

Obchodní akademie Orlová

Základy programování

Jan Grünwald

Září 2006

Abstract

Tato studijní opora se zabývá základy programování prezentovaných v prostředí jazyka PASCAL v implementacích FreePascal nebo Borland Pascal.

Obsah

1. PŘEDMLUVA	1
2. POUŽITÉ GRAFICKÉ SYMBOLY	3
3. ÚVOD	5
4. NA CO STUDIJNÍ OPORA NAVAZUJE	9
4.1. Prvky programovacího jazyka	9
4.2. Struktura programu	11
5. PODPROGRAMY	15
5.1. Struktura podprogramu	15
5.2. Podprogramy a proměnné	17
5.3. Parametry podprogramů	20
5.4. Členění podprogramů	23
5.5. Standardní podprogramy	25
5.5.1. Procedury pro vstup	26
5.5.2. Procedury pro výstup	27
5.5.3. Ostatní standardní podprogramy	36
6. MODULÁRNÍ PROGRAMOVÁNÍ	39
7. PROGRAMOVÉ JEDNOTKY - UNITY	43
7.1. Standardní programové jednotky	44
7.1.1. Podprogramy jednotky CRT	45
7.1.2. Ostatní standardní jednotky	46
7.2. Uživatelské programové jednotky	47
8. SOUBORY	53
8.1. Deklarace souborů	55
8.2. Práce se soubory	56
8.2.1. Čtení a zápis textových souborů	59
8.2.2. Čtení a zápis netextových souborů	59
9. DYNAMICKÉ DATOVÉ TYPY	61
10. DIREKTIVY PRO KOMUNIKACI	65
10.1. Přepínačové direktivy	65
10.2. Parametrické direktivy	70
A. ASCII tabulka	73
LITERATURA	75

1. PŘEDMLUVA

Vážená studentko,

Vážený studente,

ve studijní opoře, kterou právě berete do rukou se budeme věnovat **Základům programování**. Tato studijní opora volně navazuje na studijní oporu **Základy algoritmizace**. Přestože je tato vazba označena jako *volná* jsou znalosti z ní nutné pro úspěšné zvládnutí základů programování. Základy programování Vám umožní prakticky si vyzkoušet a ověřit to, co jste se teoreticky naučili. K vytváření programů můžeme používat různých nástrojů např. editorů, překladačů, linkerů atd. Spojíme-li však tyto izolované nástroje do jednoho celku, hovoříme pak o **vývojovém prostředí** (IDE, Integrated Development Environment) a usnadníme tím programátorovi situaci v tom, že nebude muset spouštět každý nástroj samostatně, ale najde vše pěkně pohromadě a snadno ovladatelné. Je to tedy soutava propojených nástrojů pro programování. Pro Vaši výuku byl zvolen programovací jazyk **PASCAL** a uvedené příklady můžete použít ve vývojovém prostředí nazývaném **Free Pascal** (dále bude používána zkratka FPascal), který se zdá být v současné době nejkvalitnějším volně šiřitelným vývojovým prostředím. Bez úprav nebo jen s drobnými úpravami lze uvedené příklady použít rovněž ve vývojovém prostředí **Turbo Pascal** nebo **Borland Pascal** (dále bude používána zkratka T/BPascal). Na případné úpravy budete vždy ve výkladu upozorněni. FPascal je modernější, proto v něm neplatí některá omezení T/BPascalu a může také pracovat v režimu, ve kterém je s jeho verzí 7.0 téměř plně kompatibilní. Pokud tento režim nebude nastaven standardně (default) pak dané nastavení můžete zkontrolovat nebo změnit po spuštění FPascalu v nabídce menu "Options" položka "Compiler" kde musí být "X" v řádku TP/BP 7.0 compatibility. Vývojové prostředí FPascal je volně šiřitelné a můžete si je stáhnout jednak ze školního serveru nebo přímo z následujících internetových stránek. S vývojovým prostředím T/BPascal je to složitější, volně šiřitelný je pouze Turbo Pascal verze 5.5, Turbo Pascal verze 7.0 doposud nebyl ani pro potřeby výuky uvolněn.

Free Pascal:

- Úvodní internetová stránka [<http://www.freepascal.org>]
- Stažení vývojového prostředí [<http://www.freepascal.org/download.html>]
- stačí si vybrat operační systém
- Stažení vývojového prostředí pro Windows [<http://www.freepascal.org/down/i386/win32-ftp.freepascal.org.html>]

Borland Pascal:

- Stažení vývojového prostředí Turbo Pascal verze 5.5
[<http://community.borland.com/article/20803>]
- Stažení vývojového prostředí Turbo Pascal verze 7.0
[http://www.borland.cz/products/classic_products/index.html] - placená verze



Doporučení:

- Studujte systematicky a postupně.
- Sledujte pravidelně výuku na serveru školy.
- Dodržujte stanovené termíny.
- Nezapomínejte, že korespondenční úkoly je nutno zasílat e-mailem.
- Ke komunikaci s tutorem i spolužáky využívejte výhradně formu **diskuse**.

Jan Grünwald, autor studijní opory

2. POUŽITÉ GRAFICKÉ SYMBOLY



Doba studia:



Cvičení

ÚLOHA č. x



Další zdroje



Klíčová slova



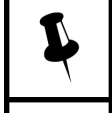
Korespondenční úkol



K zamyšlení



Potřebné znalosti



Poznámka



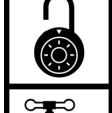
Příklad



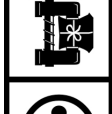
Pro zájemce



Průvodce studiem



Řešení



Shrnutí



Tip

CÍL



ÚKOL K ZAMYŠLENÍ



DOPORUČENÍ

3. ÚVOD



Tip

Cíle předmětu

Po prostudování opory budete znát:

- Další důležité příkazy programovacího jazyka Pascal.
- Základní principy programování.
- Programování jednoduchých úloh.
- Jak lépe formulovat požadavky na zadání vytvářeného programu.

Programování je specifickou činností a ještě do nedávna bylo na něj i některými odborníky nahlíženo jako na uměleckou činnost. Dnes na ně zase někteří odborníci nahlíží jako na zvládání technologie, ale většina z nich ho i nadále považuje za tvůrčí činnost. V oblasti autorských práv toto zařazení neustále platí, a protože jedno zadání lze naprogramovat několika způsoby, je každý program chápán jako autorské dílo. Ale pryč jsou ty doby, kdy program byl dílem jednotlivce nebo malé skupiny. Dnes musí být dobrý program napsán tak, aby byl stabilní aby dokázal řešit s velkým komfortem obsluhy širokou škálu úkolů a navíc aby byl snadno udržovatelný. To znamená, že se v něm musí vyznat nejen jeho autor nebo skupina autorů, ale i programátoři, kteří ve vývoji programu pokračují i když původní autorský tým již například projekt opustil. Proto dnes víc než kdy jindy musíme dbát na to, aby zápis zdrojového kódu programu byl přehledný, řádně komentovaný a zdokumentovaný. Ten, kdo chce programovat, musí dobře znát nejen vlastní programovací jazyk ve kterém píše, ale i vývojové prostředí používaného programovacího jazyka, systémové prostředí pro které je program určen a v neposlední řadě musí umět převést řešený (programovaný) problém na dílčí problémy a tyto pak na elementární příkazy programovacího jazyka. Z uvedeného vyplývá, že jako výchozí oblast musí zvládnout algoritmizaci.

Jak tedy takový počítačový program vzniká

V naprostých začátcích období pronikání počítačů do jednotlivých odvětví lidské činnosti bylo nutné psát program v tzv. strojovém kódu a později assembleru. Neexistovaly žádné programovací jazyky, které by vytváření programů usnadňovaly. Proto vývoj programu, který se z dnešního pohledu jeví jako velmi jednoduchý, byl možný pouze za účasti skutečně bystrých a znalých programátorů, tehdy většinou vědeckých pracovníků. Dnes je tomu jinak. Snaha zjednodušit a zpřístupnit vytváření programů širšímu okruhu zájemců vedla ke vzniku tzv. vyšších programovacích jazyků jejichž užití pro vytváření programů je pro člověka pochopitelnější a jednodušší.

Čím to tedy začíná? V první řadě rozbořem (analýzou) toho co má program umět, pro koho bude určen případně prostudováním zadání, ve kterém je již

rozbor proveden. Následuje zápis tzv. zdrojového kódu, který je základem každého programovacího jazyka. Zdrojový kód lze vytvářet v libovolném editoru (tento editor nesmí zapisovat do souboru formátovací znaky) nebo přímo v tzv. vývojovém prostředí programovacího jazyka. Pomocí něj programátor zapisuje v odpovídající syntaxi (struktuře) výrazy, jimiž určuje, jak a co má program provádět. Vytvářený zdrojový kód je zapisován do souboru textového typu a při použití jazyka Pascal má soubor příponu "pas". Napsaný kód je pak pomocí překladače převeden do binární, spustitelné podoby. Tuto operaci zjednodušeně nazýváme překlad a ten má několik fází. Nejprve jsou pomocí preprocesoru zpracovány zdrojové kódy a výsledek zpracování je prostřednictvím nově vzniklého souboru předán dále ke zpracování. V této chvíli zahajuje činnost **překladač**, často nazývaný také jako **kompilátor**. Překlad, neboli kompilace programu, je jednou z nejsložitějších fází při cestě za spustitelným souborem. Celkový počet fází, jejich složitost a návaznost záleží samozřejmě na konkrétním překladači jazyka a také na platformě, pro kterou je určen.

Obecně ale překladač provádí následující kroky:

Nejdříve provede tzv. *lexikální* analýzu kódu. Zdrojový kód je rozdělen na jednotky představující proměnné, jednotlivé příkazy, klíčová slova atd. **Lexikální** analýza má tedy za úkol rozpoznat v zadaném textu jednotlivé lexikální jednotky. Tyto lexikální jednotky dále jsou již zpracovány syntaktickou analýzou. Při lexikální analýze se tedy kontroluje správnost zápisu zdrojového kódu. Pokud zde překladač narazí na problém, nemůže pokračovat dál. V pokročilých vývojových prostředích je programátor o chybě podrobně informován a většinou ji také snadno opraví.

Dále nastupuje **syntaktická** analýza. Zde se zjišťuje správná posloupnost zápisu kódu v závislosti na použitém programovacím jazyku.

Nejobtížnější fází překladu je analýza **sémantická**. Zde musí překladač zkontrolovat význam jednotlivých výrazů a příkazů a zjistit, zda jsou logicky na správných místech. Výsledkem této třetí fáze je již téměř finální verze spustitelného programu.

Moderní překladače mají také různé optimalizační fáze, při kterých upravují rychlost provádění programu. Překladač tedy vytvoří jakýsi binární kód, který předává dál. Na řadě je pak tzv. linker, který vezme kód vytvořený překladačem, dále si vezme soubory, které mají být včleněny do výsledného souboru (ikony, obrázky a další zdroje), a vytvoří konečný spustitelný soubor v závislosti na použité platformě. Z obecného pohledu tento výsledný spustitelný soubor může být například běžný EXE pro Windows nebo DOS, DLL knihovna atd. Můžete však narazit také na programovací jazyky, které pracují na principu kódu a interpretu. *Jazyk Pascal ale do této skupiny nepatří.* Ke spuštění programu je totiž zapotřebí tzv. interpret jazyka, který čte kód a postupně provádí instrukce v něm obsažené. Pokud chcete takto vytvořený program předat dalšímu uživateli, je nutné, aby měl na svém počítači také nainstalovaný interpret tohoto jazyka. Rád bych ale upozornil, že to neznamená, že dotyčný uživatel si přečte vaše zdrojové kódy. Většina interpretů pracuje s binární podobou zdrojových kódů.

Jak je vidět, cesta od napsání zdrojového kódu ke spuštění programu na počítači není zrovna jednoduchá. Naštěstí pro programátory většinu práce odvede opět počítač. Na programátorech zůstává pouze povinnost dodržovat správnost zápisu, vymyslet správně činnost programu, jeho smysluplné ovládání a vzhled. Ale i tak to obnáší mnohdy (většinou) hodně práce.

4. NA CO STUDIJNÍ OPORA NAVAŽUJE



Tip

Cíl

Stručně připomenout základní znalosti, které jsou nutné pro další studium.

Tato studijní opora navazuje na studijní oporu *Základy algoritmizace*, ve které jste se dozvěděli co je to *algoritmus*, *algoritmická struktura*, seznámili se se *základními příkazy a datovými typy* programovacího jazyka Pascal i některými základními *programovými konstrukcemi*.

4.1. Prvky programovacího jazyka

Množina symbolů

jsou všechny znaky, které můžeme pro zápis programu použít. Jsou jimi písmena, číslice a zvláštní symboly.

Implementace FPascal i T/BPascal nerozlišují velká a malá písmena, při zápisu programu je používáme pro jeho zpřehlednění.

Identifikátory

jsou jednoznačná pojmenování prvků, které si v programovacím jazyce definujeme. Např. jména proměnných, programů a podprogramů a při jejich volbě musí být dodržována následující pravidla.

Identifikátor

1. se skládá z písmen, číslic a podtržítka, **nesmí** obsahovat **mezeru**,
2. **nesmí** se shodovat s žádným z klíčových slov,
3. **nesmí** začínat číslicí,
4. jeho maximální délka je neomezený počet znaků, ale implementace FPascal i T/BPascal **rozlišují pouze prvních 63 znaků**

Čísla

rozdělujeme na čísla celá a čísla s desetinnou částí. Čísla s desetinnou částí nazýváme čísla racionální a lze je zapisovat v desetinném nebo semilogatnickém tvaru. POZOR - při desetinném zápisu je nutno psát desetinnou **tečku**

nikoli čárku.

Poznámky

zapisujeme pomocí nich komentář k programu. Jednořádková poznámka začíná dvěma lomítky, více řádková začíná a končí závorkou a to složenou nebo kulatou doplněnou o hvězdičku.

Zvláštním případem je zápis tzv. **direktiv překladače** pomocí nichž ovlivňujeme chování překladače a následně tedy i chování programu. Direktivu zapisujeme následovně, za otevírací složenou závorkou bez mezery následuje znak \$ za ním pak jsou vedeny direktivy. Zápis tedy vypadá např takto: `{ $I- }`.



Příklad

Příklady zápisu poznámky:

`// jednořádková poznámka`

`{ více řádková poznámka - jeden řádek`

`další řádek`

`další řádek ...}` - tento způsob zápisu poznámky je vhodné používat výhradně pro zápis komentářů

`(*více řádková poznámka - jeden řádek`

`další řádek`

`další řádek ...*)` - tento způsob zápisu poznámky je vhodné používat pro dočasné "zapoznámkování" části programu při jeho ladění chceme-li zamezit jejímu provádění

Znaky a řetězce znaků

jsou z ASCII tabulky. Řetězcem znaků označujeme posloupnost znaků ohraničenou (uzavřenou mezi) znaky apostrof (odsuvník).



Příklad

Příklady zápisu řetězců:

`'Programování v jazyce Pascal.'`

`'!' - řetězec obsahující vykřičník`

`' ' - řetězec obsahující mezeru`

`" - prázdný řetězec - neobsahuje žádný znak`

`"" - řetězec obsahující jeden apostrof`

Oddělovače

vzájemně oddělují jednotlivé symboly a jsou jimi - **mezera, tabulátor, konec řádku a poznámka**.

Proměnné, datové typy a konstanty

Proměnnou nazýváme úsek operační paměti kam se ukládá hodnota, které proměnná nabývá.

Nejdříve je nutné připomenout, co to vlastně datový typ je a co jej definuje. **Datový typ** je souhrnné označení dvou vlastností proměnných a to:

- množiny přípustných hodnot a
- množiny operací.

Vždy, když se bude v následujícím textu hovořit o typu proměnné nebo výrazu, je tím myšlen **datový typ** proměnné nebo výrazu.

Operace nad datovými typy

Operace společné všem ordinálním datovým typům jsou tři:

- zjištění následníka
- zjištění předchůdce
- stanovení ordinální hodnoty

Další operace jsou stanoveny vždy pro daný datový typ.

4.2. Struktura programu

Dříve než si připomeneme vlastní strukturu programu, z našeho pohledu přesněji řečeno zdrojového kódu, zmiňme se také ještě o těch nejdůležitějších souborech, ve kterých bude vše uloženo. Při programování ve vývojovém prostředí FPascal nebo T/BPascal pro nás nejsou důležité pracovní soubory, které si vytvářejí jednotlivé překladače, ale ty se kterými budeme pracovat. Ty nejdůležitější mají příponu:

- **.PAS** - zde je uložen zdrojový text programu.
- **.EXE** - zde je uložen výsledek překladu zdrojového textu - výsledný přeložený program ve spustitelném tvaru.
- **.TPU** - programová jednotka T/BPascalu přeložená do tvaru pro připojení k programu v reálném režimu.

.PPU - programová jednotka FPascalu přeložená do tvaru pro připojení k programu v reálném režimu.

Struktura zdrojového kódu programu.

Nejdříve ukázka obecné struktury programu.

```
program <jméno programu>; {direktivy překladače}
uses <seznam jednotek>;
label <deklarace návěští>;
const <deklarace konstant>;
type <definice datových typů>;
var <deklarace proměnných>;
<deklarace uživatelských procedur a funkcí>
begin
<tělo hlavního programu>
end.
```

První řádek ve struktuře programu je označován jako **hlavička programu**. Je uveden klíčovým slovem **program** a za ním následuje *jméno programu*. Hlavička a další řádky tvoří tzv. **blok** a tento se dále dělí na *část deklarací* a *příkazovou*. Příkazová část začíná klíčovým slovem **begin** a končí klíčovým slovem **end**. Uvedená klíčová slova též označujeme jako **příkazové závorky**. Jednotlivé implementace jazyka Pascal mohou umožňovat opakování deklarací a příkazových částí a pořadí jednotlivých deklarací v deklarací části nemusí být dodrženo. Program **musí vždy končit tečkou**. Program napsaný v jazyce Pascal se skládá z bloků a pro jejich rozlišení používá mimo klíčového slova **program** ještě dvou dalších klíčových slov, která si řekneme a vysvětlíme později.

Hlavička:

jak již bylo řečeno zahajuje blok, určuje jeho jméno a definuje tak jeho vazbu na okolí.

