry* a v předepsané formě je zapsán do deklarační části. Požadavek na každé provedení algoritmu procedury se pak může kdekoliv v oboru platnosti deklarace procedury vyjádřit jejím *voláním*, tj. uvedením jména procedury a příslušných *skutečných parametrů*.

Výklad

Termín *procedura* se však v Delphi užívá v poněkud jiném významu než v algoritmizaci. Jelikož v dalším textu bude užívána terminologie obvyklá pro Delphi, budou v následujícím rozdělení v závorkách uvedeny pro srovnání příslušné termíny z části 1.6. *Procedury*.

Podprogramy (procedury) se dělí na:

- **funkce** (funkční procedury),
- **procedury** (obecné procedury).

Deklarací funkce získá uživatel možnost používat tuto funkci obdobně jako funkci standardní. V následující ukázce bude deklarováno několik matematických funkcí, které v Delphi „chybí“ mezi funkcemi standardními. Nejprve bude deklarována funkce x^3, potom funkce *n!*, dále pak bude následovat *n*-tá odmocnina z *x*, a na závěr funkce *Kostka*, která při každém volání poskytne náhodně některé z celých čísel 1 až 6 (obdobně jako při hodu hrací kostkou).

```
program UkazkaFunkci;
~~~
function Moc3(X:Real):Real;
  begin
    Moc3:=X*Sqr(X);
  end;
function Fakt(N:Integer):Integer;
  var
    I,NF:Integer;
  begin
    NF:=1;
```

```
    for I:=1 to N do
      NF:=NF*I;
    Fakt:=NF;
  end;
function Odmoc(N:Integer;X:Real):Real;
  begin
    if X=0 then
      Odmoc:=0
    else
      Odmoc:=Exp(1/N*Ln(X));
  end;
function Kostka:Integer;
  begin
    Kostka:=Random(6)+1;
  end;
var
  X,Y:Real;
begin
  WriteLn(Moc3(5.1));
  WriteLn(Fakt(4));
  WriteLn(Odmoc(2,3));
  WriteLn(Odmoc(3,2));
  WriteLn(Kostka);
  ~~~
  if Y>=0 then
    WriteLn(Odmoc(4,Y))
  else
     WriteLn(' --- ');
  ~~~
end.
```

Deklarace funkce je tvořena záhlavím, deklarační částí a příkazovou částí. Záhlaví začíná klíčovým slovem **function** a názvem (identifikátorem) funkce. Záhlaví dále obsahuje informaci o tom, jaké má funkce parametry (kolik a jakého typu) a jakého typu je výsledek (jedna výsledná hodnota). V Delphi však výsledek funkce nemůže být libovolného typu, ze strukturovaných typů to může být jen typ *řetězec*. Typem výsledku funkce tedy může být např. libovolný jednoduchý typ.

V příkazové části funkce (tj. mezi **begin** a **end** funkce) je třeba (alespoň jednou) přiřadit identifikátoru funkce, resp. standardnímu identifikátoru *Result*, nějakou hodnotu. „Poslední“ hodnota přiřazená identifikátoru funkce, resp. identifikátoru *Result*, pak představuje výslednou funkční hodnotu.

V deklaracích uvedených funkcí není zohledněn jejich definiční obor, tzn., že požadavek na užití funkce mimo její definiční obor by vyvolal chybové hlášení obdobně jako např. u standardní funkce *Sqrt*.

Řešené úlohy

Příklad 2.16.1. Sestavte program, podle kterého se pro danou čtveřici třísložkových aritmetických vektorů $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$ vypočte vektor $(\boldsymbol{a} \times \boldsymbol{b}) \times (\boldsymbol{c} \times \boldsymbol{d})$.

Řešení: K získání požadovaného výsledku je třeba po načtení čtyř zadaných vektorů třikrát počítat vektorový součin, nejprve např. $\boldsymbol{u} = \boldsymbol{a} \times \boldsymbol{b}$, pak $\boldsymbol{v} = \boldsymbol{c} \times \boldsymbol{d}$, a na závěr $\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v}$. Osnovu programu by tedy bylo možno vyjádřit např. následujícím způsobem:

1. čtení vektoru **a**
2. čtení vektoru **b**
3. čtení vektoru **c**
4. čtení vektoru **d**
5. **a** × **b** → **u**
6. **c** × **d** → **v**
7. **u** × **v** → **w**
8. tisk vektoru **w**

I v Delphi však lze zavedením vhodných prostředků (např. zavedením, tj. deklarací, procedur *Čtení*, *Součin* a *Tisk*) dosáhnout toho, že příkazová část požadovaného programu bude mít tvar:

```
begin
  Cteni(A);
  Cteni(B);
  Cteni(C);
  Cteni(D);
  Soucin(A,B,U);
  Soucin(C,D,V);
  Soucin(U,V,W);
  Tisk(W);
  ~~~
end.
```

tj. tvar, který se uvedené osnově bude velmi podobat. Z hlediska uživatele je totiž možno zjednodušeně pohlížet na deklaraci procedury jako na prostředek pro zavedení nového vlastního příkazu.

Např. pro výpočet vektorových součinů je zde použit příkaz *Součin*, který má tři parametry. Při výpočtu vektorového součinu je totiž třeba zadat, ze kterých vektorů se má součin počítat. Na rozdíl od matematiky, je však třeba též zadat, kam se má uložit výsledek. Dále, na rozdíl od předchozí osnovy, však příslušný příkaz pro výpočet součinu v Delphi ztrácí podobu přiřazovacího příkazu. Výsledkem

vektorového součinu je totiž pole, a to již v Delphi nemůže být výsledkem funkce.

Výklad

Každý dílčí algoritmus, ať již je jeho výsledkem jedna jednoduchá hodnota nebo více jednoduchých hodnot či strukturovaných hodnot, může být v Delphi naprogramován formou procedury. Výsledky procedury se pak obvykle předávají rovněž prostřednictvím parametrů, a z hlediska algoritmizace (tj. podle směru předávaných hodnot) se proto rozlišují ***parametry vstupní***, ***parametry výstupní*** a ***parametry vstupně-výstupní***.

Např. při zavádění příkazu *Součin* (při deklaraci procedury *Součin*) z předchozího příkladu bude tedy třeba mj. předepsat následující skutečnosti:

- jaký je název procedury,
- jaké jsou parametry procedury (kolik jich je, jak budou nazývány, jakého jsou typu a které z nich jsou vstupní a které výstupní),
- jaký je vlastní algoritmus procedury (konkrétně zde: jak se počítají jednotlivé složky výsledného vektoru).

Pokračování řešení př. 2.16.1.: Záhlaví a deklarační část požadovaného programu by mohly být vytvořeny např. následujícím způsobem:

```
program UkazkaProcedur;
~~~
type
  TVektor=array[1..3] of Real;
procedure Soucin(A,B:TVektor; var C:TVektor);
  begin
    C[1]:=A[2]*B[3]-A[3]*B[2];
    C[2]:=A[3]*B[1]-A[1]*B[3];
    C[3]:=A[1]*B[2]-A[2]*B[1];
  end;
procedure Cteni(var V:TVektor);
  begin
    ReadLn(V[1],V[2],V[3])
  end;
procedure Tisk(X:TVektor);
  begin
    WriteLn(X[1]:8:3,X[2]:8:3,X[3]:8:3)
  end;
var
  A,B,C,D,U,V,W:TVektor;
begin
  ~~~
end.
```

Procedura *Součin* má tři parametry *A*, *B*, *C* typu *TVektor*. První dva parametry jsou vstupní, třetí je výstupní (výstupnímu parametru musí předcházet klíčové slovo **var**).

Procedura popisuje, jak se vypočte vektorový součin nějakých fiktivních (pomyslných, formálních) vektorů ***a*** a ***b*** s tím, že výsledek je uložen do nějakého fiktivního (formálního) vektoru ***c***. V příkazové části programu je pak tato procedura použita (volána) nejprve pro vektory ***a***, ***b***, ***u***, pak pro vektory ***c***, ***d***, ***v*** a na závěr pro vektory ***u***, ***v***, ***w***.

Při deklaraci parametrů procedury není možno přímo použít popis typu ***array****[1..3]* ***of*** *Real*, ale je nutno předem tento typ deklarovat (viz deklaraci typu *TVektor*).

Deklarace procedury *Čtení* popisuje, jak se načítá „jakýsi“ fiktivní (pomyslný, formální) třísložkový vektor ***v***. V příkazové části programu je pak procedura *Čtení* použita (volána) postupně pro konkrétní (skutečné) vektory ***a***, ***b***, ***c***, ***d***.

Deklarace podprogramu má obecně tři části, kterými jsou:

1. ***záhlaví***,
2. ***deklarační část***,
3. ***příkazová část***.

Podprogram má tedy stavbu analogickou se stavbou programu. Posloupnost deklarační části a příkazové části podprogramu se nazývá ***tělo podprogramu***. Deklarační část podprogramu může být i prázdná, příkazová má vždy tvar složeného příkazu.

Záhlaví podprogramu:

- dává podprogramu jméno (jménem je identifikátor),
- určuje, zda jde o proceduru či o funkci, a v případě funkce též jaký je typ funkční hodnoty,
- specifikuje ***formální parametry***, tj. fiktivní objekty, které se při volání podprogramu nahrazují konkrétními objekty (***skutečnými parametry***).

Rozdělení identifikátorů v podprogramu: Identifikátory, které se vyskytují v záhlaví podprogramu, v příkazové části podprogramu a na základní úrovni v deklarační části podprogramu, kromě identifikátoru tohoto podprogramu, se dělí na:

- **lokální identifikátory** (jsou deklarovány na základní úrovni deklarační části podprogramu nebo to jsou identifikátory formálních parametrů),
- **nelokální identifikátory** (nejsou lokální a jejich deklarace by měla předcházet deklaraci podprogramu – přesněji viz dále oddíly *Bloková struktura programu* a *Rozsah použitelnosti deklarovaných objektů*).

Lokální identifikátor „zastiňuje“ v těle podprogramu případné další významy téhož identifikátoru deklarované vně podprogramu.

Realizace volání podprogramu: Při volání podprogramu se za formální parametry dosadí (způsobem předepsaným v deklaraci) skutečné parametry, a teprve potom se provede algoritmus podprogramu.

Počet skutečných parametrů se musí shodovat s počtem formálních parametrů. Každý skutečný parametr musí být přípustným skutečným parametrem k odpovídajícímu formálnímu parametru.

Přiřazení skutečných parametrů k parametrům formálním je dáno jen jejich pořadím, tj. nezávisí na jejich označení.

Záhlaví procedury má jeden ze dvou tvarů:

```
procedure IP
procedure IP (SFP ; SFP ; ~~~ ; SFP)
```

kde

IP je identifikátor procedury (*jméno procedury*),

SFP je ***specifikace formálních parametrů***.

Záhlaví funkce má jeden ze dvou tvarů:

```
function IdF:ITFH
function IdF(SFP ; SFP ; ~~~ ; SFP):ITFH
```

kde

IdF je identifikátor funkce (*jméno funkce*),

ITFH je identifikátor typu funkční hodnoty (jednoduchý typ nebo typ řetězec či ukazatel),

SFP je ***specifikace formálních parametrů***.

Volání procedury i funkce má jeden z dvou tvarů

```
identifikátor podprogramu
identifikátor podprogramu (seznam skutečných parametrů)
```

rozdílný je však význam volání:

- volání procedury představuje příkaz (příkaz procedury),
- volání funkce představuje výraz.

Formální parametry podprogramů lze rozdělit:

- **z hlediska algoritmizace** (tj. podle směru předávaných hodnot) na:
  - ***parametry vstupní***,
  - ***parametry výstupní***,
  - ***parametry vstupně-výstupní***,
- **z hlediska Delphi** na:
  - ***parametry volané hodnotou***,
  - ***parametry volané odkazem*** (vztahuje se na ně klíčové slovo **var**).

Formální parametr volaný hodnotou představuje v těle podprogramu jen lokální proměnnou, které je na počátku provádění podprogramu přiřazena hodnota skutečného parametru. Přípustným skutečným parametrem může být proto libovolný výraz, jehož hodnota je kompatibilní vzhledem k přiřazení s typem formálního parametru. Případné změny hodnoty formálního parametru při provádění podprogramu nemění hodnotu skutečného parametru.

Formální parametr volaný odkazem představuje v těle podprogramu vždy tu konkrétní proměnnou, která je určena skutečným parametrem. Přípustným skutečným parametrem proto může být obecně jen proměnná, jejíž typ je totožný s typem formálního parametru (jisté výjimky se v Delphi připouští např. pro typ *řetězec*).

Parametry volané hodnotou proto reprezentují v podprogramu jeho vstupní hodnoty, kdežto parametry volané odkazem reprezentují v podprogramu obvykle výstupní nebo vstupně-výstupní proměnné, tj. proměnné, jejichž hodnoty budou provedením podprogramu definovány.

V některých případech však lze i vstupní parametry volat odkazem, kdežto parametry výstupní nebo vstupně-výstupní je třeba volat odkazem vždy.

Specifikace formálních parametrů volaných hodnotou má tvar

IFP, *IFP*, ~~~, *IFP* : *ITFP*

kde

IFP je identifikátor formálního parametru,

ITFP je identifikátor typu formálních parametrů.

Specifikace formálních parametrů volaných odkazem má tvar

var *IFP*, *IFP*, ~~~, *IFP* : *ITFP*

kde význam symbolů *IFP* a *ITFP* je stejný jako u předchozí specifikace.

Seznam skutečných parametrů má tvar

SP, *SP*, ~~~, *SP*

kde *SP* je skutečný parametr (oddělovačem skutečných parametrů je vždy čárka).

Řešené úlohy

Příklad 2.16.2. Vstupní údaje: přirozené číslo *n* z intervalu <2, 100> a dále posloupnost *n* reálných čísel z intervalu (-1000, 1000), z nichž alespoň dvě jsou různá. Sestavte program pro výpočet aritmetického průměru těch prvků posloupnosti, které jsou větší než aritmetický průměr všech prvků posloupnosti, a aritmetického průměru těch prvků posloupnosti, které jsou menší než aritmetický průměr všech prvků posloupnosti.

Řešení: Před vlastním výpočtem požadovaných průměrů je třeba vypočítat průměr všech prvků posloupnosti, o kterých je známo, že jsou z intervalu (-1000, 1000). Program lze tedy navrhnout tak, že se bude třikrát počítat aritmetický průměr z těch prvků posloupnosti, které jsou vždy z nějakého vhodného intervalu (*a*,*b*). Pro výpočet tohoto aritmetického průměru, tj. průměru z prvních *n* prvků nějakého pole, jejichž hodnota současně patří do intervalu (*a*,*b*), bude v prvních dvou řešeních použita procedura *Průměr*.

```
program Prumery;
~~~
type
  TPole=array[1..100] of Real;
procedure Prumer(X:TPole;N:Integer;A,B:Real;var P:Real);
  var
    I,K:Integer;
    S:Real;
  begin
    S:=0;
    K:=0;
    for I:=1 to N do
      if (X[I]>A) and (X[I]<B) then
      begin
        S:=S+X[I];
        K:=K+1;
      end;
    P:=S/K
  end {procedure Prumer};
var
  X:TPole;
  I,N:Integer;
  P,PV,PM:Real;
begin
  ReadLn(N);
  for I:=1 to N do
    ReadLn(X[I]);
  Prumer(X,N,-1000,1000,P);
  Prumer(X,N,P    ,1000,PV);
  Prumer(X,N,-1000,P   ,PM);
  WriteLn('Prumer vetsich = ',PV);
  WriteLn('Prumer mensich = ',PM);
  ReadLn;
end.
```

V programu deklarovaná procedura *Průměr* má 5 parametrů. Při všech třech voláních procedury *Průměr* jsou však první dva parametry shodné. S využitím nelokálních proměnných *X* a *N* by tedy bylo možno program upravit např. takto:

```
~~~
type
  TPole=array[1..100] of Real;
var
  X:TPole;
  N:Integer;
procedure Prumer(A,B:Real;var P:Real);
  ~~~
  end {procedure Prumer};
var
  I:Integer;
  P,PV,PM:Real;
begin
  ~~~
  Prumer(-1000,1000,P);
  Prumer(P    ,1000,PV);
  Prumer(-1000,P   ,PM);
  ~~~
```

Vzhledem k tomu, že algoritmus podprogramu *Průměr* slouží jen k získání jediného čísla, je možno v prvně navrženém programu místo procedury *Průměr* deklarovat funkci *Průměr*. Např.

```
~~~
type
  TPole=array[1..100] of Real;
function Prumer(X:TPole;N:Integer;A,B:Real):Real;
  var
    I,K:Integer;
    S:Real;
  begin
    ~~~
    for I:=1 to N do
      ~~~
      end;
    Prumer:=S/K
  end {function Prumer};
var
  X:TPole;
  I,N:Integer;
  P,PV,PM:Real;
begin
  ReadLn(N);
  for I:=1 to N do
    ReadLn(X[I]);
  P :=Prumer(X,N,-1000,1000);
  PV:=Prumer(X,N,P    ,1000);
  PM:=Prumer(X,N,-1000,P   );
  ~~~
```

Poznámka

V předchozích ukázkách deklarací procedur a funkcí byl a i dále bude používán zápis do úrovní (způsob odsazování), který se jen jedním pravidlem liší od formy zápisu ukázek z helpů v Delphi. Pro snadnější pochopení a větší názornost deklarací je tělo podprogramu posunuto vzhledem k záhlaví podprogramu o jednu úroveň vpravo. Smysl uvedeného pravidla, které zavádí další úroveň při zápisu deklarací podprogramů, vynikne zejména, je-li v deklarační části podprogramu deklarován další podprogram (tj. *vnořený podprogram*), z úrovně odsazení je pak více patrná i úroveň vnoření příslušného podprogramu (viz zejména řešení př. 2.16.3.).

Příklad: Má-li záhlaví procedury tvar

```
procedure Ukazka(U,V,X:Real; var Y:Real; Z:Boolean);
```

pak parametry *U*, *V*, *X*, *Z* jsou volány hodnotou a jen parametr *Y* je volán odkazem.

V následující části programu jsou naznačena přípustná i nepřípustná volání procedury *Ukázka*, chybně volané parametry jsou označeny šipkami pomocí komentářů v následujícím řádku.

