Flex 4.0

Učebnice programování v jazyce Flex 4.0

Verze: 1.0

Flex 4.0

Učebnice programování v jazyce Flex 4.0

Flex 4.0
Učebnice programování v jazyce Flex 4.0

Autor: Aleš Procháska

Verze: 1.0
Datum tisku: 3.6.2004
Jméno souboru:.pdf

Obsah

Seznam použitých tabulek

1. Úvod

Učebnice programovacího jazyka Flex je určena k seznámení s jazykem formou příkladů a neformálního popisu. Jsou v ní uvedeny důležité jazykové konstrukce v rozsahu dostatečném pro běžné programování v klasickém i objektovém stylu. Je vhodná jak pro programátory se začátečnickými znalostmi, tak pro zkušené programátory v některém vyšším programovacím jazyce jako rychlý úvod do programovacího jazyka.

2. Základní příkazy a datové typy

2.1 Struktura programu, volání procedur

Nejjednodušší program ve Flexu, který jenom napíše na obrazovku pár slov, vypadá takto:

```
program dobry_den =

with
  support;

begin
  write_line_txt('Dobrý den, jak se máte?');
  end dobry_den;
```

Program ve Flexu začíná klíčovým slovem `program` a jménem programu. Jméno může být zvoleno libovolně, pouze musí dodržovat pravidla Flexu pro pojmenovávání jednotlivých entit – smí být složeno jen z písmen anglické abecedy, číslic a znaku podtržení, nesmí obsahovat mezery a nesmí začínat číslicí. Mohou být libovolně použita malá i velká písmena, která se pro účely pojmenování považují za shodná. Délka jména není omezena.

Za jménem je seznam použitých modulů, který začíná klíčovým slovem `with` a v tomto programu obsahuje jediný modul `support`. Více o modulech řekneme o pár řádků níže, zatím jen ujištění pro céčkaře, že je to přibližně totéž co `#include` v programovacím jazyce C.

Vlastní programový kód, neboli tělo programu, začíná klíčovým slovem `begin` a končí slovem `end`, za kterým se povinně opakuje jméno programu – to proto, aby mohl překladač snáze odhalit a přesněji lokalizovat jednu z nejčastějších programátorských chyb, zapomenutí nebo pomíchání "endů". Zde je vhodné se zmínit o principu, který nás bude provázet celým jazykem – Flex obsahuje celou řadu podobných opatření proti chybám z nepozornosti. Jde o významný rys jazyka, který v daleko větší míře než většina ostatních jazyků (zejména C, ale i C++, Java, Pascal) kontroluje formální správnost programu.

Tělo programu v příkladu obsahuje jeden jediný příkaz. Je to volání procedury `write_line_txt`, která, jak již napovídá její jméno, umí zapsat řádek textu na obrazovku. Text, který chcete zapsat, je parametrem procedury, jinými slovy je zapsán v závorkách za procedurou. Vlastní text, neboli textový literál, jak se také říká textu přímo zapsanému v programu, musí být ještě v souladu s pravidly syntaxe Flexu uveden mezi apostrofy. Na samotný text není kladeno žádné omezení, můžete použít všechny znaky, které se dají napsat. Procedura `write_line_txt` zapíše tento text na první řádek obrazovky, druhé volání procedury `write_line_txt` zapíše uvedený text na druhý řádek atd.

Jak jste asi správně usoudili, vaše programy budou obsahovat hodně volání procedur. Součástí instalace jsou tzv. knihovny, což jsou připravené sady procedur pro nejrůznější účely, kreslení na obrazovku, matematické výpočty, zpracování textů, práci s databází, používání služeb operačního systému a podobně. Knihovní procedury jsou rozděleny do skupin a podskupin podle oblasti použití. Každá taková skupina či podskupina tvoří jeden modul, což je programová jednotka určená přímo pro tyto účely.