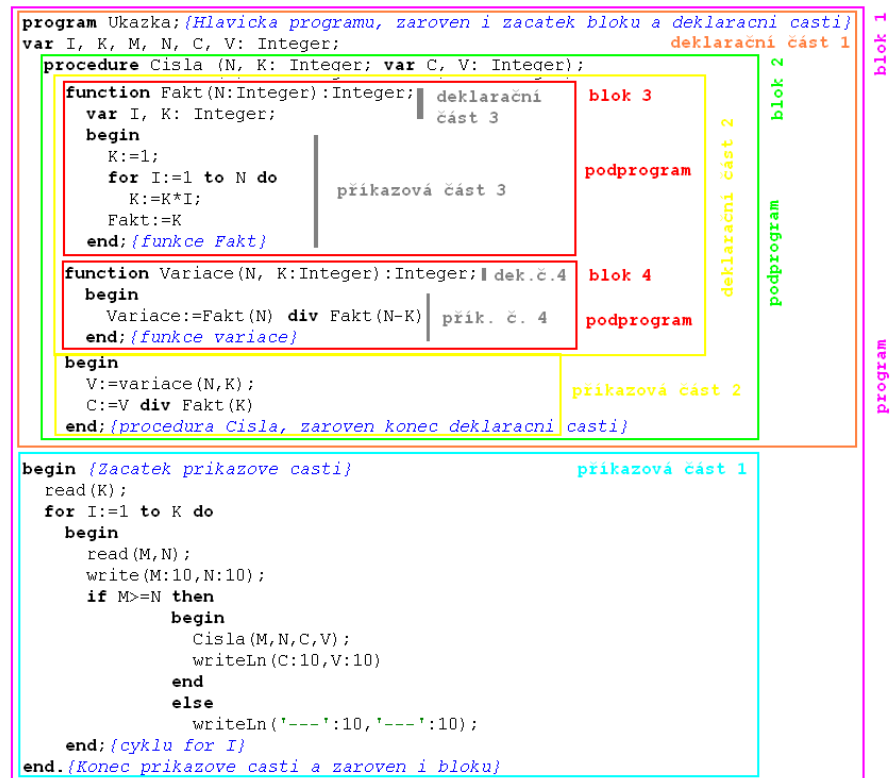
Deklarací část též Deklarace:

obsahují výčet všech prvků, které budeme později v příkazové části bloku používat. V Pascalu platí, že **veškeré prvky, které budou použity v příkazové části, musí být nejdříve zde v deklarací části definovány**. V ukázce výpisu programu jsou vyjmenovány všechny možnosti deklarací, ale je nutno zdůraznit, že ve skutečném programu **se všechny vyskytovat nemusí**. Deklarací část je vlastně takovou popisnou částí a říká (definuje) co je co.

Příkazová část:

obsahuje všechny příkazy, které realizují vlastní **činnost bloku**. Tato část je tedy tou **výkonnou** částí programu na rozdíl od části deklarací. Tato část bloku bývá též nazývána "**tělo bloku**" a je vymezena klíčovými slovy **begin** a **end**. **Jednotlivé příkazy** navzájem oddělujeme **středníkem**.

Ukázka struktury funkčního programu



Poznámka

Zapamatujte si:

Hlavička, blok, deklarací část, deklarace, příkazová část, příkazové závorky, tělo, datový typ.

5. PODPROGRAMY



Tip

Cíl

Objasnit pojem podprogram, procedura a funkce, vysvětlit užití standardních procedur pro vstup a výstup, naučit vytváření podprogramů a práci s jejich parametry.



Klíčová slova

Podprogram, podprogramy standardní a uživatelské, procedura, Read, ReadLn, Write, WriteLn, formátování, funkce, parametry, parametr formální, parametr skutečný, nahrazení parametru hodnotou, nahrazení parametru odkazem.



Doba studia: 6 hodin

Jak již sám název kapitoly napovídá budeme, hovořit o podprogramech. Je proto nutné si říct co to vlastně podprogram je. **Podprogram** můžeme definovat jako **relativně samostatnou část hlavního programu**. Nejdříve si řekneme proč vůbec podprogramy používáme. Odpověď je poměrně jednoduchá podprogram nám umožní **zpřehlednit a zkrátit hlavní program**. K výraznému zkrácení dojde zejména v případech kdy se část vyčleněná jako podprogram vícekrát v hlavním programu opakuje. V tomto případě podprogram **deklarujeme** (zapišeme) **jen jednou** a to v úseku deklarací a pak v hlavním programu už jen určíme, kdy se má podprogram provést. Odborně tento příkaz označujeme jako **volání podprogramu**. Dalším významným kladem podprogramů je omezení zavlékání chyb neustálým přepisováním částí zdrojového textu. Abychom vše mohli používat tak jak je uvedeno, je ještě nutné podprogram **pojmenovat**, přidělit mu tedy **identifikátor**. Zde je nutno zdůraznit, že každý podprogram musí být *nejdříve* deklarován, tj. zvolenému identifikátoru (jménu) podprogramu je přidělena definovaná posloupnost příkazů, která řeší nějakou dílčí úlohu. I zde je vidět, že vše co jsme si řekli o blokové struktuře programu, platí i pro podprogramy. Rozdíl je právě jen v klíčových slovech hlavičky. Protože se podprogramy se dělí na dvě základní skupiny a to na **procedury** a **funkce**, budou i ona klíčová slova dvě a to **procedure** a **function**. Nesmíme ještě zapomenout zdůraznit, že pro pojmenovávání podprogramů platí stejná pravidla jako je tomu u programu. Identifikátor podprogramu musí být souvislý text, nesmí začínat číslicí ani se shodovat s některým klíčovým slovem jazyka Pascal. Tolik obecně o podprogramech. Nyní se s nimi seznámíme podrobněji.

5.1. Struktura podprogramu

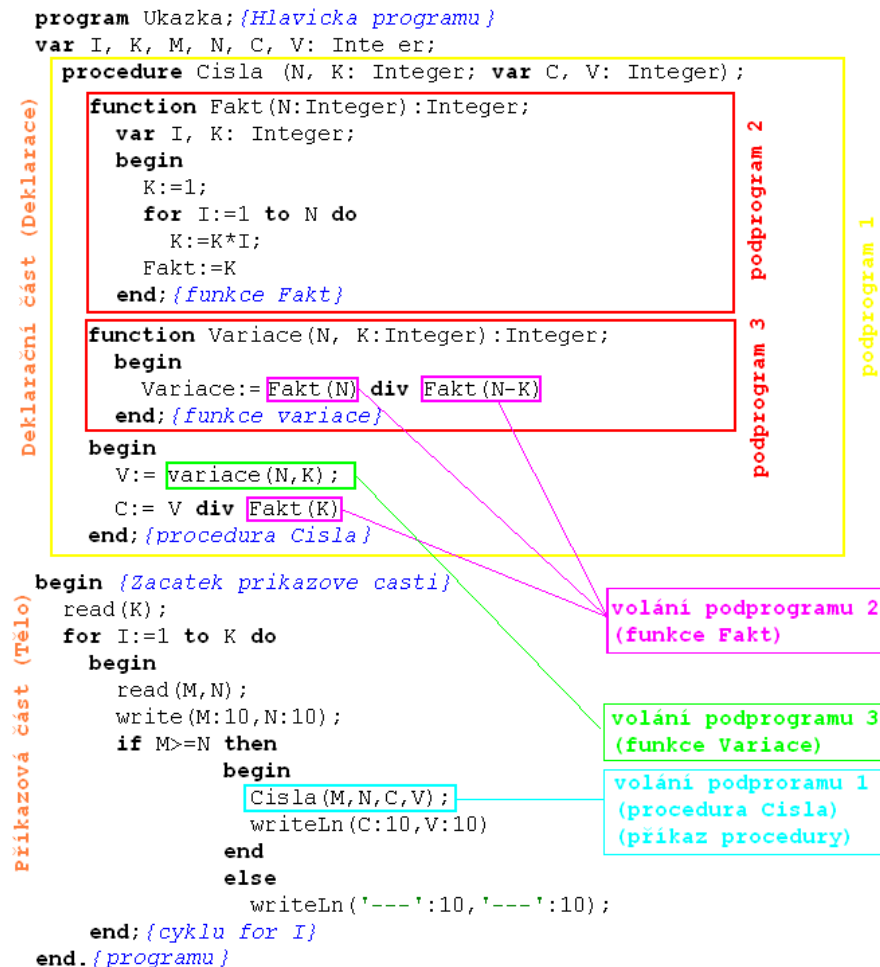
Deklarace podprogramu vypadá obdobně jako je tomu u programu. Pro

snadnější porozumnění uvádím obecný příklad.

```
procedure <jméno_procedury><definice parametrů>;
uses <seznam jednotek>;
label <deklarace návěští>;
const <deklarace konstant>;
type <definice datových typů>;
var <deklarace proměnných>;
<deklarace uživatelských procedur a funkcí>
begin
  <tělo procedury>
end;
```

V uvedeném příkladu vidíme, že v deklarační části podprogramu mohou být i další podprogramy. Musíme si ale uvědomit, že **všechny** prvky deklarované uvnitř podprogramu existují pouze uvnitř tohoto podprogramu (nejsou tedy vně podprogramu přístupné), a proto je označujeme, na rozdíl od prvků deklarovaných v hlavním programu, jako prvky **lokální**. Prvky deklarované v hlavním programu jsou prvky **globální** a ty jsou přístupné i v podprogramech. Protože potřebujeme spouštět podprogramy za různých podmínek, tj. bez vstupních údajů nebo s různými vstupními údaji, je nutné tyto ke zpracování nějak předat. To se děje prostřednictvím tzv. **parametrů**. Toto je hlavní rozdíl mezi programem a podprogramem. Jak již bylo uvedeno, ne vždy jsou parametry u podprogramů žádoucí. Pak hovoříme o podprogramech **bez parametrů** tedy podprogramech uzavřených, které s okolím nekomunikují. Ty podprogramy, které parametrů využívají, budeme nazývat **podprogramy s parametry** a ještě navíc budeme rozlišovat, zda jsou parametry **povinné** či **nepovinné**.

Ukázka struktury funkčního programu se třemi programátorem definovanými podprogramy:



5.2. Podprogramy a proměnné

V podprogramu pracujeme s proměnnými obdobně jako v hlavním programu. Musíme však rozlišovat proměnné, které byly deklarovány v podprogramu, tyto nazýváme proměnné **lokální** a lze s nimi pracovat **pouze v podprogramu, ve kterém byly deklarovány**. A proměnné, které byly deklarovány v hlavním programu, ty nazýváme proměnné **globální** a s nimi lze pracovat jak v **hlavním programu tak i všech podprogramech**.

Nevyžadují-li to zvláštní okolnosti, globální proměnné by neměly být v podprogramech používány. To znamená, že veškerá komunikace podprogramu s ostatními částmi programu (tj. předání vstupních údajů do podprogramu a předání výstupních údajů z procedury) by měla být prováděna pomocí parametrů. Použití globálních proměnných v podprogramu zhoršuje jeho použitelnost v jiném programu.

```
program Ukazka; {Hlavicka programu, zaroven i zacatek bloku a deklaracni casti}
var I, K, M, N, C, V: Integer; // deklarace globalnich promennych
procedure Cisla (N, K: Integer; var C, V: Integer);
function Fakt(N: Integer): Integer;
var I, K: Integer; // deklarace lokalnich promennych
begin
  K:=1;
  for I:=1 to N do
    K:=K*I;
  Fakt:=K
end; {funkce Fakt}

function Variace(N, K: Integer): Integer;
begin
  Variace:=Fakt(N) div Fakt(N-K)
end; {funkce variace}
begin
  V:=variace(N,K);
  C:=V div Fakt(K)
end; {procedura Cisla, zaroven konec deklaracni casti}

begin {Zacatek prikazove casti}
  read(K);
  for I:=1 to K do
  begin
    read(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; {cyklu for I}
  end. {Konec prikazove casti a zaroven i bloku}
```

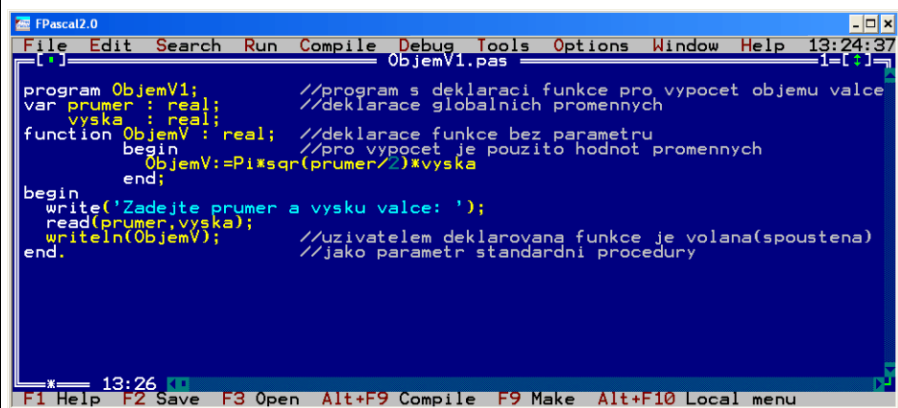
Následující příklady podprogramů jsou zde uvedeny jako ilustrativní. Při programování dodržujte předchozí doporučení, že globální proměnné se v podprogramech používají jen vyjimečně.



Příklad

V následujícím výpisu programu je ukázána deklarace podprogramu **bez parametrů**, ve dvou variantách.

U varianty první jsou použity globální proměnné.



```
FPascal2.0
File Edit Search Run Compile Debug Tools Options Window Help 13:24:37
ObjemV1.pas
program ObjemV1; //program s deklaraci funkce pro vypocet objemu valce
var prumer : real; //deklarace globalnich promennych
    vyska : real;
function ObjemV : real; //deklarace funkce bez parametru
begin //pro vypocet je pouzito hodnot promennych
  ObjemV:=Pi*sqr(prumer/2)*vyska
end;
begin
  write('Zadejte prumer a vysku valce: ');
  read(prumer,vyska);
  writeln(ObjemV); //uzivatelem deklarovana funkce je volana(spoustena)
end. //jako parametr standardni procedury
F1 Help F2 Save F3 Open Alt+F9 Compile F9 Make Alt+F10 Local menu
```

U varianty druhé jsou použity lokální proměnné.

Následující dva výpisy dokladují použitelnost lokálních proměnných pouze v podprogramech, ve kterých byly deklarovány.


```
program ObjemV1; //program s deklaraci funkce pro vypocet objemu valce
function ObjemV : real; //deklarace funkce bez parametru
var prumer : real; //deklarace lokalnich promennych
    vyska : real;
begin //pro vypocet je pouzito hodnot promennych
    ObjemV:=Pi*sqr(prumer/2)*vyska
end;
begin
    write('Zadejte prumer a vysku valce: ');
    read(prumer,vyska);
    writeln(ObjemV); //uzivatelem deklarovana funkce je volana (spoustena)
end. //jako parametr standardni procedury
(Jedna se o program shodny s predchozim, ale deklarace promennych jsou provede-
ny jako deklarace lokalni.
Pokuste-li se tento program prelozit bude Vam prekladac hlasit chybu.
Ve vypisu chyboveho hlaseeni bude uvedeno, ze jsou pouzity nezname promenne
a to na radku 11:8 - read(prumer...)
```

Následuje výpis včetně chybového hlášení překladače.

```
program ObjemV1; //program s deklaraci funkce pro vypocet objemu valce
function ObjemV : real; //deklarace funkce bez parametru
var prumer : real; //deklarace lokalnich promennych
    vyska : real;
begin //pro vypocet je pouzito hodnot promennych
    ObjemV:=Pi*sqr(prumer/2)*vyska
end;
begin
    write('Zadejte prumer a vysku valce: ');
    read(prumer,vyska);
    writeln(ObjemV); //uzivatelem deklarovana funkce je volana (spoustena)
end. //jako parametr standardni procedury
(Jedna se o program shodny s predchozim, ale deklarace promennych jsou provede-
ny jako deklarace lokalni.
Pokuste-li se tento program prelozit bude Vam prekladac hlasit chybu.
Ve vypisu chyboveho hlaseeni bude uvedeno, ze jsou pouzity nezname promenne
a to na radku 11:8 - read(prumer...)
```

Compiler Messages

objemv1.pas(11,8) Error: Identifier not found "prumer"
objemv1.pas(11,15) Error: Identifier not found "vyska"
objemv1.pas(11,21) Error: Illegal expression
objemv1.pas(18,37) Fatal: There were 3 errors compiling module, stopping
objemv1.pas(18,37) Fatal: Compilation aborted

Poslední výpis ukazuje funkční řešení s použitím lokálních proměnných.

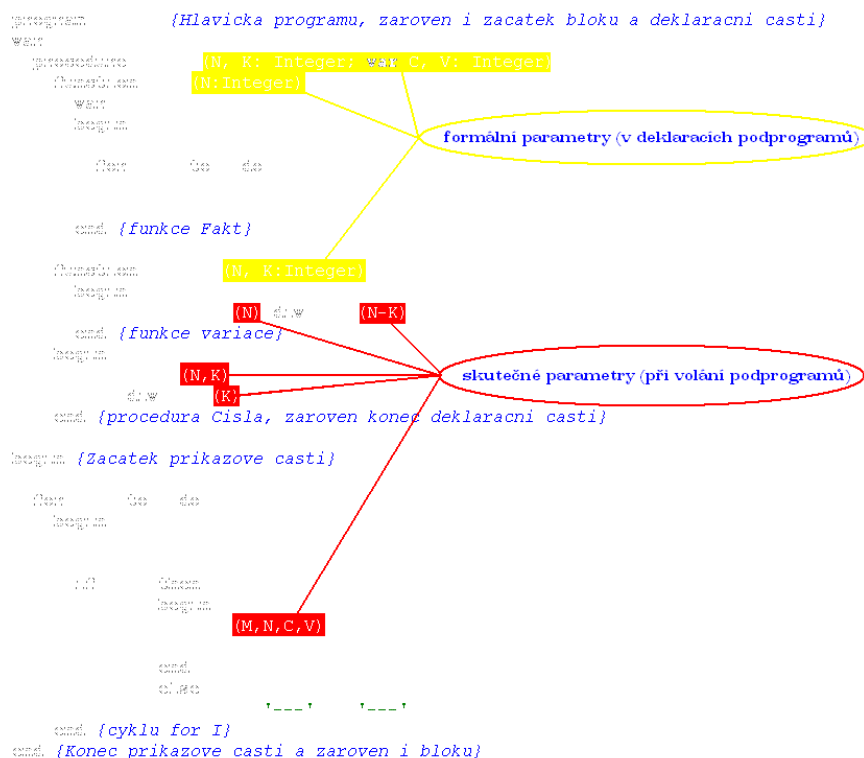
```
program ObjemV1; //program s deklaraci funkce pro vypocet objemu valce
function ObjemV : real; //deklarace funkce bez parametru
var prumer : real; //deklarace lokalnich promennych
    vyska : real;
begin //pro vypocet je pouzito hodnot promennych
    read(prumer,vyska);
    ObjemV:=Pi*sqr(prumer/2)*vyska
end;
begin
    write('Zadejte prumer a vysku valce: ');
    writeln(ObjemV); //uzivatelem deklarovana funkce je volana(spoustena)
end. //jako parametr standardni procedury
(Takto upraveny program bude opet funkcní.)
```

Hovoříme-li o lokálních a globálních proměnných a práci s nimi je nutné se také zmínit i o tzv. **zastínění identifikátoru**. Co tento pojem v programátorské praxi znamená a jaké má důsledky. V příkladě na začátku této podkapitoly je vidět, že podprogramy lze vzájemně zanořovat. Lze tedy v podprogramu deklarovat další podprogram tuto skutečnost označujeme jako **hierarchickou blokovou strukturu**. Použijeme-li v této struktuře identifikátor, který bude deklarován i s různými datovými typy, jak v hlavním programu, tak i několika do sebe zanořených podprogramech, dojde k následujícímu efektu. Použitím identifikátoru v podřízeném bloku (podprogram, podprogram v podprogramu) dojde vždy k vytvoření nové (lokální) proměnné (s vlastnostmi určenými použitým datovým typem) a znepřístupnění pro-

měnné se stejným identifikátorem v bloku vyšší úrovně a právě tento jev nazýváme **zastínění identifikátoru**.

