PASCAL

pro začátečníky

Jakub Marian

Výuka jazyka od základů

Praktické příklady pro snadné pochopení

Přátelský styl výkladu

Verze z roku 2003

OBSAH

Poznámka k textu z roku 2009

Tento text jsem sepisoval před šesti lety, v době, kdy mi bylo 14 let. Proto prosím omluvte různé poněkud úsměvné (rozumějte hloupé) formulace, jež jsem tou dobou používal, a také častou nedotaženost výkladu (v té době jsem samozřejmě neměl do programování dostatečný vhled na to, abych byl schopen podat nějaký hluboký a ucelený výklad, a zkušenosti s psaním učebnic jsem neměl žádné).

Poděkování a poznámka

Zde bych chtěl poděkovat **Pavlu Tománkovi** za grafickou úpravu tohoto textu, návrh titulní strany a vypracování rejstříku a úpravu obsahu. V této verzi se oproti původní verzi z roku 2003 změnila pouze úprava úvodních stránek a vzhledu všech stránek (původní verze byla totiž určena k tisku), emailový kontakt na konci textu a několik slov v předmluvě.

Předmluva

Když jsem kdysi dávno začínal s programováním, tak mi vždycky štvaly všechny ty stupidní obrázky, které měly začátečníka naučit, jak funguje jakési větvení v programu. Jejich problémem je to, že začátečníkovi stejně nic neřeknou, protože je nechápe, a pokročilejším jsou taky k ničemu, protože jsou moc primitivní :). Dále mi vadil styl výkladu. K čemu začátečníkovi je, že se na začátku knihy dozví, že v Pascalu existují datové typy ty a ty, jaké podmínkové příkazy je možno použít, atd. Člověk, který čte takovou knihu a na prvních dvaceti stránkách nalézá pouze toto, to pravděpodobně vzdá a na programování se vykašle. A právě z tohoto důvodu jsem se rozhodl napsat tuto učebnici, kde se Vás pokusím jednoduše a od základů naučit základy jazyka Pascal.

Úvod

Tato učebnice by Vás měla naučit jazyk Pascal. Pokusím se v ní popsat Pascal od programu Hello world po dynamické proměnné. Samozřejmě, pokud chcete dělat programy v Pascalu, musíte mít kompilátor. Já osobně používám *Borland Pascal 7.0*, ale ten není freeware. Nejlepší je asi *Free Pascal Compiler*, který najdete na *http://www.freepascal.org*. Jeho nevýhodou je velká velikost, takže pokud se připojujete přes běžnou telefonní linku, budete ho asi dost dlouho stahovat.

Ještě jedno upozornění pro nadšence: v této učebnici budu popisovat vývoj pouze konzolových aplikací, takže pokud jste si mysleli, že po přečtení této knihy naprogramujete Quake 4, tak jste na omylu. Naučíte se zde však základy, které jsou nezbytné pro jakékoliv další programování (i v grafice). Později můžete přejít třeba na Delphi, kde můžete snadno programovat aplikace pro Windows. Navíc s komponentami *DelphiX*, sloužícími pro práci s *DirectX*, můžete klidně naprogramovat pěknou hru ve 3D. Ale to už je mimo rámec tohoto textu.

Kapitola 1 – Hello world

Co je to ten hello world? Tak se říká prográmkům, které na obrazovku vypíší nějaký text (většinou hello world :). K výpisu textu slouží v Pascalu příkaz „**write**“. První parametr příkazu je text nebo proměnná textového typu. Ale parametrů je vlastně neomezeně, takže se to prakticky bere jako jeden parametr, který je vlastně pole o N prvcích. Příkaz „write“ se dá použít i pro zápis do souboru, kde první parametr je proměnná typu file a další parametr proměnná stejného typu jako soubor… Ne, ale teď vážně :). Tyhle debilitky, které Vás určitě zatím nezajímají opravdu popisovat nebudu, takže několik předchozích vět rovnou vypusťte z hlavy. Tady je slibovaný hello world:

```
begin
write('Hello world');
end.
```

Když program zkompilujete a spustíte, měl by se vám na obrazovku vypsat text „Hello world“. Pozor! Za každým příkazem (až na pár výjimek) se píše středník. Za posledním „end“ se píše tečka, ale proč to tak je, to nevím ani já sám :). Příkaz „**writeln**“ funguje stejně jako „write“, akorát po vypsání textu přejde na další řádek. Příklad:

```
begin
write('Hello ');
write('world');
end.
```

Vypíše normálně:

Hello world

Ale:

```
begin
writeln('Hello');
writeln('world');
end.
```

Vypíše:

Hello
world

No prostě odřádkování. Doufám že to chápete, protože jestli ne, tak máte opravdu IQ tykve :).

Zde na začátku bych se měl ještě zmínit o **komentářích**. Komentář je text, který napíšete do zdrojového kódu, ale nebere se jako příkaz. Komentáře se uzavírají mezi znaky „(* *)“ nebo „{ }“. Zde je příklad:

```
begin
writeln('Hello '); (* Napíše hello *)
writeln('world');  { napíše world }
end.
```

Prostě je to věc, která tam je, ale bere se to tak, jakoby tam nebyla :).

Kapitola 2 – Proměnné

Co je to vlastně proměnná? Je to informace uložená v paměti počítače, se kterou můžeme dále pracovat. Z důvodu rozlišení způsobu práce s proměnnými existují „**datové typy**“. Zjednodušeně – když budete mít dvě čísla, a napíšete výraz třeba `1 + 2`, tak budete chtít, aby vám vyšlo 3. Ale když budete mít dva textové řetězce a napíšete výraz `'1' + '2'` tak dostanete „12“ a právě také z tohoto důvodu se typy proměnných rozlišují. Všechna jména, co v Pascalu používáte a která nejsou součástí překladače, musíte nějak deklarovat. Deklarace proměnných se provádí příkazem „`var jméno:typ`“. Nejlépe to pochopíte z příkladu:

```
var text: string;
begin
text := 'Ahoj lidi'; (* přiřazujeme hodnotu *)
write(text);
end.
```

Zde se nám objevila podivná věc „**:=**“. To slouží k přiřazení hodnoty do proměnné. Pak se nám tady objevil „`string`“. Je to jeden z datových typů – textový. Text se vždy uzavírá do apostrofů! Přiřazení hodnoty znamená, že se obsah proměnné přepíše zadanou hodnotou. Tzn.:

```
var text: string;
begin
text := 'Ahoj lidi';
text := 'Co děláte?';
write(text);
end.
```

Vypíše pouze text: **Co děláte?**

Se všemi proměnnými stejného typu se pracuje stejně. Tzn. že třeba `Integer` a `Byte` (popíšu později) se od sebe liší jen rozsahem (tzn. i tím, kolik místa zabírají v paměti), ale operace s nimi jsou stejné. Ale třeba těžko by se vypočítalo toto:

```
'Ahoj'*3+2*('Lidi'+1)
```

To je jednoduše řečeno úplná hovadina.

Jen ještě jedna poznámka – pokud chcete číslo zapsat šestnáctkově, použijete znak dolaru (např. `$2FA`).

Kapitola 3 – Operátory

Stejně jako v normální matematice, jsou i v Pascalu aritmetické funkce. Používáme tyto operátory:

Operátor	Význam
+	Sčítání
-	Odčítáni
*	Násobení
/	Dělení
div	Celočíselné dělení
mod	Zbytek po celočíselném dělení

Příklad použití:

```
begin
writeln(55+3*7);
writeln((55+3)*7);
end.
```

Toto vypíše nejdříve 76 a potom 406. Z toho je jasně vidět, že operátory mají v Pascalu stejnou prioritu jako v normální matematice. Tato priorita se dá obejít závorkami.

Příkaz „`readln`“ načte hodnotu zadanou uživatelem. A nyní už můžeme vytvořit první užitečný program :).

```
var cislo: integer;
begin
write('Zadejte cislo: ');
readln(cislo);
write('Cislo x 2 je: ', cislo*2);
end.
```

Tím jsme se dostali k dalšímu datovému typu – „`Integer`“. Je to číselný datový typ, který může obsahovat čísla -32768 až 32767. Pokud hodnota překročí limit, bere se to, jako by to šlo odzadu – když se do proměnné tohoto typu pokusíte přiřadit třeba 33000 tak to hodí nějaké záporné číslo. S tímto typem souvisí i typ „`Word`“, který má rozsah 0 až 65535 Pokud chcete pracovat s většími čísly, je k dispozici typ „`Longint`“ s rozsahem -2147483648 až 2147483647 (s tím už se dá něco dělat). Další věc, kterou ukazuje tento prográmek je, že parametr příkazu může být i výraz (tím myslím to „`cislo*2`“). Stejně tak je to i u přiřazení hodnoty do proměnné.

```
var cislo: integer;
begin
cislo := 50*8/5+7;
end.
```

Když se vrátíme k příkladu z minulé kapitoly, tak z toho vyplývá způsob, jak proměnnou nepřepsat, ale přidat k ní hodnotu:

```
var text: string;
begin
```

```
text := ‘Ahoj lidi’;
text := text + ‘Co děláte?’;
write(text);
end.
```

U textových řetězců lze logicky použít pouze operátor „+“ ke spojení řetězců. Já osobně si teda nedovedu představit, jak by se mohly dva řetězce odečíst.