```
var
  X,Y,Z:Real;
  I:Integer;
  L:Boolean;
begin
  ~~~
  Ukazka(2.0, X, Sqrt(X), X, True);
  Ukazka(2  , I, Abs(I) , Z, L   );
  Ukazka(X  , Y, Sqrt(I), Y, 3<X );
  ~~~
  Ukazka(L  , I, I+1    , I, L   );
        {↑}              {↑}
  Ukazka(X  , Y, Z     , X+Y, Z   );
                          {↑} {↑}
```

Bloková struktura programu

Výklad

Stavba každého podprogramu se shoduje se stavbou programu v tom smyslu, že program i podprogram jsou tvořeny posloupností záhlaví, deklarační části a příkazové části (přičemž deklarační část může být i prázdná). Pro programy i podprogramy se proto zavádí souhrnný název ***blok***.

Typy, konstanty a proměnné deklarované (na základní úrovni) v deklarační části určitého bloku jsou v tomto bloku lokální, tj. jsou dostupné jen v rámci tohoto bloku.

V deklarační části každého bloku, tedy i v deklarační části každého podprogramu, však mohou být deklarovány nejen lokální proměnné, konstanty a typy, ale též lokální podprogramy. Vzhledem k tomu, že blok může obsahovat deklaraci podprogramu, který je opět blokem, může docházet k vnořování bloků do sebe. V tomto smyslu je tedy možno na každý program pohlížet jako na určitou hierarchickou strukturu bloků.

Bloková struktura programu je vyjádření hierarchické struktury všech bloků, které se v tomto programu vyskytují. Úroveň vnoření bloků obecně není omezena, závisí jen na možnostech konkrétního překladače.

Příkladem netriviální blokové struktury je např. řešení následujícího př. 2.16.3.

Rozsah použitelnosti deklarovaných identifikátorů

K vymezení rozsahu použitelnosti deklarovaných identifikátorů (tj. identifikátorů typů, konstant, proměnných a podprogramů) se používají následující pojmy a pravidla vycházející z blokové struktury programu.

Lokální identifikátor: Vzhledem k určitému bloku je identifikátor lokální, je-li deklarován v záhlaví tohoto bloku a není to identifikátor tohoto bloku, nebo je-li deklarován na základní úrovni tohoto bloku, tj. ne v některém bloku vnořeném do uvažovaného bloku. Identifikátor bloku je lokální vzhledem k nejblíže nadřazenému bloku.

Nelokální identifikátor: Vzhledem k určitému bloku je identifikátor nelokální, není-li vzhledem k tomuto bloku lokální a je-li lokální v některém z nadřazených bloků, tj. v některém z bloků, do něhož je uvažovaný blok vnořen, přičemž v bloku programu jsou lokální i identifikátory, které tu „přinášejí“ jednotky ze seznamu jednotek (za klíčovým slovem **uses**).

Vzhledem k určitému bloku se mohou v programu vyskytovat identifikátory, které nejsou (vzhledem k tomuto bloku) ani lokální, ani nelokální. Např. identifikátor, který je vzhledem k nějakému bloku lokální, není vzhledem k blokům nadřazeným tomuto bloku ani lokální, ani nelokální (tyto bloky o jeho existenci „neví“, pro tyto bloky tedy neexistuje).

Globální identifikátor je identifikátor, který je lokální v bloku programu a který je tedy deklarován na základní úrovni v programu (tj. ne v některém vnořeném bloku) nebo se tu „dostává“ prostřednictvím jednotek ze seznamu jednotek.

Zastíněný identifikátor: Shoduje-li se, vzhledem k nějakému bloku, lokální identifikátor s identifikátorem nelokálním, pak v celém tomto bloku (včetně vnořených bloků) je nelokální identifikátor zastíněn, tj. je tu nedostupný (neviditelný).

Rozsah použitelnosti identifikátoru: Identifikátor lze používat jen v tom bloku, v němž je deklarován (a tedy i v podprogramech, které jsou do uvažovaného bloku vnořeny), ale jen na místech, která splňují následující podmínky:

- místo následuje za deklarací identifikátoru,
- nejedná se o místo, v němž je daný identifikátor zastíněn.

Řešené úlohy

Příklad 2.16.3. Vstupní údaje tvoří přirozené číslo k a dále k dvojic přirozených čísel m, n. Sestavte program, podle kterého se pro každou dvojici m, n vytisknou čísla $C_n(m)$ a $V_n(m)$, pokud ovšem existují.

Řešení: V definičních vztazích pro čísla $C_n(m)$ a $V_n(m)$ se vyskytují faktoriály. V požadovaném programu by proto mohla být nejdříve deklarovaná funkce pro výpočet $n!$, a teprve potom (na stejné úrovni, tj. ve stejném bloku) funkce pro výpočet

čísla $V_n(m)$, resp. $V_k(n)$, a dále funkce pro výpočet čísla $C_n(m)$, resp. $C_k(n)$. Aby však bylo možno ukázat bohatší blokovou strukturu, bude na úrovni programu deklarována jen jedna procedura pro výpočet obou požadovaných čísel, přičemž teprve v této proceduře bude deklarována funkce pro výpočet *n!* a funkce pro výpočet $V_k(n)$.

```
program UkazkaBlokoveStruktury;
~~~
var
  I,K,M,N,C,V:Integer;
procedure Cisla(N,K:Integer;var C,V:Integer);
  function Fakt(N:Integer):Integer;
   var
      I,K:Integer;
    begin
      K:=1;
      for I:=1 to N do
        K:=K*I;
      Fakt:=K
    end {function Fakt};
  function Variace(N,K:Integer):Integer;
    begin
      Variace:=Fakt(N) div Fakt(N-K)
    end {function Variace};
  begin
    V:=Variace(N,K);
    C:=V div Fakt(K)
  end {procedure Cisla};
begin {začátek příkazové části programu}
  ReadLn(K);
  for I:= 1 to K do
  begin
    ReadLn(M,N);
    Write(M:10,N:10);
    if M>=N then
    begin
      Cisla(M,N,C,V);
      WriteLn(C:10,V:10)
    end
    else
      WriteLn('---':10,'---':10);
  end {for I};
  ReadLn;
end.
```

Uvedený program byl přednostně navržen tak, aby měl netriviální blokovou strukturu a aby na něm bylo možno dostatečně demonstrovat problematiku rozsahů použitelnosti deklarovaných objektů. Výpočet kombinačního čísla *C* a variačního čísla *V* je proto

v proceduře *Čísla* prováděn pomocí příslušných faktoriálů, což značně omezuje rozsah použitelnosti programu (v Delphi je již 13! > *MaxInt*). Vzhledem k tomu, že v deklarační části procedury *Čísla* je nejprve deklarována funkce *Fakt* pro výpočet faktoriálu, a teprve potom funkce *Variace*, může funkce *Variace* již využívat funkci *Fakt*.

V programu je identifikátor *K* použit ve čtyřech různých významech: $K_{program}$, $K_{Čísla}$, K_{Fakt}, $K_{Variace}$ (označení vychází z bloků, v nichž je příslušný objekt *K* deklarován). Tedy např. v bloku *Čísla* má identifikátor *K* význam $K_{Čísla}$, ovšem s výjimkou bloků *Fakt* a *Variace*, v nichž má jiné významy, tj. v blocích *Fakt* a *Variace* je objekt *K* ve významu $K_{Čísla}$ zastíněn. Identifikátor *N* je v programu rovněž používán ve čtyřech významech, kdežto identifikátor *M* jen v jednom. Pokud by např. v příkazové části bloku *Čísla* byl místo příkazu

```
V:=Variace(N,K)
```

„omylem“ zapsán příkaz

```
V:=Variace(N,M)
```

znamenalo by to

$$\texttt{V:=Variace(}N_{\text{Čísla}}\texttt{,}M_{\text{program}}\texttt{)}$$

V záhlaví procedury *Čísla* a ani v její deklarační části není sice deklarován identifikátor *M* (je tu tedy nelokální), ale je deklarován na úrovni bloku programu, tj. jedná se přímo o globální identifikátor. Vzhledem k tomu, že při volání procedury *Čísla* v příkazové části programu je formální parametr *N* ($N_{Čísla}$) nahrazen skutečným parametrem *M* ($M_{program}$), poskytoval by pak program jako hodnotu „variačního čísla“ vždy *m!*.

V Delphi i v TP, na rozdíl od referenční verze jazyka Pascal, ale nemusí v deklarační části předcházet deklarace proměnných deklaraci podprogramů, což by bylo možno využít k vhodnějšímu (jen naznačenému) řazení řádků uvedeného programu:

```
program UkazkaBlokoveStruktury;
~~~
procedure Cisla(N,K:Integer;var C,V:Integer);
  ~~~
  end {procedure Cisla};
var
  I,K,M,N,C,V:Integer;
begin {začátek příkazové části programu}
  ~~~
end.
```

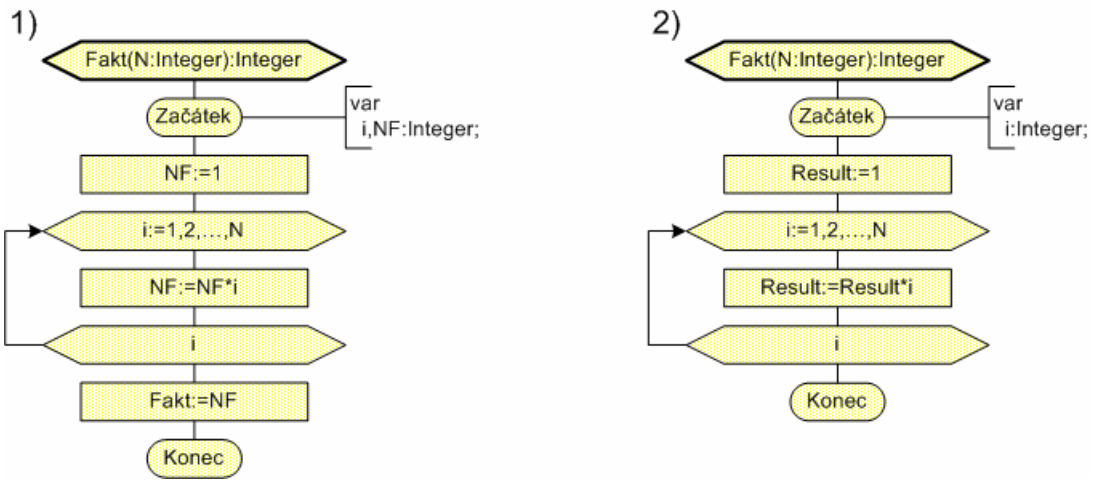
Pokud by v takto zapsaném programu došlo k dříve uvedenému „omylu“, nedošlo by k výpočtu, byla by hlášena chyba již při překladu (při překladu procedury *Čísla* totiž ještě není známa proměnná *M)*.

Jak již bylo uvedeno, předchozí řešení př. 2.16.3. bylo přednostně navrženo tak, aby příslušný program měl netriviální blokovou strukturu a aby na něm bylo možno dostatečně demonstrovat problematiku rozsahů použitelnosti deklarovaných objektů. Praktičtější by ale bylo následující, opět jen naznačené řešení:

```
program UkazkaBlokoveStruktury;
~~~
function Fakt(N:Integer):Integer;
  ~~~
  end {function Fakt};
function Variace(N,K:Integer):Integer;
  begin
    Variace:=Fakt(N) div Fakt(N-K)
  end {function Variace};
function Kombinace(N,K:Integer):Integer;
  begin
    Kombinace:= Variace(N,K) div Fakt(K)
  end {function Variace};
procedure Cisla(N,K:Integer;var C,V:Integer);
  begin
    V:=Variace(N,K);
    C:=Kombinace(N,K)
  end {procedure Cisla};
var
  I,K,M,N,C,V:Integer;
begin {začátek příkazové části programu}
  ~~~
end.
```

Výklad

Již u prvních ukázek funkcí bylo uvedeno, že v příkazové části funkce je třeba (alespoň jednou) přiřadit identifikátoru funkce, resp. standardnímu identifikátoru *Result*, nějakou hodnotu. „Poslední“ hodnota přiřazená identifikátoru funkce, resp. identifikátoru *Result*, pak představuje výslednou funkční hodnotu. Použití speciální proměnné *Result* přibližuje druhá část následující ukázky deklarace funkce pro výpočet *n!*, přičemž pro srovnání je v první části ukázky použito již dříve uvedené řešení (viz př. 1.6.2. a úvodní ukázka z části 2.16.).




```
{1. část ukázky}
function Fakt(N:Integer):Integer;
  var
    I,NF:Integer;
  begin
    NF:=1;
    for I:=1 to N do
      NF:=NF*I;
    Fakt:=NF;
  end;
```

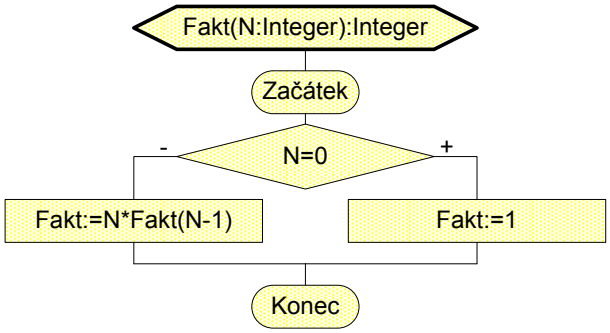
```
{2. část ukázky }
function Fakt(N:Integer):Integer;
  var
    I:Integer;
  begin
    Result:=1;
    for I:=1 to N do
      Result:=Result*I;
  end;
```

V druhé části ukázky odpadá užití pomocné proměnné *NF*, a tedy i poslední přiřazovací příkaz *Fakt:=NF* z první části ukázky. Při pokusu nahradit v druhé části ukázky identifikátor *Result* identifikátorem funkce *Fakt* však dojde k chybovému hlášení. Nevyvolal by je však příkaz *Fakt:=1*, ale až pravá část přiřazovacího příkazu *Fakt:=Fakt*I*. Identifikátor *Fakt* byl totiž již záhlavím podprogramu deklarován jako identifikátor funkce, a proto je na pravé straně zmíněného přiřazovacího příkazu za identifikátorem *Fakt* očekáváno celé volání funkce (tj. i závorka s příslušným skutečným parametrem).

Uvedené omezení vyplývá ze skutečnosti, že v Delphi je umožněna rekurze.

Rekurze znamená, že podprogram volá sám sebe (*rekurzivní volání*), ať již přímo nebo prostřednictvím jiných podprogramů.

K ukázce rekurze poslouží opět deklarace funkce pro výpočet *n!*. Kromě náznakového vztahu *n!=1·2·...·n* lze *n!* definovat rekurentním vztahem $n! = \begin{cases} n \cdot (n-1)! & \text{pro } n > 0, \\ 1 & \text{pro } n = 0. \end{cases}$




```
function Fakt(N:Integer):Integer;
  begin
    if N<=0 then
      Fakt:=1
    else
      Fakt:=N*Fakt(N-1)
  end;
```

V uvedeném vyjádření funkce *Fakt* v Delphi je místo podmínky *n=0* uvedena podmínka *n≤0* . Bez této změny by došlo k problémům, jestliže by funkce byla volána se záporným parametrem, neboť rekurzivní volání by se pak měla teoreticky nekonečněkrát opakovat.

U všech tří předcházejících deklarací funkce pro výpočet *n!* byl použit stejný identifikátor *Fakt*. Z hlediska užití v programu jsou totiž všechny tři deklarace rovnocenné, tj. např. při ladění jinak stejného programu by bylo možno použít kteroukoliv z nich.

Další jednoduché řešené příklady viz část 1.6. *Procedury* př. 1.6.1. až 1.6.4.

Řešené úlohy

Příklad 2.16.4. Deklarujte proceduru, která zajistí seřazení prvních *N* prvků jednorozměrného pole s indexy 1..100 do nerostoucí posloupnosti.

Řešení: Dva algoritmy požadovaného třídění jsou blíže popsány v př. 1.4.4., v následujícím řešení bude v proceduře *Třídění* použit první z nich. Prvky budou v proceduře *Třídění* setříděny přímo v tom poli (bude označeno identifikátorem *A*), ve kterém budou dodány, tzn., že toto pole bude nejen parametrem vstupním, ale i výstupním, tedy parametrem vstupně-výstupním.

V zadání není určen typ prvků pole. Řešení by sice mohlo být provedeno např. jen pro prvky typu *Real*, z metodických důvodů však bude provedeno pro prvky jakéhosi typu *TPrvek*, kterému je pak třeba v programu používajícím proceduru přiřadit konkrétní typ, opět např. typ *Real* nebo třeba ***string****[9]*, pak by se třídilo podle abecedy pole nejvýše 9-ti prvkových řetězců (ale sestupně).

Navržené řešení nebude tentokrát obsahovat jen požadovanou proceduru, ale bude se jednat o celý program (aby bylo možno podprogram ladit, je třeba jej volat ve vhodném programu).

```
program Trid3;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
 {TPrvek=Integer;}
  TPrvek=string[9];
  TPole=array[1..100] of TPrvek;
procedure Trideni(var A:TPole;N:Integer);
  var
    I,J:Integer;
    P:TPrvek;
  begin
    for I:=1 to N-1 do
      for J:=I+1 to N do
        if A[I]<A[J] then
        begin
          P   :=A[I];
          A[I]:=A[J];
          A[J]:=P;
        end;
  end;
var
  A:TPole;
```

```
  I,N:Integer;
begin
  ReadLn(N);
  for I:=1 to N do
    ReadLn(A[I]);
  Trideni(A,N);
  for I:=1 to N do
    Write(A[I]:10);
  ReadLn;
end.
```

Příklad 2.16.5. Deklarujte proceduru:

a) pro nalezení řádkového maxima *K*-tého řádku reálné matice *A* typu (*M*,*N*),

b) pro nalezení řádkových maxim (všech) řádků reálné matice *A* typu (*M*,*N*).

Proceduru použijte v programu pro nalezení řádkových maxim reálné matice *A* typu (*K*,*K*), kde *K* je z intervalu <2, 5> a je to první vstupní údaj, za kterým pak následují prvky matice A v pořadí po sloupcích. Zadanou matici a požadovaná maxima vytiskněte tak, že jednotlivá maxima budou za příslušnými řádky matice.

Řešení: Z metodických důvodů budou v programu deklarovány obě požadované procedury, nazvané postupně *Maximum* a *Maxima*. Požadované výsledky pak budou tištěny celkem třikrát. V části programu nazvané *Řešení A1* bude volána procedura *Maximum* až při tisku (přesněji po tisku) jednotlivých řádků matice, tj. opakovaně se tu bude provádět v jednom cyklu tisk řádku matice a hledání příslušného maxima. V *Řešení B* se nejprve provedou všechny výpočty jediným voláním procedury *Maxima*, uloží se do vektoru *MaxR*, a teprve potom se provedou všechny požadované tisky. V *Řešení A2* se opět nejprve provedou všechny výpočty, ale tentokrát s použitím procedury *Maximum*, která tedy bude muset být volána pro každý řádek zvlášť, tj. opakovaně.