5.3. Parametry podprogramů

Hlavním úkolem parametrů podprogramů je přenášet do podprogramů hodnoty, se kterými pak podprogramy pracují. Proto podprogramy navrhujeme vždy tak, aby parametry byly využity co nejvíce a abychom v maximální míře omezili použití proměnných deklarovaných v hlavním programu. Parametry v místě deklarace podprogramu nazýváme **formální (předepsané) parametry**, zatímco v místě volání (spuštění) podprogramu je nazýváme **skutečné parametry**.



Formální parametry podprogramů můžeme při jejich volání nahrazovat (předávat) dvojím způsobem:

- **hodnotou**

- při zavolání podprogramu vznikne nová proměnná, která je před zahájením vykonávání podprogramu inicializována na hodnotu skutečného parametru, při návratu z podprogramu (po ukončení jeho vykonávání) tato proměnná zaniká. Můžeme tedy konstatovat, že:

Formální parametr nahrazený hodnotou představuje v těle podprogramu jen lokální proměnnou, které je na počátku provádění podprogramu přiřazena hodnota skutečného parametru.

- přípustným skutečným parametrem může být **proměnná, konstanta nebo libovolný výraz**, hodnota však musí odpovídat datovému typu,

kteřý byl přiřazen formálnímu parametru.

- **odkazem**

- zde formální parametr představuje zástupný symbol, skutečný parametr udává, kterou proměnnou má podprogram aktuálně použít. Můžeme tedy konstatovat, že:

Formální parametr nahrazený odkazem představuje v těle podprogramu vždy tu konkrétní proměnnou, která je určena skutečným parametrem.

- při tomto způsobu předávání parametrů je vždy před jménem formálního parametru použito klíčového slova **var**
- přípustným skutečným parametrem může být **pouze proměnná**, jejíž datový typ je totožný s datovým typem, který byl přiřazen formálnímu parametru.



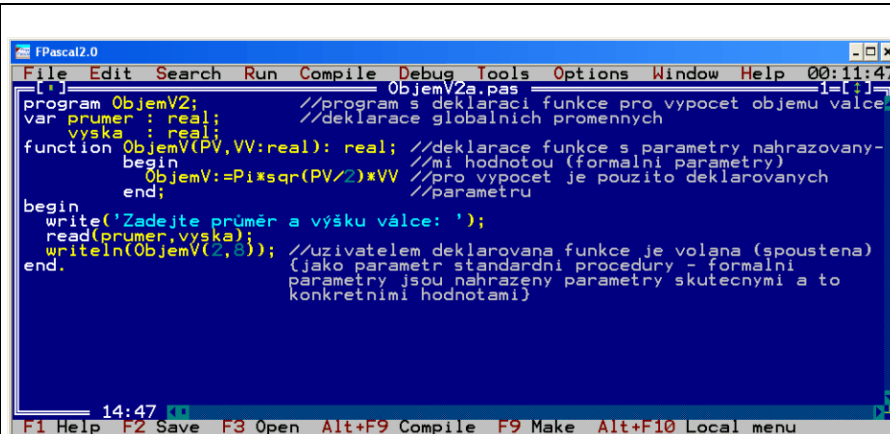
Příklad

V následujícím výpisu programu je ukázána deklarace podprogramu s parametry. V tomto příkladě budou formální parametry nahrazovány hodnotou, což je patrné již z deklarace funkce (**před** formálními parametry **není** uvedeno klíčové slovo **var**).

První varianta programu ukazuje použití proměnných jako skutečných parametrů.

```
FPascal2.0
File Edit Search Run Compile Debug Tools Options Window Help 00:02:41
ObjemV2.pas
program ObjemV2; //program s deklaraci funkce pro vypocet objemu valce
var prumer : real; //deklarace globalnich promennych
    vyska : real;
function ObjemV(PV,VV:real): real; //deklarace funkce s parametry nahrazovany-
begin //mi hodnotou (formalni parametry)
    ObjemV:=Pi*sqr(PV/2)*VV //pro vypocet je pouzito deklarovanych
end; //parametru
begin
    write('Zadejte průměr a výšku válce: ');
    read(prumer,vyska);
    writeln(ObjemV(prumer,vyska)); //uzivatelem deklarovana funkce je volana
end; //((spoustena) jako parametr standardni
//procedury - formální parametry jsou nahrazeny
//parametry skutečnými a to proměnnými)_
```

Druhá varianta programu ukazuje použití konstant (hodnot) jako skutečných parametrů. Z příkladu je vidět, že hodnoty proměnných, přestože byly načteny **nejsou** pro výpočet použity).



```
FPascal2.0
File Edit Search Run Compile Debug Tools Options Window Help 00:11:47
[.] ObjemV2.pas 1-[.]
program ObjemV2;           //program s deklaraci funkce pro vypocet objemu valce
var prumer : real;        //deklarace globalnich promennych
    vyska : real;
function ObjemV(PV,VV:real): real; //deklarace funkce s parametry nahrazovany-
begin                     //mi hodnotou (formalni parametry)
    ObjemV:=Pi*sqr(PV/2)*VV //pro vypocet je pouzito deklarovanych
end;                       //parametru

begin
    write('Zadejte průměr a výšku válce: ');
    read(prumer,vyska);
    writeln(ObjemV(2,8)); //uzivatelem deklarovana funkce je volana (spoustena)
                        //jako parametr standardni procedury - formalni
                        //parametry jsou nahrazeny parametry skutecnymi a to
                        //konkretnimi hodnotami
end.

14:47
F1 Help F2 Save F3 Open Alt+F9 Compile F9 Make Alt+F10 Local menu
```

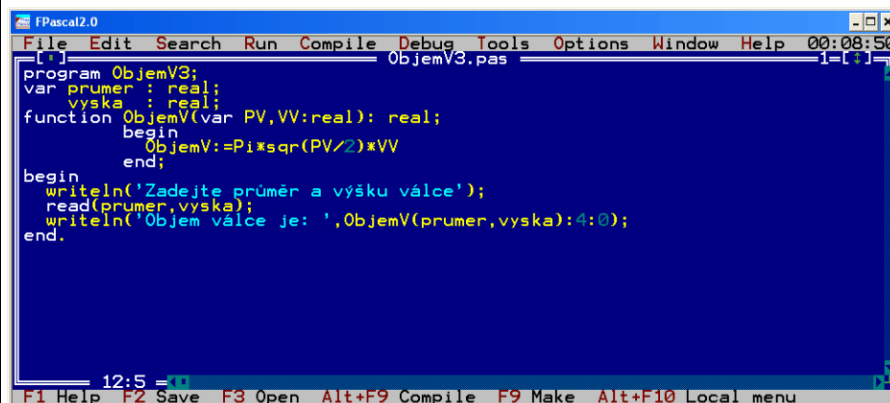
Třetí varianta programu ukazuje použití výrazů jako skutečných parametrů. Při provádění programu, je-li volána funkce "ObjemV", provede se nejdříve vyhodnocení prvního výrazu. Tím se získá hodnota prvního parametru. Následuje vyhodnocení druhého výrazu, tím se získá hodnota druhého parametru. A teprve pak je proveden vlastní výpočet. Je tedy vyhodnocen výraz definující vzorec pro výpočet objemu válce.



Příklad

V následujícím výpisu programu je ukázána deklarace podprogramu s parametry. V tomto příkladě budou formální parametry nahrazovány odkazem, což je patrné již z deklarace funkce (**před** formálními parametry **je** uvedeno klíčové slovo **var**).

A z toho co jsme si již řekli vyplývá, že formální parametry mohou v tomto případě být nahrazeny **pouze proměnnou**.

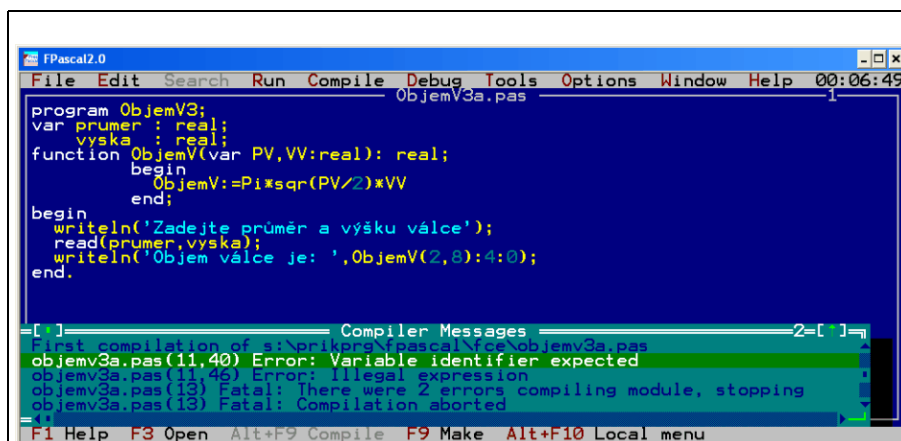


```
FPascal2.0
File Edit Search Run Compile Debug Tools Options Window Help 00:08:50
[.] ObjemV3.pas 1-[.]
program ObjemV3;
var prumer : real;
    vyska : real;
function ObjemV(var PV,VV:real): real;
begin
    ObjemV:=Pi*sqr(PV/2)*VV
end;

begin
    writeln('Zadejte průměr a výšku válce!');
    read(prumer,vyska);
    writeln('Objem válce je: ',ObjemV(prumer,vyska):4:0);
end.

12:5
F1 Help F2 Save F3 Open Alt+F9 Compile F9 Make Alt+F10 Local menu
```

Pokusíte-li se jako skutečný parametr použít určitou hodnotu, nebude možné program přeložit a překladač překlad zastaví s chybovým hlášením, že očekával ve volání funkce "ObjemV" jako skutečný parametr identifikátor proměnné. Vše je vidět z následujícího výpisu.

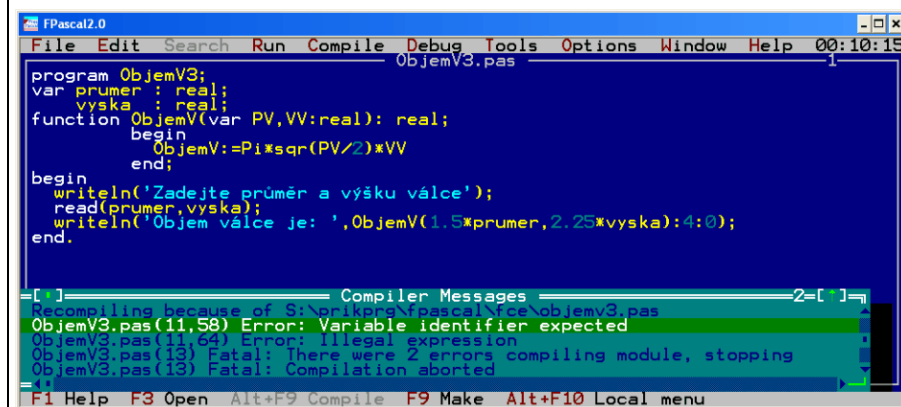


```
FPascal2.0
File Edit Search Run Compile Debug Tools Options Window Help 00:06:49
ObjemV3a.pas
program ObjemV3;
var prumer : real;
    vyska : real;
function ObjemV(var PV,VV:real): real;
begin
    ObjemV:=Pi*sqr(PV/2)*VV
end;
begin
    writeln('Zadejte průměr a výšku válce!');
    read(prumer,vyska);
    writeln('Objem válce je: ',ObjemV(2,8):4:0);
end.
```

```
Compiler Messages
First compilation of S:\prikprg\fpascal\ObjemV3a.pas
ObjemV3a.pas(11,40) Error: Variable identifier expected
ObjemV3a.pas(11,46) Error: Illegal expression
ObjemV3a.pas(13) Fatal: There were 2 errors compiling module, stopping
ObjemV3a.pas(13) Fatal: Compilation aborted
```

Pozice (11,40) v chybovém hlášení je uzavírací závorka volání funkce - ObjemV(2,8).

Obdobná bude situace i v případě, kdy se jako skutečný parametr pokusíte použít výraz.



```
FPascal2.0
File Edit Search Run Compile Debug Tools Options Window Help 00:10:15
ObjemV3.pas
program ObjemV3;
var prumer : real;
    vyska : real;
function ObjemV(var PV,VV:real): real;
begin
    ObjemV:=Pi*sqr(PV/2)*VV
end;
begin
    writeln('Zadejte průměr a výšku válce!');
    read(prumer,vyska);
    writeln('Objem válce je: ',ObjemV(1.5*prumer,2.25*vyska):4:0);
end.
```

```
Compiler Messages
Recompiling because of S:\prikprg\fpascal\ObjemV3.pas
ObjemV3.pas(11,58) Error: Variable identifier expected
ObjemV3.pas(11,64) Error: Illegal expression
ObjemV3.pas(13) Fatal: There were 2 errors compiling module, stopping
ObjemV3.pas(13) Fatal: Compilation aborted
```

Všimněte si, že při práci s podprogramy, vždy při nahrazování formálních parametrů parametry skutečnými je dodržováno pořadí nahrazování. Vždy musí první skutečný parametr odpovídat prvnímu formálnímu parametru, druhý - druhému, třetí - třetímu atd.

5.4. Členění podprogramů

Podprogramy můžeme členit podle dvou základních hledisek. Tím prvním je hledisko výsledku činnosti podprogramů, kterou podprogramy provádějí a tím druhým je hledisko "autorství". Proč vůbec tato členění vznikla? To první hledisko je nutné proto, že pro různé druhy podprogramů dostáváme odlišné výstupy. To druhé proto, že je nutné odlišit podprogramy vytvářené uživateli a podprogramy, jež jsou součástí implementace programovacího jazyka pro danou platformu, tedy daný operační systém.

Členění podprogramů:

1. Dle výsledku činnosti, kterou provádějí se dělí na:

- a. **Procedury.**
 - b. **Funkce.**
2. Dle autorství se dělí na:
- a. **Standardní** - to jsou ty, o které se už postarali programátoři při implementaci programovacího jazyka.
 - b. **Uživatelské** - to jsou ty, které vytváříme při programování zadání.

Nyní si vysvětlíme čím se vzájemně procedura a funkce od sebe liší. Zatím co procedura je "klasickým" podprogramem, který při svém spuštění provede činnost, která je uvnitř naprogramována a pracuje se zde s parametry a to jen pokud jsou deklarovány. Funkce se chová odlišně. Může také pracovat jak bez parametrů tak i s parametry ale navíc po ukončení její činnosti je jejímu identifikátoru přiřazena hodnota, která byla její činností vytvořena, říkáme že funkce **vrací hodnotu**. Při programování funkce musí programátor **identifikátoru funkce uvnitř jejího těla definovat hodnotu**. Definování hodnoty se provede **přiřazovacím příkazem, na jehož levé straně je uveden identifikátor funkce a na straně pravé hodnota**, která vznikla činností funkce. Z tohoto důvodu musíme navíc při deklaraci funkce také určit jakého **datového typu** bude její **výstupní hodnota**. To jsou zásadní rozdíly mezi procedurou a funkcí. Jak to prakticky programátor provede uvádějí následující příklady.



Příklad

deklarace procedury s identifikátorem (jménem) "Cisla":

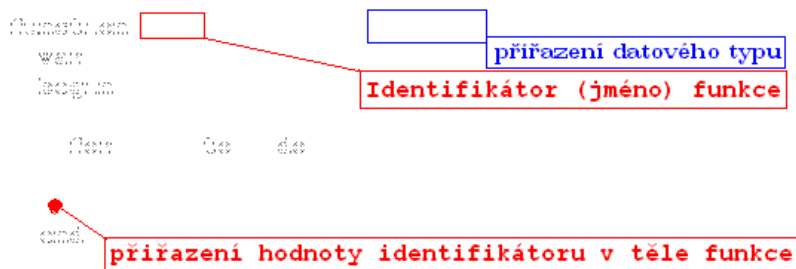
```
procedure Cisla (N, K: Integer; var C, V: Integer);  
  function Fakt (N: Integer): Integer;  
    var I, K: Integer;  
    begin  
      K:=1;  
      for I:=1 to N do  
        K:=K*I;  
      Fakt:=K  
    end; {funkce Fakt}  
  function Variace (N, K: Integer): Integer;  
    begin  
      Variace:=Fakt (N) div Fakt (N-K)  
    end; {funkce variace}  
  begin  
    V:=variace (N, K);  
    C:=V div Fakt (K)  
  end; {procedura Cisla}
```

Žádné přiřazení hodnoty identifikátoru procedury zde není



Příklad

deklarace funkce s identifikátorem (jménem) "Fakt":

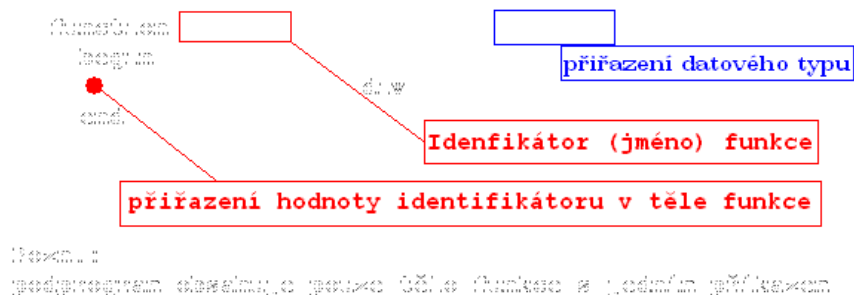


Upozornění: Na rozdíl od deklarace procedury je v deklaraci funkce jejímu identifikátoru určen **datový typ**.