Kapitola 4 – Podmínkové příkazy a operátory

Co je to podmínka? To je snad jasné. Prostě pokud něco platí, něco dalšího se vykoná. Vše předvedu na jednoduchém příkladu:

```
var cislo: integer;
begin
cislo := 5;
if cislo = 5 then write('Ahoj');
end.
```

Do proměnné číslo se přiřadí hodnota 5 a pokud je hodnota proměnné „`cislo`“ 5, tak se vypíše „**Ahoj**“. Tu podmínku zajišťuje klíčové slovo „`if`“. Takže kdyby tam bylo „`cislo := 6`“, tak by se nic nevypsalo. Úplně stejně lze porovnávat textové řetězce:

```
var txt: string;
begin
readln(txt);
if txt = 'ahoj' then write('Cau');
end.
```

Pokud chceme, aby se vykonalo více příkazů, nemusíme tam psát podmínku „`if`“ několikrát, ale použijeme další „`begin`“ a „`end`“:

```
var txt: string;
begin
readln(txt);
if txt = 'ahoj' then begin (* když uživatel zadal „ahoj“ *)
 write('Cau');
 write(':-)');
 end;
end.
```

Podmínkové operátory jsou:

Operátor	Význam
=	Rovná se
<>	Není rovno
>=	Je větší nebo rovno
<=	Je menší nebo rovno
<	Je menší
>	Je větší

Teď je vhodná chvíle, abych se zmínil o spojování více podmínek pomocí dalších operátorů. Jsou to operátory „**and**“, „**or**“, „**xor**“ a „**not**“.

Použití operátorů:

Operátor	Význam
And	pokud platí obě podmínky
Or	pokud platí alespoň jedna podmínka
Xor	pokud platí pouze jedna podmínka
Not	pokud podmínka neplatí

Příklad:

```
var cislo: integer;
begin
cislo := 5;
if (cislo >= 2) and (cislo < 4) then write('Ahoj');
if not (cislo = 1) then write('Cislo neni jedna');
end.
```

Jak vidíme, ahoj se nevypíše, protože podmínka splněna není. Pozor! Podmínky je nutné uzavírat do závorek, jinak by to bylo bráno jako bitová operace s čísly, ale tím se teď zabývat nebudu. Operátor `not` znamená negaci, tzn. že zápis „`not (cislo = 1)`“ znamená to samé jako „`cislo <> 1`“. Podmínky lze utvářet i složitější a to pomocí závorek:

```
var cislo: integer;
begin
cislo := 5;
if ((cislo > 2) and (cislo < 7)) or (cislo = 10)  then write('Ahoj');
end.
```

Ahoj se tedy vypíše, pokud je „`cislo`“ větší než 2 a menší než 7, nebo je 10.

V souvislosti s „`if`“ se musím zmínit ještě o příkazu „**`else`**“. Všechno co je za příkazem „`else`“ se vykoná, pokud předcházející podmínka nebyla splněna:

```
var cislo: integer;
begin
cislo := 4;
if cislo = 5 then write('Cislo je pet')
else write('cislo je nejake jine');
end.
```

Pozor! U posledního příkazu před `else` se nepíše středník. Pokud se má za else vykonat více příkazů, je samozřejmě nutné je uvést mezi „begin“ a „end“.

Pascal má ještě jeden typ podmínky, a to „**`case`**“. `Case` se používá místo více podmínek. Použití je následující:

```
var cislo: integer;
begin
cislo := 5;
case cislo of (* zjistíme hodnotu *)
```

```
1: write('cislo je jedna');
2: write('cislo je dva');
5: write('cislo je tri');
7: begin
     write('cislo je jedna');
     cislo := 15;
   end;
else write('cislo je jine');
end;
end.
```

Tedy „**case** *proměnná* **of**“ určuje proměnnou, ze které se má zjišťovat hodnota, následuje výčet hodnot. Za každou hodnotou je uvedena dvojtečka a příkaz, který se má vykonat. Pokud chceme vykonat více příkazů, uzavřeme je mezi „**begin**“ a „**end**“. Pokud žádná hodnota neplatí, vykoná se příkaz za „**else**“. Výčet hodnot končí slovem „**end**“. Příkaz **case** je bohužel možné použít pouze na číselné proměnné.

V souvislosti s podmínkami bych měl zmínit datový typ „**boolean**“, který nabývá hodnot „**true**“ (pravda) a „**false**“ (nepravda). Tento typ může být uveden místo podmínky. Příklad:

```
var plati: boolean;
begin
plati := true;
if plati then write('Podminka plati');
end.
```

Hodnota přiřazená do tohoto typu může být i výsledkem výrazu vracejícího logickou hodnotu:

```
var plati: boolean;
begin
plati := 5 > 3;
if plati then write('Podminka plati');
end.
```

Kapitola 5 – Programové jednotky

Standardní Pascal neumožňuje používat jiné příkazy, než ty vestavěné v překladači. A právě proto přišel *Borland* s programovými jednotkami, neboli „**Unit**“. Tyto unity jsou vlastně jakési „předkompilované“ programy v Pascalu, ze kterých je možné používat procedury, funkce a proměnné (viz kapitola 8). Pro vás je zatím jen důležité vědět, že použitím unity si prostě přidáte možnost používat další příkazy. Použití unity oznámíte překladači klíčovým slovem „**uses**“ – příklad:

```
uses crt;
begin
write('Ahoj lidi');
readkey;
end.
```

„**crt**“ je asi nepoužívanější unita, protože obsahuje většinu důležitých funkcí pro práci se hardwarem. Vzhledem k tomu, že jsme se ještě nezabývali funkcemi, nemohu zde přesně říci, co „**readkey**“ znamená. Mohu jen vysvětlit, že funguje jako jakási pauza a pokud napíšete „**proměnná := readkey**“, kde proměnná musí být typu „**char**“ (znak), tak do ní uloží stisknutý znak, se kterým je možné dále pracovat:

```
uses crt;
var znak: char;
begin
znak := readkey;
case znak of     (* zjistíme, jaký znak byl vlastně stisknut *)
'1': writeln('Stiskl jste jednicku');
'2': writeln('Stiskl jste dvojku');
'i': writeln('Stiskl jste i');
else writeln('Stiskl jste neco jineho');
end.
```

Jen ještě jedna poznámka k unitě „crt“ – normální „crt“ v *Borland Pascalu* způsobuje „runtime error 200“ (dělení nulou), řešení spočívá v unitě „**fdelay**“, která se dá najít na internetu. Stačí pak napsat pouze:

```
uses fdelay, crt;
var znak: char;
begin
write('Ahoj lidi');
end.
```

A vše je v pořádku.

Popisem jednotky „crt“ se budu zabývat později.

Kapitola 6 – Cykly

Co je to cyklus? Cyklus je něco pořád dokola se opakujícího (lat. cyklus = koloběh (nebo tak něco :)). Takže cyklus v Pascalu je nějaká část kódu, která se bude vykonávat pořád dokola.

Pascal obsahuje 3 druhy cyklů:

```
for prom := ? to ? do něco
while podmínka do něco
repeat until podmínka
```

Cyklus FOR

Jak vidíte, zápis cyklu `for` vypadá trochu zmateně :). Předvedu na příkladu:

```
var i: integer;
begin
for i := 1 to 10 do writeln(i);
end.
```

Pokud tento prográmek spustíte, vypíše se vám:

```
1
2
3
...
10
```

Cyklus for funguje tak, že přiřadí do proměnné hodnotu, vykoná příkaz který je za „**do**“, zvedne hodnotu proměnné o 1 a znovu vykoná příkaz za do. Takto se to pořád opakuje, dokud je hodnota proměnné (v našem případě „`i`“) menší než hodnota uvedená za „**to**“. Pokud chcete, aby šel cyklus „shora dolů“, použijte místo „`to`“ „**downto**“. Příklad:

```
var i: integer;
begin
for i := 10 downto 1 do writeln(i);
end.
```

Vypíše:

```
10
9
8
...
1
```

Cyklus WHILE

Cyklus `while`, funguje jako normální podmínka „`if`“, s tím rozdílem, že místo „`then`“ se uvádí „**do**“ a příkaz za „`do`“ se provádí pořád opakovaně, dokud podmínka platí. Příklad:

```
var i: integer;
begin
i := 1;
while i < 50 do begin (* dokud je „i“ menší než 50 *)
writeln(i);
i := i + 5;
 end;
end.
```

Chápete? Doufám že jo :).