U procedury *Maxima* jsou v obdélnících uvedeny dvě možné varianty jejího těla. Druhá varianta má kratší zápis, neboť maxima na jednotlivých řádcích se tu získávají pomocí volání předcházející procedury *Maximum*.

```
program RadMax;
~~~
type
  TMat=array[1..5,1..5] of Real;
  TVek=array[1..5] of Real;
procedure Maximum(A:TMat;M,N,K:Integer;var Max:Real);
  var
    J:Integer;
  begin
    Max:=A[K,1];
    for J:=1 to N do
      if A[K,J]>Max then
        Max:=A[K,J];
  end;
procedure Maxima(A:TMat;M,N:Integer;var Max:TVek);
```

```
  var
    I,J:Integer;
  begin
    for I:=1 to M do
    begin
      Max[I]:=A[I,1];
      for J:=1 to N do
        if A[I,J]>Max[I] then
          Max[I]:=A[I,J];
    end;
  end;
```

↔

```
var
  I:Integer;
begin
  for I:=1 to M do
    Maximum(A,M,N,I,Max[I]);
end;
```

```
var
  A:TMat;
  I,J,K:Integer;
  Max:Real;
  MaxR:TVek;
begin
  ReadLn(K);
  for I:=1 to K do
    for J:=1 to K do
      ReadLn(A[I,J]);
  WriteLn;
  WriteLn('Reseni A1');
  for I:=1 to K do
  begin
    for J:=1 to K do
      Write(A[I,J]:6:1);
    Maximum(A,K,K,I,Max);
    WriteLn(Max:10:1);
  end;
  WriteLn;
  WriteLn('Reseni B');
  Maxima(A,K,K,MaxR);
  for I:=1 to K do
  begin
    for J:=1 to K do
      Write(A[I,J]:6:1);
    WriteLn(MaxR[I]:10:1);
  end;
  WriteLn;
  WriteLn('Reseni A2');
  for I:=1 to K do
    Maximum(A,K,K,I,MaxR[I]);
  for I:=1 to K do
  begin
    for J:=1 to K do
      Write(A[I,J]:6:1);
    WriteLn(MaxR[I]:10:1);
  end;
  ReadLn;
end.
```

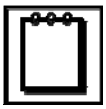
V obou procedurách *Maximum* i *Maxima* je výstupním parametrem proměnná *Max*, přitom jednou se jedná o jednoduchou proměnnou, podruhé o vektor. Ukázka má naznačit, že v různých podprogramech může být stejný identifikátor využit k různým účelům. Vzhledem k tomu, že v uvedeném programu je volána jak procedura *Maximum*, tak i procedura *Maxima*, nemůže mít skutečný parametr, nahrazující formální parametr *Max*, při voláních obou procedur stejné jméno. V *Řešení A1* byl „náhodou" použit opět shodný identifikátor *Max*, v *Řešení B* pak jiný identifikátor *MaxR*. V *Řešení A2* je třeba při opakovaném volání procedury *Maximum* vždy ukládat výsledek pro pozdější použití, v programu byl pro ukládání opět použit vektor *MaxR*, a skutečným parametrem při volání procedury *Maximum* je tu tedy *MaxR[I]*, tj. jméno příslušného prvku vektoru *MaxR*.

V programu je dále v různých významech používán identifikátor *K*, při deklaraci procedury *Maximum* je *K* index řádku, ve kterém se provádí hledání, v příkazové části programu je *K* řád zpracovávané čtvercové matice (u níž se jako řádkový index používá *I*).

Upozornění pro začátečníky: Požadovaný program má sice zpracovávat jen čtvercové matice typu (*K*,*K*), v zadání jsou však požadovány poněkud obecnější procedury pro matice typu (*M*,*N*), tedy procedury, které by byly obecně použitelnější (v jiných programech).

Vzhledem k tomu, že podprogram *Maximum* má jediný výstupní parametr, a to jednoduchého typu, bylo by možno (z cvičných důvodů) místo požadované procedury použít funkci.

Poznámka



Pokud je třeba do podprogramu dodat vektor nebo matici, nestačí obvykle mezi vstupními parametry uvést identifikátor příslušného pole, ale často je třeba dodat údaje o skutečně využívané části pole.

V př. 2.16.1. (vektorové součiny) se v procedurách pracovalo vždy s třísložkovými vektory, které byly v třísložkových polích, údaje o obsazení pole tedy nebylo třeba přenášet.

V předchozím příkladě však matice obecně nezabírá celé pole, a proto je třeba (některé) údaje o rozměrech matice přenášet. Mezi formálními parametry obou procedur *Maximum* i *Maxima* jsou sice oba parametry *M*, *N*, ale např. v proceduře *Maximum* je parametr *M* zbytečný a bylo by ho možno zrušit (nejen v deklaraci, ale samozřejmě i ve voláních).

Úlohy k samostatnému řešení

Aby bylo možno dále požadované podprogramy ladit, je třeba je zařadit do nějakého vhodného programu (viz předchozí dva příklady). Pro počáteční ladění lze výchozí vektory nebo matice vytvořit jen „natvrdo“, tj. např. zápisem několika vhodných přiřazovacích příkazů nebo vhodných cyklů. Kromě načítání lze dále výchozí vektory nebo matice generovat pomocí generátoru náhodných čísel (viz funkce *Random* za zadáním př. 2.10.1.).

Příklad 2.16.6. Deklarujte podprogram, který zajistí transponování reálné matice *A* typu (*M*,*N*), kde *M*, *N* jsou z intervalu <2, 8>, přičemž:

a) transponovaná matice se uloží do pole *B*,

b) transponovaní se provede přímo v poli *A*.

Příklad 2.16.7. Deklarujte podprogram, který pro reálnou matici typu (*M*,*N*), kde *M*, *N* jsou z intervalu <2, 8>, nalezne:

a) maximum z hodnot prvků hlavní diagonály,

b) aritmetický průměr prvků hlavní diagonály,

c) aritmetický průměr kladných prvků hlavní diagonály,

d) aritmetický průměr prvků pod hlavní diagonálou a aritmetický průměr prvků nad hlavní diagonálou,

e) *minimax matice*, tj. minimum z řádkových maxim,

f) dvojici indexů výskytu (jednoho z výskytů) minimaxu matice,

g) maximum z hodnot kladných prvků hlavní diagonály,

h) index výskytu maximálního prvku, resp. jednoho z maximálních prvků hlavní diagonály,

i) minimum z absolutních hodnot všech prvků hlavní diagonály,

j) dvojici indexů výskytu (jednoho z výskytů) minima z absolutních hodnot všech prvků hlavní diagonály.

Předpokládané znalosti

Další příklady jsou zaměřeny na práci s řetězci a část z nich úzce navazuje na *Úlohy k samostatnému řešení* z předchozí části 2.15. *Typ řetězec (string)*, viz např. v úvodu těchto úloh zavedené termíny *obvyklý zápis jména a příjmení*, *běžný zápis časového údaje* a *obvyklý zápis časového údaje* (tj. zápis ve formátu *hh:mm*).

Úlohy k samostatnému řešení

Příklad 2.16.8. Deklarujte podprogram (proceduru a případně i funkci), který k vstupnímu parametru typu **string**, jímž je řetězec s obvyklým zápisem jména a příjmení, poskytne:

a) příslušný monogram jako dvouznakový řetězec,

b) příslušný monogram jako dva samostatné znaky,

c) délku (počet znaků) jména a délku příjmení,

d) řetězec, v němž bude nejprve příjmení a potom jméno,

e) řetězec, v němž bude nejprve příjmení a potom jméno, přičemž celé příjmení bude vypsáno velkými písmeny,

d) údaj kolikrát se v příjmení a jméně vyskytuje české písmeno *ch*.

Příklad 2.16.9. Deklarujte funkci „inverzní“ k standardní funkci *UpCase* (viz poznámka za př. 2.15.2.), tj. funkci se vstupním parametrem typu *Char*, která ke vstupujícímu velkému písmenu anglické abecedy poskytne příslušné malé písmeno a ostatní znaky ponechá beze změny.

Příklad 2.16.10. Deklarujte podprogram (proceduru a případně i funkci), který k vstupnímu parametru typu **string**, jímž je řetězec s běžným časovým údajem (tj. místo dvojtečky mohou být hodiny a minuty odděleny jiným oddělovačem a mohou se tu vyskytovat i nadbytečné mezery), poskytne:

a) řetězec s časovým údajem v obvyklém formátu, tj. ve formátu *hh:mm*,

b) číselný údaj o hodinách a číselný údaj o minutách.

2.17. Dynamická deklarace, typ ukazatel (typ *Pointer*)

Výklad

Všechny dosud popisované proměnné byly deklarovány staticky (***statická deklarace proměnných***).

Staticky deklarovaná proměnná existuje, tj. zabírá (obsazuje) místo v paměti po celou dobu existence bloku, v němž je deklarována, a během této doby nemění svou velikost, která je (plně) určena již při zápisu programu, tj. před zahájením chodu programu.

Za chodu programu např. může být využíván různý počet složek nějakého pole, ale velikost tohoto pole a typ jeho složek byly zatím pevně dány příslušnou deklarací již při zápisu programu. V Delphi je však umožněna i ***dynamická deklarace proměnných***.

Dynamicky deklarovaná proměnná vzniká, tj. zabírá (obsazuje) místo v paměti až po provedení speciálního příkazu za chodu programu a její velikost bývá často určována až za chodu programu. Místo zabrané dynamicky deklarovanou proměnnou lze rovněž za chodu programu uvolnit, tj. dynamicky deklarovanou proměnnou lze zase zrušit za chodu programu (i během provádění bloku, v němž byla vytvořena).

Pro dynamickou deklaraci jsou třeba ***proměnné typu ukazatel***. Pro uložení každé proměnné nějakého typu v paměti počítače je třeba určitý počet bajtů (*byte* - bajt je nejmenší přímo adresovatelná jednotka paměti), tento počet se nazývá ***délka uložení*** a adresa prvního bajtu uložení konkrétní proměnné se nazývá ***adresa proměnné***.

Proměnné typu ukazatel slouží pro práci s adresami. Do každé proměnné typu ukazatel může být uložena právě jedna adresa (pro proměnnou typu ukazatel se často užívá též jen termín *ukazatel*). Delphi rozlišuje ***ukazatele typové*** a ***ukazatele netypové***.

Ukazatel typový je ukazatel vázaný jen k jednomu určitému typu, který se vzhledem k typovému ukazateli nazývá ***doménový typ***. V typovém ukazateli tedy obvykle bývá adresa prvního bajtu uložení některé proměnné příslušného doménového typu.

Ukazatel netypový je ukazatel, který není vázán na žádný typ, může se v něm nacházet jakákoliv adresa.

Adresa *nil* - v každém ukazateli (typovém i netypovém) může být adresa **nil**, tj. adresa ukazující „nikam“ (**nil** = nula, nic).

Při deklaraci proměnné typu ukazatel se v paměti jen zabere místo na uložení adresy.

Hodnotu, tj. adresu, lze (jen) do typového ukazatele uložit pomocí procedury *New*, kterou se za chodu programu, tj. dynamicky, vytváří proměnná příslušného typu. Místo v paměti zabrané proměnnou dynamicky deklarovanou pomocí procedury *New* lze uvolnit pomocí procedury *Dispose*.

K uložení adresy do kteréhokoliv ukazatele lze použít proceduru *GetMem*, operátor @ nebo funkci *Ptr*. Místo v paměti zabrané kteroukoliv dynamicky deklarovanou proměnnou lze uvolnit pomocí procedury *FreeMem* (tato procedura se obvykle používá pro proměnné dynamicky deklarované procedurou *GetMem*).

V Delphi jsou dva operátory pro práci s adresami: ***operátor adresy*** (značení: @) a ***operátor indirekce*** (značení: ^).

Operátor @ (operátor adresy) slouží k získání adresy.

Je-li např. *N* proměnná typu *Integer*, pak *@N* je její adresa.

Operátor ^ (oprátor indirekce) umožňuje zpřístupnit proměnnou, na niž ukazuje určitý ukazatel.

Je-li např. *UkI* ukazatel na typ *Integer* (a je-li naplněn), pak *UkI^* označuje (představuje) příslušnou celočíselnou proměnnou.

Příklad: Tisíciprvkové reálné vektory ***a*** a ***b*** by mohly být dynamicky deklarovány např. takto:

```
~~~
type
  TPole=array[1..1000] of Real;
var
  UkA,UkB:^TPole;    {deklarace ukazatelů na zamýšlená pole}
  ~~~
begin
  ~~~
  New(UkA);          {dynamická deklarace pole s fiktivním označením A}
  New(UkB);
```

Příkazem *New(UkA)* se dynamicky vytváří jedno pole typu *TPole* a ukazatel na toto pole se uloží do proměnné *UkA*. Zápis *UkA^* pak představuje označení tohoto pole a

např. *UkA^[5]* je označení jeho pátého prvku. V programu by pak mohly být např. příkazy:

```
UkA^[2]:=UkA^[1]+1;
UkB^:= UkA^;            {obsah celého pole s ukazatelem UkA
                         se zkopíruje do pole s ukazatelem UkB}
```

Výklad

Ukazatele, který ještě není využíván nebo už přestal být využíván, bývá často vhodné naplnit hodnotou **nil**, což pak lze využít např. v podmínkách příkazů **if**, **repeat** nebo **while**.

Příklad:

```
~~~
var
  Uk1,Uk2:^Integer;
  N:Integer;
begin
  N   :=1;
  Uk1 :=@N;          {do Uk1 se uloží adresa proměnné N}
  Uk1^:=2;           {změní se hodnota staticky deklarované proměnné N}
  WriteLn(N);        {kontrolní tisk }
  New(Uk2); {vytvoří se celočíselná proměnná a její ukazatel se uloží do Uk2}
  Uk2^:=Sqr(Uk1^); {do proměnné s ukazatelem Uk2 se uloží hodnota}
  WriteLn(Uk2^);     {kontrolní tisk }
  Dispose(Uk2);      {uvolní se místo proměnné, na niž ukazuje Uk2}
  Uk2:= nil;         {teprve teď se adresa v Uk2 přemaže hodnotou nil};
  if Uk2=nil then
    ~~~
```

Při programování algoritmů s většími nároky na uložení dat v paměti počítače musel programátor v jazyku Turbo-Pascal často používat dynamickou deklaraci, neboť části paměti vyhrazené pro uložení staticky deklarovaných proměnných (***datový segment*** a ***zásobník***) byly poměrně malé vzhledem k části paměti pro ukládání dynamicky deklarovaných proměnných (***halda*** - ***heap***).

V následujícím metodickém (jinak vcelku nesmyslném) příkladě má programátor pracovat se třemi desetitisíciprvkovými reálnými vektory ***a***, ***b*** a ***c***. První bude v programu označen *A* a deklarován staticky, další dva budou deklarovány dynamicky s ukazateli *UkB* a *UkC*. *UkB^* tedy bude označení vektoru ***b*** a *UkC^* označení vektoru ***c***. Vektor ***b*** bude deklarován pomocí procedury *New*, vektor ***c*** (z metodických důvodů) pomocí procedury *GetMem*. Vzhledem k tomu, že procedura *GetMem* není vázána na typ ukazatele, je třeba při

jejím volání zadávat i rozsah deklarované proměnné, který lze zjistit např. pomocí funkce *SizeOf* (u pole lze jednoduše zjistit jeho rozsah, tj. velikost pole, i výpočtem).

Příklad:

```
type
  TPole   =array[1..10000] of Real;
  TUkPole =^TPole;
var
  A:TPole;
  UkB,UkC:TUkPole;
  Uk:Pointer;
  RozsahPole:Integer;
begin
  A[1]:=1;                      {s polem A je možno pracovat ihned}
  New(UkB);                     {pole „B“ je třeba nejprve vytvořit}
  RozsahPole:=SizeOf(TPole);
  GetMem(UkC,RozsahPole);       {avizované jiné vytvoření pole „C“ }
  ~~~
  UkB^:=A;                      {celé pole A se kopíruje do pole „B“ }
  UkC^[1]:=UkB^[1]+1;           {příklad práce s prvky polí „B“a „C“}
  WriteLn(UkC^[1]:10:1);
  ~~~
  FreeMem(UkC,RozsahPole);      {zrušení pole „C“ }
  Dispose(UkB)                  {zrušení pole „B“ };
  ~~~
```

Výklad

Dynamicky deklarované proměnné je vhodné rušit v opačném pořadí než v jakém byly vytvářeny, aby v paměti (na haldě) nevznikaly obtížně využitelné „díry“.

Při programování srovnatelných algoritmů obvykle programátor nemusí v Delphi používat ukazatele tak často jako při programování v jazyku Turbo-Pascal. Skutečnost, že programátor v Delphi pracuje s ukazateli přímo jen zřídka, je právě zajímavou předností prostředí Delphi. Přitom „v zákulisí“ Delphi se právě ukazatelů užívá dosti často, a proto pochopení ukazatelů je důležité jak pro pokročilé programování, tak pro pochopení objektového modelu Delphi, neboť každá proměnná datového typu třída, tj. objekt (viz část 2.20. *Objektově orientované programování (OOP)*), je totiž jen implicitním ukazatelem na místo v paměti, kam byl objekt uložen.

2.18. Jednotka (*unit*)

Výklad

Jednotka (***unit***) je knihovnou deklarací (typů, konstant, proměnných, procedur, funkcí, …), kterou lze připojit do programu nebo ji zpřístupnit z jiné jednotky.

Jednotky jsou základem modulárního programování v Delphi. Mohou sloužit jako knihovny, které lze připojovat k různým programům bez zpřístupnění zdrojového kódu, a také k rozčlenění rozsáhlých programů do logicky souvisejících modulů. Jednotka je samostatným souborem, zdrojovým tvarem jednotky je soubor s příponou *.pas*, při překladu se z něj vytváří soubor s příponou *.tpu*.

Obecná struktura jednotky:

```
unit název jednotky;
interface
odkazy na další jednotky
deklarace exportovaných typů, konstant a proměnných,
    seznam exportovaných procedur a funkcí
implementation
skryté deklarace a kód exportovaných procedur a funkcí
inicialization
volitelná inicializační část
finalization
volitelná úklidová část
end.
```

V modulech, tj. v programech nebo dalších jednotkách, které určitou jednotku používají, lze využívat (je viditelné) vše, co je uvedeno v její ***části*** (***sekci***) ***interface***, přičemž vše další ze ***sekce implementation*** je pro ostatní moduly neviditelné. Jednotka začíná záhlavím, které je tvořeno klíčovým slovem **unit** a jménem (identifikátorem) jednotky. Za záhlavím následuje nejprve sekce interface a pak sekce implementation. Dále může následovat ***sekce inicialization*** a příp. ***sekce finalization***. Jednotka končí, stejně jako program, klíčovým slovem **end** a tečkou.