Příklad

deklarace funkce s identifikátorem (jménem) "Variace":



Upozornění: Na rozdíl od deklarace procedury je v deklaraci funkce jejímu identifikátoru určen **datový typ**.

5.5. Standardní podprogramy

Dříve než bude popsáno vytváření vlastních podprogramů, řekněme něco více o nejpoužívanějších standardních podprogramech a to o procedurách pro **vstup** a **výstup**. Tyto procedury budeme zkoumat podrobněji, proto že pomocí nich může probíhat komunikace mezi uživatelem a programem. Vyjde-li ze základní charakteristiky programu, že transformuje vstupní data na data výstupní. Zdůrazňuje pak tato charakteristika skutečnost, že je nutné nějak vstupní data programem převzít a ta výstupní nějak zase předat zpět. Právě k tomuto účelu jsou procedury pro vstup i výstup a mezi nejpoužívanější standardní se řadí právě proto, že je každý program používá. Abychom nemusili neustále data vkládat jen prostřednictvím klávesnice, ukládáme je do **souborů**. Zejména pokud je těchto dat větší množství. Soubor je datový typ, který bude podrobněji popsán v kapitole "Soubory". Aby bylo možné popsat činnost vstupně-výstupních procedur, je nutné informovat o dvou zvláštních souborech, kterými jsou soubory **input** a **output**. Jak je již z názvu patrné

soubor **input** slouží pro **vstup** a soubor **output** pro **výstup**. Tyto dva soubory označujeme také jako **standardní vstupní soubor** a **standardní výstupní soubor**. Soubory jsou textové a to znamená, že řádky jsou v nich odděleny dvojicí znaků s ordinálními hodnotami #13 <CR> a #10 <LF>. V implementacích jazyka Pascal, o kterých hovoříme, jsou oba tyto soubory přístupné **automaticky**. Použití procedur pro vstup je použitím standardního vstupního souboru input a u procedur pro výstup je to použití standardního výstupního souboru output. Chování s ostatními druhy souborů bude popsáno v již zmíněné kapitole "Soubory". Seznam většiny standardních podprogramů, mimo ty které jsou zde popsány podrobněji, najdete v podkapitole "Ostatní standardní podprogramy".

5.5.1. Procedury pro vstup

Čtení procedurou Read:

Read (seznam proměnných);

Pro každou z uvedených proměnných přečte procedura ze vstupu všechny znaky tvořící vnější reprezentaci hodnoty, provede její konverzi na reprezentaci vnitřní (například z posloupnosti dekadických číslic vypočte odpovídající číslo) a výslednou hodnotu proměnné přiřadí. Proměnná musí být znakového, řetězcového, číselného nebo logického datového typu (pro hodnoty jiných typů nejsou definovány výstupní konverze) blíže viz. tabulka "Přehled možností čtení a zápisu datových typů procedurami pro vstup a výstup".

Při **čtení do libovolné číselné proměnné** může zápisu čísla předcházet libovolný počet oddělovačů (mezer, konců řádků, tabulátorů). Tuto posloupnost bude procedura ignorovat. Pokud nebude posloupnost znaků ve vstupním souboru odpovídat definici čísla, pak nelze provést přiřazení hodnoty, a proto program ohlásí chybu. Chyba bude též hlášena v případě, že správný zápis čísla bude po načtení přiřazován nesprávnému datovému typu. Zapisujeme-li tedy více čísel za sebe, vkládáme mezi ně nejméně jeden oddělovač.

Pokud má být současně čteno více hodnot, lze použít odpovídajícího počtu volání procedury Read zvlášť pro každou z nich nebo jediného volání s více parametry. Jednotlivé proměnné se pak v seznamu vzájemně oddělují čárkou. Syntakticky správné je i volání procedury *Read bez parametrů* (seznam proměnných je pak prázdný a neuvádí se ani závorka, která jej ohraničuje) *nemá však žádný efekt*.



Příklad

Trojici příkazů pro postupné načtení hodnot do proměnných A, B a C:

```
Read (A); Read (B); Read (C)
```

Lze nahradit jediným příkazem ve tvaru:

```
Read (A, B, C)
```

Čtení procedurou ReadLn:

ReadLn (seznam proměnných);

Jméno procedury se až na písmena "Ln", která jsou zkratkou anglického slova *line* tedy řádek, shoduje s procedurou předchozí. Z toho lze vyvodit, že uvedená procedura pracuje stejně jako Read jen s tím rozdílem, že **po načtení údaje přejde na nový řádek**. Z toho vyplývá, že ostatní údaje uvedené na stejném řádku **zůstanou nepřečteny a nelze se k nim vrátit**.

Při čtení čísel opět platí, že konce řádků jsou chápány jako oddělovače, které se ignorují až do nejbližšího znaku jenž oddělovačem není.

Při čtení do proměnné datového typu char se ze vstupního souboru přečte pouze jeden znak a přiřadí se do proměnné. Budeme-li se při čtení nacházet na konci řádku, bude přečten první znak z dvojice <CR><LF> (znak s ordinální hodnotou #13) a ten se přiřadí proměnné, ale druhý znak z dvojice <LF> (znak s ordinální hodnotou #10) zůstane nepřečten.

5.5.2. Procedury pro výstup

Výstup procedurou Write:

Write (seznam prvků tisku);

Procedura Write postupně vypíše od aktuální pozice kurzoru vnější reprezentace hodnot prvků tisku uvedených v seznamu. Jednotlivé prvky jsou v seznamu vzájemně odděleny čárkami. Syntakticky přípustné je i volání procedury *Write bez parametrů*, nemá však žádný efekt.

Základním tvarem každého prvku tisku je proměnná (vypisuje se hodnota proměnné ne její identifikátor, výraz (jehož hodnota je nejdříve vypočtena) nebo text uzavřený do apostrofů.

Za proměnnou, výrazem nebo textem smějí být v prvku tisku uvedeny požadavky na formát jeho výpisu, podrobnější popis k nim najdete v podkapitole "Formátování výstupních informací".

Výpis celého čísla - při výpisu celého čísla je počet vystupujících znaků vnější reprezentace dán počtem cifer jeho dekadického zápisu, případným záporným znaménkem (kladné se nevypisuje) a příslušným počtem formátovacích mezer.



Příklad

výpisu celého čísla s hodnotou -123 na 20 pozic | -123|

Výpis reálného čísla - výslednou hodnotou výrazu reálného typu je číslo typu Real resp. Extended, podle nastavení direktivy {\$N-} resp. {\$N+}. Pokud **není specifikován formát**, obsahuje vnější reprezentace reálného čísla následující znaky - takovýto formát výpisu označuje jako semilogaritmický tvar:

- znaménko mantisy (mezera nebo mínus)
- první platná číslice mantisy
- desetinná tečka
- 10 resp. 14 číslic desetinné části mantisy
- oddělovač exponentu (písmeno E)
- znaménko exponentu (plus nebo mínus)
- 2 resp. 4 číslice exponentu



Příklad

výpisu reálného čísla 5.6666666667 | 5.6666666667E+00|

Výpis znaku a řetězce - při výpisu hodnoty typu znak resp. řetězec vystupuje na displej příslušný znak resp. všechny platné znaky řetězce (v počtu daném jeho aktuální délkou) a příslušný počet formátovacích mezer. Při výpisu prázdného řetězce tedy vystupují pouze formátovací mezery. Některé znaky jsou při výpisu těchto hodnot používány jako řídicí (viz tabulka "Řídicí znaky výstupu na obrazovku"), ostatní jsou zobrazeny „normálně“.

Výpis logické hodnoty - při výpisu logické hodnoty vystupuje na obrazovku řetězec "FALSE" resp. "TRUE" doplněný zleva o příslušný počet formátovacích mezer.

Výstup procedurou WriteLn:

WriteLn (seznam prvků tisku);

Volání procedury WriteLn má stejnou syntaxi a funkci jako volání procedury Write. Procedura WriteLn však navíc po výpisu hodnot všech uvedených prvků odřádkuje - přesune kurzor na začátek další řádky (tj. doplní výpis znaky <CR><LF>). Ve tvaru bez parametrů WriteLn pouze *odřádkuje*. Toho je možné využít například k výpisu prázdných (vynechaných) řádek.

5.5.2.1. Formátování výstupních informací

Procedury Write a WriteLn provádějí před výpisem požadované hodnoty její konverzi z vnitřního na vnější (textový) tvar. Prvek tisku tedy proměnná i výraz musí být znakového, řetězcového, číselného nebo logického datového typu (pro hodnoty jiných typů nejsou definovány výstupní konverze). Blíže viz tabulka "Přehled možností čtení a zápisu datových typů procedurami pro vstup a výstup".

Pro proměnnou nebo výraz kteréhokoliv z povolených typů lze definovat celkový počet znaků vnější reprezentace jeho hodnoty takto:

proměnná: počet_znaků, výraz: počet_znaků

Specifikovaný počet znaků musí být celočíselný výraz. Pokud je pak počet znaků vnější reprezentace hodnoty vypisovaného výrazu nebo proměnné menší než počet takto předepsaný, je vnější reprezentace doplněna zleva potřebným počtem mezer. V opačném případě je předepsaný počet znaků **ignorován** a je použito tzv. **minimálního** formátu a informace **vždy vystoupí v nezkráceném tvaru**, žádný údaj tedy nebude zkreslen.



Příklad

Write ('*' : 5, 7 * 10 : 5, 'soudků' : 5)

Tímto příkazem bude vytištěno celkem 16 znaků: předepsané 4 formátovací mezery a vnější reprezentace řetězce '*' (dohromady požadovaných 5 znaků); další 3 formátovací mezery a 2 číslice hodnoty 50 výrazu $5 * 10$ (dohromady požadovaných 5 znaků) a konečně 6 písmen řetězce 'soudků' (předepsaných 5 znaků se neuplatní, protože řetězec je o jeden znak delší).

Pro proměnné a výrazy reálných typů může specifikace formátu výpisu obsahovat navíc i požadavek na počet desetinných míst vnější reprezentace hodnoty:

proměnná: počet_znaků : počet_desetinných_míst

výraz : počet_znaků : počet_desetinných_míst

Hodnota udávající počet_znaků je celkový počet znaků, na který bude hodnota vypsána, tzn. do tohoto počtu se započítávají všechny znaky výpisu tedy i znaménka, desetinná tečka u reálných typů a počet desetinných míst.

Hodnota, získaná výpočtem výrazu, je zaokrouhlena na předepsaný počet desetinných míst a konvertována na tvar s pevnou desetinnou tečkou (bez exponentu). Záporný počet desetinných míst je ignorován.



Příklad

Write (-5/6 : 7 : 2)

Tímto příkazem bude vytištěno požadovaných 7 znaků: 2 formátovací mezery a číslo -0.83 (5 znaků).

Pokud je pro výstup reálných typů specifikován jen počet znaků a není specifikován počet desetinných míst, může být počet číslic v desetinné části mantisy 1 až 10 resp. 1 až 17 (podle nastavení direktivy $\{ \$N- \}$ resp. $\{ \$N+ \}$). Pokud je specifikován i počet desetinných míst, odpovídá vnější reprezentace běžnému zápisu reálného čísla s pevnou desetinnou tečkou (bez exponentu). Maximální počet zobrazených desetinných míst je pak 11 resp. 18. Nezáporná čísla jsou v tomto případě zobrazena bez znaménka.

Přepíšete-li si následující program do vývojového prostředí, přeložíte jej a spustíte, provede formátovaný výpis číselných údajů.

Pozn.: Z důvodu exportu do pdf formátu musily být příliš dlouhé řádky s pří-

kazy WriteLn rozděleny do dvou řádků, při přepisu je **nutné zapsat je do jednoho řádku**. Pokračovací řádky jsou na první pozici označeny *, která však do zápisu programu nepatří.

```
program UkazFormat;
var
  I : integer;
  X : real;
begin
  WriteLn('Vypisovana cisla jsou uvadena v [ ] zavorkach');
  WriteLn('Celkovy pocet mist vypisu je uveden v prvnim
* sloupci');
  WriteLn('Vypis cisla "-123"');
  for I:=1 to 20 do
    WriteLn(I:3, ' [' , -123:I, ']' );
  Write ('Stisknete Enter');
  ReadLn;
  X:= 17/3;
  WriteLn('Vypis realneho cisla ',X,' bez uvedeni poctu
* desetinnych mist. ');
  for I:=1 to 20 do
    WriteLn(I:3, ' [' , X:I, ']' );
  WriteLn('Cislo je vypsano v semilogaritmicke tvaru se
* zaokrouhlenou posledni pozici. ');
  Write ('Stisknete Enter');
  ReadLn;
  WriteLn('Vypis tehoz realneho cisla ',X,' na 7 desetinnych
* mist. ');
  for I:=1 to 20 do
    WriteLn(I:3, ' [' , X:I:7, ']' );
  WriteLn('Cislo je vypsano v pevne formatu (s pevnu
* desetinnou teckou) se zaokrouhlenou ');
  WriteLn('posledni pozici a bez exponentu. ');
  Write ('Stisknete Enter');
  ReadLn;
  I:=20;
  WriteLn('Vypis vseh tri nejdelsich formatu. ');
  WriteLn(I:3, ' [' , -123:I, ']' );
  WriteLn(I:3, ' [' , X:I, ']' );
  WriteLn(I:3, ' [' , X:I:7, ']' );
end.
```



Příklad

výpisu čísla "-123" (pro přehlednost je výpis uveden v [] závorkách) na stanovený (celkový) počet míst, který je uveden v prvním sloupci.

Výpis proveden příkazy:

```
for I:=1 to 20 do
  WriteLn(I:3, ' [' , -123:I, ']' );
1 [-123]
2 [-123]
3 [-123]
```

```
4 [-123]
5 [ -123]
6 [  -123]
7 [   -123]
8 [    -123]
9 [     -123]
10 [      -123]
11 [       -123]
12 [        -123]
13 [         -123]
14 [          -123]
15 [           -123]
16 [            -123]
17 [             -123]
18 [              -123]
19 [               -123]
20 [                -123]
```

Na uvedeném výpise je patrné, že ještě na 4-tém řádku se formátovací informace neuplatňují, protože čtyři pozice jsou nezbytně nutné pro výpis - číslo se vypisuje v tzv. **minimálním formátu**.



Příklad

výpisu reálného čísla 5.6666666667E+00 (výpis je uveden v [] závorkách) na stanovený (celkový) počet míst, který je uveden v prvním sloupci. Ve formátu však **není** uveden počet desetinných míst.

Výpis proveden příkazy:

```
X:= 17/3;
```

```
for I:=1 to 20 do
```

```
  WriteLn (I:3, ' [', X:I, ']' );
```

```
1 [ 5.7E+00]
```

```
2 [ 5.7E+00]
```

```
3 [ 5.7E+00]
```

```
4 [ 5.7E+00]
```

```
5 [ 5.7E+00]
```

```
6 [ 5.7E+00]
```

```
7 [ 5.7E+00]
```

```
8 [ 5.7E+00]
```

```
9 [ 5.67E+00]
```

```
10 [ 5.667E+00]
```

```
11 [ 5.6667E+00]
```

```
12 [ 5.66667E+00]
```

```
13 [ 5.666667E+00]
```

```
14 [ 5.6666667E+00]
```

```
15 [ 5.66666667E+00]
```

```
16 [ 5.666666667E+00]
```

```
17 [ 5.6666666667E+00]
```

```
18 [ 5.6666666667E+00]
```

```
19 [ 5.6666666667E+00]
```

```
20 [ 5.6666666667E+00]
```

Z výpisu je patrné, že:

- a) Číslo je vypsáno v **semilogaritmickém** tvaru se zaokrouhlenou poslední pozicí.
- b) Ještě na 8-mém řádku se formátovací informace neuplatňují, protože osm pozic je nezbytně nutných pro výpis (u záporného čísla by na první pozici místo mezery bylo znaménko) - číslo se vypisuje v tzv. minimálním formátu.



Příklad

výpisu téhož reálného čísla 5.666666667E+00 (výpis je uveden v [] závorkách) na stanovený (celkový) počet míst, který je uveden v prvním sloupci. Ve formátu **je** uveden počet desetinných míst, a to 7.

Výpis proveden příkazy:

```
X:= 17/3;
for I:=1 to 20 do
  WriteLn (I:3, ' [', X:I:7, ']' );
1 [5.6666667]
2 [5.6666667]
3 [5.6666667]
4 [5.6666667]
5 [5.6666667]
6 [5.6666667]
7 [5.6666667]
8 [5.6666667]
9 [5.6666667]
10 [ 5.6666667]
11 [ 5.6666667]
12 [ 5.6666667]
13 [ 5.6666667]
14 [ 5.6666667]
15 [ 5.6666667]
16 [ 5.6666667]
17 [ 5.6666667]
18 [ 5.6666667]
19 [ 5.6666667]
20 [ 5.6666667]
```

Z výpisu je patrné, že:

- a) Číslo je vypsáno v **pevném formátu** (s pevnou desetinnou tečkou) se zaokrouhlenou poslední pozicí a **bez exponentu**.
- b) Ještě na 9-tém řádku se formátovací informace neuplatňují, protože devět pozic je nezbytně nutných pro výpis - číslo se vypisuje v tzv. minimálním formátu.

Při výpisu v pevném formátu se na první pozici u kladných čísel nezobrazuje mezera ani znaménko plus.



Příklad

výpisu všech tří nejdelších formátů z výše uvedených příkladů.

Výpis proveden příkazy:

```
I:=20;
WriteLn (I:3, ' [' , -123:I, ']' );
WriteLn (I:3, ' [' , X:I, ']' );
WriteLn (I:3, ' [' , X:I:7, ']' );
20 [          -123]
20 [    5.66666666667E+00]
20 [          5.6666667]
```



Příklad

výpis téhož reálného čísla 5.6666666667E+00 (výpis je uveden v [] závor-
kách) na stanovený (celkový) počet míst 19. Ve formátu je uveden proměn-
ný počet desetinných míst, a to od 0 po20. Počet desetinných míst platný
pro příslušný řádek je uveden v prvním sloupci.