Cyklus REPEAT UNTIL

Cyklus „**repeat until** podmínka“ funguje na stejném principu, jako cyklus `while` s tím rozdílem, že se opakuje do té doby, dokud podmínka uvedená za `until` NEPLATÍ:

```
var i: integer;
begin
i := 1;
repeat
writeln(i);
i := i + 5;
until i > 50; (* dokud „i“ není větší než 50 *)
end.
```

Teď si možná říkáte, k čemu existují 2 cykly, se stejnou funkcí, lišící se pouze zadáním podmínky? Rozdíl je v tom, že cyklus „`repeat`“ se vždy vykoná alespoň jednou, kdežto cyklus „`while`“ se vykoná pouze pokud na začátku podmínka platí:

```
var i: integer;
begin
i := 1;
repeat
writeln('Until probehlo');
until i = 1;
while i <> 1 do
writeln('While probehlo');
end.
```

Tento program vypíše pouze „**Until probehlo**“.

V souvislosti s cykly se musím ještě zmínit o příkazu „`break`“, který běh cyklu ukončí. Př.:

```
var i: integer;
begin
for i := 1 to 10 do begin
 if i = 6 then break; (* když je i 6, cyklus se ukončí *)
 writeln(i);
 end;
end.
```

Kapitola 7 – Strukturované datové typy

Pole

Co si představíte pod pojmem pole? Hromadu hlíny s obilím? Podobně je to i v Pascalu. Pole je datový typ, který obsahuje hromadu proměnných. Příklad:

```
var pole: array[1..3] of integer;
begin
pole[1] := 50; (* první položka má hodnotu 50 *)
pole[2] := 10;
pole[3] := 20;
writeln(pole[1]);
writeln(pole[2]+pole[3]);
end.
```

Pole se deklaruje klíčovým slovem „**array**“. Za ním je uveden rozsah uzavřený do hranatých závorek. Následuje klíčové slovo „**of**“ a typ proměnné. Když chceme do proměnné v poli přiřadit hodnotu, tak se zase index proměnné píše do hranatých závorek. Doufám, že je to jasné.

Nyní si předvedeme, jak je možné vytvořit vícerozměrné pole:

```
var pole: array[1..3, 1..3] of integer;
begin
pole[1,1] := 50;
pole[1,3] := 10;
pole[3,2] := 20;
end.
```

Z tohoto příkladu by vám mělo být jasné, jak se vícerozměrné pole používá. Je to vlastně jako kdybychom si vytvořili devět různých proměnných.

Teď když jsme se dostali k poli, musím ještě zmínit, že s datovým typem „**string**“ je možné pracovat jako s polem znaků, a to jak při deklaraci, tak při čtení nebo zápisu:

```
var retezec: string[50];
```

Vytvoří řetězec o délce maximálně 50 znaků. V programu následně použijeme:

```
writeln(retezec[12]);
```

Což vypíše 12. znak v řetězci. Toto čtení určitého znaku samozřejmě můžeme použít i když si řetězec nedeklarujeme s přesnou délkou, protože když napíšeme:

```
var retezec: string;
```

tak to má stejný význam jako:

```
var retezec: string[255];
```

Záznam

Co to je a k čemu slouží záznam? Slouží k uložení více typů informací do jednoho datového typu. Např. když budete třeba dělat něco jako databázi, tak budete chtít uložit jméno, příjmení atd. To byste udělali asi takhle:

```
var jmeno, prijm, telefon, adresa: string;
    vek: integer;
```

No jo, ale co když budete chtít uložit 10 záznamů? To by nebylo snadné. A proto je tu typ „**record**“. Teď se ještě musím zmínit o jednom klíčovém slově – „**type**“, pomocí nějž si můžete vytvořit vlastní datový typ:

```
type mujtyp = integer;
var vek: mujtyp;
```

Tenhle zápis je celkem k ničemu, protože jsme si vlastně vytvořili „zástupné jméno“ pro „integer“, ale pro typ record je potřeba:

```
type zaznam = record
              jmeno, prijm, telefon, adresa: string;
              vek: integer;
              end;
var prom: zaznam;
begin
prom.jmeno := 'petr';
               (* prom.něco funguje jako samostatná proměnná *)
prom.vek := 20;
end.
```

Jak vidíte, k jednotlivým položkám se přistupuje pomocí tečky. Tento zápis pro nás nijak výhodný není, protože je vlastně delší než ten původní. Vraťme se ale k problému s 10 záznamy. Jak na to? Jednoduše – pomocí pole.

```
type zaznam = record
              jmeno, prijm, telefon, adresa: string;
              vek: integer;
              end;
var pole: array[1..10] of zaznam;
begin
pole[1].jmeno := 'petr';
pole[5].vek := 20;
end.
```

V souvislosti s tímto je nutné se zmínit o příkazu „**with**“. Příklad:

```
type zaznam = record
              jmeno, prijm, telefon, adresa: string;
              vek: integer;
              end;
var prom: zaznam;
```

```
begin
prom.jmeno := ‘petr’;
prom.prijm := ‘novák’;
prom.telefon := ‘0609112567’;
prom.adresa := ‘U salónu 25’;
prom.vek := 20;
end.
```

Tento zápis je trošku zdlouhavý. Neleze vám tam na nervy to věčně se opakující „prom“? A to lze řešit právě příkazem „with“:

```
type zaznam = record
              jmeno, prijm, telefon, adresa: string;
              vek: integer;
              end;
var prom: zaznam;
begin
with prom do begin
 jmeno := ‘petr’;
 prijm := ‘novák’;
 telefon := ‘0609112567’;
 adresa := ‘U salónu 25’;
 vek := 20;
 end;
end.
```

To už je lepší ne?

Množina

Co je to množina si asi dovede představit každý, kdo prošel první třídou základní školy. Je to soubor prvků stejného typu (vzpomínáte na jabka a hrušky? :). Množina se vytváří jako datový typ pomocí příkazu „**set**“. Nebudu se tady zabývat principy množiny, ale předvedu ji na nejčastějších použitích. Pozor! Množina musí být číselného typu. Použití:

1. **Vytvořit si vlastní typ použitý místo číselného zápisu:**

```
type ramecek = (jednoduchy, dvojity, trojity, levy, pravy);
var typ: ramecek;

begin
typ := dvojity;
case typ of
 jednoduchy: writeln(‘Ramecek ma jednu caru’);
 dvojity: writeln(‘Ramecek ma dve cary’);
 else writeln(‘Ramecek je nejakej jinej’);
end;
end.
```

Jak vidíte, vytvořili jsme si typ „ramecek“, který tvoří množinu čísel. Tato čísla jsou nahrazena zástupnými jmény, která můžeme v programu použít – hodí se ke zlepšení přehlednosti.

2. Množina znaků nebo čísel

```
uses crt; (* readkey je definováno v unitě crt, pokud to *)
          (* způsobuje error 200, použijte „fdelay“ *)
const znaky  = [‘a’,’n’,’1’,’2’,’3’];
begin
repeat
until readkey in znaky;
end.
```

Zde jsme se setkali s novým operátorem – „in“. Význam: „hodnota in množina“ vrací „true“, když se hodnota v množině nachází a „false“ když ne. Tzn. že tento prográmek bude čekat na stisk jedné z kláves uvedených v množině. Jistě jste si všiml, že se v tomto případě přímého zápisu hodnot (ne zástupných jmen) uzavírá množina do hranatých závorek. Stejně tak to může být i množina čísel:

```
[255, 85, 41 ,56]
```

Kombinovat číselné a znakové množiny není možné

Teď si možná říkáte, proč jsem napsal, že se množina deklaruje klíčovým slovem „set“. Je to z toho důvodu, že jak vidíte, do množin, které jsem zde uvedl není možné zadat hodnotu. V příkladu 2 byla množina konstanta, a proto se neuvádí datový typ. Ve skutečnosti je to však „set of typ“:

```
var mnoz: set of char;
begin
mnoz := [‘A’, ‘B’, ‘C’];
end.
```

S proměnnou pracujeme stejně jako s konstantou z příkladu 2 s tím rozdílem, že do ní můžeme přiřadit hodnotu.

Kapitola 8 – Procedury a funkce

Procedury

Pojem procedura znamená část kódu, který tvoří jakýsi podprogram. Zatím jsme zadávali příkazy pouze mezi „begin“ a „end.“. Ale když budete chtít nějakou část programu vykonat několikrát, nemusíte ji několikrát psát, ale vložíte ji do procedury. Vše objasní příklad:

```
procedure ahoj;
begin
writeln('Ahoj lidi');
writeln('Jak se mate?');
end;

begin
ahoj;
ahoj;
ahoj;
end.
```

Tento program vypíše 3x text uvedený v proceduře „ahoj“. Procedura se tedy deklaruje klíčovým slovem „**procedure**“, následuje jméno procedury a parametry. Parametry jsou hodnoty předané proceduře:

```
procedure ahoj(kdo: string);
begin
writeln('Ahoj '+kdo);
end;

begin
ahoj('Petře'); (* vypíše „Ahoj Petře“ *)
ahoj('Jano');
ahoj('Honzo');
end.
```

Program vypíše 3x Ahoj + zadaný text. Deklarace parametrů se píše za jméno do závorek, jako deklarace proměnných a také uvedený parametr používáme jako proměnnou. Pokud chcete použít více parametrů, oddělují se čárkami:
`procedure ahoj(kdo,pozdrav: string)` a pokud chcete použít více parametrů různého typu, oddělují se středníkem:
`procedure ahoj(kdo: string; cislo: integer)`.