Jednotka může používat i jiné jednotky, pokud je v ní jejich jméno uvedeno v klauzuli („příkazu“) *uses*. Klauzule uses se může v jednotce objevit na dvou místech. Poprvé to může být za klíčovým slovem **interface**. V tomto případě vše, co je deklarováno v sekcích interface uvedených jednotek, může být použito v kterékoliv sekci této jednotky. Podruhé se klausule uses může vyskytnout za klíčovým slovem **implementation**. V tomto případě lze vše, co je deklarováno v sekcích interface jednotek této klausule uses, použít nejdříve až v sekci implementation (tj. ne již v sekci interface) uvažované jednotky.

Sekce interface („veřejná“ část) jednotky začíná klíčovým slovem **interface** (a končí před klíčovým slovem **implementation**). Tato sekce definuje vše, co je viditelné pro kterýkoliv modul (tj. program nebo další jednotku), který tuto jednotku používá, tj. užívající programy či jednotky mají přístup ke všem deklaracím ze sekce interface.

Procedury a funkce deklarované v sekci interface jednotky jsou sice viditelné pro kterýkoliv modul, který tuto jednotku používá, ale jejich skutečný kód (tj. implementace) je skryt v sekci implementation. V sekci interface jsou z deklarovaných procedur a funkcí uvedena jen jejich záhlaví, těla jsou uvedena až v sekci implementation.

Sekce implementation („soukromá“ část) jednotky začíná klíčovým slovem **implementation**. Vše co bylo deklarováno v sekci interface, je v sekci implementation viditelné. Navíc však sekce implementation může zavádět své vlastní deklarace, které pro moduly užívající uvažovanou jednotku nebudou viditelné, tj. užívající modul „neví“ o existenci těchto deklarací a nemůže na ně odkazovat, resp. je volat. Přitom ale tyto skryté deklarace mohou být (a většinou jsou) využívány „viditelnými“ procedurami a funkcemi uvažované jednotky.

Inicializační sekce, která je nepovinnou částí jednotky a začíná klíčovým slovem **initialization**, umožňuje přiřadit počáteční hodnoty proměnným a datovým strukturám, které jednotka sama užívá, resp. je prostřednictvím sekce interface nabízí užívajícím modulům. Inicializační sekce může být použita i k otevření souborů, se kterými se bude později pracovat. Při spuštění programu, který volá nějakou jednotku, je její inicializační sekce volána ještě předtím, než se začne provádět vlastní příkazová část programu.

Sekce finalization je opět nepovinná část, která může následovat po inicializační části. Tato „úklidová“ část jednotky se provede při ukončení programu.

Řešené úlohy

Příklad : Vytvořte „na ukázku“ jednotku s matematickou náplní, která poskytne část prostředků pro práci s body a vektory a dále prostředky pro výpočet harmonického průměru, geometrického průměru a kvadratického průměru dvou reálných čísel. Jednotku použijte v programu pro výpočet uvedených průměrů z čísel 2 a 4 a pro „převod“ nějakého bodu *X* na vektor *OX* (*O* – počátek souřadné soustavy).

Řešení: V jednotce *Matem* budou deklarovány typy *TVektor* a *TBod* pomocí konstanty *Dim* (na jednom místě zdrojového tvaru je tedy možno změnit dimenzi příslušného geometrického prostoru). V jednotce bude rovněž deklarována procedura *Vektor*, která vypočte vektor určený dvěma body, a dále tu bude „jen“ formou proměnné deklarován bod *O*, představující počátek souřadné soustavy (bod *O* by bylo vhodnější deklarovat jako konstantu příslušného typu).

Kromě funkcí pro výpočet požadovaných průměrů bude v implementační části jednotky deklarována funkce *PrumerAr* pro výpočet aritmetického průměru ze dvou čísel, která bude využita při deklaraci funkcí *PrumerKv* a *PrumerHa*.

Soubor *Matem.pas* - zdrojový soubor požadované jednotky:

```
unit Matem;

interface

const
  Dim=3;

type
  TVektor=array[1..Dim] of Real;
  TBod   =array[1..Dim] of Real;

var
  O:TBod;

function PrumerGe(A,B:Real):Real;
function PrumerKv(A,B:Real):Real;
function PrumerHa(A,B:Real):Real;
procedure Vektor(A,B:TBod; var AB:TVektor);
```

```
implementation

function PrumerGe(A,B:Real):Real;
  begin
    PrumerGe:=Sqrt(A*B);
  end;

function PrumerAr(A,B:Real):Real;
  begin
    PrumerAr:=(A+B)/2;
  end;

function PrumerKv(A,B:Real):Real;
  begin
    PrumerKv:=Sqrt(PrumerAr(Sqr(A),Sqr(B)));
  end;

function PrumerHa(A,B:Real):Real;
  begin
    PrumerHa:=1/PrumerAr(1/A,1/B);
  end;

procedure Vektor(A,B:TBod; var AB:TVektor);
  var
    I:Byte;
  begin
    for I:=1 to Dim do
      AB[I]:=B[I]-A[I];
  end;

var
  I:Byte;

inicialization

  for I:=1 to Dim do
    O[I]:=0;

end.
```

Soubor *Test.dpr* - zdrojový soubor požadovaného programu:

```
program Test;
{$APPTYPE CONSOLE}
uses
  SysUtils,Matem;
var
  X :TBod;
  OX:TVektor;
```

```
begin
  WriteLn(PrumerHa(2,4):10:5);
  WriteLn(PrumerGe(2,4):10:5);
  WriteLn(PrumerKv(2,4):10:5);
  WriteLn(Dim);
  X[1]:=5; X[2]:=5; X[3]:=1;
  Vektor(O,X,OX);
  WriteLn(OX[1]:10:5,OX[2]:10:5,OX[3]:10:5);
  ReadLn;
end.
```

V implementační části jednotky *Matem* je sice deklarována funkce *PrumerAr*, ale v programu *Test* ji nelze použít, neboť její záhlaví není uvedeno v interferenční části jednotky *Matem*.

Úlohy k samostatnému řešení

Aby bylo možno dále požadované jednotky plně ladit, je třeba je použít v nějakém vhodném programu (viz předchozí příklad).

Příklad 2.18.1. Vytvořte jednotku *Komplex* usnadňující práci s komplexními čísly. V interferenční části jednotky *Komplex* by bylo nejprve vhodné deklarovat typ pro komplexní proměnné (i komplexní konstanty) jako záznam, který má dvě položky typu *Real* (např. *Re* pro reálnou a *Im* pro imaginární složku komplexního čísla). V jednotce by mohly být např. deklarovány následující procedury a funkce:

a) pro výpočet velikosti a argumentu komplexního čísla,

b) pro výpočet komplexně sdruženého čísla,

c) pro součet, rozdíl, součin a podíl komplexních čísel,

d) pro výpočet přirozené mocniny komplexního čísla,

e) pro výpočet odmocnin komplexního čísla.

Příklad 2.18.2. Vytvořte jednotku *Vektor* usnadňující práci s reálnými vektory, které mají nejvýše *NMax* souřadnic, kde $NMax \leq 100$. V interferenční části jednotky *Vektor* by bylo vhodné nejprve deklarovat konstantu *NMax* a typ příslušného jednorozměrného pole. V jednotce by mohly být např. deklarovány následující procedury a funkce:

a) pro čtení, resp. generování vektoru a pro tisk vektoru,

b) pro výpočet skalárního součinu vektorů,

c) pro výpočet velikosti vektoru,

d) pro nalezení maximální hodnoty z hodnot souřadnic vektoru,

e) pro setřídění souřadnic vektoru do nerostoucí, resp. neklesající, posloupnosti.

Příklad 2.18.3. Vytvořte jednotku *Matice* usnadňující práci s reálnými maticemi typu (*M*,*N*), kde *M*,*N* jsou z intervalu <1,9>. V interferenční části jednotky *Matice* by bylo vhodné deklarovat nejen typ příslušného dvourozměrného pole, ale (předtím) i typ jednorozměrného pole pro řádky matice. V jednotce by mohly být např. deklarovány následující procedury a funkce:

a) pro čtení, resp. generování matice a pro tisk matice,

b) pro součet, rozdíl a součin matic,

c) pro výpočet normy matice,

d) pro požadavky z př. 2.16.6. až 2.16.7.,

e) pro práci se čtvercovými maticemi.

2.19. Direktivy překladače

Výklad

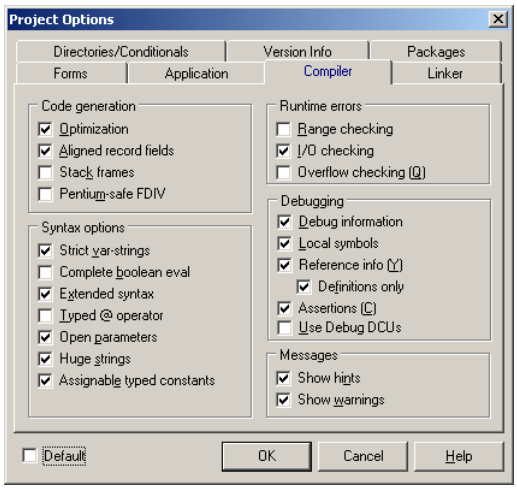
Direktivy překladače (direktivy kompilátoru) jsou pokyny pro překladač, které umožňují ovlivnit jeho práci, což se pak projeví na vlastnostech, parametrech a způsobu realizace výsledného programu.

Jednou z možností zadání direktivy překladače je její zápis přímo do zdrojové formy programu formou „připomínající“ komentář zapsaný pomocí složených závorek, jehož prvním znakem je znak *$*. Zápis direktivy překladače má pak tvar:

{$text direktivy}

Např. direktiva *{$APPTYPE CONSOLE}* používaná v předcházejících programech informuje překladač, že vytvářený program nemá být běžnou standardní aplikací, ale jen aplikací pro konzolu.

Na konkrétním programu bude dále ukázáno, že různé překladače mohou stejný zdrojový program (resp. alespoň významově stejný zdrojový program) různě realizovat. Např. má-li být *i*-tému prvku pole *A* přiřazena hodnota, pak některé překladače zajistí, že se před vlastním přiřazením provede kontrola, zda index *i* je z rozsahu, který je dán deklarací pole *A*, a pokud tomu tak není dojde k chybovému stavu. Každá kontrola prováděná za chodu programu však představuje nároky na strojový čas, a proto některé překladače např. zmíněnou kontrolu samy nezajišťují (a záleží pak na programátorovi zda ji do programu výslovně předepíše). V Delphi si ale programátor může formou direktivy překladu zvolit, zda překladač sám má, či nemá uvedenou kontrolu zajišťovat (direktiva *Range checking*, krátce též jen direktiva *R*). Direktivy překladače lze nastavit prostřednictvím menu volbou *Project / Options…* na kartě *Compiler*, nebo, což je často výhodnější, direktivy lze též zapsat přímo do zdrojového programu.

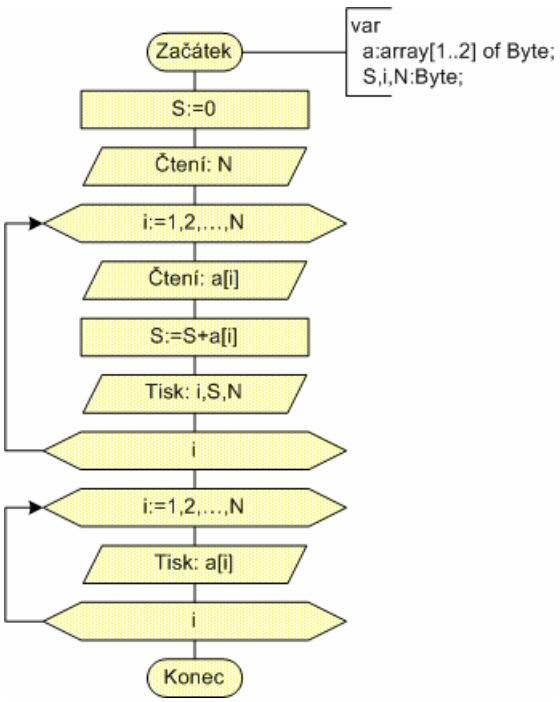


Chyby, ke kterým může dojít při nezajištění zmíněné kontroly, vyplývají ze skutečnosti, že adresy jednotlivých prvků pole se (kromě výjimek) počítají až za chodu programu z adresy počátku pole (tj. z adresy prvního prvku pole) a z délky uložení jednoho prvku pole. Např. tedy adresa třetího prvku pole se vypočte tak, že se k adrese prvního prvku pole přičte dvojnásobek délky uložení jednoho prvku pole. Má-li však příslušné pole jen dva prvky, pak vypočtená adresa „třetího“ prvku pole ukazuje do paměti bezprostředně za konec pole, kde se již může nacházet další deklarovaná proměnná (viz následující př. 2.19.1.).

Řešené úlohy

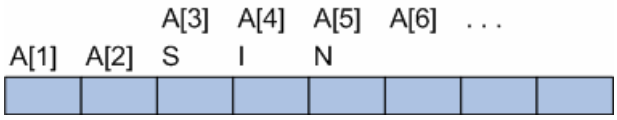
Příklad 2.19.1. Vstupní údaje tvoří přirozené číslo $N \leq 20$ a dále posloupnost N přirozených nejvýše dvojciferných čísel. Sestavte program, který po načtení každého prvku posloupnosti zajistí tisk součtu dosud načtených prvků a na závěr, tj. po načtení všech prvků posloupnosti, tisk této posloupnosti.

Řešení: Z požadavku na závěrečný tisk všech prvků posloupnosti vyplývá, že je třeba prvky posloupnosti uložit do pole. Při deklaraci pole však „došlo“ k chybě, místo intervalu 1..20, je tu jen interval 1..2.




```
program DirektivaR;
{$APPTYPE CONSOLE}
{$R-}
uses
  SysUtils;
var
  A:array[1..2] of Byte;
  S,I,N:Byte;
begin
  S:=0;
  ReadLn(N);
  for I:=1 to N do
  begin
    ReadLn(A[I]);
    S:=S+A[I];
    WriteLn('S[',I,'] = ',
            S,' N = ',N);
  end;
  WriteLn('Posloupnost: ');
  for I:=1 to N do
    Write(A[I]:4);
  ReadLn;
end.
```

Pořadím deklarací proměnných je určeno i pořadí příslušných paměťových míst v paměti počítače, které je tedy v daném případě následující:



V daném případě jsou všechny proměnné typu *Byte*, tzn., že příslušná paměťová místa mají velikost 1 B. Direktiva *R* pro kontrolu rozsahů se zapíná/vypíná pomocí zaškrtávacího políčka *Range checking* v okně *Project Options*, nebo, a to přednostně, uvedením direktivy *{$R+}*, resp. *{$R-}* přímo v programu (viz třetí řádek programu). Pokud je direktiva *R* vypnuta, je paměťové místo *S* dostupné i pod jménem *A[3]*, paměťové místo *I* pod jménem *A[4]* atd. Uvedený program v Delphi zajišťuje pro přehlednost a snadnější pochopení kromě požadovaných částečných součtů i tisk aktuální hodnoty proměnné *I* a proměnné *N* (včetně doprovodného textu).

Např. pro vstupní údaje: 6, 1, 1, 1, 1, 3, 3,

je pak obsah vstupně výstupního okna příslušné aplikace následující:

```
6
1
S[1] = 1 N = 6
1
S[2] = 2 N = 6
1
S[3] = 2 N = 6
1
S[1] = 3 N = 6
3
S[2] = 6 N = 3
3
S[3] = 9 N = 3
Posloupnost:
   1   1   9
```

Ve výpisu byla červeně dodatečně zvýrazněna jen místa, na nichž se poprvé v proměnných *S*, *I* a *N* objevuje jiná hodnota, než v případě programu se správným rozsahem pole *A*. Poprvé je to po načtení třetího prvku posloupnosti. Do *A[3]*, a tedy i do proměnné S se načte číslo 1, pak se provede příkaz *S:=S+A[3]*, a proměnná *S* tedy nabude hodnotu 2.

Pokud by v programu (ve třetím řádku) místo direktivy *{$R-}* byla použita direktiva *{$R+}*, bude chod programu při třetím provádění příkazu *ReadLn(A[I])* přerušen chybovým hlášením *Range check error*. Pokud by ani jedna z uvedených direktiv *{$R+}*, resp. *{$R-}*,

nebyla v programu zapsána, bude chod programu záviset na stavu položky *Range checking* v okně *Project Options* při překladu programu (nikoliv při jeho užití).

Výklad

Direktiva *R*, jíž byl „věnován“ předchozí příklad, je ***direktiva-přepínač*** (***switch***) (viz dále), a je to ***direktiva lokální***, tzn., že může být použita jen v části programu, resp. jednotky, kdežto ***direktiva globální*** musí být použita na celý program, resp. jednotku.

Zápis jednotlivé direktivy kompilátoru začíná za otevírající komentářovou závorkou ***{*** znakem ***$***, za nímž bezprostředně následuje jméno (jeden nebo více znaků), které určuje konkrétní direktivu.

Direktivy kompilátoru se dělí následujícím způsobem:

Direktiva kompilátoru:

- **Direktiva – přepínač** (***switch***): Přepínač zapíná nebo vypíná určitou vlastnost kompilátoru. Zapínání se zadává znakem + nebo slovem *ON*, vypínání znakem – nebo slovem *OFF*.
- **Direktiva s parametrem**: Její částí je doplňkový údaj, kterým je u některých direktiv např. jméno souboru, jenž má být zařazen do kompilace .
- **Podmíněná direktiva**: Řídí podmíněnou kompilaci, např. jedna z nich umožňuje zařadit do kompilace jednu ze dvou vymezených částí zdrojového programu.

Vhodným použitím direktiv kompilátoru je možno např. zkrátit zdrojovou formu programu, usnadnit ladění programu, či poměrně lehce vyřešit některé problémy (zejména u aplikací pro konzolu, neboť při jejich programování nemá programátor k dispozici mnohé prostředky Delphi).

I pokročilý programátor však manipuluje obvykle s direktivami kompilátoru jen zřídka, obvykle si je „vhodně a natrvalo“ nastaví v okně *Project Options*. Při přenášení nebo poskytování zdrojové formy programů, či jejich částí, je však někdy vhodné mít direktivy zapsané přímo ve zdrojovém formě.