Výpis proveden příkazy:

```
X:= 17/3;
for I:=0 to 20 do
  WriteLn (I:3, ' [' , X:19:I, ']' );
0 [          6]
1 [          5.7]
2 [          5.67]
3 [          5.667]
4 [          5.6667]
5 [          5.66667]
6 [          5.666667]
7 [          5.6666667]
8 [          5.66666667]
9 [          5.666666667]
10 [          5.6666666667]
11 [          5.66666666667]
12 [          5.666666666667]
13 [          5.6666666666667]
14 [          5.66666666666667]
15 [          5.666666666666667]
16 [ 5.6666666666666670]
17 [ 5.6666666666666670]
18 [ 5.6666666666666670]
19 [ 5.6666666666666670]
20 [ 5.6666666666666670]
```

Z výpisu je patrné, že:

- a) Číslo je vypsáno v pevném formátu se zaokrouhlenou poslední pozicí a bez exponentu.
- b) Při 16-ti desetinných místech je na poslední pozici doplněna 0, protože

číslo má jen 15 cifer za desetinnou tečkou a od této pozice se formát výpisu již nemění.



Příklad

výpis téhož reálného čísla 5.666666667E+00 (výpis je uveden v [] závorkách) na stanovený (celkový) počet míst 8. Ve formátu je uveden proměnný počet desetinných míst, a to od 0 po 20. Počet desetinných míst platný pro příslušný řádek je uveden v prvním sloupci.

Výpis proveden příkazy:

```
X:= 17/3;
```

```
for I:=0 to 20 do
```

```
  WriteLn (I:3, ' [', X:8:I, ']' );
```

```
0 [      6]
```

```
1 [     5.7]
```

```
2 [    5.67]
```

```
3 [   5.667]
```

```
4 [  5.6667]
```

```
5 [ 5.66667]
```

```
6 [5.666667]
```

```
7 [5.6666667]
```

```
8 [5.66666667]
```

```
9 [5.666666667]
```

```
10 [5.6666666667]
```

```
11 [5.66666666667]
```

```
12 [5.666666666667]
```

```
13 [5.6666666666667]
```

```
14 [5.66666666666667]
```

```
15 [5.666666666666667]
```

```
16 [5.6666666666666670]
```

```
17 [5.6666666666666670]
```

```
18 [5.6666666666666670]
```

```
19 [5.6666666666666670]
```

```
20 [5.6666666666666670]
```

Z výpisu je patrné, že:

a) Číslo je vypsáno v pevném formátu se zaokrouhlenou poslední pozicí a bez exponentu.

b) Jakmile počet desetinných míst přesáhne 6 neuplatňuje se ve formátu celkový počet pozic, který byl ve výpisu zvolen jen 8, celkový počet pozic pak narůstá až do okamžiku kdy přesáhne o jednu pozici skutečný počet cifer za desetinnou tečkou tj. při 16 des. místech.

c) Při 16-ti desetinných místech je na poslední pozici doplněna 0, protože číslo má jen 15 cifer za desetinnou tečkou a od této pozice se formát výpisu již nemění.

Uvedené příklady potvrzují skutečnost, že celkový počet pozic ve výpisu je skutečně dán součtem počtu cifer před des. tečkou, jednou pozicí na tečku a

počet cifer za des. tečkou.

Tabulka 5.1. Přehled možností čtení a zápisu datových typů procedurami pro vstup a výstup

Datový typ	Read	Write
	ReadLn	WriteLn
Jednoduché datové typy předdefinované (ordinální)		
byte	ano	ano
shortint	ano	ano
integer	ano	ano
word	ano	ano
longint	ano	ano
char	ano	ano
boolean	NE	ano
Jednoduché datové typy předdefinované (neordinální)		
real a ostatní datové typy pro racionální čísla	ano	ano
Jednoduché datové typy uživatelské		
výčet	NE	NE
interval (bázový datový typ předdefinovaný - ordinální)	ano	ano
interval (bázový datový typ - výčet)	NE	NE
Strukturované datové typy		
řetězec	ano	ano
pole znaků	NE	ano
ostatní datové typy	NE	NE

Tabulka 5.2. Řídící znaky výstupu na obrazovku

znak	řídící funkce
BELL (#7)	akustický signál (pípnutí)
BACKSPACE (#8)	posun kurzoru o pozici vlevo (není-li na začátku řádky)
LF (#10)	posun kurzoru o pozici dolů
CR (#13)	posun kurzoru na začátek řádky
EOF (#26)	ukončení výpisu právě aktivní procedury Write nebo WriteLn — žádné další znaky (ani znaky konce řádky) již nejsou vypsaný

5.5.3. Ostatní standardní podprogramy

Pro přehlednost jsou tyto podprogramy rozděleny do několika skupin, podle toho jakou činnosti provádějí.

1. Aritmetické podprogramy:

Abs - absolutní hodnota.

ArcTan - funkce arcus tangens úhlu.

Cos - funkce cosinus úhlu.

Sin - funkce sinus úhlu.

Exp - exponenciální funkce.

Ln - přirozený logaritmus.

Sqr - druhá mocnina.

Sqrt - druhá odmocnina.

Frac - desetinná část.

Trunc - celá část - vrací celočíselnou hodnotu získanou odříznutím desetinné části z reálného argumentu.

Int - celá část.

Round - zaokrouhlení racionálního čísla - vrací celočíselnou hodnotu, získanou zaokrouhlením reálného argumentu k nejbližšímu celému číslu.

Pi - Ludolfovo číslo.

Hi - vrací horní byte argumentu.

Lo - vrací dolní byte argumentu.

Random - vrací náhodné číslo z intervalu $<0..1>$.

2. Konverzní podprogramy:

Ord - převádí znak na ordinální číslo dle ASCII tabulky, je inverzní funkcí k funkci **Chr** s argumentem typu ordinální číslo.

Chr - převádí ordinální číslo na znak, dle ASCII tabulky, je inverzní funkcí k funkci **Ord** s argumentem typu znak.

Upcase - pokud je argumentem **malé písmeno anglické abecedy**, vrací funkce **odpovídající písmeno velké**, jinak je vrácená hodnota rovna ar-

gumentu.

3. **Ordinální podprogramy:**

Inc - zvýšení ordinální hodnoty.

Dec - snížení ordinální hodnoty.

Pred - zjištění předchůdce.

Succ - zjištění následníka.

Odd - určení lichosti.

4. **Podprogramy pro práci s řetězci:**

Concat - spojení několika řetězců v jeden celek - zřetězení.

Lenght - vrací hodnotu aktuální délky řetězce.

Copy - kopíruje určenou část řetězce.

Pos - vrací hodnotu pozice prvního znaku zadaného podřetězce.

Insert - vloží daný řetězec do jiného řetězce na určenou pozici.

Delete - od dané pozice odstraní v řetězci daný počet znaků.

Str - převede výraz číselného datového typu na znakovou reprezentaci tzn. např. číslo 123 převede na číslice 123.

Val - převede znakovou reprezentaci číslic do číselného datového typu, jedná se o proceduru opačnou k proceduře **Str**.

5. **Podprogramy pro práci se soubory:**

Assign - provede propojení skutečného souboru s identifikátorem - **tuto proceduru musíme provést vždy před použitím souboru v následujících podprogramech.**

Reset - otevře již připojený soubor pro čtení.

Rewrite - otevře již připojený soubor pro zápis.

Append - otevře již připojený soubor **pro přidání dat na konec souboru.**

Close - provede uzavření souboru.

Flush - provede vyprázdnění vyrovnávací paměti.

6. **Podprogramy pro práci s dynamickými proměnnými:**

New - založí novou dynamickou proměnnou o velikosti báze datového typu parametru.

Dispose - uvolní paměťový prostor obsazený dynamickou proměnnou.

GetMem - analogie **New** - založí novou dynamickou proměnnou.

FreeMem - opak **GetMem** - uvolní dynamickou proměnnou.

Mark - zaznamená stávající adresu hromady.

Release - zruší všechny dynamické proměnné vytvořené po **Mark**.



Cvičení

V ukázce struktury funkčního programu se třemi programátorem definovanými podprogramy zjistěte, kolik celkem standardních podprogramů tato ukázka obsahuje a kolik je jich různých druhů.



Cvičení

Určete kolik znaků se vypíše následující procedurou, je-li dáno:



Poznámka

Zapamatujte si:

Podprogram, globální a lokální proměnná, procedura, funkce, parametr formální a skutečný, nahrazení parametru hodnotou a odkazem.

6. MODULÁRNÍ PROGRAMOVÁNÍ



Tip

Po prostudování tohoto oddílu budete schopni vysvětlit a při programování použít metody i prostředky strukturovaného a modulárního způsobu programování.



Klíčová slova

Metody návrhu algoritmu shora dolů a dola nahoru, strukturované programování, modulární programování, modul.



Doba studia: 1 hodina

Než se pustíme do vysvětlování metody modulárního programování, nebude určitě na škodu připomenout metody návrhu algoritmů, o kterých jste již získali informace v algoritmizaci.

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 stupnovitému rozkladu (dekompozici) problému na podproblémy pak odpovídá i návrh algoritmu, který je nejprve hrubý a pak se postupně zjemňuje až do konečného tvaru. Tento konečný tvar se dá již plně vyjádřit příslušným programovacím jazykem. Návrh programu tedy vlastně začíná jen jakousi osnovou, pro jejíž zápis je možno využít programovacího jazyka jen v omezené míře.

Metoda návrhu algoritmu zdola nahoru je obvykle od počátku vázána na konkrétní programovací jazyk a je pro ni charakteristický postup od jednoduchých řídicích struktur, přes složitější řídicí struktury až po celkové řešení. I při tomto způsobu řešení je zřejmé, že celý postup se neobejde bez určitého minimálního počátečního 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,

u metody návrhu algoritmu zdola nahoru - jen přibližný, orientační návrh řešení.

Každá z uvedených metod má svoje přednosti, jejichž uplatnění závisí nejen na složitosti a povaze řešeného problému, ale rovněž i na schopnostech programátora. Přednosti metody návrhu algoritmu zdola nahoru se uplatní zejména při řešení více příbuzných problémů, kdy je možno vytvořené zá-

kladní složky použít ve větším počtu řešení. Dále přednosti této metody vyniknou při příliš volném nebo neúplném zadání úlohy.

Strukturované programování

Strukturované programování

je metoda systematického a metodického přístupu k vytváření takové řídicí struktury, 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 obrátů), 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 považována 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 algoritmů pak přispívá nejen k snadnějšímu pochopení algoritmu, ale může rovněž výrazným způsobem usnadnit úpravy i vlastní vytváření algoritmu.

Modulární programování

Modulární programování

je metoda vytváření programů, která je založena na myšlence *rozložit řešený problém na pokud možno izolované podproblémy*, jejichž jednotlivá řešení jsou 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

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, podnikových informačních systémů.

Modulární programování však nesouvisí jen s tvorbou programových systémů, ale jeho výhod lze rovněž 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 takových v nichž uživatel může vytvářet 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. Každá z programových jednotek se zpracovává (překládá) nezávisle na ostatních programových jednotkách. V každé verzi programovacího jazyka Pascal lze činnost vytváření a užívání modulů lehce napodobit, jestliže procedury deklarované přímo na základní úrovni bloku programu mají lokální proměnné. Vyšší verze jazyka T/BPascal a FPascal, na rozdíl od referenční verze jazyka Pascal, však již poskytují velmi silné prostředky pro vytváření programu z více programových jednotek a dovolují tedy metodu modulárního programování plně realizovat. K realizaci této metody slouží i programové jednotky, se kterými se seznámíte v následující kapitole.

7. PROGRAMOVÉ JEDNOTKY - UNITY



Tip

Cíl

Po prostudování tohoto oddílu budete schopni vysvětlit pojem jednotka (unita), popsat její strukturu a definovat možnosti použití.



Průvodce studiem

Práce s jednotkami (Unit) je velice efektivní. Při zpracování rozsáhlejších projektů je to jediná cesta, jak efektivně projekt zpracovat. Pomocí jednotek si můžete vytvářet své vlastní knihovny, které lze pak používat v různých programech a urychlit tak jejich naprogramování. Další výhodou programových jednotek je i skutečnost, že jejich prostřednictvím lze do vytváření rozsáhlého programu zapojit několik programátorů současně.



Klíčová slova

Unit, uses, CRT, DOS, GRAPH, PRINTER, OVERLAY, interface, implementation, segment.



Doba studia: 3 hodiny

Na úvod tohoto oddílu je nutno podotknout, že referenční verze jazyka Pascal programové jednotky nepoužívá a tato možnost je k dispozici až u implementací tohoto jazyka.

Programové jednotky, které v daném programu využíváme, uvádíme za klíčovým slovem **uses** v deklarační části hlavního programu a vzájemně je oddělujeme **čárkou**. Zápis v části uses musí končit **středníkem**. Tímto zápisem se provede tzv. připojení programové jednotky a prostředky v ní obsažené jsou dostupné k užití z vytvářeného programu. A stejně tak jako jsme měli u podprogramů podprogramy standardní, které jsou součástí implementace jazyka a podprogramy uživatelské, které si vytváří programátoři sami, obdobně je tomu i s programovými jednotkami. Zde ale jsou možnosti větší. Protože programová jednotka je samostatným souborem, lze k vytvářenému programu připojovat programové jednotky nejen **standardní**, ale i jednotky **uživatelské**, které jsme si vytvořili sami nebo nám je poskytli jiní programátoři ať již v komerčním nebo open source vztahu. Velmi často se místo pojmu **programová jednotka** nebo **unita** používá též pojmu **knihovna**.



Příklad

deklarační části programu s oblastí uses:

Uses CRT, GRAPH, Moje, Petrova, Tabulky;

V udedeném příkladu jsou jak unity standardní (jsou vypsány pouze velkými písmeny) tak i uživatelské.

7.1. Standardní programové jednotky

Programovací jazyk Pascal ať již v implementaci FreePascal nebo Turbo Pascal nabízí několik standardních programových jednotek. Tvůrcem těchto programových jednotek je autor implementace programovacího jazyka. Podprogramy v programových jednotkách doplňují a rozšiřují základní možnosti dané implementace jazyka.

- **SYSTEM** - obsahuje systémové podprogramy a připojí se k přeloženému programu automaticky.
- **CRT** - obsahuje nástroje pro lepší práci s uživatelskou textovou obrazovkou, zejména v barevném režimu.
- **GRAPH** - umožňuje práci v grafickém, obecně barevném režimu. Může spolupracovat s libovolným běžným grafickým ovladačem.
- **DOS** - umožňuje využívat služeb operačního systému, které by jinak nebyly z jazyka Pascal přístupné.
- **PRINTER** - umožňuje jednoduše používat tiskárny pro výstup výsledků.
- **OVERLAY** - umožňuje segmentaci vytvářeného programu.
- **WINDOS** - obsahuje obdobné prostředky jako jednotka DOS a je součástí verze 7.0 jazyka Turbo Pascal.

V dalším textu nebudeme možnosti a podprogramy jednotlivých unit podrobně rozebírat, protože nejsou nezbytně nutné pro zvládnutí probíraného učiva. Standardní unity distribuované společně s implementací jazyka jsou souhrnně označovány zkratkou RTL (Run-Time Library). Jen pro přiblížení práce s unitami bude podrobněji pojednáno o unitě CRT.



Pro zájemce

Podrobný popis standardních unit je vždy součástí instalace vývojového prostředí a lze jej nalézt ve složkách dokumentace. Pro FPascal najdete tuto podsložku ve složce s instalací FPascalu. Je to podložka *Drive:\hlavní_složka_instalace_FPascal\doc\rtl\index.html*. Tuto verzi doku-

mentace si můžete prohlížet internetovým prohlížečem. Verze vhodná i pro tisk je v souboru **RTL.pdf** (pozor má 1408 stran). Zde uvedený popis je platný i pro T/BPascal.

7.1.1. Podprogramy jednotky CRT

Unita **CRT** implementuje převážně prostředky pro řízení vstupu z klávesnice a výstupu na obrazovku. Standardní podpora těchto zařízení je implementována unitou **SYSTEM**, která je zpracovává jako textové soubory, spojené s proměnnými **Input** a **Output**. Unita **CRT** nepoužívá pro jejich ovládání univerzální přístup, nýbrž instaluje vlastní ovladače konkrétních fyzických zařízení klávesnice a obrazovky. Uvedený přístup umožňuje implementaci řady specializovaných mechanismů využívající zvláštních vlastností těchto zařízení, které jiná textová zařízení obecně nemají (například výstup textu na obrazovku není orientovaný pouze sekvenčně, pozici prvního znaku vypisovaného textu lze volit).

V následující části najdete vybrané podprogramy jednotky **CRT** včetně jejich parametrů a stručného popisu.

AssignCRT (var **F**:text) - připojení libovolného textového souboru k **CRT**, implicitně je k **CRT** připojen standardní vstup (klávesnice, soubor **input**) a výstup (obrazovka, soubor **output**).

ClrScr - smazání obrazovky a umístění kurzoru do levého horního rohu.

DelLine - smazání řádku na pozici kurzoru.

ClrEOL - smazání znaků od pozice kurzoru do konce řádku.

InsLine - vložení řádku před řádek s kurzorem.

NormVideo; HighVideo; LowVideo; - tyto tři příkazy slouží k nastavení jasu textu - normální, vysoký, nízký.

TextMode (modus: integr) - určení textového režimu.

TextBackGround (barva: byte) - nastavení barvy pozadí v rozsahu 0 .. 7.

TextColor (barva: byte) - nastavení barvy textu v rozsahu 1 .. 15.

Sound (Hz: word) - zapnutí vnitřního generátoru na kmitočet Hz.

NoSound - vypnutí vnitřního generátoru.

Delay (ms: word) - pozastaví výpočet na stanovený počet milisekund.

GoToXY (X,Y: byte) - přesune kurzor na pozici (X,Y).

WhereX - vrací X - souřadnici aktuální pozice kurzoru.

WhereY - vrací Y - souřadnici aktuální pozice kurzoru.

Window (x1,y1,x2,y2: byte) - nastaví aktuální okno a umístí kurzor do levé-

ho horního rohu, na absolutní polohu (x1,y1), od tohoto okamžiku se používá relativních souřadnic, které platí v aktuálním okně, tzn. levý horní roh je "bod" o souřadnicích (1,1).

KeyPressed - testuje stisk klávesy a pokud byla libovolná klávesa stisknuta, vrací hodnotu TRUE.

ReadKey - přečte jeden znak z klávesnice, ale **nezobrazuje** jej, datový typ načtené hodnoty je char.