Program `dobry_den` používá knihovní modul `support`, který obsahuje různé definice a jednoduché procedury pro vypisování textů na obrazovku a zadávání textů z klávesnice. Modul `support` byl vytvořen speciálně pro použití v příkladech v této učebnici. Ve skutečném programování budete místo něj používat řadu jiných, specializovaných modulů ve kterých je k dispozici podstatně více předdefinovaných prostředků.

Kromě knihovních neboli výrobcem dodaných procedur a modulů samozřejmě můžete vytvářet i své vlastní. Vytváření vlastních procedur je popsáno dále v této učebnici (kapitola 4), moduly jsou popsány v referenční příručce (viz kapitolu 9).

2.2 Proměnné, přiřazovací příkaz

Mezi základní stavební kameny každého programu patří proměnné. Proměnná je místo vyhrazené v paměti počítače, do kterého můžeme uložit nějakou hodnotu – číslo, text nebo jiná data. Pojem proměnné nejlépe ukážeme na příkladu.

```
program dobry_den_2 =

with
  support;

var
  hlaseni: text;

begin
  hlaseni:='Dobrý den, jak se máte?';
  write_line_txt(hlaseni);
  end dobry_den_2;
```

Jak vidíte, jde o upravenou verzi programu z předchozí kapitoly.

Do programu je zavedena část deklarací proměnných (uvedená klíčovým slovem var), ve které je deklarována proměnná jménem `hlaseni` datového typu `text`. Datový typ `text` určuje, že do proměnné `hlaseni` může být uložen text. Deklaraci proměnné chápeme tak, že proměnná existuje (je použitelná) od místa deklarace až do konce programu.

V programu se text do proměnné uloží pomocí přiřazovacího příkazu **:=**. Na levé straně přiřazovacího příkazu je uvedeno jméno proměnné jejíž obsah se má změnit, na pravé straně je hodnota, kterou má proměnná mít po provedení příkazu.

V proceduře `write_line_txt` už na rozdíl od předchozího příkladu není uveden přímo text, ale je zde jen jméno proměnné. Výsledek programu ale bude stejný, protože v proměnné `hlaseni` je uložen stejný text, jaký je v předchozím příkladu uveden přímo v parametru procedury.

2.3 Výrazy

Prográmek z předchozích odstavců nám poslouží ještě jednou, tentokrát pro objasnění výrazů.

Nejdříve se seznámíme s další užitečnou procedurou, `input_line_txt`. Stejně jako `write_line_txt` má jeden parametr typu `text`, ale funkci opačnou než `write_line_txt`. Po zavolání procedura čeká, až uživatel napíše z klávesnice nějaký text a stiskne klávesu (**Enter**). Poté text uloží do proměnné, která byla uvedena v parametru a skončí. Všimněte si, že zde může být parametrem pouze proměnná, zatímco u procedury `write_line_txt` to mohla být proměnná nebo přímá hodnota (nebo obecně jakýkoliv výraz – viz dále).

```
program dobry_den_3 =

with
  support;

var
  hlaseni: text;
  odpoved: text;

begin
  hlaseni:='Dobrý den, jak se máte?';
  write_line_txt(hlaseni);
  input_line_txt(odpoved);
  hlaseni:='Děkuji, já se mám také ' & odpoved;
  write_line_txt(hlaseni);
  end dobry_den_3;
```

Vidíme, že program ze začátku funguje stejně jako předchozí program. Po vypsání textu "Dobrý den, jak se máte?" ale čeká, až uživatel něco napíše. Dejme tomu, že zadá z klávesnice "dobře" a stiskne klávesu (**Enter**). Program odpoví "Děkuji, já se mám také dobře" a skončí.

Jak je to uděláno? Program volá proceduru `input_line_txt` a jako parametr jí předá proměnnou `odpoved`. Procedura podle očekávání počká na zadání uživatele a po ukončení procedury bude v proměnné `odpoved` uložen uživatelem napsaný text, například "dobře".