V proceduře je také možné deklarovat **lokální** proměnné. Lokální proměnná je proměnná použitelná pouze v proceduře, kde je deklarována a nikde jinde. Naproti tomu **globální** proměnnou je možné použít kdekoliv – jak v hlavním programu, tak ve všech procedurách:

```
var text: string;

procedure ahoj;
begin
```

```
writeln(text); (* vypíše proměnnou text *)
end;

begin
text := 'ahoj lidi';
ahoj;
writeln(text);
end.
```

Lokální proměnnou lze použít pouze v místě, kde je nadeklarována:

```
procedure ahoj;
var text: string;
begin
writeln(text); (* vypíše proměnnou text *)
end;

begin
text := 'ahoj lidi'; (* zde překladač zahlásí chybu *)
ahoj;     (* proměnná text pro tuto část programu neexistuje *)
end.
```

Funkce

Teď už snad chápete, jak procedury fungují. Nyní se podívejme na funkce. Funkce má stejnou stavbu jako procedura, akorát ještě vrátí hodnotu způsobem „`jménofunkce := hodnota`“. Pozor! Funkce může být pouze základních typů Pascalu a ne vlastních strukturovaných. Deklaruje se „`function jmeno(parametry): typ`“. Příklad:

```
function dvakrat(co: integer): integer;
begin
dvakrat := co*2;
end;

begin
writeln(dvakrat(5));
end.
```

Tato funkce moc užitečná není, ale dají se samozřejmě napsat i užitečnější. Příklad: zkuste vymyslet funkci, která by počítala stranu „c“ pravoúhlého trojúhelníka a jako parametry by dostala strany „a“ a „b“. Napovím: funkce pro odmocninu je „**`sqrt(číslo)`**“ a datový typ pro desetinná čísla je „**`real`**“. Nevíte pořád jak na to? Chce to vzpomenout si na známou rovnici „$\mathbf{a^2+b^2=c^2}$“ ze které snadno zjistíte stranu „c“: je to odmocnina z „$\mathbf{a^2+b^2}$“. Pořád nic? No dobře, tady je řešení:

```
function stranac(a,b: real): real;
begin
stranac := sqrt(a*a+b*b);
end;
```

```
begin
writeln('Strana c (a=3,b=4): ',stranac(3,4));
end.
```

Zde bych ještě měl zmínit příkaz **exit**, který vyskočí z procedury nebo funkce. To je vhodné především pro ošetřování chyb. Příklad:

```
function vydel(co,cim: real): real;
begin
 if cim = 0 then begin     (* pokud se dělitel = 0 tak se z funkce vyskočí *)
 writeln('Nelze delit nulou');     (* a vypíše se text „Nelze delit nulou“ *)
 vydel := 0;       (* funkce vydel vrátí hodnotu 0, aby bylo jasné že neproběhla *)
 exit;             (* vyskočení *)
end;
vydel := co / cim; (* pokud se dělitel není nula, provede se dělení *)
end;

begin
writeln(vydel(50,8));
writeln(vydel(88.2,0)); (* zkusíme dělit nulou *)
end.
```

Kdyby to nebylo ošetřeno vyskočením z funkce, aplikace by hodila „runtime error 200: division by zero“. Tím se dostávám k dalšímu problému. Jak se chránit před runtime errory pokud nemůžete nijak zajistit ošetření chyby před jejím provedením? K tomu složí **direktiva překladače** „**{$I+}**“ a „**{$I-}**“. Teď se asi ptáte, co je to direktiva překladače. Direktiva slouží ke sdělení zprávy o něčem překladači a není to příkaz – proto se píše do komentáře (musí to být komentář typu „**{ }**“). Popis jednotlivých direktiv najdete v nápovědě k překladači.

Direktiva $I slouží ke změně ošetření vstupních a výstupních chyb. Pokud použijete direktivu „**{$I-}**“, můžete pomocí funkce „**ioresult**“ zjistit zda nastala při poslední operaci chyba. Funkce „ioresult“ vrátí číslo chyby, a pokud chyba nenastala, vrátí hodnotu 0:

```
var f: file;

begin
assign(f,'neexistujici');
{$I-}
reset(f);
{$I+}
if ioresult <> 0 then writeln('soubor se nepodarilo otevrit');
end.
```

Zde uvedené funkce slouží k práci se soubory a budou popsány v příští kapitole. Program se pokusí otevřít neexistující soubor. Nemohl jsem zde použít minulý příklad, protože „**{$I-}**“ nezachytí chybu při dělení nulou, takže tu opravdu musíte ošetřit podmínkou.

Odkaz na proměnnou

No jo, ale jak to udělat, aby nám funkce vrátila hodnotu jinou, než normálního Pascalového typu? Použijeme odkaz na proměnnou. V deklaraci procedury použijeme slovíčko „**var**“:

```
procedure ahoj(var x: mujtyp);
```

Tím jsme řekli překladači, že „x“ není proměnná jako parametr, ale je to odkaz na proměnnou uvedenou ve volání procedury. Zjednodušeně řečeno to znamená, že když přiřadíme hodnotu do „x“ tak změníme obsah proměnné uvedené v parametru zavolání funkce. Vím, zní to sice trochu složitě, ale z příkladu je to hned jasné:

```
var prom: string;

procedure zmen(var txt: string);
begin
txt := 'Zmeneny text'; (* ve skutečnosti měníme hodnotu proměnné „prom“ *)
end;

begin
prom := 'ahoj';
zmen(prom);          (* Jako „var“ parametr musí být vždy uvedena proměnná *)
writeln(prom);       (* Vypíše „Zmeneny text“ *)
end.
```

Stejným způsobem je to možné použít i u strukturovaných datových typů.

Kapitola 9 – Práce se soubory

Pro práci se soubory slouží v Pascalu datový typ „**file**“ a pokud chceme pracovat s textovým souborem, využijeme typ „**text**“. Je sice trochu zvláštní, že se souborem pracujeme jako s proměnnou, ale je to jednodušší. Nejdříve je nutné si deklarovat proměnnou typu `file`. Ale pozor! `File` je taky určitého typu (jako pole) a (stejně jako u pole) se tento typ definuje slovíčkem „**of**“:

```
var f: file of byte;
```

Soubor může být jakéhokoliv typu (vyjma typu `file` – „`var f: file of file`“ je blbost), tzn. i vlastního typu, který si nadefinujeme pomocí příkazu „**type**“, což se hodí zejména pro uložení nějakých dat s více položkami. Pro práci se soubory slouží tyto procedury (funkce):

Funkce	Význam
assign(f: file, jmeno)	Přiřadí proměnné typu file jméno souboru
reset(f: file)	Otevře soubor
rewrite(f: file)	Vytvoří nový soubor (nebo přepíše starý se stejným jménem)
append(f: text)	Otevře textový soubor a nastaví pozici na konec souboru
close(f: file)	Ukončí práci se souborem
Filesize(f: file)	Vrátí počet dat stejného typu jako soubor
seek(f: file, pozice)	Přejde na pozici v souboru
filepos(f: file)	Vrátí pozici v souboru
rename(f: file, jmeno)	Přejmenuje soubor na zadané jméno
erase(f: file)	Smaže soubor
eof(f: text): boolean	Logická hodnota – je pozice na konci souboru?
eoln(f: text): boolean	Logická hodnota – je pozice na konci řádku?
readln(f: text, prom)	Načte řádek do textové proměnné
read(f: file, prom)	Načte hodnotu do proměnné stejného typu jako soubor
writeln(f: text, text)	Zapíše text do textového souboru
write(f: file, prom)	Zapíše hodnotu stejného typu jako soubor

Nelekejte se, není to nic těžkého. Pro začátek si ukážeme, jak pracovat s textovými soubory:

```
var f: text;
    pom: string;

begin
assign(f, 'muj.txt');       (* přiřadíme jméno *)
reset(f);                   (* otevřeme soubor *)
while not  eof(f) do begin (* dokud nedojdeme na poslední řádek *)
 readln(f, pom);            (* načteme řádek do proměnné *)
 writeln(pom);              (* vypíšeme řádek na obrazovku *)
 end;
close(f);                   (* ukončíme práci se souborem *)
end.
```

Jak vidíte, je zde použita funkce „**readln**“, kterou už jsme používali pro vstup z klávesnice. Pokud je v této funkci uveden jako první parametr soubor, načte se text ze souboru. Stejně tak je to i s funkcemi „**read**“, „**write**“ a „**writeln**“. Nyní si ukážeme, jak do textového souboru zapisovat:

```
var f: text;
    pom: string;

begin
assign(f, 'muj.txt');  (* přiřadíme jméno *)
rewrite(f);            (* vytvoříme soubor - nemusíme ho otvírat pomocí *)
                       (*reset, protože funkce rewrite ho rovnou i otevře *)
repeat
 readln(pom);          (* uživatel zadá text *)
 writeln(f, pom);      (* zapíšeme text do souboru *)
until pom = '';        (* dokud nebude zadáno nic *)
close(f);              (* ukončíme práci se souborem *)
end.
```

Dalo by se říct, že jste právě vytvořil svůj první textový editor :).