Pro zájemce

Následující výpis programu demonstruje užití některých podprogramů jednotky CRT.

```
program DemoCRT;
uses CRT;
begin
{na pozici 10, 10 se vypise text}
  TextBackGround(4);           {barva pozadi cervena}
  TextColor(14);               {barva popredi zluta}
  ClrScr;                      {smazani obrazovky}
  GoToXY(10,10);               {nastaveni polohy kurzoru}
  WriteLn('Text pred vyvolanim procedury okno');
  Write('Stiskni Enter');
  ReadLn;                     {cekani na stisk klavesy Enter}
{definice okna - tisk do okna}
  Window(4,3,65,13);           {definuje textove okno}
  TextBackGround(blue);        {barva pozadi v okne}
  TextColor(white);            {barva popredi v okne}
  ClrScr;                      {smazani okna - vyplni barvou pozadi}
  GoToXY(5,5);                 {nastaveni polohy kurzoru v okne}
  WriteLn('Text v okne');
  Sound(440); Delay(500); NoSound; {pipnuti}
  Write('Stiskni Enter');
  ReadLn;                     {cekani na stisk klavesy Enter}
  Window(1,1,80,25);           {cela obrazovka}
  TextBackGround(0);           {barva pozadi cerna}
  ClrScr; {smazani okna - vyplni barvou pozadi celou obraz.}
  GoToXY(15,15); {nastaveni polohy kurzoru v novem okne}
  WriteLn('Text po vyvolani procedury okno');
  WriteLn('Stiskni cokoliv');
  Repeat
    Until KeyPressed; {ceka na stisk libovolne klavesy}
end.
```

7.1.2. Ostatní standardní jednotky

Unita **SYSTEM** implementuje základní prostředky jazyka Pascal — standardní konstanty, typy, proměnné a podprogramy, mechanismus ošetření běhových chyb a další. Je automaticky užita všemi pascalovskými moduly (programy i programovými jednotkami), aniž by bylo nutné uvádět její identifikátor v klauzulích uses. Všechny pascalovské moduly jsou tudíž na jed-

notce **SYSTEM** závislé a v každém z nich tvoří **SYSTEM** hierarchicky nejvyšší blok.

Unita **GRAPH** je samostatným grafickým systémem, který je třeba inicializovat před započítím práce a po práci řádně ukončit. Obsahuje 75 podprogramů podporujících grafické operace.

Unita **DOS** implementuje řadu podprogramů na bázi rutin operačního systému především pro práci se soubory na disku. Chybové stavy procedur a funkcí implementovaných unitou **DOS** jsou automaticky ošetřeny tak, že nenastane chyba běhu programu (program běží dál) a číslo chybového stavu je uloženo do proměnné `DosError`. Další ošetření chyby je pak zcela v kompetenci programu.

Unita **PRINTER** je velmi malého rozsahu. Deklaruje pouze textový soubor `Lst`, spojí jej s tiskárnou a otevře pro zápis. Po skončení programu, který unitu **PRINTER** používá, je soubor `Lst` automaticky uzavřen. Unita tedy nepřináší žádné speciální prostředky, pouze ušetří zápis pro definování výstupu textu na tiskárnu.

Unita **OVERLAY** obsahuje prostředky pro vytváření překryvných (rozšiřujících) modulů. Zabírá-li program v paměti příliš mnoho místa a tím omezuje datovou oblast, lze program rozdělit na dvě části. První je **jádro**, které obsahuje základní část programu nutnou pro jeho chod a správu překryvného modulu a pak **překryvnou unitu**, z níž se do operační paměti zavede vždy jen ta část, která je v daném okamžiku potřebná pro zpracování. Jakmile tato část již není potřebná, paměť se uvolní a zavede se jiná nutná skupina podprogramů. My však nebudeme tak rozsáhlé programy vytvářet, abychom tuto unitu využívali.

7.2. Uživatelské programové jednotky

Jak již bylo řečeno v úvodu tohoto oddílu, tvůrcem uživatelské programové jednotky může být kdokoliv. Výhodou je existence **zdrojového textu** dané unity a možnost využívat ji v různých verzích dané implementace. Je-li však unita přeložena do strojového kódu, je její užití omezeno pouze na tu verzi jazyka, pod kterou byla přeložena.

Důvod tvorby programových jednotek částečně vyplývá i z předchozí kapitoly "Modulární programování" a jsou pro ni následující důvody:

- Snadnější orientace v rozsáhlejších programech - do unit se spolu umísťují datové položky a podprogramy, které spolu souvisejí. Rozčlenění rozsáhlejšího programu do "modulů".
- Do tvorby rozsáhlejšího programu lze zapojit více programátorů.
- Jednou naprogramované podprogramy obecného charakteru, se již nemusí programovat, ale lze je užít ve více programech.

- Posledním důvodem je způsob organizace paměti. My jsme si ji zatím neobjasňovali, učiníme to tedy nyní.

Každý program napsaný v jazyce Pascal může obsahovat maximálně 64 kB programového kódu (**programový segment - code segment**), 64 kB dat (**datový segment - data segment**) a 64 kB zásobníkového prostoru (**zásobníkový segment - stack segment**). Toto omezení lze pomocí unit rozšířit, protože každá unita má přidělen vlastní programový segment (nikoliv datový). Potřebuje-li další zvětšení kódu programu, je to možné tak, že unity koncipujeme jako překryvné (overlay). V paměti je pak vždy jen jeden z overlayů a ostatní jsou uloženy na vnějším paměťovém mediu (viz. popis standardní unity OVERLAY).

Tvorba vlastních unit

A stejně tak jako hlavní program a podprogramy má i vytvářená vlastní unita obdobnou strukturu. Začíná hlavičkou, za ní následují další tři části:

1. **Rozhraní (interface)** - v této části se nachází soupis všech deklarací, které mají být přístupné z ostatních knihoven nebo z hlavního programu.
2. **Zdrojový text unity (implementation)** - zde jsou těla všech podprogramů, které byly uvedeny v rozhraní, dále zde mohou být deklarace dalších datových typů, konstant, proměnných a podprogramů jako lokální deklarace v této unitě. Jsou tedy **mimo** unitu **nepřístupné**. Deklarace hlaviček jednotlivých podprogramů musí odpovídat příslušným deklaracím hlaviček v rozhraní.
3. **Inicializační** - tato část je nepovinná a lze v ní provést nějaké počáteční nebo přípravné činnosti, např. nastavení hodnot, kontrolu souborů a podobně.

Obecná struktura unity:

```
Unit <název_unity>;
Interface
< veřejné deklarace >
Implementation
< deklarace podprogramů unity >
begin
< inicializační sekvence příkazů >
end.
```

Pokud za implementační částí **nenásleduje** část inicializační končí implementační část klíčovým slovem end a klíčové slovo begin se neuvádí. Příkazy uvedené v inicializační části se provádějí vždy automaticky po spuštění programu, který unitu využívá.



Příklad

uživatelské unity.

V tomto příkladu je vidět shodnost hlavičky procedury **inic** v části **interfa-**
ce s hlavičkou téže procedury v části **implementation** a obdobně shoda
hlaviček procedury **search**. Dále je z toho příkladu patrné, že v části **intefa-**
ce jsou pouze hlavičky obou procedur, z důvodu zajištění vazeb na vnější
prostředí. Kompletní deklarace procedur vč. jejich těla je patrná v části **im-**
plementation. Tato unita neobsahuje inicializační část.

```

unit us;
interface ◀
  type
    ref = ^member;
    member = record
      key : integer;
      count : integer; {pocet vyskytu}
      next : ref;
    end;
    Psez = ^sez;
    sez = record
      Root,      { fiktivni prvni prvek seznamu }
      Tail :ref; {fiktivni posledni prvek seznamu }
    end;

  procedure inic(var P:psez);
    {vytvari prazdny seznam P }
  procedure search( X:integer; P:psez);
    { hleda prvek s klicem X v usporadanem seznamu P
      najde-li ho zvetsi mu count o 1
      nenajde-li ho tam, vlozi ho na spravne misto
      s hodnotou count 1
    }

implementation ◀
  procedure inic(var P:psez);
    {vytvari prazdny seznam P }
  begin
    new(P);
    with P^do
      begin
        new(Root);
        new(Tail);
        Root^.next:=Tail;
      end;
  end;

  procedure search( X:integer; P:psez);
    {hleda prvek s klicem X v usporadanem seznamu P
      najde-li ho zvetsi mu count o 1
      nenajde-li ho tam, vlozi ho na spravne misto s hodnotou count 1}
    var W1,W2 :ref;
      {W1 jde o krok pred W2 }
  begin
    with P^ do
      begin
        W2:=Root; W1:=W2^.next;
        Tail^.key:=X;  {zarazka}
        while W1^.key < X do
          begin W2:=W1; W1:=W1^.next; end;
        { W1^.key >= X}
        if (W1^.key=x) and (w1<>Tail) then {nasel jsem ho }
          W1^.count:= W1^.count+1
        else
          begin {neni tam a musim ho vlozit pred W1 }
            new(W2^.next);
            with W2^.next^ do
              begin key:=X; count:=1; next:=W1; end;
            end;
          end {of with}
        end {of search};
      end;
  end.

```


Využívá-li program nějakou unitu, má k dispozici všechny deklarace uvedené v části rozhraní (*interface*) jako kdyby byly definovány uvnitř vlastního programu.



Poznámka

Zapamatujte si:

- 1. Strukturu jednotky.*
- 2. Podmínku, že název jednotky se musí shodovat s názvem souboru pod kterým je tato unita uložena.*
- 3. Názvy jednotlivých segmentů a jejich maximální velikost.*

8. SOUBORY



Tip

Cíl

Po prostudování kapitoly budete umět vysvětlit pojem soubor, deklarovat datový typ soubor a naučíte se se soubory pracovat, tedy naučíte se jich využívat pro uložení a čtení dat.



Průvodce studiem



Klíčová slova

Soubor, textový, netextový, typový, netypový, čtení, zápis, postupný přístup, sekvenční přístup.



Doba studia: 2 hodiny

Nejdříve musíme říci, co vlastně pod pojmem **soubor** chápeme. Určitě víte již z úvodu do výpočetní techniky, že pojmem soubor označujeme základní logickou jednotku (uzavřenou skupinu) dat, kterou ukládáme na vnější paměťové medium. Nezáleží nám na tom o jaký druh vnějšího paměťového media se jedná, zda je tímto mediem pevný disk, disketa, CD-ROM, DVD nebo flash disk, protože se k souborům v prostředí programovacího jazyka přistupuje stejným způsobem. Fyzické provedení vlastní operace totiž řeší použitý operační systém, a proto se nemusíme při psaní programů zabývat vlastnostmi použitého nosiče dat. Dále víte, že k jednoznačnému určení souboru je nutné znát jeho název a přístupovou cestu.

Z hlediska zpracování souborů můžeme tyto rozdělit podle různých kritérií:

1. Podle použití řídicích znaků:

- a. Soubory **textové** - pro takto chápané soubory jsou uvnitř uložená data složená ze znaků s ordinální hodnotou 32 až 255 nositelem textové informace, znaky s ordinální hodnotou 0 až 31 jsou nositelem řídicí informace, tzn., že po jejich přečtení se provede předem stanovená operace.
- b. Soubory **netextové s udaným typem** (soubor **netextový typový**) - obsahují pouze data stejného typu, např. jen čísla "real", znaky nebo záznamy. Při zpracování obou druhů netextových souborů se znaky s ordinální hodnotou 0 až 31 nepovažují za znaky řídicí, ale zpracují se jako ostatní znaky s ordinální hodnotou 32 až 255.
- c. Soubory **netextové bez udání typu** (soubor **netextový netypový**)

- mohou obsahovat údaje různých datových typů, tj. např. směs čísel, znaků, záznamů, množin, řetězců atd. V tomto případě ale musíme znát jaké druhy údajů a v jakém pořadí se v souboru nacházejí nebo mají nacházet jedná-li se o zápis souboru. Tento druh souborů však není definován v referenčním jazyce Pascal.

2. Podle druhu práce se soubory:
 - a. Soubory určené **pouze ke čtení**.
 - b. Soubory určené **pouze pro zápis**.
 - c. Soubory určené **jak ke čtení, tak k zápisu** - tento druh souborů není definován v referenčním jazyce Pascal.
3. Podle způsobu zpracování dat v souboru:
 - a. Soubory zpracováváné **postupně** tedy **sekvenčně**.
 - b. Soubory **s přímým přístupem** (tento druh souborů není definován v referenčním jazyce Pascal).

Uvedené hlavní rozdělení je obecné a nezáleží na použitém programovacím jazyce. Musím zde zdůraznit, že fyzická tedy skutečná podoba souboru je vždy stejná a že uvedené rozdělení se *týká výhradně práce se souborem a logického přístupu k souboru*. Protože textový soubor má mezi soubory zvláštní postavení, bude jeho struktura popsána detailněji. V případě textového souboru se jedná o soubor znaků, vnitřně uspořádaných do řádků, kde konec řádku je vyznačen dohodnutými řídicími znaky. V prostředí operačního systému DOS je těmito dohodnutými znaky dvojice znaků s ordinálními hodnotami #13#10 (<CR><LF>).

Dále objasníme, jak s taktovýmto pohledem na soubory s nimi pracovat.

Abychom mohli se soubory začít pracovat, je potřebné provést určitou skupinu činností, které nám tuto práci umožní.

Tou první činností je **otevření souboru**. Po úspěšném provedení uvedené činnosti nám operační systém umožní další práci se souborem a jeho obsahem. Soubor otevřený pro čtení umožňuje pouze zikávání dat. Soubor otevřený pro zápis naopak jen jejich uložení. Jedině soubor otevřený pro čtení i zápis nám umožní jak data získávat tak i ukládat.

Další činností je **uzavření souboru**. Tím operačnímu systému oznámíme, že soubor již nebude používán.

Otevření i uzavření souboru musíme provádět s každým souborem bez ohle-

du na to jak budeme zpracovávat uvnitř uložená data, zda textově nebo netextově. Nezohledňujeme ani tu skutečnost, zda budeme data jen číst nebo jen zapisovat a nebo obojí. S textovými soubory pracujeme vždy jen jako se soubory pro čtení nebo pro zápis, ale **nikdy** pro **obojí**. Zato s netextovými soubory můžeme pracovat všemi třemi způsoby.

Zpracování dat v souboru provádíme buď **postupným zpracováním (sekvenčně)** nebo **přímým přístupem**. Sekvenční zpracování znamená, že data jsou ze souboru čtena postupně v tom pořadí, jak jsou zapsána. Při čtení se nelze vracet zpět ani přeskakovat jinam. Uvedený způsob tedy **sekvenční** je **jediným** způsobem jak zpracovat **textový** soubor. Přímý přístup k datům pak znamená, že každá položka souboru má svoje pořadové číslo a toto číslo lze použít pro směřování na čtenou nebo zapisovanou položku. Tato metoda se využívá výhradně pro práci s netextovými soubory.

8.1. Deklarace souborů

Než budeme ve výkladu pokračovat, je nutné zavedení dvou zkratk pro zkrácení zápisu deklarací příkazů pro práci se soubory:

Identifikátor souboru = IdS.

Identifikátor datového typu = IdDt.

Deklaraci souboru lze provést dvěma způsoby:

1. *V deklarační části proměnných **var** přímo následující formou zápisu:*

IdS1 : file; = **netextový netypový** soubor - obecný soubor

IdS2 : file of real; = **netextový typový** soubor - číselná data typu real

IdS3 : file of char; = **netextový typový** soubor - znaková data typu char

IdS4 : text; = **textový** soubor - pro textový soubor je zaveden **zvláštní typ**

- ze zápisu je vidět, že za ":" se neobjevuje klíčové slovo "file" ale "text"

2. *V deklarační části definice datových typů **type** následující formou zápisu:*

IdDt1 = file of real; = **netextový typový** soubor - číselná data typu real

IdDt2 = file of char; = **netextový typový** soubor - znaková data typu char

IdDt3 = file of [1..10]; = **netextový typový** soubor - soubor množin

Při tomto způsobu deklarace je však navíc nutno v části proměnných **var** provést ještě přiřazení datového typu následující formou zápisu:

IdS5 : IdDt1; = **netextový typový** soubor - číselná data typu real

IdS6 : IdDt3; = **netextový typový** soubor - soubor množin



Příklad

deklarace souborů různých typů:

```
type RealFile = file of Real;
   Kniha      = record
                       Autor, Nazev : string [30];
                       Editor       : string [15];
                       RokVydani    : Word
                   end;
var  F1       : RealFile;
     F2       : file of Kniha;
     F3       : file;
```

Těmito deklaracemi jsou zavedeny tři proměnné typu soubor. Každý fyzický soubor, který bude později spojen s proměnnou F1, bude chápán jako posloupnost reálných čísel, soubor spojený s proměnnou F2 jako posloupnost komponent typu Kniha a soubor spojený s proměnnou F3 jako posloupnost bloků bytů, bez definované vnitřní struktury bloků.

8.2. Práce se soubory

Ukázali jsme, jak se datový typ soubor pro práci deklaruje. V následujícím textu popíšeme procedury a funkce, které nám umožní s těmi již deklarovanými soubory pracovat. V předchozí kapitole je uvedeno, že práce se soubory je závislá na vlastnostech operačního systému. Jednotlivé implementace jazyka Pascal mají proto různé prostředky pro zpracování souborů. My se budeme zabývat pouze těmi, které lze použít v implementacích FPascal a TPascal.

Nyní popíšeme kroky, které je nutné realizovat abychom mohli se soubory pracovat. Prvním krokem, který musíme provést, je **propojení skutečného souboru s identifikátorem**. Pro tuto operaci využijeme předdefinovanou proceduru **assign**. Procedura má dva parametry - identifikátor a název souboru.

V programu voláme proceduru následovně:

Assign (**var** identifikátor; jméno_souboru: **string**);

Tuto proceduru můžeme použít pro práci jak se soubory textovými tak i netextovými a nezáleží ani na tom, jakým způsobem budeme ze souborem pracovat, jestli z něj budeme jen číst nebo jen zapisovat a nebo obojí. Příkaz této procedury musíme provést **vždy** před použitím souboru v následujících podprogramech.

Otevření souboru - tímto úkonem určujeme, jaké činnosti budeme se souborem dále provádět. Jak již víme, jsou tři způsoby práce se souborem, a proto máme i tři způsoby otevření souboru:

1. **Otevření pro čtení** - provádíme procedurou **Reset**. Pokud je již soubor otevřen, procedura jej nejprve uzavře a poté otevře. Po otevření souboru je aktuální pozice nastavena na začátek souboru. Aktuální pozice je místo, kde se v souboru nacházíme. Je to místo, kde se začne se čtením údajů.