Většinu informací o jednotlivých direktivách kompilátoru poskytuje příslušný help, k vyhledání helpů obvykle postačí klíčová slova z okna *Project Options*. Dále tedy budou následovat jen formou příkladů informace o dvou dalších direktivách kompilátoru, které je

možno efektivně využít i v jednoduchých programech. Nejprve ale bude uvedena na ukázku část helpu pro direktivu *R*, jež byla použita a popsána v řešení předchozího příkladu:

Range checking

Type	Switch
Syntax	{$R+} or {$R-} {$RANGECHECKS ON} or {$RANGECHECKS OFF}
Default	{$R-} {$RANGECHECKS OFF}
Scope	Local

Z ukázky je patrné, že se jedná o lokální direktivu-přepínač (*switch*), kterou je možno do programu zapisovat dvěma formami (krátkou nebo dlouhou), a že její přednastavená hodnota (*default*) je *OFF*.

Řešené úlohy

Příklad 2.19.2. Napište část programu, která zajistí bezchybné načtení hodnoty proměnné *X* typu *Real* z klávesnice, tj. která zajistí aby program nehavaroval při běžných chybách nebo překlepech (zadání čárky místo desetinné tečky, zdání písmene místo cifry, zadání dvou teček apod.).

Řešení: Pro řešení zadané úlohy bude využita direktiva *I* (*I/O checking*). Je-li její hodnota *ON* (default), pak po zmíněných chybách dojde k chybovému hlášení a k přerušení chodu programu.

Každá procedura nebo funkce zajišťující vstupní či výstupní operaci (např. procedura *ReadLn*, či *Reset*) poskytuje totiž jako vedlejší produkt svého volání chybový kód. Je-li vše „v pořádku“, je chybový kód 0, jinak je nenulový a jeho hodnota se využívá např. právě pro výběr textu chybového hlášení.

Je-li direktiva *I* vypnuta, pak nenulový chybový kód vstupní či výstupní operace nezpůsobí (bezprostřední) přerušení chodu programu, ale může být prostřednictvím funkce *IOResult* a vhodných příkazů využit např. pro zajištění nápravy:

```
program IORes1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  X:Real;
begin
  {$I-}
  repeat
    Write('X: ');
    ReadLn(X);
  until IOResult=0;
  {$I+}
  WriteLn('X = ',X);
  ReadLn;
end.
```

```
~~~
var
  X:Real;
  Kod,N:Integer;
begin
  N:=0;
  {$I-}
  repeat
    N:=N+1;
    Write(N,'. X: ');
    ReadLn(X);
    Kod:=IOResult;
  until (Kod=0) or (N=2);
  {$I+}
  if Kod=0 then
    WriteLn('X = ',X)
  else
    WriteLn('X = ?');
~~~
```

Výklad

Direktiva *I* se dříve v TP často užívala např. při programování prostředků (procedur, funkcí, či částí programu) pro práci se soubory, např. pro zjištění, zda soubor zadané specifikace existuje (neexistující soubor nelze otevřít pro čtení). V Delphi má sice uživatel mnoho těchto prostředků k dispozici, často se však jedná o metody vizuálních komponent, takže pro začátečníka je jejich užívání poněkud problematické.

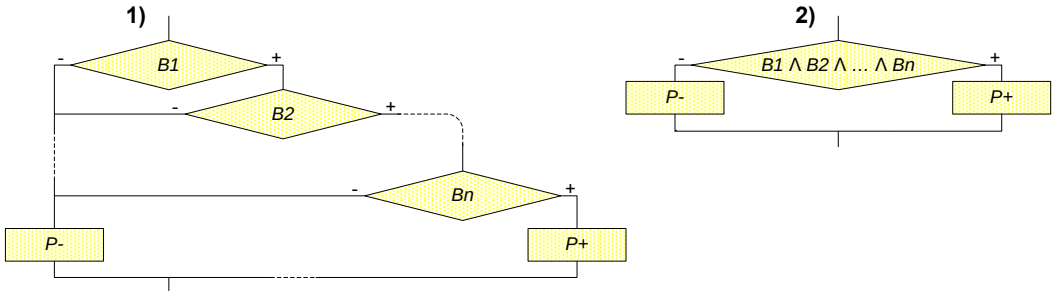
Další direktivou kompilátoru, se kterou přijdou běžně do styku i začátečníci (i když často o tom ani neví) je direktiva-přepínač *B* (*Komplete boolean eval*, v helpu pod názvem *Boolean short-circuit evaluation*).

Direktiva-přepínač *B* slouží k zapínání/vypínání plného vyhodnocování booleovských výrazů, které obsahují logické operátory **and** a **or**. Při zkráceném vyhodnocování se výraz postupně vyhodnocuje zleva (pokud závorky nestanoví jinak), a je-li o výsledku „již rozhodnuto“, tak se vyhodnocování přeruší, tj. v dalším vyhodnocování se již nepokračuje. Je-li např. více dílčích booleovských výrazů spojeno spojkou **and**, je vyhodnocování přerušeno, jakmile se dojde k prvnímu booleovskému výrazu, jehož hodnota je *False*. Zkrácené vyhodnocování booleovských výrazů (tj. direktiva *{$B-}*) je tedy zajímavá i z hlediska algoritmizace, neboť mj. umožňuje zkrátit a zpřehlednit jednu z často používaných řídicích struktur.

Boolean short-circuit evaluation

Type	Switch
Syntax	{$B+} or {$B-} {$BOOLEVAL ON} or {$BOOLEVAL OFF}
Default	{$B-} {$BOOLEVAL OFF}
Scope	Local

Zmíněnou řídicí strukturu a její zkrácení vyjadřují následující části vývojového diagramu. Místo posloupnosti rozhodování v nezkrácené řídicí struktuře (1) je ve zkrácené řídicí struktuře (2) uvedeno jen jedno rozhodování s konjunkcí příslušných dílčích podmínek B_1, B_2, …, B_n.



Řešené úlohy

Příklad 2.19.3. Napište část programu, ve které booleovská proměnná *L* nabude hodnotu *True*, nebo *False*, podle toho, zda hodnota reálné proměnné *X* je, či není, z definičního oboru funkce $f(x)=\sqrt{\sin x-\sqrt{\sin\sqrt{x}}}$.

Řešení: Požadavky na definiční obor funkce f(x) se v matematice běžně vyjadřují zápisem

$$x \geq 0 \wedge \sin\sqrt{x} \geq 0 \wedge \sin x - \sqrt{\sin\sqrt{x}} \geq 0 \quad ,$$

který však není zcela korektní, neboť jednotlivé dílčí podmínky na sebe navazují. Pokud by např. nebyla splněna první dílčí podmínka, nemá již druhá podmínka vůbec smysl (hodnota $\sin\sqrt{x}$ není definována). Pokud by tedy v následující části programu byla použita direktiva *{$B+}*, došlo by při užití uvedeného příkazu **if** k chybě nejen pro *X*<0, ale např. i pro *X*=10.

```
~~~
{$B+}
if (X>=0) and (sin(sqrt(X))>=0)
          and (sin(x)-sqrt(sin(sqrt(X)))>=0) then
  L:=True
else
  L:=False;
```

V další části programu je uvedena další varianta řešení, která naznačuje, jak lze „přeprogramovat“ v Delphi (bez ohledu na stav direktivy *B*) a i v mnoha dalších programovacích jazycích výše uvedenou řídicí strukturu (1).

```
~~~
L:=False;
if X>=0 then
  if sin(sqrt(X))>=0 then
    if sin(X)-sqrt(sin(sqrt(X)))>=0 then
      L:=True;
~~~
```

Úlohy k samostatnému řešení

Příklad 2.19.4. Napište část programu (resp. proceduru či funkci), pomocí níž se zjistí zda existuje soubor zadané specifikace.

Návod: Pokus otevřít pro čtení neexistující soubor při direktivě *{$I+}* skončí chybovým hlášením, při direktivě *{$I-}* poskytne funkce *IOResult* číslo 0 jen v případě, že se pokus zdaří (otevřený soubor je pak třeba zavřít!), jinak je výsledkem funkce *IOResult* číslo, které udává proč se otevření nezdařilo (např., že neexistuje zadaný adresář).

2.20. Objektově orientované programování (OOP)

Výklad

Objektově orientované programování je metoda tvorby programů (vyžadující ***objektově orientovaný jazyk***) založená na spojení dat a jejich vlastností či příslušných metod (procedur a funkcí), která vychází ze tří základních principů, jimiž jsou:

- **obalení** (**zapouzdření – *encapsulation***) - spojení datových struktur a jim příslušejících metod a vlastností do jednoho typu (třídy),
- **dědičnost** (***inheritance***) – od definovaného typu lze odvodit potomka, tj. typ, který od předka „dědí“ datové struktury i vlastnosti a metody,
- **polymorfismus** (**mnohotvarost** též **pozdější vazba**) – předek a potomek mohou mít vlastnost, resp. používat metodu, se stejným jménem, jejíž realizace však může být různá.

Objektově orientovaný jazyk je jazyk, který poskytuje prostředky pro realizaci uvedených tří základních principů OOP.

Např. pro člověka, který je dostatečně obeznámen s maticemi, nepředstavuje zápis matice jen pouhou tabulku čísel, ale tato tabulka je pro něj objektem, z jehož zápisu je dále zřejmý nejen počet řádků a počet sloupců matice, ale i objektem, kterému přísluší např. hodnost, norma a v případě čtvercové matice též determinant, příp. i inverzní matice atd.

V řešených příkladech z předchozích částí byla např. matice typu *(M,N)* uložena ve vhodném dvourozměrném poli a údaje o jejím typu v proměnných *M*, *N*. Pro tisk matice, či pro výpočet hodnosti matice by pak mohly být deklarovány podprogramy, do nichž by se např. formou parametrů dodával název pole a údaje o rozměrech matice. Příslušné pole i proměnné *M*, *N* by však mohly být zahrnuty do jednoho záznamu vhodného typu, a zmíněné podprogramy by pak měly jediný parametr (identifikátor příslušného záznamu). V Delphi je však možno deklarovat nejen typ záznam (**record**), ale i typ třída (**class**), který kromě příslušné datové struktury (u matice obvykle dvourozměrné pole a dvě celočíselné proměnné) 1může formou podprogramů zahrnovat i některé metody či vlastnosti, které se k datové struktuře váží (u matic např. podprogram pro výpočet normy matice). Jak lze pomocí zmíněného typu *„třída pro matice“* pro každou matici deklarovat, vytvořit a používat samostatný objekt naznačí následující příklad. Při deklaraci typu třída je dále např. možno

některé její položky či metody deklarovat jako veřejné (**public**) nebo neveřejné (**private**). Klíčové slovo **private** označuje položky a metody třídy, které nejsou přístupné (tj. jsou „neviditelné“) vně jednotky (zdrojového souboru) deklarující danou třídu.

Objektově orientovaný jazyk může vycházet z různých objektových modelů (konkrétně např. Delphi používá jiný objektový model než Turbo-Pascal) a také terminologie a označování používané v souvislosti s OOP se v různých programovacích jazycích značně liší. V Delphi se vychází z termínů třída a objekt.

Třída (***class***) je „jen“ typ (buď standardní, nebo definovaný uživatelem) a **objekt** je instancí třídy, tj. proměnná typu definovaného třídou, čili skutečný útvar, který za běhu programu zabírá určitou část paměti. Vztah mezi objektem a třídou je tedy stejný jako mezi proměnnou a datovým typem (*Pozor!* Turbo-Pascal používal klíčové slovo **object** k deklaraci třídy a pro skutečné objekty se pak obvykle používal termín *instance objektu*).

Řešené úlohy

Příklad 2.20.1. Pro práci s celočíselnými maticemi typu *(M,N)*, kde *M*, *N* jsou z intervalu <1, 3> deklarujte třídu *TMat*, která kromě vlastního dvourozměrného pole *H* a dvou celočíselných proměnných *M*, *N* (tj. kromě položek) obsahuje i proceduru *GenMat* pro „vytvoření“ matice např. pomocí generátoru náhodných čísel, proceduru *Tisk* pro tisk matice a funkci pro výpočet euklidovské normy matice. Typ *TMat* dále použijte pro deklaraci třídy *TMatQ* pro čtvercové matice, přičemž v této třídě bude zahrnuta i funkce pro výpočet příslušného determinantu.

V obou třídách budou uvedeny se stejným identifikátorem *GenMat* procedury pro vytvoření matice. Zatímco v rámci typu *TMat* bude generována matice typu *(nM,nN)*, v rámci typu *TMatQ* bude generována čtvercová matice řádu *min(nM,nN)*. Aby bylo možno stejný identifikátor použít pro nestejné metody předka a potomka (*polymorfismus*), je třeba u „prvního“ předka s touto metodou uvést za záhlavím příslušné metody klíčové slovo **virtual**, u „dalších“ potomků pak klíčové slovo **override**. Pro bližší objasnění bude v obou třídách různě deklarována i procedura *Tisk*, která kromě tisku vlastní matice bude zajišťovat též tisk normy matice v prvním případě a tisk determinantu matice v druhém případě.

V deklaracích tříd se z metod (procedur a funkcí) uvádí jen záhlaví, celá deklarace metody se uvádí „někde“ až za deklarací třídy, záhlaví celé deklarace metody pak musí obsahovat i identifikátor příslušné třídy.

```
unit MaticeUn;

interface

uses
  SysUtils,Math;

type
  TMat=class
  private
    M,N:1..3;
    H:array[1..3,1..3] of Integer;
    procedure GenH;
    procedure TiskH;
  public
    procedure GenMat(nM,nN:Byte);virtual;
    procedure Tisk; virtual;
    function NormaE:Real;
    constructor Copy(A:TMat);
  end;
  TMatQ=class(TMat)
  public
    procedure GenMat(nM,nN:Byte);override;
    procedure Tisk; override;
    function Det:Integer;
  end;

implementation

procedure TMat.GenH;
  var
    I,J:Byte;
  begin
    for I:=1 to M do
      for J:=1 to N do
        H[I,J]:=Random(21)-10;
  end;

procedure TMat.GenMat(nM,nN:Byte);
  begin
    M:=nM;
    N:=nN;
    GenH;
  end;
```

```
procedure TMat.TiskH;
  var
    I,J:Byte;
  begin
    WriteLn;
    for I:= 1 to M do
      begin
        for J:=1 to N do
          Write(H[I,J]:4);
        WriteLn;
      end;
  end;

procedure TMat.Tisk;
  begin
    TiskH;
    WriteLn('Norma = ',NormaE:10:3);
  end;

function TMat.NormaE:Real;
  var
    I,J:Byte;
    S:Word;
  begin
    S:=0;
    for I:= 1 to M do
      for J:=1 to N do
        S:=S+sqr(H[I,J]);
    Result:=sqrt(S);
  end;

constructor TMat.Copy(A:TMat);
  begin
    M:=A.M;
    N:=A.N;
    H:=A.H;
  end;

procedure TMatQ.GenMat(nM,nN:Byte);
  begin
    M:=Min(nM,nN);
    N:=M;
    GenH;
  end;

procedure TMatQ.Tisk;
  begin
    TiskH;
    WriteLn('Deter = ',Det);
  end;
```

```
function TMatQ.Det:Integer;
  begin
    case N of
      1: Result:=H[1,1];
      2: Result:=H[1,1]*H[2,2]-
                 H[1,2]*H[2,1];
      else Result:= H[1,1]*~~~;
    end;
  end;
end.

program MaticePr;
{$APPTYPE CONSOLE}
uses
  SysUtils,MaticeUn;
var
  A,C:TMat;
  B:TMatQ;
begin
  Randomize;
  A:=TMat.Create;
  A.GenMat(2,2);
  C:=TMat.Copy(A);
  A.Tisk;
  C.Tisk;
  C.Destroy;
  B:=TMatQ.Create;
  B.GenMat(3,3);
  B.Tisk;
  WriteLn(B.NormaE:10:3);
  C.Tisk;
  ReadLn;
end.
```

V uvedeném řešení jsou požadované deklarace v unitu *MaticeUn*, a ten je užit v programu *MaticePr*.

Výklad

V Delphi se všechny objekty vytvářejí dynamicky. Deklarací proměnné některé třídy (v programu *MaticePr* např.: ***var*** *A,C TMat; B:TMatQ;*) se zavádí jen proměnná typu ukazatel „na objekt“, která nabude hodnotu teprve po vytvoření příslušného objektu.

Vlastní vytvoření objektu (v paměti počítače) se zajišťuje až za chodu programu voláním konstruktoru, což je metoda, tj. podprogram, který má stejnou stavbu jako procedura, v jehož záhlaví je však místo klíčového slova **procedure** klíčové slovo **constructor**.

K vytvoření objektu je možno využít i standardní konstruktor *Create*, viz např. příkaz *A:=TMat.Create* či příkaz *B:=TMatQ.Create* (identifikátor třídy zde zajišťuje, aby pomocí „stále“ stejného identifikátoru *Create* byly vytvořeny objekty příslušných správných typů).

V uvedeném předchozím programu je na ukázku deklarován vlastní konstruktor jen u typu *TMat* (konstruktor *Copy*), který byl využit jen pro vytvoření matice *C* (viz příkaz *C:=TMat.Copy(A)*). Uvedený konstruktor *Copy* kromě vlastního vytvoření objektu zajistí i naplnění jeho datové struktury zkopírováním datové struktury z objektu, který je uveden jako skutečný parametr tohoto konstruktoru. Provedením příkazu *C:=TMat.Copy(A)* je tedy nejen v paměti zabráno místo pro objekt (matici) *C*, ale jsou sem i zkopírována data matice *A*.

Objekt lze za chodu programu nejen vytvořit, ale i zrušit, tj. místo, které objekt (po svém vytvoření) zabere v paměti počítače lze rovněž za chodu programu zase uvolnit. Ke zrušení objektu lze využít standardní destruktor *Destroy*.

Pro každou třídu může být deklarován nejen jeden nebo více konstruktorů, ale i destruktorů (v záhlaví destruktoru je klíčové **destructor**). Pomocí vlastního destruktoru lze pak zajistit nejen zrušení objektu, ale i provedení nějakých „úklidových“ prací.

Pro lepší pochopení způsobu vytváření a rušení objektů je v programu *MaticePr* v třetím řádku od konce uveden „nesmyslný“ příkaz *C.Tisk*. Nemá totiž smysl požadovat tisk matice *C*, která již byla pomocí destruktoru z paměti odstraněna. Zmíněné uvedení příkazu však tentokrát nevyvolá žádné chybové hlášení, ale zajistí (poněkud překvapivě) tisk matice *B*, jejíž vytvoření je v programu *MaticePr* požadováno bezprostředně po zrušení matice *C*. Vzhledem ke způsobu využívání paměti začíná umístění matice *B* na stejné adrese, na které předtím začínalo umístění matice *C*, přičemž tato adresa se pouhým zrušením matice *C* z příslušné proměnné *C* typu *TMat* neodstraní. Problémy tohoto typu by bylo možno řešit podobně jako v druhém příkladě z části 2.17. *Dynamická deklarace, typ ukazatel (typ pointer)*, tj. po destrukci objektu vždy naplnit příslušnou proměnnou adresou **nil**.

Deklarace typů *TMat* a *TMatQ* požadované v př. 2.20.1. mohly být uvedeny přímo v programu *MaticePr*, tj. nemusely být uvedeny v samostatném unitu. Deklarace typů *TMat* a *TMatQ* v samostatném unitu *MaticeUn*, však umožňuje demonstrovat rozdíl mezi položkami či metodami, které jsou deklarovány jako veřejné (**public**) nebo neveřejné (**private**).

Položky a metody třídy *TMat*, předznačené klíčovým slovem **private**, nejsou v programu *MaticePr* přímo dostupné, např. uvedení příkazu *A.TiskH* v programu *MaticePr* by vyvolalo již při překladu chybové hlášení o nedeklarovaném identifikátoru *TiskH*.

V programu *MaticePr* tedy také není možno přímo pracovat s polem *H* jednotlivých objektů, tj. nelze tu „zatím“ např. zadat požadavek na změnu hodnoty jednotlivých prvků některé z matic *A*, *B*, *C*. Tento stav by bylo možno změnit dvěma způsoby: převést v unitu *MaticeUn* složku *H* z neveřejné na veřejnou nebo v unitu *MaticeUn* deklarovat vhodnou metodu. Změnu zmíněného stavu by tedy bylo možno zajistit jen změnou v unitu (nikoliv v programu).

Pro lepší pochopení některých výhod, které přináší OOP, bude porovnána metoda *TMat.Tisk* s „klasicky“ psanou procedurou *TiskMat* pro tisk matice *A* typu (*M*,*N*). Procedura *TiskMat* by mohla být např. využita v př. 2.16.5. pro tisk matice *A* typu (*K*,*K*) nebo pro tisk původní matice a matice k ní transponované v př. 2.16.6.

Pokud by procedura *TiskMat* měla záhlaví