Jen ještě jedna poznámka – ke každému příkazu „`reset`“ je vhodné umístit i vlastní ošetření chyb. Jednoduše uděláte:

```
{$I-}
reset(f);
{$I+}
if ioresult <> 0 then begin
 writeln('Soubor se nepodarilo otevrit');
 halt;
end;
```

Příkaz „`halt`“ ukončí běh programu. To je zde nutné z toho důvodu, že kdyby se soubor otevřít nepodařilo, proběhly by nějaké příkazy třeba pro čtení dat ze souboru a to by také způsobilo chybu.

S binárními (prostě jinými než textovými) soubory se pracuje trochu jinak. U každého souboru existuje jakási virtuální pozice, která určuje kde se právě v souboru nacházíte. Neurčuje však byte, ale pozici v souboru závislou na datovém typu. Příklad:

```
type zaznam = record     (* vytvoříme si vlastní typ záznamu *)
              jmeno, prijm: string[20];
              vek: byte;
              end;

var f: file of zaznam;   (* soubor *)
    pom: zaznam;         (* pomocná proměnná stejného typu jako soubor *)
    pos: integer;        (* pozice v souboru *)

begin
assign(f, 'data.dat');   (* přiřadíme jméno *)
rewrite(f);              (* vytvoříme nový soubor *)
pos := 0;                (* pozici nastavíme na 0 *)
repeat                           (* cyklus se opakuje do nekonečna *)
 writeln('Jmeno: '); readln(pom.jmeno); (* uživatel zadá data *)
 if pom.jmeno = '' then break;   (* pokud je jméno prázdné, cyklus se ukončí *)
```

```
 writeln('Prijmeni: '); readln(pom.prijm);
 writeln('Vek: '); readln(pom.vek);
 seek(f,pos);                    (* přejdeme na zadanou pozici v souboru *)
 write(f,pom);                   (* zapíšeme data do souboru *)
 pos := pos + 1;                 (* pozici zvedneme o 1 *)
until false;                     (* opakování cyklu *)
close(f);                        (* ukončíme práci se souborem *)
end.
```

Tak co je tady nového? Přibyla nám procedura „**seek**“, která posune pozici v souboru na zadanou hodnotu. My chceme, aby se další záznamy pořád přidávaly, a proto hodnotu proměnné „pos“ zvýšíme pokaždé o 1. Procedura „**write**“ zapíše data na zadanou pozici v souboru. Na konci práce soubor zavřeme procedurou „**close**“, což je velice důležité, jinak se zadaná data neuloží!!!

Teď bychom měli udělat nějaký prohlížeč pro tu naši „databázi“:

```
type zaznam = record     (* vytvoříme si vlastní typ záznamu *)
              jmeno, prijm: string[20];
              vek: byte;
              end;

var f: file of zaznam;   (* soubor *)
      pom: zaznam;       (* pomocná proměnná stejného typu jako soubor *)
      pos: integer;      (* pozice v souboru *)

begin
assign(f, 'data.dat');           (* přiřadíme jméno *)
reset(f);                        (* otevřeme náš soubor *)
pos := 0;                        (* pozici nastavíme na 0 *)
while pos < filesize(f) do begin (* dokud nejsme na konci souboru *)
 seek(f,pos);
 read(f,pom);                    (* načteme data *)
 writeln('Jmeno: '+pom.jmeno+', Prijmeni: '+pom.prijm+', Vek: ',pom.vek);
                                 (* vypíšeme *)
 pos := pos + 1;
end;
close(f);                        (* ukončíme práci se souborem *)
end.
```

Funkce „**read**“ načte data z pozice do proměnné.

Teď už máme systém databáze, jen by měla být trochu víc „user friendly“, ale to už nechám na vás. Následující kapitola se už věnuje unitě CRT, která slouží pro základní komunikaci s hardwarem a softwarem a díky níž už budete mít nástroj např. pro naprogramování procedury na vykreslení rámečku apod.

Kapitola 10 – Unita CRT

Unita „**CRT**“ slouží pro základní komunikaci s hardwarem a operačním systémem. Tím samozřejmě nemyslím, že byste pomocí unity CRT mohl naprogramovat třeba připojení scanneru přes USB, ale složí pro základní práci v textovém režimu *DOSu*. Zde je malý příklad:

```
uses (* fdelay, *) crt;
begin
textbackground(black);                          (* změna barvy pozadí *)
clrscr;                                         (* smaže obrazovku *)
textcolor(14); writeln('Text vypsany zlute'); (* změna barvy textu *)
textcolor(15); writeln('Text vypsany bile');
textcolor(blue); writeln('Text vypsany modre');
end.
```

Jak vidíte, není zde moc co vysvětlovat, názvy procedur mluví samy za sebe. Ale jak jste si jistě všimli, někdy jsem použil jméno barvy a někdy číslo. Tato jména jsou konstanty nadefinované v unitě „CRT“. Jejich popis najdete v nápovědě k překladači. Já osobně používám číselný zápis, protože to není až tak těžké na zapamatování – barev je pouze 16 (počítají se od 0 do 15) a pro barvu pozadí je možné použít pouze prvních 8. Další důležitá procedura je „`gotoXY(x,y)`“:

```
uses crt;
begin
gotoXY(20,5);        (* přejde na 20 znak na pátém řádku *)
write('Ahoj lidi'); (* vypíše zde text *)
readkey;
end.
```

Jak vidíte, procedura „`gotoXY`“ přejde na zadaný sloupec a řádek. Teď už byste mohl vymyslet proceduru vykreslující třeba rámeček z písmen „o“ (pro zjednodušení):

```
uses (* fdelay, *) crt;

procedure ramecek(x,y,x2,y2: byte);
var i: integer;
begin
for i := x to x2 do begin
 gotoXY(i,y); write('o');  (* horní strana *)
 gotoXY(i,y2); write('o'); (* dolní strana *)
end;
for i := y to y2 do begin
 gotoXY(x,i); write('o');  (* levá strana *)
 gotoXY(x2,i); write('o'); (* pravá strana *)
end;
end;

begin
clrscr;
ramecek(5,5,50,15);         (* vykreslíme rámeček *)
readkey;
end.
```

Samozřejmě pokud chcete, aby tento rámeček vypadal trochu k světu, chce to použít jiné znaky a tady může být trochu problém. Jak zobrazit znak, který není možné zapsat pomocí klávesnice, ale je zobrazitelný? K tomu slouží v Pascalu „**#**“. Když napíšete „**#**“ a číslo znaku, tak se to bere jako znak. Např.:

```
begin
write(#48); (* vypíše ascii znak 48 – „0“ *)
write(#82); (* vypíše ascii znak 82 – „R“ *)
end.
```

Když budete chtít použít znak s číslem uloženým v proměnné, použijete funkci „**chr**(cislo)“:

```
var i: integer;
begin
i := 82;
write(chr(i));    (* vypíše znak *)
write(chr(82));   (* tento zápis je také možný, ale pokud znáte předem *)
                  (* hodnotu znaku, tak je lepší použít zápis s „#“ *)
end.
```

A pokud budete naopak chtít zjistit číslo znaku, použijete funkci „**ord**(znak)“.:

```
begin
writeln(ord(readkey)); (* vypíše číslo stisknuté klávesy *)
end.
```

Ale to už jsem trochu odbočil od unity CRT. S pozicí kurzoru souvisí ještě funkce „**wherex**“ a „**wherey**“. Tyto funkce vrací sloupec (řádek) ve kterém se nachází kurzor. Pak jsou zde ještě dvě zajímavé procedury pro mazání textu a to „**ClrEol**“, která smaže všechny znaky od kurzoru do konce řádku, a „**DelLine**“, která smaže vše na řádku, na kterém se nachází kurzor.
Další důležitá procedura je „**textmode**“, která změní textový mód obrazovky. Její parametr se dá zapsat buď číselně, nebo pomocí konstant s celkem výstižnými jmény, všechny možné kombinace naleznete v tabulce:

Konstanta	Význam
CO80	80 sloupců 25 řádků
CO40	40 sloupců 25 řádků
CO80 + Font8x8	80 sloupců 49 řádků
CO40 + Font8x8	40 sloupců 49 řádků

Daný mód se tedy vyvolá takto:

```
uses (* fdelay, *) crt;
begin
textmode(CO80+Font8x8);
write('Maly font – pro zkousku');
readkey;
end.
```

Pokud budete chtít dělat textové hry (něco ve stylu střílení po hvězdičkách), je nejlepší používat mód „CO80 + Font8x8“, protože v něm jsou znaky přibližně stejně vysoké, jako široké. Dříve byla velice používanou procedurou procedura „**delay**“. Používala se takto:

```
uses (* fdelay, *) crt;
begin
writeln('Text');
delay(500); (* počká 500 milisekund *)
writeln('Text 2');
end.
```

Problém spočívá v tom, že na moderních rychlých počítačích běhá velice zrychleně, takže je nepoužitelná.