V programu voláme proceduru následovně:

Reset (var F[: soubor; velikost: word]);

Pozor! Důležitou podmínkou pro činnost této procedury je existence skutečného souboru na disku. V případě, že se budeme snažit otevřít soubor který neexistuje, program ukončí svou činnost a ohlásí chybu. V nepovinné části parametru (je to část uzavřená do [] závorek) příkazu procedury uvádíme parametry, které používáme pro práci s netextovými netygovými soubory.

2. **Otevření pro zápis** - můžeme provést dvěma způsoby:

První způsob volaná procedura vytvoří **nový** soubor nebo u existujícího souboru **zruší jeho starý obsah** a aktuální pozice je tím pádem nastavena na začátek.

V programu voláme proceduru následovně:

Rewrite (var F[: soubor; velikost: word]);

Použití nepovinné části je shodné jako u procedury Reset.

Druhý způsob je charakterizován otevřením pro zápis **na konec** existujícího souboru. O tomto způsobu hovoříme jako o **přidání k obsahu souboru**. **Uvedený způsob je však možný pouze u textových souborů**. Aktuální pozice v souboru je automaticky nastavena **za poslední** pozici. To platí pouze tehdy, neobsahuje-li soubor znak #26 <EOF>. Je-li v souboru tento znak, pak se aktuální pozice nastaví právě na něj a případným zápisem se tento znak přepíše. Při uzavření textového souboru se na jeho konec zapisuje znak #26 <EOF>.

V programu voláme proceduru následovně:

Append (var F: text);

3. **Otevření pro čtení i zápis současně** - tento způsob práce můžeme použít pouze pro netextové soubory. **POZOR - tímto způsobem nelze pracovat s textovými soubory.**

Pro práci s netextovým souborem použijeme procedur *reset* a *rewrite* a nerozlišujeme při ní, zda byl soubor deklarován jako typový nebo nety-

pový.

Uzavření souboru - provádí pro všechny typy souborů procedura `close`.

V programu voláme proceduru následovně:

Close (**var** F);

Vyprázdnění vyrovnávací paměti -

V programu voláme proceduru následovně:

Flush (**var** F: text);

Procedury, které jsme si doposud v tomto oddíle popsali, můžeme nazvat procedurami "přípravnými". Uvedené procedury totiž nemanipulují žádným způsobem s datovým obsahem souborů, pouze tuto manipulaci zpřístupňují. S některými procedurami, které budou fyzicky pracovat s daty jsme se už seznámili. Jsou to procedury `Read`, `ReadLn`, `Write`, `WriteLn`. Při jejich volání jen doplníme jeden parametr. Tímto parametrem bude identifikátor souboru, se kterým se má pracovat a v seznamu parametrů jej budeme uvádět na prvním místě. Bohužel tyto procedury neumožní zpracovat data všech druhů souborů. Je proto nutné jejich počet rozšířit ještě o dvě procedury, a to *BlockRead* a *BlockWrite*, které budou pracovat pouze s daty netextových netypových souborů. Popis těchto souborů si uvedeme později. I přes výše uvedená doplnění dalšími procedurami není náš aparát dostačující abychom kompletně zvládli práci se soubory. Potřebujeme ještě další nástroje. Jde o nástroje na zjištění pozice a také na vyhledávání v souboru. Těmito nástroji jsou následující funkce:

Funkce pracující se všemi typy souborů:

EOF (End of File) - testuje stav konec souboru a vrací hodnotu boolean.

U textového souboru to bude hodnota *True* právě tehdy, když je na vstupu aktuálním znakem znak <EOF> (#26).

U netextového souboru to bude hodnota *True* v případě, kdy se ukazatel aktuální pozice v souboru nachází za jeho poslední komponentou.

IOResult - vrací aktuální hodnotu proměnné *InOutRes* (chybový kód poslední vstupně-výstupní operace) a současně ji vynuluje (přiřadí proměnné hodnotu nula), čímž umožní normální funkci vstupně-výstupních operací.

Funkce pracující pouze s textovými soubory:

EOLn (End of Line) - testuje stav konec řádky a vrací hodnotu boolean. Bude to hodnota *True* právě tehdy, když je na vstupu aktuálním znakem <CR> (#13) nebo <EOF> (#26).

SeekEOF - používá se pro test stavu konec souboru při zpracování posloupnosti čísel a vrací hodnotu boolean. Testu předchází „přeskočení“ všech od-

dělovačů, tj. všech znaků menších nebo rovných znaku mezera (#32) a současně různých od znaku <EOF> (#26).

SeekEOLn - pracuje obdobně jako **SeekEOF**, testován je však stav konec řádky a mezi oddělovače není počítán ani znak <CR> (#13).

Funkce pracující pouze s netextovými soubory:

FilePos - vrací hodnotu ukazatele aktuální pozice v souboru — index aktuální komponenty (počáteční komponenta má index 0), vracená hodnota je datového typu **LongInt**.

FileSize - vrací hodnotu velikosti souboru v komponentách (aktuální celkový počet komponent v souboru), vracená hodnota je datového typu **LongInt**.

Seek - slouží k nastavení ukazatele aktuální pozice v souboru na hodnotu udanou v parametru.

Truncate - slouží k odstranění (výmazu) aktuální komponenty a všech následujících ze souboru.

Vzhledem ke specifickému postavení standardních textových souborů **input** a **output** se o nich zde ještě zmíníme blíže. My jsme o nich hovořili v souvislosti s procedurami *Read*, *ReadLn*, *Write* a *WriteLn*. Rád bych zde opět připoměl, že uváděné soubory se **nemusí** deklarovat, protože to zařídili již programátoři při implementaci jazyka. Zároveň jsou uvedené soubory přiřazeny určitým fyzickým souborům (soubor *input* zpravidla *klávesnici*, soubor *output* obvykle *monitoru*). Zvláštní jsou ale i tím, že při realizaci programu jsou automaticky otevírány a zavírány a že není třeba tyto akce programovat (neuvádějí se v příkazech vstupu a výstupu, nenajdeme je tedy v parametrech zmiňovaných procedur).

8.2.1. Čtení a zápis textových souborů

Práce s textovým souborem bude pro nás asi tou nejčastější prací se soubory a proto bude probrána podrobněji. V předchozím textu jsme vyjmenovali procedury a funkce sloužící k práci s vlastními daty v souborech a nyní ukážeme jejich praktické použití.

8.2.2. Čtení a zápis netextových souborů



Poznámka

Zapamatujte si:

1. Členění souborů na textové a netextové, netextových na typové a netypové.
2. Tři druhy otevření souboru pro čtení, zápis, čtení i zápis současně.

3. Čtení postupné (sekvenční) a s přímým přístupem.
4. Otevření textového souboru - pro přidání na konec.
5. Uzavření souboru.
6. Vyprázdnění paměti.
7. Zvláštnosti práce se standardními soubory input a output.

9. DYNAMICKÉ DATOVÉ TYPY



Tip

Cíl

Vysvětlit deklaraci a práci s dynamickými datovými strukturami



Klíčová slova

Datový typ ukazatel, dynamicky alokované proměnné, datový segment, nepřímá adresace, NIL.



Doba studia: 2 hodiny

Datový typ ukazatel

Všechny datové typy, se kterými jste se doposud seznámili, obsahují konkrétní hodnoty – data a hovoříme o nich jako o statických datových typech. Proměnná tohoto datového typu je při deklaraci umístěna do konkrétního místa v operační paměti a na tomto setrvává po celou dobu své existence. Přístup k hodnotám těchto proměnných je zajištěn prostřednictvím **identifikátorů** proměnných a jejich položek. Odlišný typ informace uchovává datový typ **ukazatel (pointer)**. Uvedený datový typ uchovává na rozdíl od těch ostatních **adresy prvků v operační paměti počítače**, skutečná data však **neobsahuje**. Ukazatel tedy ukazuje na blok paměti, jehož organizace je obvykle dána jiným datovým typem. Ač je datový typ ukazatel datovým typem statickým, vytváříme jeho pomocí tzv. **dynamicky alokované proměnné**, které se od těch statických liší tím, že okamžik jejich vzniku nebo zániku není dán **deklarací**, ale zvláštním příkazem pro vytvoření nebo zánik. Pokusíme-li se shrnout tento obsáhlý výklad do jedné věty, lze říci: *V proměnné statického datového typu **ukazatel** je uchovávána adresa (o velikosti 4 B) dynamicky alokované proměnné*. Dynamicky alokované proměnné a z nich vytvořené konstrukce vznikají až za běhu programu a není nutné jako u statických struktur znát jejich počet již v okamžiku překladu programu. To je hlavním důvodem využívání těchto datových struktur.

Pro snadnější pochopení si deklaraci a užití datového typu ukazatel ukážeme na příkladě a doplníme grafickým znázorněním.



Příklad

práce s datovým typem ukazatel:

```
var DCislo: ^Real;  
    DString: ^String;
```

```
.  
. .  
begin  
  New (DCislo); {vyhrazení (alokace) paměti}  
  DCislo^ := 55.1; {naplnění bloku paměti  
                                     reálným číslem 55.1}  
  .  
  .  
  .  
  Dispose(DCislo); {uvolní vyhrazenou paměť}  
  .  
  .  
  New (DString); {vyhrazení (alokace) paměti}  
  DString^ := 'Text retezce'; {naplnění bloku  
                               paměti znaky "Text retezce"}  
  .  
  .  
  Dispose(DString); {uvolní vyhrazenou paměť}  
  .  
  .  
end;
```

Popis činnosti:

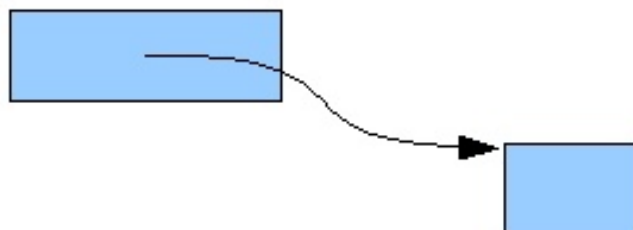
V deklarační části **var** jsou deklarovány dvě proměnné datového typu **ukazatel** s identifikátory "DCislo" a "DString", každá z těchto proměnných zabírá 4 B v datovém segmentu paměti. Příkazem **New** (DCislo) vytvoříme úsek o velikosti 6 B (což je velikost proměnné datového typu real) na určitém místě operační paměti a uložení adresy tohoto místa do proměnné DCislo. Při provádění příkazu DCislo^ := 55.1 proběhnou dvě akce: nejdříve se zjistí hodnota v proměnné DCislo. Touto hodnotou je adresa ukazující na určité místo v paměti a do tohoto místa se následně uloží (přiřadí) zadaná hodnota (55.1). Pokud již proměnnou nepotřebujeme, uvolníme paměť příkazem **Dispose** (DCislo). Obdobně je tomu s proměnnou DString pouze s tím rozdílem, že alokovaný úsek paměti nebude 6 B ale 256 B (což je velikost proměnné datového typu řetězec).

Programový příkaz a jeho grafické znázornění:

a1) Alokace pro cislo "real".

New (DCislo)

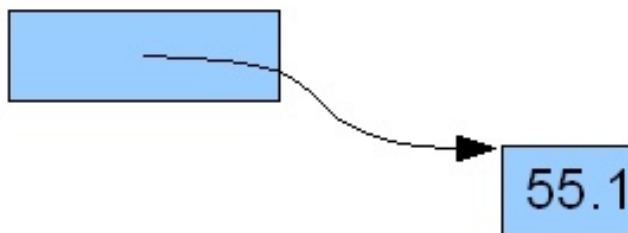
DCislo



a2) Přiřazení hodnoty do dynamicky alokované proměnné "DCislo".

DCislo[^] := 55.1

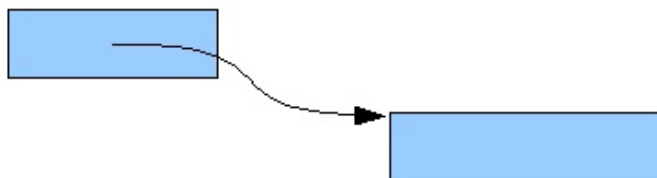
DCislo



b1) Alokace pro řetězec.

New (DString)

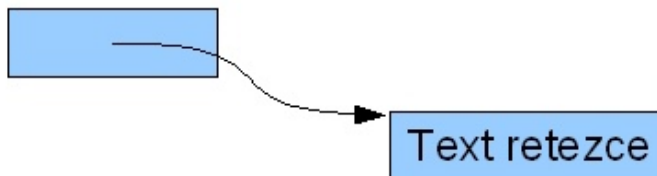
DString



b2) Přiřazení hodnoty do dynamicky alokované proměnné "DString".

DString[^] := 'Text retezce'

DString



Před použitím dynamicky alokované proměnné jí musíme určit básový datový typ, přidělit úsek (místo v) paměti a adresu úseku uložit do proměnné typu ukazatel. Ve výše uvedeném příkladě je to DCislo a DString. Místo v paměti vyhradíme (alokujeme) prostřednictvím procedury **New**. Dle básového datového typu se kterým je ukazatel provázán, vytvoří tento příkaz v paměti prostor o velikosti odpovídající básovému datovému typu. S daty, tedy se skutečným obsahem nebo-li hodnotou na kterou ukazuje typ ukazatel, pracujeme pomocí **operátoru nepřímé adresace** [^] (stříška - Alt94). Viz příkaz přiřazení ve výše uvedeném příkladě. Vlastní hodnoty se pak uloží tam, kam ukazují ukazatele. Po ukončení operací s typem ukazatel, musíme přidělenou paměť opět uvolnit. K tomuto účelu slouží procedura **Dispose**, která přidělenou paměť uvolní pro další užití. Je nutné mít na zřeteli skutečnost, že příkazem Dispose sice dojde k uvolnění paměti, ale pouze té dynamicky alokované, proměnná ukazatel je i nadále deklarována a její obsah je ve stavu **nedefinován**. V této souvislosti nesmíme zapomenout, že pro užití s datovým typem ukazatel je také rezervováno slovo **NIL**. Jenž reprezentuje konstantu s

hodnotou ukazatele, který neukazuje na **žádné** místo v paměti. Ukazatel s hodnotou **NIL** je kompatibilní se všemi ostatními datovými typy. Je nutné velmi pečlivě rozlišovat ukazatel s nedefinovanou hodnotou a ukazatel s hodnotou **NIL**. V případě nedefinované hodnoty ukazatele a pokusu o její použití dojde k chybě. Zatím co má-li ukazatel přiřazenu hodnotu **NIL** je jeho užití naprosto správné. Když hovoříme o dynamicky alokovaných proměnných uvedeme ještě operace, které při práci s nimi můžeme použít. Tou první operací je přiřazení, které však můžeme použít jen mezi ukazateli stejného typu. Dále můžeme ukazatele stejného typu navzájem porovnávat, avšak jsou povoleny pouze dva testy a to test na rovnost a nerovnost. Vzhledem k tomu, že porovnáváme adresy ostatní testy nemají smysl. Dva ukazatele jsou si rovny, pokud obsahují stejnou adresu, pak ukazují na tutéž dynamickou proměnnou.



Poznámka

Zapamatujte si:

1. *Dynamicky alokovaná proměnná.*
2. *New - vyhrazení (alokace) paměti.*
3. *Dispose - uvolnění alokované paměti.*
4. *^ - (stříška) operátor nepřímé adresace.*
5. *NIL - hodnota v ukazateli, který neukazuje na žádné místo v paměti.*

10. DIREKTIVY PRO KOMUNIKACI



Tip

Cíl

Cílem je objasnit užití základních direktiv pro řízení překladu programu.



Klíčová slova

Direktiva, lokální, globální, direktivy přepínačové a parametrické.



Doba studia: 1/2 hodiny

Jak již bylo uvedeno implementace FPascal a T/BPascal obsahují sadu direktiv pro řízení procesu překladu. Tyto direktivy jsou uváděny ve zdrojových textech ve složených závorkách, přičemž za levou (otevírací) závorkou bezprostředně následuje znak dolar (\$), odlišující direktivu od komentáře (poznámky), a název (značka) direktivy. Na velikosti písmen názvu direktivy nezáleží.

Podle funkce lze direktivy rozdělit na přepínačové a parametrické.

10.1. Přepínačové direktivy

Přepínačové direktivy přepínají způsob kompilace zdrojového textu vzhledem k jisté vlastnosti mezi dvěma stavy. Název direktivy je tvořen jediným písmenem a volba stavu se provádí symbolem plus (+) resp. minus (-), uvedeným bezprostředně za názvem. Například: `{ $N+ }`

Uvnitř jedné složené závorky lze současně uvést i více názvů a stavů přepínačových direktiv, které se pak navzájem oddělují čárkami (mezery ani jiné nadbytečné oddělovače se mezi nimi vyskytovat nesmí): `{ $N+,E+ }`

Podle rozsahu platnosti lze přepínačové direktivy klasifikovat na **globální** a **lokální**. Globální direktivy smí být uvedeny nejpozději za hlavičkou programu resp. programové jednotky a jimi definovaný stav je platný během kompilace celého textu modulu. Lokální direktivy lze naopak uvádět kdekoliv, kde je povolen výskyt komentáře, a jimi definované stavy jsou platné až do dalšího výskytu téže direktivy s opačným znaménkem, opět však nejvýše do konce textu modulu.

Implicitní stavy přepínačových direktiv jsou uplatněny vždy na začátku kompilace každého dílčího modulu. Modifikovat je lze v zaškrťávacích polích v nastavení vývojového prostředí nebo dialogu Compiler Options, který je přístupný příkazem Compiler nabídky Options vývojového prostředí. Za-

škrtnuté resp. nezaškrtnuté pole pak odpovídá stavu plus resp. minus příslušné direktivy.

Nejdříve si uvedeme ty tři nejpoužívanější.