```
procedure TiskMat(A:TMat;M,N:Integer);
```

potom by zmíněná volání měla tvar:

```
TiskMat(A,K,K);

TiskMat(A,M,N);
TiskMat(B,N,M);
```

V uvedených voláních samotný programátor musí zajistit správné uvedení těch proměnných, které nesou informaci o typu matice (přičemž některé chyby by ani nemusely vyvolat chybové hlášení). V případě OOP by však zmíněná volání mohla mít jednoduchý, přehledný a „bezproblémový“ tvar:

```
A.Tisk;

A.Tisk;
B.Tisk;
```

Použití OOP pro řešení úloh z kap. 1. a z předchozích částí kap. 2. by bylo zřejmě zbytečně pracné a náročné. Počáteční práce vyplývající z objektově orientovaného přístupu k programování se však obvykle brzy zúročí při řešení rozsáhlých úloh nebo přímo skupin příbuzných rozsáhlých úloh. Je např. poměrně jednoduché doplnit deklaraci třídy *TMat* z předchozího příkladu o některé další metody (viz př. 2.20.2.).

Úlohy k samostatnému řešení

Aby bylo možno dále požadované deklarace tříd prověřit, je třeba vytvořit i jednoduché programy, v nichž se bude pracovat s příslušnými objekty, přičemž požadované deklarace tříd by měly být v samostatné jednotce (viz př. 2.20.1.).

Příklad 2.20.2. Doplňte deklaraci třídy *TMat* z př. 2.20.1. o metody, které např. poskytují:

- řádkové maximum, resp. minimum, *K*-tého řádku matice,
- řádkový součet *K*-tého řádku matice,
- řádkový průměr *K*-tého řádku matice.

Pro každý z požadavků je možno vytvořit samostatnou metodu nebo je možno vytvářet metody, které plní dva nebo více požadavků. Další metody by mohly kromě uvedených řádkových hodnot poskytovat příslušné hodnoty např. pro sloupce, hlavní diagonálu, či pro celou matici.

Poznámka

Pro matice by měly praktický význam metody poskytující např. inverzní matici, vlastní čísla a vlastní vektory matice, či LU-rozklad matice, programování těchto metod by si však vyžádalo další matematické znalosti.

Příklad 2.20.3. Deklarujte (v samostatné jednotce) třídu *TKomplex*, jejímiž objekty budou komplexní čísla např. s položkami *Re* pro reálnou a *Im* pro imaginární složku komplexního čísla. Položky *Re* a *Im* by mohly být nejprve **public** později **private**, tj. z programu, který by vytvářenou jednotku používal, by pak tyto položky byly dostupné jen prostřednictvím vhodných metod. Mezi metodami deklarujte i metody pro výpočet velikosti a argumentu komplexního čísla. V jednotce dále deklarujte procedury:

- pro výpočet komplexně sdruženého čísla,
- pro součet, rozdíl, součin a podíl komplexních čísel,
- pro výpočet přirozené mocniny komplexního čísla,
- pro výpočet odmocnin komplexního čísla.

Příklad 2.20.4. Deklarujte (v samostatné jednotce) třídu *TVektor*, jejímiž objekty budou reálné vektory, které mají nejvýše *NMax* souřadnic, kde *NMax*≤100 (*NMax* - konstanta). Vlastní vektory (tj. pole jejich souřadnic) by mohly být nejprve **public** později **private**, tj.

z programu, který by vytvářenou jednotku používal, by pak tyto položky byly dostupné jen prostřednictvím vhodných metod. Kromě metod, které slouží pro čtení, resp. generování vektoru a pro tisk vektoru, deklarujte i metody, které např. poskytují:

- velikost vektoru,
- normu vektoru,
- maximum z hodnot souřadnic vektoru,
- setřídění souřadnic vektoru do nerostoucí, resp. neklesající, posloupnosti.

V jednotce dále deklarujte procedury a funkce:

- pro násobení vektoru reálným číslem,
- pro součet či rozdíl dvou vektorů,
- pro skalární součin vektorů.

3. KONZOLOVÉ APLIKACE V DELPHI

Výklad

I když původně bylo vývojové prostředí Delphi vyvinuto firmou Borland pro vytváření aplikací pro Windows s takzvaným grafickým uživatelským rozhraním (GUI – *Graphical User Interface*), může také být použito jen pro vytváření konzolových aplikací. Použití těchto čistě textových programů je velmi vhodné pro výuku základů algoritmizace a programování.

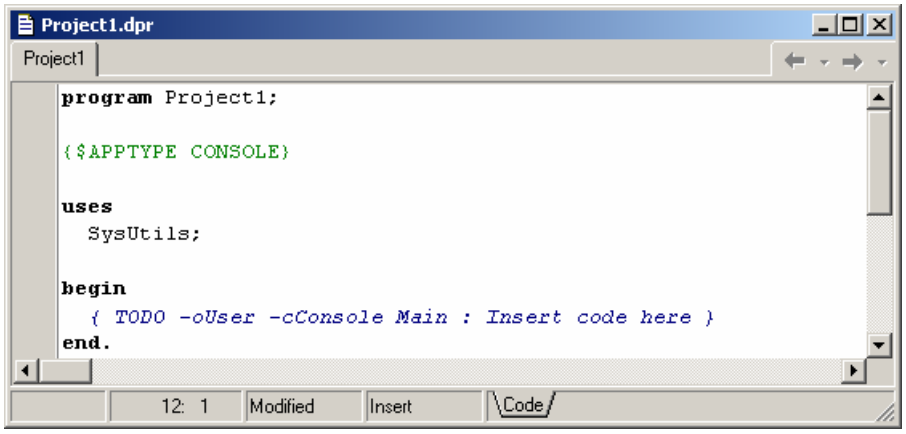
Používání Delphi, kterým firma Borland navázala na své dřívější úspěšné kompilátory jazyka Turbo-Pascal (vytvořené jen pro operační systém DOS), přináší následující výhody:

- Delphi je standardní aplikace Windows se standardním ovládáním (práce se soubory, …),
- součástí Delphi je kvalitní editor pro psaní programů se standardním ovládáním (práce s bloky textu, použití schránky (*clipboard*), ...),
- Delphi poskytuje kvalitní ladící prostředí,
- pro účely výuky je možné získat příslušnou verzi Delphi zdarma.

Konzolová aplikace je čistě 32-bitový program bez grafického rozhraní. Konzolová aplikace se spouští v konzolovém okně, které simuluje původní prostředí MS DOS. Okno je textové, má rozměry 80 znaků na šířku a 25 řádků na výšku a standardní výstup je bílým písmem na černé pozadí. Vstup a výstup se provádí stejně jako v původních kompilátorech jazyka Turbo-Pascal pomocí příkazů *Read, ReadLn, Write, WriteLn*.

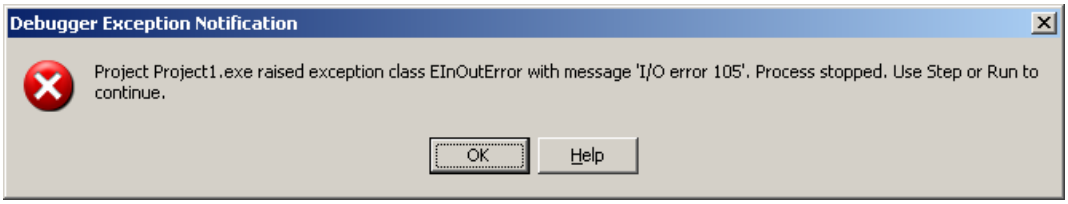
Zdrojový kód (zdrojový forma programu) pro konzolovou aplikaci je uložen v textovém souboru s příponou *.dpr* (*Delphi project*). Přípona *.pas* se používá pro ukládání zdrojových forem knihoven (unitů).

Zdrojový kód konzolové aplikace můžeme buď vytvořit sami, nebo použít (od verze 5) pro vytvoření kostry programu kouzelníka (*console application wizard*). Vyvoláme jej (ve verzi 7) z hlavního menu volbou *File/New/Other.../Console application*. Vznikne následující jednoduchý projekt:



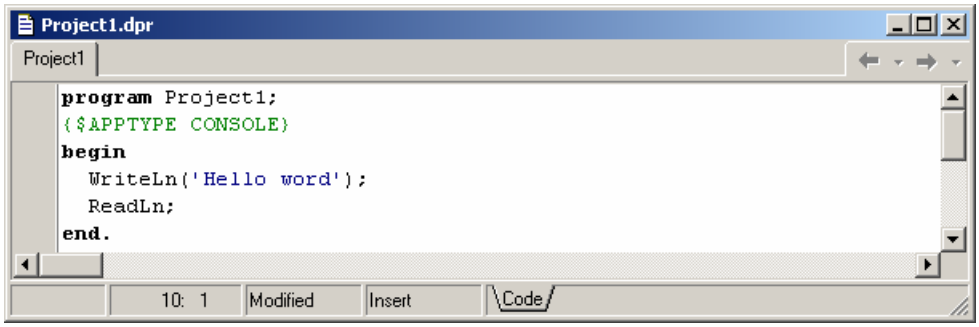
Klíčové slovo **program** říká kompilátoru, že se jedná o hlavní program. Název programu *Project1* se musí shodovat s názvem souboru, do nějž je zdrojová verze programu s příponou *.dpr* uložena. Kompilátor pak vytvoří spustitelný program se stejným názvem a příponou *.exe*. Pokud ale zdrojový soubor uložíme s jiným názvem, kompilátor sám zajistí i změnu názvu za klíčovým slovem **program**. Název programu a tedy i název souboru musí proto vyhovovat podmínkám pro identifikátor, což je posloupnost písmen a číslic, která začíná písmenem, nesmí tedy obsahovat mezeru, místo písmene se může použít znak _ (podtržítko).

Direktiva *{$APPTYPE CONSOLE}* kompilátoru říká, že se jedná o konzolovou aplikaci. Pokud ji zapomeneme uvést a použijeme např. příkaz *Read* pro čtení nebo *Write* pro zápis, dostaneme následující chybové hlášení (s číslem chyby 105 pro zápis nebo 106 pro čtení):

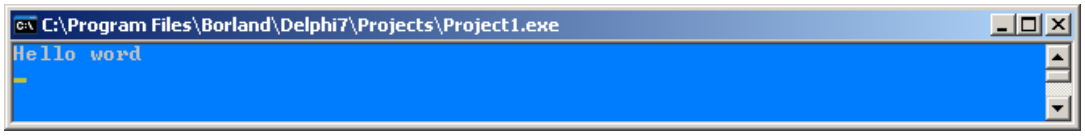


Za klíčovým slovem **uses** se uvádí seznam použitých knihoven. Knihovna *SysUtils* je nepovinná a může být ze seznamu odebrána.

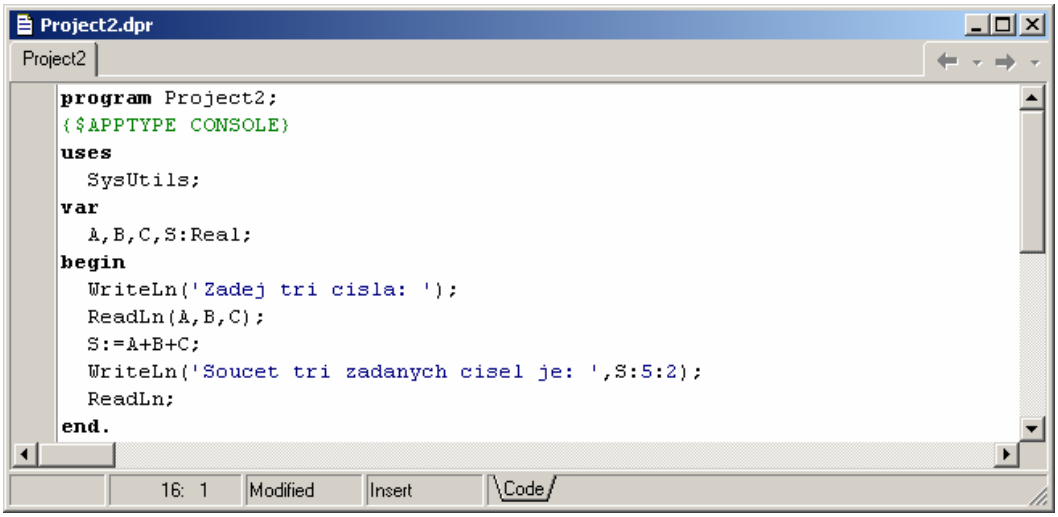
Vlastní program zapíšeme mezi závorky **begin** a **end**. Automaticky vygenerovaný komentář mezi nimi můžeme také odstranit. Nejjednodušší program vypadá např. následovně:

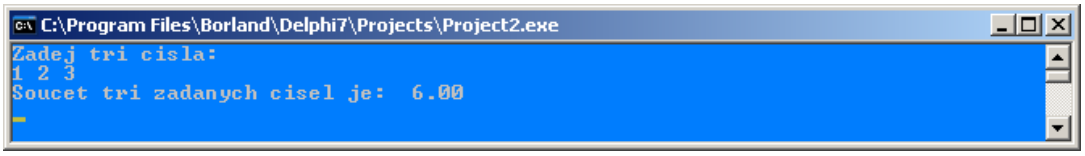


Při jeho spuštění (položka *Run/Run* hlavního menu, blíže viz 4.1. *Položky menu určené pro ladění* a 4.2. *Použití bodu zastavení*) se vytvoří konzolové okno, vypíše se text uvedený v příkazu *WriteLn* a díky příkazu *ReadLn* program čeká na stisknutí klávesy *Enter*.



Složitější příklad pro výpočet součtu tří zadaných čísel:





Poznámka

V programech zapsaných v jazyku Turbo-Pascal se často pro práci s textovou obrazovkou („předchůdcem“ konzolového okna) využívaly příkazy z knihovny *Crt*, např. *Clrscr* (pro smazání obrazovky), či *GotoXY* (pro přemístění kurzoru na požadovanou pozici). Existují sice knihovny, které knihovnu *Crt* v Delphi „umí“ nahradit a které by tedy umožnily mechanický převod (jen) některých starších programů do Delphi, ale žádná z těchto knihoven není standardní součástí Delphi. Pro vytváření standardních aplikací má totiž Delphi mnohem lepší prostředky, a vytváření takových konzolových aplikací, které by se bez náhražky knihovny *Crt* neobešly, je „nerozumné“ a nemá praktický význam.

4. LADĚNÍ PROGRAMŮ V DELPHI

Výklad

Hlavní výhoda integrovaného vývojového prostředí Delphi (IDE - *integrated development environment*) je integrovaný ladící prostředek, tzv. ***debugger*** (*bugg* - veš, *debugger* - doslovně odvšivovač). Ladící prostředí umožňuje snadno nastavovat v programu ***body zastavení*** (*breakpoints*), prohlížet hodnoty proměnných, krokovat program a mnohem více. Použitím ladícího prostředí můžete rychle a snadno zjistit co se děje (nebo neděje) s vaším programem, když je spuštěn. Dobré ladící prostředí je nezbytné pro účinný vývoj programů.

Ladění programů není jen hledání a odstraňování chyb, je to vývojový nástroj. Proto je důležité a užitečné naučit se používat všechny možnosti, které integrované ladící prostředí poskytuje.

Poznámka

Termín *program* má velmi široké užití. Někdy je však třeba odlišit zda se jedná např. jen o zdrojovou formu programu (zápis v Delphi) nebo o cílový program (soubor typu .exe). Bude-li třeba v dalším textu oba uvedené významy odlišit bude pro zdrojovou formu programu používán termín ***zdrojový kód*** a pro cílový program termín ***cílový kód***.

Pro soubor se zápisem zdrojového kódu se užívá termín ***zdrojový soubor***. Zdrojový kód konzolové aplikace v Delphi je zapsán buď jen v jednom zdrojovém souboru (soubor typu *.dpr*), nebo ve více zdrojových souborech (jeden základní zdrojový soubor typu *.dpr* může obsahovat odkazy na více souborů typu *.pas*, obsahem každého souboru typu *.pas* je zdrojový kód jedné jednotky – unitu).

4.1. Položky menu určené pro ladění

Výklad

Před tím, než přistoupíme k detailům ladícího prostředí, si projdeme položky menu, které se vztahují k ladění programů. Některé z těchto položek jsou v hlavním menu pod položkou *Run*, ostatní jsou přístupné v kontextovém menu editoru (vyvolá se pravým tlačítkem myši v okně editoru zdrojového kódu) pod položkou *Debug*.

Tab. 1. Položky pro ladění menu *Run*.

Položka	*Zkratka*	*Popis*
Run	F9	Přeloží program (je-li to zapotřebí) a potom program spustí pod kontrolou integrovaného ladícího prostředí.
Parameters		Umožní vložit parametry příkazového řádku pro program.
Step Over	F8	Krokování programu. Provede jeden řádek zdrojového kódu a zastaví se na dalším řádku. Pokud je na řádku volání podprogramu, provede se celý podprogram najednou.
Trace Into	F7	Krokování programu. Provede jeden řádek zdrojového kódu a zastaví se na dalším řádku, je-li však je na řádku volání podprogramu a je k dispozici zdrojový kód podprogramu, bude se podprogram také krokovat.
Run to Cursor	F4	Spustí program (pokud neběží) a provádí jej až po řádek editoru, na kterém je kurzor.
Show Execution Point		Zobrazí řádek, na kterém je zastaveno provádění programu v okně editoru. Pracuje pouze je-li provádění programu pozastaveno.
Program Pause		Pozastaví provádění programu, jakmile se má provést řádek, jehož zdrojový text je k dispozici.
Program Reset	Ctrl+F2	Bezpodmínečně přeruší provádění programu a řízení se vrátí vývojovému prostředí Delphi.
Evaluace/Modify	Ctrl+F7	Zobrazí dialogové okno *Evaluate/Modify*, které umožní prohlédnout si a případně změnit hodnotu proměnné za běhu programu.
Add Watch	Ctrl+F5	Zobrazí okno *Watch list* (okno, které umožňuje prohlížet si hodnoty vložených proměnných) a přidá proměnnou pod kurzorem do okna.
Add Breakpoint		Zobrazí podmenu pro zadání různých druhů bodů zastavení.

Tab. 2. Položky menu *Debug* kontextového menu editoru

Položka	*Zkratka*	*Popis*
Toggle Breakpoint	F5	Zapíná nebo vypíná bod zastavení (*breakpoint*) pro aktuální řádek editoru. V editoru se objeví červený pruh.
Run to Cursor	F4	Spustí program (pokud neběží) a provádí jej až po řádek editoru, na kterém je kurzor.
Evaluace/Modify	Ctrl+F7	Zobrazí dialogové okno *Evaluate/Modify*, které umožní prohlédnout si a případně změnit hodnotu proměnné za běhu programu.
Add Watch at Kursor	Ctrl+F5	Zobrazí okno *Watch list* (okno, které umožňuje prohlížet si hodnoty vložených proměnných) a přidá proměnnou pod kurzorem do okna.

Uvedené položky menu se často používají při ladění programů. Je užitečné seznámit se s klávesovými zkratkami (*horkými klávesami*) pro ladící operace.

4.2. Použití bodů zastavení

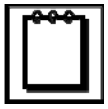
Výklad

Ladění programu začíná jeho spuštěním v ladícím prostředí. Ladící prostředí je použito automaticky při spuštění programu stisknutím tlačítka *Run* na tlačítkové liště (*toolbar*). Program lze také spustit položkou *Run/Run* hlavního menu nebo horkou klávesou *F9*. Program spuštěný ve vývojovém prostředí Delphi běží plnou rychlostí a zastaví se pouze na nastavených bodech zastavení.