Dále se občas hodí procedura „**window**“, která nastaví velikost virtuální obrazovky. Funguje to tak, že např. „window(10, 10, 70, 15)“ způsobí, že třeba procedura „gotoXY(1,1)“ přejde ve skutečnosti na „(10,10)“ nebo třeba „clrscr“ smaže pouze znaky v tomto „okně“.

Pak jsou zde ještě dvě procedury pro práci s PC speakerem. Je to „**sound**“ a „**nosound**“. Příklad vše objasní:

```
uses (* fdelay, *) crt;
begin
sound(500); (* začne přehrávání zvuku na frekvenci 500 Hz *)
readkey;
nosound;    (* vypne přehrávání zvuku *)
end.
```

Poslední velice důležitá funkce, o které se zmíním, je „**keypressed**“. Je typu boolean a určuje, zda se nachází v bufferu klávesnice čekající klávesa, neboli zda od posledního zavolání funkce „readkey“ bylo něco stisknuto. To je velice důležité, protože díky ní můžeme čtení klávesy podmínit a to stylem:

```
uses (* fdelay, *) crt;
var ch: char;
begin
repeat
if keypressed then ch := readkey;
                   (* když je stisknuta klávesa, zjistíme její hodnotu *)
write('abc');      (* pořád dokola vypisujeme abc *)
delay(500);
until ch = #27;    (* #27 je escape *)
end.
```

Tento program vypisuje pořád dokola „abc“, ale kdyby nebylo načtení klávesy podmíněno zjištěním, jestli nějaká byla stisknuta, program by vždy čekal na stisk klávesy a po každém proběhnutí cyklu by se na klávesu čekalo znovu.

Kapitola 11 – Unita DOS

Úplný popis unity DOS najdete v nápovědě k překladači. Já se zde budu zabývat pouze nejzákladnějšími a nejpoužívanějšími procedurami a funkcemi. Pro zjištění volného místa na disku použijete funkci „**DiskFree**“ a pro zjištění velikosti disku funkci „**DiskSize**“. Parametr těchto funkcí tvoří číslo disku, kde se postupuje podle pravidla `1 = A`, `2 = B`, `3 = C` atd. Pokud disk neexistuje, funkce vrátí `-1`. Takže když si budete chtít napsat prográmek, který vám vypíše všechny disky a jejich velikost, uděláte to takto:

```
uses dos;
const disky: string = 'ABCDEFGHIJKLMN'; (* to snad stačí :) *)
var size: longint;
    disk: byte;
begin
disk := 3; (* začneme u trojky, abychom přeskočili disketové mechaniky *)
repeat
size := DiskSize(disk);
if size = -1 then break;
writeln('Disk '+disky[disk]+ ' ma ', size div (1024*1024), ' MB');
disk := disk + 1;
until false;
end.
```

Háček je v tom, že longint má rozsah jenom něco přes 2 miliardy, takže nám to správně zjistí maximálně tak velikost CD nebo diskety.

Další funkcí, která se může hodit je „**DosVersion**“, která vrátí číslo verze *DOSu*.

„**EnvCount**“ vrací počet systémových „proměnných“ a „**EnvStr**“ vrací řetězec ve tvaru „`jmeno=hodnota`“:

```
uses Dos;
var i: Integer;
begin
 for i := 1 to EnvCount do (* vypíšeme všechny systémové proměnné *)
  writeln(EnvStr(i));
end.
```

S tímto souvisí i funkce „**GetEnv**`(jméno: string)`“, která vrací hodnotu proměnné uvedené v parametru:

```
uses dos;
begin
writeln('Pouzita command lajna: '+GetEnv('COMSPEC'));
writeln('Path: '+GetEnv('PATH'));
end.
```

Pro vyhledávání souborů jsou v unitě DOS zabudovány procedury „**FindFirst**“ a „**FindNext**“. Procedura „`FindFirst`“ je deklarována takto:

```
procedure FindFirst(Path: String; Attr: Word; var F: SearchRec);
```

„Path“ určuje cestu k souborům, „Attr“ určuje atributy souboru, které se dají zapsat buď číselně, nebo pomocí konstant:

Konstanta	Hodnota	Význam
ReadOnly	$01	Jen pro čtení
Hidden	$02	Skrytý
SysFile	$04	Systémový soubor
VolumeID	$08	Disk
Directory	$10	Složka
Archive	$20	Archivovat
Anyfile	$3F	Jakýkoliv soubor

„Searchrec“ je definován takto:

```
type SearchRec = record
     Fill: array[1..21] of Byte;
     Attr: Byte;
     Time: Longint;
     Size: Longint;
     Name: string [12];
end;
```

Samozřejmě je definován v unitě DOS, takže si ho nemusíte definovat sám. A teď malé vysvětlení jednotlivých položek:

Fill – rezervováno pro *DOS*
Attr – atributy (podle tabulky)
Time – zapakované datum a čas poslední změny (jak ho rozpakovat se dozvíte v další části)
Size – velikost v bytech
Name – Jméno souboru i s příponou

Procedura „FindNext“ je deklarována takto:

```
procedure FindNext(var F: SearchRec);
```

Tato hledá další výskyt souboru odpovídajícího podmínkám při hledání „FindFirst“. Pokud není možné další soubor najít, vrátí funkce „DosError“ jinou hodnotu než 0. Takže program který vypíše obsah C:\ je:

```
uses dos;
var soub: searchrec;
begin
FindFirst('c:\*.*',anyfile,soub);   (* najdeme první soubor odpovídající zadání *)
while DosError = 0 do begin         (* dokud se soubor podaří najít *)
 writeln(soub.name, ' - ',soub.size, ' bytů'); (* vypíšeme jméno + velikost *)
 findnext(soub);                    (* najdeme další soubor *)
 end;
end.
```

A nyní slibované pakování a odpakování času a data. K tomu slouží procedury „PackTime“ a „UnpackTime“, které jsou deklarovány takto:

```
procedure PackTime(var T: DateTime; var Time: Longint);
procedure UnpackTime(Time: Longint; var DT: DateTime);
```

„DateTime“ je deklarováno takto:

```
DateTime = record
 Year,Month,Day,Hour,Min,Sec: Word;
end;
```

Když tedy použijeme předchozí příklad, ale místo velikosti vypíšeme datum, tak to bude vypadat takto:

```
uses dos;
var soub: searchrec;
    d: datetime;
begin
FindFirst('c:\*.*',anyfile,soub);
while DosError = 0 do begin
 unpacktime(soub.time,d);
 writeln(soub.name,' - ',d.day,'.', d.month,'.', d.year,' ', d.hour,':', d.min,':', d.sec);
 findnext(soub);
 end;
end.
```

Pokud chcete zjistit datum a čas v OS, tak k tomu použijete procedury „GetDate“ a „GetTime“, které jsou deklarovány takto:

```
procedure GetDate(var Year, Month, Day, DayOfWeek: Word);
procedure GetTime(var Hour, Minute, Second, Sec100: Word);
```

Procedura „GetDate“ je snad jasná, akorát se musím zmínit o tom, že „DayOfWeek“ vrací číslo od 0 do 6, kde 0 je neděle. U procedury „GetTime“ znamená „Sec100“ setinu vteřiny. K nastavení data a času máme k dispozici procedury „SetDate“ a „SetTime“:

```
procedure SetDate(Year, Month, Day: Word);
procedure SetTime(Hour, Minute, Second, Sec100: Word);
```

Díky proceduře „GetTime“ už máme možnost vytvořit si vlastní proceduru „Delay“, která by čekala určitý počet setin vteřiny. Nevíte jak na to? Je to prosté. Vytvoříme si proměnnou „min“, do které přiřadíme čas, kdy naše procedura delay začíná a proměnnou „souc“, do které přiřadíme při každém proběhnutí cyklu aktuální čas. Cyklus pak bude probíhat do té doby, než bude aktuální čas větší než čas konce což je „min+doba“ nebo je aktuální čas menší než původní doba. To proto, že když se jakoby dostaneme ke konci minuty, budou zase vteřiny „menší“, a cyklus by mohl trvat třeba i minutu, i když v zadání bylo třeba jen 5 setin. Zní to sice složitě, ale tak složité to není:

```
uses dos;

procedure mydelay(stn: integer);
var n,vt,st: word;
    min,souc: longint;
begin
gettime(n,n,vt,st);
```

```
min := vt*100+st;  (* zjistíme čas, kdy procedura začala *)
repeat
gettime(n,n,vt,st);
souc := vt*100+st; (* vypočítáme aktuální čas *)
until (souc >= min + stn) or (souc < min);
                    (* dokud nedosáhneme požadovaného času *)
end;

begin
write('Text');
mydelay(50);
write('Text vypsany po polovine vteriny');
end.
```

Pokud to nechápete, tak se musíte smířit s tím, že to funguje :).