{SIstav}

- Řídí zpracování vstupních a výstupních operací (V/V operací, I/O operací)
- Je-li nastaveno {SI+}, provádí se u všech vstupních a výstupních operací kontrola, zda nedošlo k chybě. Při výskytu chyby se běh programu ukončí a vypíše se chybové hlášení.
- Je-li naopak nastaveno {SI-}, musí programátor kontrolovat chyby vstupu a výstupu pomocí standardní funkce IOResult
- Chyby vstupu a výstupu pak nezpůsobují zastavení programu, ale způsobí, že všechny následující operace vstupu a výstupu jsou ignorovány až do doby, kdy se vyvolá funkce IOResult
- Funkce IOResult vrací hodnotu nula jestliže předchozí operace vstupu a výstupu proběhla úspěšně. Jinak vrací hodnotu různou od nuly, která odpovídá vzniklé chybě.
- Uvedeného mechanismu je možné využít ke zjištění, zda daný soubor již existuje či nikoliv

Uvedené můžeme též formulovat takto:

Lokální direktiva ošetření chyb vstupních a výstupních operací. Ve stavu {SI+} je důsledkem každé chyby při vstupně-výstupních operacích běhová chyba programu. Ve stavu {SI-} je číslo vzniklé chyby pouze uloženo do speciální proměnné. Program pak běží dál, ale veškeré vstupně-výstupní operace jsou ignorovány, dokud není chybový příznak vynulován voláním funkce IOResult — chyba musí být ošetřena programem.

A ještě poznámka: Vyvolání funkce IOResult je „destruktivní“. To znamená, že při prvním vyvolání funkce IOResult vrací hodnotu odpovídající chybě, ale při dalším bezprostředním vyvolání již vrací hodnotu nula. Je to dáno tím, že následné vyvolání IOResult vrací výsledek předešlé I/O operace, kterou bylo úspěšně provedené vyvolání IOResult

{SRstav}

Lokální direktiva kontroly rozsahu indexů polí a řetězců a hodnot, přiřazovaných ordinálním proměnným. Ve stavu {SR+} vsune překladač do kódu přístupu k prvku pole (řetězce) resp. do kódu přiřazovacího příkazu kontrolní test, který při překročení deklarovaného rozsahu indexu resp. hodnoty proměnné generuje běhovou chybu programu. Ve stavu {SR-} se kontrolní kód nevkládá a překročení rozsahu je ignorováno — program běží dál, ale důsledky překročení rozsahu nejsou nijak ošetřeny.

Testy překročení rozsahu poněkud zpomalují běh programu a zvětšují rozsah

jeho kódu, proto se obvykle používají jen ve fázi ladění. V konečné verzi programu by již měly být všechny chyby, které překročení rozsahu způsobují, odstraněny.

{SVstav}

Lokální direktiva kontroly identity typů formálních a skutečných řetězcových parametrů podprogramů. Při překladu volání podprogramů provádí překladač kontrolu identity typů formálních a skutečných parametrů volaných odkazem. Ve stavu {V–} však není striktní identita vyžadována u parametrů řetězcových typů — skutečný parametr pak smí být libovolného typu řetězec.

A nyní ty zbývající v abecedním pořádku.

{SAstav}

Globální direktiva zarovnávání dat. Ve stavu {A+} jsou všechny proměnné a typové konstanty větší než jeden byte zarovnávány na hranici slova tak, že začínají na sudých adresách. Na procesorech 8086 je přístup k datům na sudých adresách rychlejší než přístup k datům na adresách lichých.

{SBstav}

Lokální direktiva režimu vyhodnocování booleovských výrazů. Týká se vyhodnocování binárních logických operací and a or. Ve stavu {B+} (režim úplného vyhodnocování) jsou vždy vyhodnoceny oba operandy, i když je výsledek operace znám již po vyhodnocení prvního z nich, zatímco ve stavu {B–} (režim zkráceného vyhodnocování) se v takovém případě již vyhodnocení druhého operandu neprovádí.

Pokud má totiž první operand operace and resp. or hodnotu False resp. True, je hodnotou operace False resp. True, nezávisle na hodnotě operandu druhého. Vyhodnocení druhého operandu pak má smysl pouze tehdy, je-li jeho důsledkem nějaký vedlejší efekt — je-li například druhým operandem volání logické funkce, která před poskytnutím výsledku vykoná nějakou přidruženou akci.

{Dstav}

Globální direktiva generování ladicích informací o zdrojovém textu modulu pro debugger. Ve stavu {D+} direktiva generování ladicích informací povoluje a tím umožňuje krokování běhu modulu, nastavení breakpointů a lokalizaci případné běhové chyby ve zdrojovém textu.

Při překladu programové jednotky jsou povolené ladicí informace automaticky uloženy do tpu-souboru, do cílového kódu programu (exe-souboru) se však povolené ladicí informace ukládají pouze v případě nastavení položky Debugging dialogu Debugger na hodnotu Standalone. Dialog je přístupný příkazem Debugger nabídky Options vývojového prostředí. Zahrnutí ladicích informací do cílového kódu programu umožňuje jeho ladění mimo vývojové prostředí, externím ladicím programem (Turbo Debugger).

{SEstav}

Globální direktiva emulace aritmetického koprocessoru. Má smysl pouze při současném nastavení direktivy {N+} (použití koprocessoru povoleno). Ve stavu {E+} je do kódu programu zahrnuta knihovna pro komunikaci s koprocessorem, která umožňuje jeho případnou emulaci. Program pak buď používá přímo koprocessor (je-li přítomen) nebo jej programově emuluje. Ve stavu {E-} je do kódu programu zahrnuta podstatně menší knihovna, která emulaci koprocessoru neumožňuje (program pak lze spustit pouze za přítomnosti koprocessoru).

Direktiva je ignorována při zakázaném použití koprocessoru. Do kódu programu se pak vkládá pouze knihovna pro výpočty s reálnými čísly základního typu Real.

{SFstav}

Lokální direktiva implicitního modelu volání podprogramů. Týká se všech následujících podprogramů ve zdrojovém textu modulu, které nejsou veřejnými podprogramy jednotky (ty jsou přeloženy vždy pro daleký model volání) ani nejsou vnořené v bloku jiného podprogramu (ty jsou přeloženy vždy pro blízký model volání) ani nemají požadovaný model volání explicitně vyznačen direktivou far resp. near. Ve stavu {F+} resp. {F-} jsou všechny tyto podprogramy přeloženy pro daleký resp. blízký model volání.

Podprogramy s blízkým modelem volání jsou adresovány pouze ofsetovou částí adresy, proto mohou být volány pouze z toho segmentu kódu programu, ve kterém jsou alokovány (tj. z toho modulu zdrojového textu, ve kterém jsou deklarovány) a nemohou být předávány jako parametry ani nemohou být ovládány prostřednictvím procedurálních proměnných.

{SGstav}

Lokální direktiva generování kódu pro procesor 80286. Ve stavu {G-} je cílový kód tvořen pouze instrukcemi procesorů 8086 a 8088, kterými byly vybaveny starší modely počítačů. Ve stavu {G+} je množina povolených instrukcí rozšířena o speciální instrukce procesoru 80286, což se může projevit zvýšením rychlosti výpočtu a snížením rozsahu kódu programu. Takto kompilovaný program pak ovšem není slučitelný s procesory 8086 a 8088.

{SLstav}

Globální direktiva generování ladicích informací o lokálních symbolech v blocích podprogramů a v implementačních částech programových jednotek. Ve stavu {L+} je generování ladicích informací o lokálních symbolech povoleno.

Direktiva {L+} je ignorována, pokud je v modulu zakázáno generování ladicích informací direktivou {D-}.

{Nstav}

Globální direktiva použití aritmetického koprocessoru. Ve stavu `{N+}` povoluje provádění výpočtů s reálnými čísly využitím aritmetického koprocessoru 8087 (nebo jeho emulace) a použití speciálních číselných typů koprocessoru (Single, Double, Extended a Comp). Ve stavu `{N-}` může být pro reprezentaci reálných čísel použit pouze typ Real.

`{Qstav}`

Lokální direktiva kontroly přetečení při výpočtu hodnoty ordinálních výrazů. Ve stavu `{Q+}` vsune kompilátor do kódu vyhodnocení výrazu kontrolní test, který při vzniku přetečení generuje běhovou chybu programu. Ve stavu `{Q-}` se kontrolní kód nekládá a přetečení je ignorováno — program běží dál, ale výsledek výpočtu výrazu je chybný.

Testy přetečení poněkud zpomalují běh programu a zvětšují rozsah jeho kódu, proto se obvykle používají jen ve fázi ladění. V konečné verzi programu by již měly být všechny chyby, které přetečení způsobují, odstraněny.

`{Pstav}`

Globální direktiva předefinování významu rezervovaného slova string v deklaracích odkazem volaných formálních parametrů podprogramů. Ve stavu `{P-}` je předefinování zakázáno — formální parametr, v jehož deklaraci je uvedeno rezervované slovo string, je typu string [255] (jako ve starších verzích Turbo Pascalu). Ve stavu `{P+}` má rezervované slovo string v deklaraci odkazem volaného formálního parametru stejný význam jako identifikátor OpenString — typem formálního parametru je otevřený řetězec string [X], kde X je deklarovaná maximální délka skutečného parametru, kterým pak smí být proměnná libovolného řetězcového typu.

`{Sstav}`

Lokální direktiva kontroly přeplnění zásobníku. Ve stavu `{S+}` je součástí každého volání podprogramu kontrolní kód, který otestuje, zda je na zásobníku dostatek místa pro lokální proměnné podprogramu a jiná dočasná uložení dat. Při záporném výsledku je pak generována chyba běhu programu. Pokud dojde k přeplnění při vypnuté kontrole, program pravděpodobně havaruje.

`{Tstav}`

Lokální direktiva režimu typové kontroly kompatibility ukazatelových typů. Ve stavu `{T-}` je kontrola striktní — dva ukazatelové typy jsou uznány kompatibilními pouze pokud jsou identické nebo pokud je jeden z nich ukazatelem univerzálním. Ve stavu `{T+}` jsou za kompatibilní považovány i ukazatelové typy s identickými báзовými typy.

Dalším aspektem, který na stavu této direktivy závisí, je typ výrazu `@Objekt`, kde Objekt je neprocedurální proměnná (hodnotou výrazu je ukazatel na uvedenou proměnnou). Ve stavu `{T-}` je typem výrazu ukazatel univerzální (Pointer), kdežto ve stavu `{T+}` ukazatel typový, jehož báзовým typem je typ proměnné Objekt.

{SXstav}

Globální direktiva rozšířené syntaxe volání funkcí. Volání funkce je podle syntaxe jazyka výrazem a proto se standardně, tj. ve stavu {SX–}, může vyskytovat pouze tam, kde je výraz očekáván. Ve stavu {SX+} lze volání funkce použít i jako příkaz — pokud ovšem v daném kroku nezáleží na výsledku funkce nýbrž na vedlejších efektech jeho výpočtu.

10.2. Parametrické direktivy

Parametrickými direktivami jsou specifikovány vkládané a připojované soubory a požadavky na přidělení paměti. Název direktivy je opět tvořen jedním písmenem a parametry jsou od něj odděleny (aspoň jednou) mezerou.

{SI soubor}

Vložení obsahu textového souboru do zdrojového textu modulu. Při překladu zdrojového textu modulu se postupuje tak, jakoby na pozici výskytu direktivy byl uveden obsah souboru soubor.

Parametr soubor je systémová specifikace vkládaného souboru. Pokud neobsahuje příponu, bere se implicitně .pas. Pokud neobsahuje cestu k adresáři, je vkládaný soubor hledán nejprve v aktuálním adresáři a dále v adresářích, uvedených v položce Include directories dialogu Directories, který je přístupný příkazem Directories nabídky Options vývojového prostředí.

Direktiva nesmí být uvedena uvnitř příkazové části a vkládané soubory lze do sebe vnořit maximálně v patnácti úrovních.

{SL soubor}

Připojení externího kódu ke kódu modulu. Soubor soubor musí obsahovat kód v relokabilním formátu (tzv. object-kód). Může být získán například překladem assemblerského podprogramu nebo jako produkt konverze binárního souboru utilitou binobj.exe. Ve zdrojovém textu modulu se pak uvádí pouze hlavička externího podprogramu a místo jeho těla direktiva external. Identifikátor podprogramu, uvedený v deklaraci hlavičky, se ovšem musí shodovat s veřejným jménem podprogramu, které je součástí připojovaného souboru.

Parametr soubor je systémová specifikace připojovaného souboru. Pokud neobsahuje příponu, bere se implicitně .obj. Pokud neobsahuje cestu k adresáři, je soubor hledán nejprve v aktuálním adresáři a dále v adresářích, uvedených v položce Object directories dialogu Directories, který je přístupný příkazem Directories nabídky Options vývojového prostředí.

Na umístění direktivy ve zdrojovém textu modulu nezáleží (nesmí být uvedena pouze uvnitř lexikálního elementu), pro přehlednost je vhodné umístit ji buď na jeho začátek nebo poblíž deklarace hlavičky podprogramu, jehož kód připojovaný soubor obsahuje.

{\$M zásobník, minheap, maxheap}

Požadavek na přidělení paměti pro zásobník a heap programu. Direktivu lze uvést pouze na začátku zdrojového textu, nejpozději za hlavičkou programu (ve zdrojových textech programových jednotek je ignorována). Všechny tři parametry jsou celočíselné a vzájemně se oddělují jednou nebo více čárkami, středníky nebo mezerami.

Parametr zásobník je požadovaná velikost zásobníku v bytech. Povolené jsou hodnoty 1024 až 65 520. Parametr minheap resp. maxheap je požadovaná minimální (nezbytně potřebná) resp. maximální (nejvýše využitelná) velikost heapu v bytech, povolené jsou hodnoty 0 až 655 360, přičemž musí být minheap ≤ maxheap. Pokud je při spouštění programu méně volné paměti pro heap než požaduje minheap, nebude program spuštěn. Pokud je jí dostatek, ale méně než požaduje maxheap, přidělí se celá. Pokud je jí ještě více, přidělí se právě maxheap bytů.

Implicitní hodnoty parametrů direktivy (standardně 16 384, 0, 655 360) lze nastavit v dialogu Memory Sizes, který je přístupný příkazem Memory sizes nabídky Options vývojového prostředí.

A. ASCII tabulka

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
0	#0	<NUL>	32	#20	mezera	64	#40	@	96	#60	`
1	#1	<SOH>	33	#21	!	65	#41	A	97	#61	a
2	#2	<STX>	34	#22	"	66	#42	B	98	#62	b
3	#3	<ETX>	35	#23	#	67	#43	C	99	#63	c
4	#4	<EOT>	36	#24	\$	68	#44	D	100	#64	d
5	#5	<ENQ>	37	#25	%	69	#45	E	101	#65	e
6	#6	<ACK>	38	#26	&	70	#46	F	102	#66	f
7	#7	<BEL>	39	#27	'	71	#47	G	103	#67	g
8	#8	<BS>	40	#28	(72	#48	H	104	#68	h
9	#9	<TAB>	41	#29)	73	#49	I	105	#69	i
10	#A	<LF>	42	#2A	*	74	#4A	J	106	#6A	j
11	#B	<VT>	43	#2B	+	75	#4B	K	107	#6B	k
12	#C	<FF>	44	#2C	,	76	#4C	L	108	#6C	l
13	#D	<CR>	45	#2D	-	77	#4D	M	109	#6D	m
14	#E	<SO>	46	#2E	.	78	#4E	N	110	#6E	n
15	#F	<SI>	47	#2F	/	79	#4F	O	111	#6F	o
16	#10	<DLE>	48	#30	0	80	#50	P	112	#70	p
17	#11	<DC1>	49	#31	1	81	#51	Q	113	#71	q
18	#12	<DC2>	50	#32	2	82	#52	R	114	#72	r
19	#13	<DC3>	51	#33	3	83	#53	S	115	#73	s
20	#14	<DC4>	52	#34	4	84	#54	T	116	#74	t
21	#15	<NAK>	53	#35	5	85	#55	U	117	#75	u
22	#16	<SYN>	54	#36	6	86	#56	V	118	#76	v
23	#17	<ETB>	55	#37	7	87	#57	W	119	#77	w
24	#18	<CAN>	56	#38	8	88	#58	X	120	#78	x
25	#19		57	#39	9	89	#59	Y	121	#79	y
26	#1A	<EOF>	58	#3A	:	90	#5A	Z	122	#7A	z
27	#1B	<ESC>	59	#3B	;	91	#5B	[123	#7B	{
28	#1C	<FS>	60	#3C	<	92	#5C	\	124	#7C	
29	#1D	<GS>	61	#3D	=	93	#5D]	125	#7D	}
30	#1E	<BS>	62	#3E	>	94	#5E	^	126	#7E	~
31	#1F	<US>	63	#3F	?	95	#5F	_	127	#7F	

Neuvádím zde druhou část tabulky pro znaky s hodnotami od 128 (#80) do 255 (#FF), protože zobrazované tvary znaků jsou závislé na použitém kódování a navíc pro náš výklad není tato část podstatná.

Tabulka A.1. Význam sloupců:

Dec	uvádí kód v desítkové soustavě
Hex	uvádí kód v šestnáctkové soustavě
ASCII	uvádí tvar zobrazovaného znaku nebo zkratku označení řídicího kódu

Tabulka A.2. Důležité zkratky kódů:

7	BELL	bell	zvukový signál
8	BS	backspace	zpět
9	TAB	tabulator	tabulátor
10	LF	line feed	řádkový posun
12	FF	form feed	stránkový posun
13	CR	carriage return	návrat na začátek řádku
26	EOF	end of file	konec souboru
27	ESC	escape	únik - zrušení

LITERATURA

HÁLA, T. *Pascal - učebnice pro střední školy*. Praha: Computer Press, 2002, ISBN 80-7226-733-7.

SATRAPA, P. *Pascal pro zelenáče III.vydání*. Praha: Neocortex, 2001, ISBN 80-86330-03-6.

WRÓBLEWSKI, P. *Algoritmy - Datové struktury a programovací techniky*. Brno: Computer Press, 2004, ISBN 80-251-0343-9.

MCCONNELL, S. *Dokonalý kód - Umění programování a techniky tvorby software*. Brno: Computer Press, 2005, ISBN 80-251-0849-X.

POLÁCH, E. *Programování v jazyku Turbo Pascal (skriptum pro výuku předmětu Programování v jazyku Pascal)*. <http://home.pf.jcu.cz/~edpo/program/program.html> ze dne 1.9.2005 Pedagogická fakulta JU, České Budějovice 1997.

ŠTEFAN, R. *Programování*. Ostrava: Ostravská univerzita v Ostravě, pedagogická fakulta, 2002, ISBN 80-7042-254-8.

BAJGAR, L. *Základy programování. Pascal. Průvodce Turbo Pascalem 5.0*. Brno: Nakladatelství VUT v Brně, březen 1992, ISBN 80-214-0333-0.

KRČEK, B., KREML, P. *Algoritmizace a programování v jazyku Pascal*. Ostrava: VŠB - technická univerzita Ostrava, 1999, II. vydání, ISBN 80-7078-963-8.