Bod zastavení (***breakpoint***, též *bod přerušení*) je značka, která řekne ladícímu prostředí, aby zastavilo provádění programu, když na ni ve zdrojovém kódu narazí.

Nastavení a zrušení bodu zastavení

Bod zastavení můžeme nastavit kliknutím myší do okna editoru v šedém pruhu vlevo před řádkem, na kterém chceme bod zastavení nastavit. ***Značka bodu zastavení*** - červený kruh se objeví před daným řádkem a celý řádek je vybarven červeně. Bod zastavení se odstraní kliknutím myší na jeho značku (červené kolečko), značka a červené vybarvení řádku zmizí. Lze také použít horkou klávesu *F5* pro přepínání nastavení/zrušení bodu zastavení nebo položku kontextového menu *Debug/Toggle Breakpoint*.



Poznámka

Bod zastavení má smysl (je platný) pouze na řádku, ze kterého je generován nějaký příkaz cílového kódu (před těmito řádky se po překladu objeví modrý bod). Bod zastavení např. není platný, když je nastaven na prázdném řádku, na řádku s komentářem nebo na řádku s deklarací. Zadání neplatného bodu zastavení je však možné a bohužel se projeví až po spuštění programu (značka neplatného bodu zastavení je přeškrtnuta a řádek je vybarven zeleně). Bod zastavení může být nastaven na **end** procedury nebo funkce.

Když je program spuštěn v ladícím prostředím, chová se normálně, dokud nenarazí na nastavený bod zastavení. V tomto okamžiku se aktivním oknem stane okno editoru. Řádek, na němž se program zastavil, je vybarven červeně, neboť červená barva označuje řádek s nastaveným bodem zastavení.

Pokud krokujeme program, potom aktuální řádek, který se provede při následujícím kroku, je označen modrým pruhem a před tímto řádkem je zelená šipka. V okamžiku, kdy aktuální řádek přijde na řádek s nastaveným bodem zastavení, bude vybarvení řádku červené

a před řádkem zůstane zelená šipka indikující aktuální řádek.

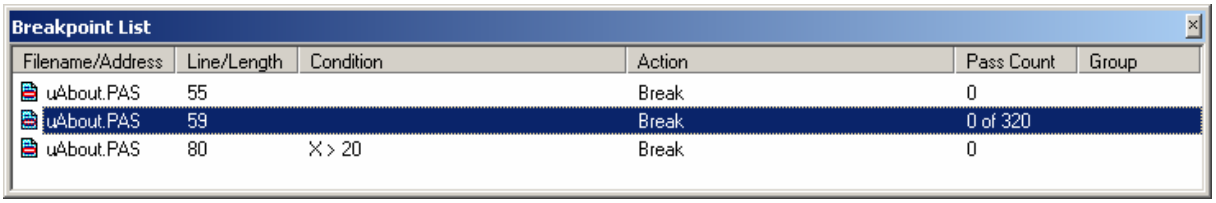
Když se program zastaví na bodu zastavení, je možno prohlížet si proměnné, měnit jejich hodnotu, prohlížet zásobník volání podprogramů (*call stack*) nebo dále krokovat program. Po vykonání všech ladících kroků lze pokračovat v normálním provádění programu stisknutím tlačítka *Run* nebo stiskem horké klávesy *F9*. Aplikace poběží dál normálně, dokud nenarazí na další bod zastavení.

Poznámka

Pokud změníme zdrojový kód programu v okamžiku krokování programu a pokusíme se dále pokračovat v provádění programu, vývojové prostředí se zeptá v okně se zprávou, zda-li chceme překompilovat zdrojový kód. Pokud odpovíme že ano, provádění programu bude přerušeno, zdrojový kód bude překompilován a program bude spuštěn znovu. Násilné přerušení provádění programu může v některých případech způsobit komplikace. Pokud překompilování programu nepovolíme a budeme pokračovat v krokování programu, nebude ladící prostředí brát provedené změny ve zdrojovém kódu v úvahu.

Okno bodů zastavení (*Breakpoint List*)

Vývojové prostředí udržuje seznam nastavených bodů zastavení. Tyto body zastavení můžeme vidět v okně bodů zastavení (okno *Breakpoint List*), které se vyvolá volbou *View/Debug Windows/Breakpoints* z hlavního menu.



Okno bodů zastavení má čtyři sloupce:

- **Sloupec *Filename/Address*** ukazuje jméno souboru se zdrojovým kódem, ve kterém je bod zastavení nastaven.
- **Sloupec *Line/Length*** ukazuje číslo řádku, ve kterém je bod zastavení nastaven.
- **Sloupec *Condition*** ukazuje podmínku zastavení, která může být pro bod zastavení nastavena.
- **Sloupec *Pass Count*** ukazuje podmínku počtu průchodů, která může být pro bod zastavení nastavena. (Podmínka zastavení a podmínka počtu průchodů je probrána podrobněji - viz dále *Podmíněné body zastavení*).

Velikost sloupců může být změněna chycením myší za čáry, které oddělují jednotlivé sloupce v řádku s nadpisy sloupců.

Poznámka

Sloupec *Pass Count* neukazuje aktuální počet průchodů přes bod zastavení, ukazuje pouze podmínku počtu průchodů přes bod zastavení, která je nastavena.

Kontextová menu okna bodů zastavení

Okno bodů zastavení má dvě kontextová menu (primární a sekundární).

Primární kontextové menu, jehož položky jsou uvedeny v tabulce 3, uvidíte, když kliknete pravým tlačítkem myši na libovolném bodu zastavení.

Tab. 3. Primární kontextové menu seznamu bodů zastavení.

Položka	*Popis*
Enable	Povolí nebo zakáže bod zastavení. Když je bod zastavení zakázán, potom jeho ikona v okně seznamu bodů zastavení zešedne. Ve zdrojovém okně ikona bodu zastavení taky zešedne a řádek s bodem zastavení je vybarven zeleně, což indikuje, že bod zastavení je zakázán.
Delete	Odstraní bod zastavení.
View Source	Odroluje zdrojový soubor v editoru kódu tak, aby byl zobrazen vybraný bod zastavení.
Edit Source	Umístí editační kurzor do řádku ve zdrojovém souboru, na němž je nastaven bod zastavení, aktivním oknem se stane okno editoru kódu.
Properties	Zobrazí dialogové okno vlastnosti bodů zastavení (*Source Breakpoint Properties*).
Dockable	Určí, jestli se bude okno bodů zastavení přichytávat k jiným oknům.

Poznámka

Pro rychlou editaci zdrojového řádku kódu, na kterém je nastaven bod zastavení, proveďte dvojklik myší na řádek bodu zastavení v okně bodů zastavení. Je to stejné, jako když vyberete položku *Edit Source* z primárního kontextového menu okna bodů zastavení.

Sekundární kontextové menu se zobrazí kliknutím pravým tlačítkem myši, když kurzor je nad částí okna bodů zastavení. Sekundární kontextové menu má položky *Add* - přidat,

Delete All - smazat všechny, *Disable All* - zakázat všechny, *Enable All* - povolit všechny, a *Dockable* - přichytnout. Význam těchto položek je zřejmý z jejich názvu.

Poznámka

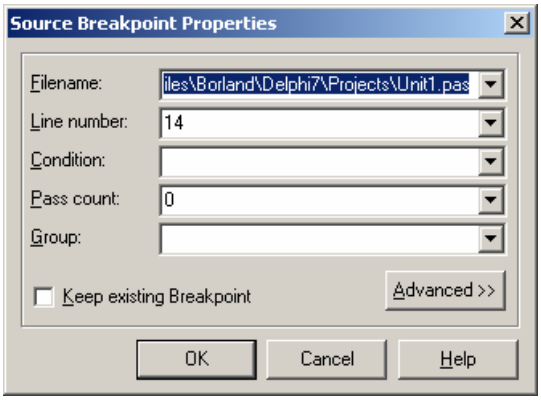
Položka *Add* (přidat) není příliš užitečná, je mnohem snadnější přidat bod zastavení v editoru kódu než přidat bod zastavení pomocí této položky kontextového menu.

Povolení a zakázání bodů zastavení

Body zastavení mohou být povoleny nebo zakázány libovolně mnohokrát. Bod zastavení zakážeme, pokud dočasně potřebujeme spustit program normálně. Bod zastavení povolíme později, aniž bychom jej museli znovu vytvářet. Ladící prostředí ignoruje body zastavení, které jsou zakázané. Pro povolení nebo zakázání bodu zastavení klikněte pravým tlačítkem myši na bod zastavení v okně bodů zastavení a vyberte položku *Enable* (povolit) v kontextovém menu. Stejná položka se nachází v kontextovém menu, které lze vyvolat kliknutím pravým tlačítkem myši na ikonu bodu zastavení (červené nebo šedé kolečko) před řádkem s bodem zastavení v editoru kódu.

Upravení bodu zastavení

Chcete-li upravit vlastnosti bodu zastavení, vyberte položku *Properties* (vlastnosti) z primárního kontextového menu okna bodů zastavení. Pokud to uděláte, zobrazí se okno *Source Breakpoint Properties*.



Hlavní důvod pro upravení vlastností bodů zastavení je přidání podmínky (*Conditional breakpoints* - podmíněné body zastavení jsou popisovány v sekci *Podmíněné body zastavení*).

Pro odstranění bodu zastavení vyberte bod zastavení v okně bodů zastavení a stiskněte klávesu *Delete* na klávesnici. Pro odstranění všech bodů zastavení vyberte položku *Delete All* v sekundárním kontextovém menu okna bodů zastavení.

Jednoduchý bod zastavení způsobí přerušení provádění programu, jakmile se program k tomuto bodu dostane.

Když je bod zastavení zadán, je to automaticky jednoduchý bod zastavení. Jednoduchý bod zastavení nepotřebuje mnoho vysvětlování. Když program dosáhne jednoduchého bodu zastavení, provádění programu je pozastaveno a ladící prostředí čeká na požadavek. Při ladění programů se nejčastěji používají jednoduché body zastavení.

Podmíněný bod zastavení je rezervován pro speciální případy, kdy je potřeba větší kontroly nad ladícím procesem.

Podmíněný bod zastavení způsobí přerušení provádění programu pouze při splnění předem definované podmínky.

Pro vytvoření podmíněného bodu zastavení nejprve nastavte bod zastavení v editoru kódu. Potom vyberte položku *View/Debug Windows/Breakpoints* hlavního menu pro zobrazení okna bodů zastavení. Klikněte pravým tlačítkem myši na bod zastavení, pro který chcete nastavit podmínky a vyberte *Properties* (vlastnosti). Zobrazí se dialogové okno *Source Breakpoint Properties* (vlastnosti bodu zastavení), ve kterém můžete nastavit podmínky pro bod zastavení.

Existují dva typy podmíněných bodů zastavení. První typ je bod zastavení s logickou podmínkou. Zadejte podmínku do pole *Condition* (podmínka) v dialogovém okně *Source Breakpoint Properties*. Když je program spuštěn, podmíněný výraz je vyhodnocen při každém průchodu přes bod zastavení. Když výraz nabude hodnoty *True*, provádění programu je zastaveno, v opačném případě je bod zastavení ignorován. Např. v uvedeném okně bodů zastavení má poslední bod zastavení nastaven podmíněný výraz $X > 20$. Jestliže někdy při provádění programu nabude proměnná X hodnotu větší než 20, provádění programu se na tomto bodě zastavení v příslušném okamžiku přeruší. Pokud hodnota proměnné X není nikdy větší než 20, program se na tomto bodu zastavení nikdy nepřeruší.

Druhý typ podmíněného bodu zastavení je bod zastavení s počtem průchodů. Provádění programu se na tomto typu bodu zastavení přeruší pouze v případě, že počet průchodů přes

bod zastavení odpovídá nastavené hodnotě. Počet průchodů přes bod zastavení se nastaví v dialogovém okně *Source Breakpoint Properties* v položce *Pass Count* (počet průchodů). Pokud nastavíte počet průchodů na hodnotu 5, provádění programu se přeruší při pátém průchodu přes bod zastavení.

Použijte bod zastavení s počtem průchodů, když potřebujete, aby program prošel přes bod zastavení určitý počet krát, než se přeruší a vy si pak budete moci prohlížet hodnoty proměnných, krokovat program nebo provádět další ladící úkoly.

Poznámka

Podmíněný bod zastavení zpomaluje normální provádění programu, protože podmínka se vyhodnocuje při každém průchodu přes bod zastavení. Jestliže je váš program při ladění pomalý, zkontrolujte seznam bodů zastavení, zda tu není nějaký zapomenutý podmíněný bod zastavení.

Tip: Fakt, že podmíněný bod zastavení zpomaluje program, můžete využít. Pokud máte nějaký proces, který potřebujete vidět zpomaleně, nastavte jeden nebo více podmíněných bodů zastavení v dané části zdrojového kódu. Nastavte podmínku, která nebude nikdy splněna, a váš program se zpomalí, ale nezastaví.

Příkaz Run to Cursor

Příkaz *Run to Cursor* (nachází se v menu *Run* hlavního menu, horká klávesa *F4*) přeloží program a spustí jej až po řádek, na kterém se nachází kurzor. Na tomto řádku se program zastaví, jakoby zde byl umístěn bod zastavení.

Příkaz *Run to Cursor* pracuje jako dočasný bod zastavení. Můžete použít tento příkaz raději než nastavovat bod zastavení na řádku, který chcete prozkoumat. Stačí jen umístit kurzor na řádek, na němž chcete program přerušit, a zvolit příkaz *Run to Cursor* (nebo stisknout klávesu *F4*). Ladící prostředí se zachová stejně, jako kdybyste umístili bod zastavení na tento řádek. Výhoda je, že nemusíte odstraňovat bod zastavení po skončení ladění této části programu.

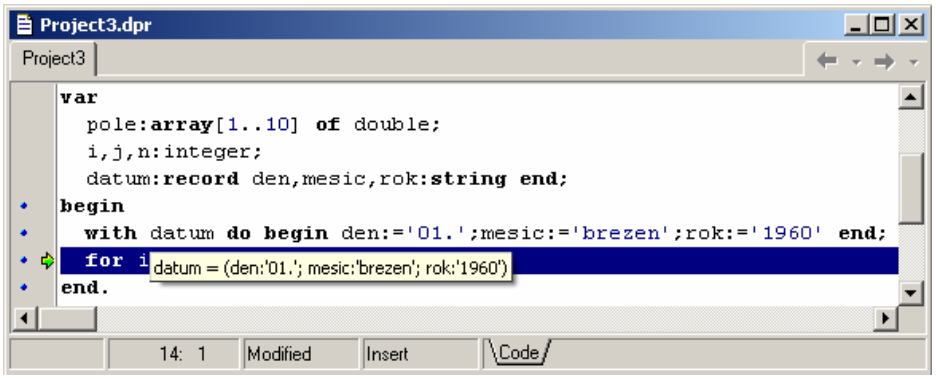
4.3. Prohlížení proměnných

Výklad

Rychlé prohlížení hodnot proměnných

Ladící prostředí umožňuje snadné prohlížení hodnot proměnných. Jakmile zastavíte program v bodě zastavení, umístěte kurzor myši nad identifikátor proměnné a po chvíli se objeví okno s hodnotou proměnné.

Pro různé typy proměnných se zobrazují hodnoty různě. Pro jednoduché datové typy *Integer*, *Char*, *Byte* a též např. pro **array** a **string** se zobrazí aktuální hodnota daného typu. Pro dynamické proměnné se zobrazí jejich adresa v paměti. Pro proměnnou typu záznam se zobrazí hodnoty všech položek záznamu. Následující obrázek ukazuje hodnoty proměnné typu záznam:



Poznámka

V některých případech se hodnota proměnné nezobrazí. Příčinou může být optimalizace překladu, kterou je vhodné pro účely ladění vypnout (v menu *Project/Options*..., záložka *Compiler*, volba *Optimization*). Hodnota proměnné se zobrazí pouze v případě, že program se zastaví v místě platnosti deklarace proměnné. Dalším případem, kdy se hodnota nezobrazí, je uvnitř příkazu **with**.

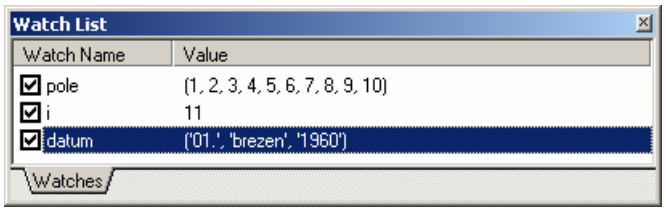
Například pro zdrojový kód:

```
with Bod do
begin
  X := 20;
  Y := 50;
end;
```

Když umístíte kurzor nad proměnnou *X*, nic se nezobrazí, protože *X* není proměnná, ale jen položka záznamu *Bod*. Její hodnotu si můžete prohlédnout umístěním kurzoru nad proměnnou *Bod* (zobrazí se všechny položky záznamu).

Okno *Watch List*

Co můžete udělat, když se program zastaví na bodu zastavení? Obvykle zastavíte provádění programu v bodě zastavení, abyste si mohli prohlédnout hodnoty jedné nebo více proměnných. Můžete zjistit, jestli proměnné mají hodnoty, které mají mít. To je základní funkce okna *Watch List* – umožňuje prohlížet hodnoty proměnných. Zde můžete umístit tolik proměnných, kolik potřebujete. Na následujícím obrázku je příklad okna *Watch List* během ladění programu.



V okně *Watch List* je zobrazeno jméno proměnné a za ním hodnota proměnné. Jak je zobrazena hodnota proměnné je určeno datovým typem proměnné a nastavením zobrazení pro sledovanou položku.

Kontextové menu okna prohlížení proměnných (*Watch List*)

Jako všechna ostatní okna Delphi má i okno *Watch List* svoje vlastní kontextové menu, jehož položky jsou uvedeny v tab. 4.

Tab. 4. Kontextové menu okna Watch List.

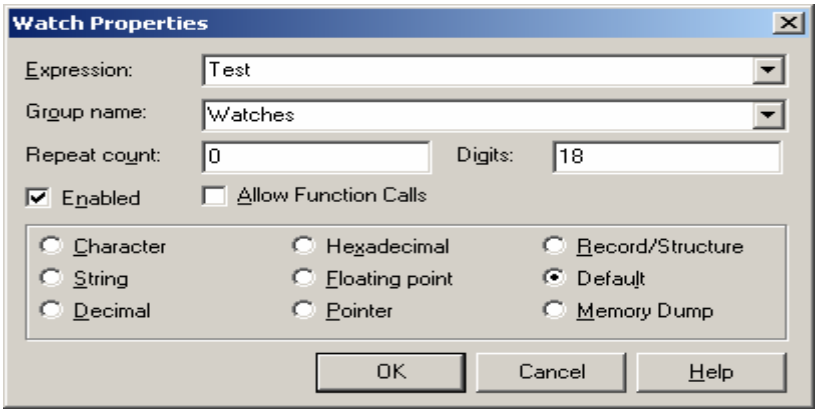
Položka	*Popis*
Edit Watch	Umožní editovat vybranou položku v okně *Watch Properties*.
Add Watch	Přidá novou položku do okna *Watch List*.
Enable Watch	Povolí zobrazování hodnoty proměnné dané položky.
Disable Watch	Zakáže zobrazování hodnoty proměnné dané položky.
Delete Watch	Odstraní danou položku z okna *Watch List*.
Enable All Watches	Povolí zobrazování hodnoty proměnné všech položek.
Disable All Watches	Zakáže zobrazování hodnoty proměnné všech položek.
Delete All Watches	Odstraní všechny položky okna *Watch List*.

Stay on Top	Přinutí okno *Watch List* zůstat nad všemi ostatními okny vývojového prostředí.
Break When Changed	Program se zastaví v okamžiku, kdy se hodnota proměnné změní. Proměnná zobrazená v okně *Watch List* je zobrazena červeně pro indikaci zastavení programu při změně hodnoty proměnné.
Dockable	Určuje, zda se okno *Watch List* připojí (přilepí) k jinému oknu, např. editoru.

První dvě položky kontextového menu *Edit Watch* a *Add Watch* vyvolají dialogové okno *Watch Properties*, které bude popsáno dále.

Použití dialogového okna *Watch Properties*

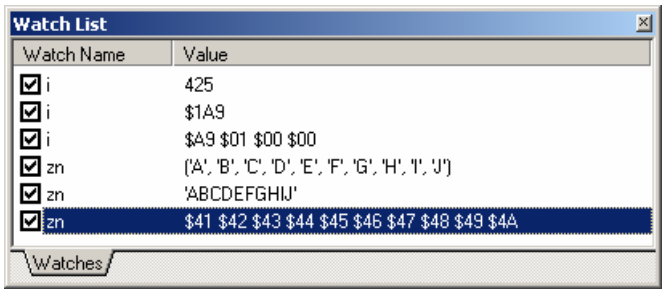
Dialogové okno *Watch Properties* použijete, když přidáváte nebo editujete položku okna *Watch*. Na následujícím obrázku je toto okno při editaci proměnné *Test.*



Do položky *Expression* v okně *Watch Properties* zadáte jméno proměnné, kterou chcete přidat do okna *Watch List*. Tuto položku lze rozvinout a vybrat dříve použitá jména. Můžete použít položku *Repeat Count* pro prohlížení pole. Například pokud máte deklarovanou proměnnou *Pole* typu ***array*** *[1..50]* ***of*** *Real* a chcete vidět deset hodnot pole počínaje dvacátou hodnotou, zadáte do položky *Expression* text *Pole[20]* a do položky *Repeat Count* hodnotu 10 (pokud zadáte jen jméno pole do okna *Watch List*, budou zobrazeny všechny hodnoty pole).

Položka *Digits* je použita pouze při prohlížení desetinných čísel. Zadejte počet významných číslic, které chcete vidět. Zobrazené číslice jsou zaokrouhleny (ne useknuty). Další položka *Enabled* v tomto dialogovém okně určuje, jestli daná položka okna *Watch List* je aktivní.

Zbývající volby v okně *Watch Properties* jsou volby způsobu zobrazení hodnot proměnných. Každý datový typ má předdefinovaný způsob zobrazení hodnoty, který se použije při výběru volby *Default*. Výběrem jiné volby pohledu můžete vidět hodnotu ve zvoleném formátu. Na následujícím obrázku je okno *Watch List* se dvěma proměnnými a s různými volbami zobrazení hodnot. Proměnná *zn* je typu pole znaků a proměnná *i* je typu *Integer*.



Pro změnu prohlížené položky klikněte pravým tlačítkem myši na položku v okně *Watch List* a vyberte volbu *Edit Watch* z kontextového menu. Tuto volbu můžete také vyvolat dvojklikem levým tlačítkem myši na vybranou položku. Zobrazí se dialogové okno *Watch Properties* a můžete upravit vlastnosti vybrané položky.

Povolení a zakázání položek okna *Watch List*

Podobně jako body zastavení, jednotlivé položky okna *Watch List* mohou být povoleny nebo zakázány. Když je položka zakázána, je zobrazena šedě a místo její hodnoty je zobrazeno <*disabled*>. Položku lze zakázat nebo povolit v kontextovém menu položky, v dialogovém okně *Watch Properties* nebo přímo v okně *Watch List* pomocí zaškrtávacího políčka před jménem proměnné.

Poznámka

Můžete chtít zakázat zobrazování hodnot proměnných, které dočasně nepotřebujete, a později opět povolit. Zobrazování hodnot příliš mnoha proměnných zpomaluje provádění programu při jeho ladění, protože všechny hodnoty musí být zobrazeny při vykonání každého řádku.

Přidání proměnné do okna *Watch List*

Proměnné můžete přidat do okna *Watch List* několika způsoby. Nejrychlejší cesta je kliknout na proměnnou v editoru a z kontextového menu vybrat položku *Add Watch* nebo

stisknou kombinaci kláves *Ctrl+F5*. Položka bude okamžitě přidána do okna *Watch List*. Potom můžete upravit její vlastnosti, je-li to zapotřebí. Pro přidání proměnné do okna *Watch List* bez jejího vyhledání ve zdrojovém kódu vyberte položku *Run/Add Watch* z hlavního menu. Zobrazí se dialogové okno *Watch Properties*, do kterého zadáte jméno přidávané proměnné.

Použití okna Watch List

Když se program zastaví v bodě zastavení, okno *Watch List* zobrazí aktuální hodnoty všech proměnných. Pokud okno *Watch List* není otevřeno, otevřete jej volbou *View/Debug Windows/Watches* z hlavního menu.

Tip: Okno *Watch List* lze přichytit ke spodní části okna editoru kódu a bude tak stále viditelné při krokování programu.

Za jistých okolností se místo hodnoty proměnné zobrazí zpráva. Například, když jsme v programu mimo oblast platnosti deklarace proměnné nebo proměnná není nalezena, zobrazí se zpráva *Undeclared identifier:'X'* vedle identifikátoru proměnné. Pokud program není spuštěn nebo není pozastaven v bodě zastavení, zobrazí se pro všechny položky zpráva *[process not accessible]*. Zakázané položky zobrazují text *<disabled>*. V případě, že je zapnuta optimalizace kódu (položka hlavního menu *Project/Options…*, záložka *Compiler*, volba *Optimalization*), bude u proměnných, pro které se optimalizace provedla, zobrazena zpráva *Variable 'X' inaccessible here due to optimization* (proměnná je nepřístupná z důvodu optimalizace.

Tip: Pro ladění programu je vhodné optimalizaci vypnout.

4.4. Krokování programu - příkazy *Step Over a Trace Into*

Výklad

Když se provádění programu zastaví v bodě zastavení, můžete udělat několik věcí pro zjištění, co se děje ve vašem programu. Můžete nastavit prohlížení hodnot proměnných v okně *Watch List*, měnit hodnoty proměnných v okně *Evaluate/Modify*, prohlížet pořadí vnořeného volání procedur v okně *Call Stack*. Můžete také krokovat program, abyste viděli, co se děje s hodnotami proměnných při provádění jednotlivých řádků zdrojového kódu.

Při krokování programu vidíte, že řádek zdrojového kódu, který bude vykonán v příštím kroku, je zvýrazněn modře. Pokud máte otevřeno okno *Watch List*, jsou hodnoty proměnných aktualizovány při vykonání každého řádku. Ladící prostředí má dva příkazy pro krokování programu: *Step Over* a *Trace Into*.

Příkaz *Step Over*

Příkaz *Step Over* znamená vykonat následující řádek zdrojového kódu a zastavit se na řádku bezprostředně následujícím. Pokud prováděný řádek obsahuje volání procedur nebo funkcí, budou volání provedena. Pro provedení příkazu *Step Over* stiskněte klávesu *F8* nebo vyberte příkaz *Run/Step Over* v hlavním menu.

Příkaz *Trace Into*

Příkaz *Trace Into* umožňuje provádět krokování procedury nebo funkce. Vyvoláme-li tento příkaz pro řádek, který obsahuje volání procedury nebo funkce, aktuálním řádkem zvýrazněným modře se stane první řádek první procedury nebo funkce. Program můžete dále krokovat po řádcích použitím *Step Over* nebo *Trace Into* podle potřeby. Klávesová zkratka pro příkaz *Trace Into* je *F7*.

Poznámka

Procedury nebo funkce je možné krokovat pouze v případě, že je k dispozici jejich zdrojový kód. Pokud se zdrojový kód nachází v jiném souboru, automaticky se tento soubor otevře v editoru.

Pokud již dále nepotřebujete program ladit, můžete jej opět spustit plnou rychlostí pomocí tlačítka *Run* na liště (horká klávesa *F9*). Program bude prováděn normálně až do konce nebo pokud nedosáhne dalšího bodu zastavení.

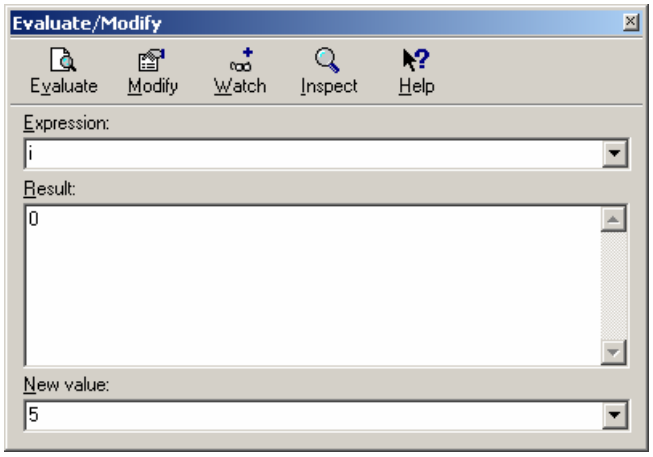
4.5. Ostatní ladící nástroje

Výklad

Delphi obsahuje další ladící nástroje, které jsou velmi užitečné v rukou zkušeného programátora.

Dialogové okno *Evaluate/Modify*

Dialogové okno *Evaluate/Modify* umožňuje prohlížet aktuální hodnoty proměnných a modifikovat tyto hodnoty, pokud je to zapotřebí. Změna hodnoty proměnné při ladění programu umožňuje testovat efekt různých hodnot proměnných bez opakovaného spouštění programu a změny vstupních údajů. Následující obrázek ukazuje okno *Evaluate/Modify* při prohlížení proměnné *i*.



Dialogové okno *Evaluate/Modify* se chová podobně jako okno *Watch List*. Pokud kliknete na identifikátor proměnné v editoru a zvolíte v hlavním menu *Run/Evaluace/Modify*, zobrazí se okno *Evaluate/Modify* a položka *Expression* bude vyplněna identifikátorem proměnné.

Položka *Expression* se používá pro zadání identifikátoru proměnné nebo výrazu, který chcete vyhodnotit. Když kliknete na tlačítko *Evaluate* (nebo stisknete klávesu *Enter*), výraz bude vyhodnocen a výsledek se zobrazí v položce *Result*.

Poznámka

Dialogové okno *Evaluate/Modify* může být použito jako rychlá kalkulačka. Můžete vložit matematický výraz obsahující čísla v desítkové nebo šestnáctkové soustavě a získat výslednou hodnotu. Například když vložíte výraz *$100+256* do položky *Expression* a stisknete *Evaluate* (nebo *Enter*), výsledek *512* se zobrazí v položce *Result*. Vyhodnocovat lze

také logické výrazy. Program musí být zastaven v bodě zastavení, pokud chcete použít hodnoty proměnných z programu.

Když chcete změnit hodnotu proměnné, zadejte novou hodnotu do položky *New Value* a stiskněte tlačítko *Modify*. Hodnota proměnné bude změněna na novou vloženou hodnotu. Pokud budete pokračovat v provádění programu, bude použita tato nová hodnota.

Poznámka

Dialogové okno *Evaluate/Modify* neaktualizuje automaticky hodnoty proměnných při krokování programu, jako to dělá okno *Watch List*. Pokud se při krokování programu hodnota proměnné změní, musíte stisknout tlačítko *Evaluate*, aby se tato změna projevila.

Okno *Call Stack*

Když je váš program spuštěn, můžete si v okně *Call Stack* prohlédnout všechny procedury a funkce, které váš program volá. Okno se vyvolá z hlavního menu *View/Debug Windows/Call Stack*. Procedury a funkce se v tomto okně zobrazí v pořadí, jak byly volány. Poslední volaná funkce je v okně nahoře.

Dvojklikem na proceduru v okně *Call Stack* se dostaneme do zdrojového kódu procedury. Obsah okna *Call Stack* je užitečný po chybě *Windows Access Violation*. Můžeme zde zjistit, která část programu se prováděla při vzniku chyby. To je první krok k tomu, abychom zjistili, co je špatně.

Tip: Když okno *Call Stack* obsahuje nesmyslné informace, může to být způsobeno chybou přetečení zásobníku (*stack overflow*) nebo přepsáním paměti. K chybě přetečení zásobníku nedojde u 32bitových programů tak snadno jako u 16bitových, ale stále se to může stát.

Okno *CPU*

Okno *CPU* se vyvolá z hlavního menu *View/Debug Windows/CPU* (*Ctrl+Alt+C*). Toto okno umožní vidět váš program na úrovni instrukcí assembleru. Program lze krokovat po jednotlivých instrukcích. Efektivní použití okna *Call Stack* (jedná se o pokročilý ladící prostředek) vyžaduje znalost jazyka assembleru.

Příkaz *Go to Address*

Příkaz *Go to Address* je také pokročilý ladící prostředek. Když váš program havaruje, Windows zobrazí chybové hlášení s adresou, na které došlo k chybě. Můžete použít příkaz *Go to Address* pro nalezení místa v programu, kde došlo k chybě.

Když uvidíte zmíněné chybové hlášení, zapište si adresu, na které došlo k chybě a potom zvolte *Debug/Go to Address* v kontextovém menu editoru. Do dialogového okna zadejte danou adresu a ladící prostředí se pokusí nalézt zdrojový řádek s chybou.

LITERATURA

[1] Cantú M.: Mistrovství v Delphi 2. Praha, Computer Press 1996.

[2] Cantú M.: Myslíme v jazyku Delphi 7 - knihovna zkušeného programátora. Praha, Grada Publishing 2003.

[3] Kadlec V.: Učíme se programovat v Delphi a jazyce Object Pascal. Praha, Computer Press 2001.

[4] Krček B. – Kolomazník I.: Algoritmizace a programování v Delphi. Ostrava, VŠB 2006.

[5] Krček B. - Kreml P.: Algoritmizace a programování v jazyku Pascal. Ostrava, VŠB 1993.

[6] Písek S.: Začínáme programovat v Delphi - podrobný průvodce začínajícího uživatele. Praha, Grada Publishing 2000.

[7] Pošta J.: Delphi - začínáme programovat. Praha, BEN 2001.

[8] Svoboda L. a dal.: 1001 tipů a triků pro Delphi. Praha, Computer Press 2001.

[9] Wirth N.: Algoritmy a štruktúry údajov. Bratislava, ALFA 1989.