Procedury sloužící ke zjištění data a času změny souboru a atributů souboru jsou „**GetFTime**“ a „**GetFattr**“. Tyto procedury nám zjistí informace o souboru vyjádřeném typem „file“ a jsou deklarovány takto:

```
procedure GetFTime(var F; var Time: Longint);
procedure GetFAttr(var F; var Attr: Word);
```

Tento soubor „F“ musí být otevřený příkazem „reset“. Naopak k nastavení těchto věcí slouží:

```
procedure SetFTime(var F; Time: Longint);
procedure SetFAttr(var F; Attr: Word);
```

Příklad:

```
uses dos;
var f: file;          (* file může být i beztypový *)
    pom: longint;
    d: datetime;
begin
assign(f,'pok.hmm'); (* na příponě nezáleží *)
rewrite(f);           (* vytvoříme soubor *)
getFTime(f, pom);
unpacktime(pom, d);   (* tím jsme vlastně zjistili aktuální datum, *)
                      (* protože jsme ten soubor právě vytvořili *)
writeln(d.day,'.', d.month,'.', d.year); (* vypíšeme základní údaje *)
end.
```

To je prozatím z unity DOS vše, protože další funkce už souvisí s Assemblerem a já předpokládám, že pokud čtete tuto knihu, tak Assembler neznáte, a proto se jimi nebudu zabývat.

Kapitola 12 – Dynamické proměnné

Ano, dostáváme se k tomu, co jsem sliboval v úvodu – dynamické proměnné. Jistě si říkáte: „Co to asi ta dynamická proměnná je?“ Dosud jsme používali pouze statické proměnné. To znamená, že jsme si je nadeklarovali a dále v programu jsme je používali. Ale co když nastane situace, kdy nebudete vědět, kolik proměnných budete potřebovat a budete nuceni je vytvořit během programu? Tak to už si se statickými proměnnými nevystačíte.

Pascal obsahuje datový typ „**pointer**“. Je to ukazatel na místo v paměti, kde je uložena hodnota. Toho, aby překladač poznal, že deklarujeme ukazatel a ne proměnnou určitého typu, docílíme znakem „^“ (pro zrychlení – tento znak napíšete pomocí ALT + 94). Příklad:

```
type ukazatel = ^integer;
```

Tím jsme si vytvořili ukazatel na typ **integer**. S tímto ukazatelem nemůžeme pracovat tak, že do něj přiřadíme hodnotu (protože tím bychom vlastně řekli ukazateli, na jaké místo v paměti má ukazovat), ale musíme přiřadit hodnotu do paměti, kam ukazatel ukazuje. Tuto paměť musíme nejdříve vyhradit procedurou „**new**“:

```
type ukint = ^integer;
var prom: ukint; (* proměnná je ukazatel (možno zapsat jako „prom: ^integer“) *)
begin
new(prom);        (* získáme paměť *)
prom^ := 50; (* do místa v paměti, kam ukazuje „prom“, zapíšeme hodnotu *)
writeln(prom^);  (* přesvědčíme se, že se hodnota opravdu uložila *)
end.
```

Pokud chceme, aby ukazatel ukazoval „nikam“, provedeme to klíčovým slovem „**nil**“ (v našem případě by to bylo „**prom := nil**“ – bez „^“!). Tím jsme ale stále nedocílili toho, abychom mohli vytvořit za běhu programu neomezený počet proměnných. To co jsme udělali je vlastně jen složitější práce s normální proměnnou. Většího počtu dosáhneme takto:

```
type pint = ^int; (* ukazatel na náš záznam *)
     int = record
           hodnota: integer;
           dalsi: pint;
           end;
var prvni, ak: pint;
    pom: integer;
begin
new(ak);           (* vyhradíme si paměť *)
prvni := ak;       (* prvni ukazuje na první položku *)
repeat
 readln(pom);      (* uživatel zadá hodnotu *)
 ak^.hodnota := pom;
 new(ak^.dalsi);  (* vytvoříme další položku *)
 ak := ak^.dalsi; (* „ak“ ukazuje na další položku *)
 ak^.dalsi := nil;  (* nechceme, aby další položka měla nějakou hodnotu *)
until pom = 0;       (* zadáváme čísla, dokud není zadána nula *)

(* nyní všechny položky vypíšeme *)
```

```
ak := prvni;            (* „ak“ zase ukazuje na první položku *)

while ak^.dalsi <> nil do begin (* dokud existuje další položka *)
 writeln(ak^.hodnota);          (* vypíšeme její hodnotu *)
 ak := ak^.dalsi;
end;

end.
```

Teď mam takový neblahý dojem, že to asi nechápete. Jestli to tak opravdu je, zkuste si prostudovat ten příklad s jednou proměnnou a až ho pochopíte, projděte si řádek po řádku v dalším příkladu s „neomezeným“ počtem proměnných. Snad to časem pochopíte.

Jdeme dál. S ukazateli souvisí i funkce „**ptr(segment, offset)**“, díky které můžete zařídit, aby ukazatel ukazoval na určité místo v paměti. Jenomže to zase zabíháme moc hluboko do znalosti hardwaru počítače, proto tady uvedu pouze příklad:

```
var m: ^byte;
begin
m := Ptr($40, $49); (* adresa videomódu v paměti *)
writeln('Videomod: ',m^);
end.
```

S tím souvisí i funkce „**addr(x)**“, jejíž parametr tvoří jméno proměnné. „Addr“ vrací adresu paměti, ve které je proměnná uložena:

```
var p: pointer;
    pom: string;
begin
pom := 'Ahoj lidi, jak se mate?'; (* zadáme text *)
p := addr(pom); (* „p“ teď ukazuje na adresu „pom“ v paměti *)
writeln(string(p^)); (* vypíšeme, co se v této paměti vyskytuje, abychom se *)
                     (* přesvědčili, že „p“ ukazuje opravdu na adresu „pom“ *)
end.
```

Proměnnou „p“ musíme nejprve přetypovat na string, protože samotné „p“ je neznámého typu. To bychom samozřejmě udělat nemuseli, pokud by „p“ nebylo typu „pointer“ ale typu „^string“, neboli „pointer to string“ (tak se to zapsat samozřejmě nedá :)

S tím souvisí i funkce „**ofs(x)**“ a „**seg(x)**“, které vrací offset, resp. segment proměnné.

K další práci s přidělením paměti pointeru slouží procedury „**getmem(pointer, velikost)**“ a „**freemem(pointer, velikost)**. Maximální velikost přidělené paměti je 65528 bytů. Příklad:

```
var p: pointer;
begin
getmem(p, 20000); (* právě máme k dispozici 20 KB paměti *)
freemem(p, 20000); (* ale my je nevyužijeme a zase je z paměti uvolníme *)
end.
```

To je zatím o dynamických proměnných vše

Kapitola 13 – Práce s textovými řetězci

Tato kapitola bude o tom, jak se pracuje s textovými řetězci (nečekaně :). Základní způsob práce s řetězci spočívá v tom, že je možné s ním pracovat jako s polem znaků. Tzn. že zápis „`var text: string[50];`“ je v principu stejný jako „`var text: array[1..50] of char;`“, a tak s ním také můžeme pracovat. K další práci s textem slouží v Pascalu tyto funkce:

```
function Copy(S: String; Index: Integer; Count: Integer): String;
procedure Delete(var S: String; Index: Integer; Count:Integer);
procedure Insert(Source: String; var S: String; Index: Integer);

function Pos(Substr: String; S: String): Byte;
function Length(S: String): Integer;
```

Názvy funkcí už vypovídají o tom, co ta funkce asi bude dělat. Takže si to rozebereme od začátku. Funkce „**copy**“ dostane jako parametr řetězec, pozici a počet a vrátí řetězec zkopírovaný z dané pozice do dané pozice + počet:

```
begin
writeln(copy('Ahoj lidi',3,5)); (* vypíše „oj li" *)
end.
```

Doufám, že je to jasné. Funkce „**delete**“ smaže zadanou část textového řetězce:

```
var txt: string;
begin
txt := 'Ahoj lidi';
delete(txt,3,5);
writeln(txt); (* vypíše „Ahdi“ *)
end.
```

A konečně funkce „**insert**“ vloží řetězec na zadanou pozici:

```
var txt: string;
begin
txt := 'Ahoj lidi';
insert('vsichni ',txt,6);
writeln(txt); (* vypíše „Ahoj vsichni lidi“ *)
end.
```

Pak zde máme funkci „**length**“, která nám vrátí délku řetězce a funkci „**Pos**“, která nám vrátí pozici podřetězce v řetězci. Příklad:

```
begin
writeln(pos('Petr','On se jmenuje Petr Novak'));
                                        (* „Petr“ začíná na 15 znaku *)
end.
```

Pokud se zadaný podřetězec v řetězci nenachází, funkce vrátí hodnotu 0. Proto se dá dobře využít k zjištění, jestli se v řetězci nachází jiný řetězec.

S těmito znalostmi už můžete vymyslet program, který by např. dokázal odizolovat z textového řetězce jednotlivé položky oddělené dvojtečkou – program by třeba dostal ke zpracování text „`ahoj lidi:petr:dalsi text:a jeste jeden`“ a z něj by vypsal:

ahoj lidi
petr
dalsi text
a ještě jeden

Zkuste to vymyslet a porovnejte svůj program se vzorovým řešením, které zde uvedu.

U tohoto zadání jsou prakticky možnosti dvě – pomocí „`pos`“:

```
var txt: string;
begin
txt := 'ahoj lidi:petr:dalsi text:a jeste jeden';
while true do begin            (* cyklus se pořád opakuje *)
if pos(':',txt) > 0 then begin (* když nejsme u poslední položky *)
 writeln(copy(txt,1,pos(':',txt)-1));
                   (* vypíšeme text od pozice 1 do dvojtečky *)
 delete(txt,1, pos(':',txt));  (* položku textu smažeme *)
end
else begin         (* za poslední položkou už není dvojtečka *)
 writeln(txt);     (* v „txt“ už zbyl pouze poslední text *)
 break;                        (* cyklus se přeruší *)
end;
end;
end.
```

Nebo pomocí práce se `stringem` jako s polem znaků:

```
var txt: string;
    i,z: integer;
begin
txt := 'ahoj lidi:petr:dalsi text:a jeste jeden';
z := 1;
for i := 1 to length(txt) do begin (* projdeme text až do konce *)
if txt[i] = ':' then begin     (* když narazíme na dvojtečku *)
 writeln(copy(txt,z,i-z));     (* vypíšeme text *)
 z := i + 1;
 end;
if i = length(txt) then writeln(copy(txt,z,i-z+1));
                               (* když jsme na konci textu *)
end;
end.
```

Jestli to nechápete, tak si projděte příklad řádek po řádku a říkejte si, co ten příkaz dělá a určitě to pochopíte.

Kapitola 14 – Trochu matematiky

Matematické funkce

Ona se ta matika taky občas hodí, takže tady popíšu pár matematických funkcí. Základní jsou samozřejmě mocnina a odmocnina. Pro ty v Pascalu existují funkce:

```
function Sqr(X): (stejný typ jako parametr); - mocnina
function Sqrt(X: Real): Real;                - odmocnina
```

Pro práci s úhly slouží tyto funkce:

```
function Cos(X: Real): Real;
function Sin(X: Real): Real;
function ArcTan(X: Real): Real;
```

Myslím, že názvy funkcí mluví za vše. POZOR! Úhly se zadávají v radiánech. Pascal nemá funkce pro tangens, ale lze ho vypočítat takto:

```
tan(x) = Sin(x) / Cos(x)
ArcSin(x) = ArcTan(x/sqrt(1-sqr(x)))
ArcCos(x) = ArcTan(sqrt(1-sqr(x))/x)
```

Pro práci s přirozeným logaritmem slouží:

```
function Ln(X: Real): Real;
function Exp(X: Real): Real;
```

Funkce „`pi`“ vrací hodnotu konstanty , neboli 3.1415926535897932385.

Pro úpravu desetinných čísel slouží tyto funkce:

```
function Int(X: Real): Real;       - vrací celou část čísla
function Frac(X: Real): Real;      - vrací desetinnou část čísla
function Round(X: Real): Longint;  - zaokrouhlí číslo
function Trunc(X: Real): Longint;  - odebere desetinnou část
```

Vše objasní příklad:

```
begin
writeln(int(128.56));   (*  128  *)
writeln(frac(128.56)); (*  0.56  *)
writeln(round(128.56)); (*  129  *)
writeln(trunc(128.56)); (*  128  *)
end.
```

Funkce „`abs(x)`“ vrací absolutní hodnotu:

```
begin
writeln(abs(-52)); (*  52  *)
end.
```

Bitové operace s čísly

Co jsou to bitové operace? Řekl bych, že to asi souvisí s bity. Co je to bit? Bit je nejmenší uložitelná informace. Nabývá hodnot 0 nebo 1. Bitové operátory pracují právě s bity. Například datový typ „word“ má rozsah od 0 do 65535 a to znamená, že má velikost 2 byty. Pozor. Nepleťte si pojem byte a bit. Byte se skládá z 8 bitů, to znamená, že se do něj vejde číslo od 0 do 255 (2^8-1). Takže když má typ rozsah 2 byty znamená to, že se do něj vejde číslo od 0 do 2^{16}-1. Nebo případně, když je se znaménkem, tak půlka rozsahu v mínusu a půlka v plusu. Možná vám to připadá jako nesmyslné žvásty, ale bitové operace se hodí například pro práci s grafikou. Nyní si předvedeme, co dělá operace „and“:

```
c := 55 and 231;
```

Co se s proměnnou „c“ stane?

```
55   =  00110111
231  =  11100111
výs  =  00100111
```

Výsledek je 39. Jistě jste si všimli, že bit výsledku je 1 jen tehdy, pokud je 1 bit prvního i druhého čísla. Jdeme dál – „or“:

```
55   =  00110111
231  =  11100111
výs  =  11110111
```

Výsledek je 247, protože bit výsledku je 1, pokud má tento bit hodnotu 1 alespoň u jednoho ze zadaných čísel. Další je „xor“:

```
55   =  00110111
231  =  11100111
výs  =  11010000
```

Výsledek je 208, protože bit výsledku je 1, jenom pokud je jeden z bitů zadaných čísel 1 a druhý ne. Další operace je „not“. Je to operace pouze s jedním operandem, takže se zapíše třeba jako:

```
c := not 231;

231  =  11100111
výs  =  00011000
```

U této operace je problém, že závisí na překladači, jak implementuje rozsah vrácené hodnoty. Pokud ji bere jako word, vypadá to takto:

```
0000000011100111
1111111111000110
```

Jak vidíte, ty jedničky na začátku jsou tam jaksi navíc. Obecně však platí pravidlo, že když přiřadíte výsledek operace not do proměnné s rozsahem zasahujícím do mínusu vyjde vám:

```
var c: integer;
begin
```

```
c := not 231; (* tyto zápisy mají stejnou hodnotu *)
c := -(231+1);
end.
```

Další operace, o kterých se zmíním jsou posuny bitů. Slouží k tomu operátory „**shl**“ a „**shr**“. Předvedu na příkladu:

```
var c: byte;
begin
c := 231 shl 3;
end.
```

Znamená to:

```
231 = 0000000011100111
```

Posuneme o tři bity doleva:

```
0000011100111000
```

A zde vám možná překladač zahlásí chybu. Číslo 231 už má totiž 8 bitů, a kdybychom se je pokusili posunout doleva, šli by ty 3 první bity kamsi pryč, a to některé překladače nedovolí, ale já jsem úmyslně zapsal číslo jako 16ti bitové, abyste pochopili, co operace „**shl**“ dělá. Jiné toto naopak umožňují, a je možné tyto bitové operace použít ke „smazaní“ několika bitů. To ovšem není nemožné udělat ani u překladačů, kterým to „vadí“:

```
var c: byte;
begin
c := 231;
c := (c shl 3) shr 3;
end.
```

Jak vidíte, není při kompilaci jasné, že proměnná „**c**“ bude moc velká, a při běhu programu to chybu nezahlásí, takže je možné to používat.

Operace „**shr**“ dělá samozřejmě to samé, akorát posouvá bity doprava.

Popis datových typů

Celočíselné datové typy:

Typ	Rozsah
Integer	-32768 až 32767
Byte	0 až 255
ShortInt	-128 až 127
Word	0 až 65535
LongInt	-2147483648 až 2147483647

Reálné datové typy:

Typ	Rozsah	Desetinných míst
Real	5.0e-39 až 1.7e38	11-12
Single	1.5-45 až 3.4e38	7-8
Double	5.0e-324 až 1.7e308	15-16
Extended	3.4e-4932 až 1.1e4932	19-20
Comp	-9.2e18 až 9.2e18	19-20

Řetězcové datové typy:

Typ	Rozsah
String	0 – 255 znaků
Char	1 znak

Na závěr

Pokud cítíte potřebu vyjádřit se k obsahu knížky nebo k její formě, pokud jste v ní nalezli nějakou chybu nebo máte-li jakoukoliv jinou připomínku, poznámku, libovolný námět či dotaz, kontaktujte prosím autora na e-mailové adrese *kubazek@gmail.com*.

REJSTŘÍK

R

S

T

U

V

W

X

Z