V dalším řádku se právě zadaný text spojí (zřetězí) s pevně daným textem a přiřazovacím příkazem uloží do proměnné `hlaseni`. Pro spojení dvou textů se používá symbol `&` (operátor zřetězení), napsaný mezi texty (operandy) které chceme sloučit.

Poslední řádek těla programu už jen vypíše připravený text na obrazovku.

> Poznámka
>
> Pokud nemáte zatím žádné zkušenosti s programováním, nenechte se zmást podobností s matematickým zápisem. V matematice by bylo nepřípustné napsat v jediném příkladu dejme tomu
>
> a = 10
>
> a = 15
>
> protože bychom tím vlastně tvrdili, že 10 = 15, což není (za obvykle přijímaných předpokladů) pravda. V programu však přiřazovací příkaz, ač je na první pohled podobný tomuto matematickému zápisu, něco podobného dovoluje. Je to tím, že proměnná nepředstavuje nějaký symbol jednou provždy přiřazený určité hodnotě, ale je to jen taková schránka, do které se ukládá text (nebo číslo nebo jiný objekt) jen na potřebnou dobu. V programu `dobry_den_3` tak měla proměnná `hlaseni` na prvních třech řádcích těla programu (myslí se tím po provedení příslušného řádku) hodnotu `'Dobrý den, jak se máte?'`, na čtvrtém řádku ale byla nahrazena hodnotou `'Děkuji, já se mám také dobře'` kterou si podržela až do konce programu.

2.4 Přirozená a celá čísla

Ve Flexu jsou samozřejmě k dispozici datové typy pro uložení čísel. Do proměnných typu `natural` lze ukládat přirozená čísla v rozsahu od nuly do přibližně čtyř miliard, do proměnných typu `integer` celá čísla přibližně v rozsahu od minus dvou miliard do dvou miliard.

S čísly lze provádět základní aritmetické operace, které jsou shrnuty v tabulce:

operátor	název	příklad
`+`	sčítání	`x+y`
`-`	odčítání	`x-y`
`*`	násobení	`x*y`
`div`	celočíselné dělení	`x div y`
`mod`	zbytek po dělení	`x mod y`

Tabulka č. 1. Základní aritmetické operace

Kromě uvedených binárních operátorů (operátorů, které mají dva operandy) existuje pro celá a přirozená čísla jeden unární operátor (s jediným operandem):

operátor	název	příklad
-	mínus	-x

Tabulka č. 2. Unární operátor

S prioritou operátorů je to stejné jako v matematice, `*`, `div` a `mod` mají přednost před + a −, takže například ve výrazu `x+y*z` se nejdřív vynásobí `y` a `z` a výsledek se sečte s `x`. Aby bylo možné pořadí výpočtu změnit, lze stejně jako v matematice použít závorky, například `(x+y)*z` napřed sečte `x` a `y` a součet vynásobí se `z`.

Následující program ukazuje různé přípustné příkazy. Vysvětlivky k jednotlivým příkazům jsou uvedeny přímo v programu. Vysvětlivky, neboli v programátorské terminologii komentáře, se mohou do programu vkládat podle libosti. Komentář začíná dvěma pomlčkami a cokoliv je za nimi napsáno až do konce řádky nemá žádný vliv na funkci programu, pouze zlepšují čitelnost programu pro člověka.

```
program aritmetika =

var
  i: integer;
  j: integer;

begin
  i:=1; -- po provedení tohoto příkazu má proměnná i hodnotu 1
  j:=i+1; -- j má hodnotu 2
  j:=i+2; -- j má hodnotu 3
  j:=j+1; -- j má hodnotu 4
  i:=j*10; -- i má hodnotu 40
  i:=(j+1)*10; -- i má hodnotu 50
  i:=((j+1)*2+1)*i; -- i má hodnotu 550
  i:=i div 10; -- celočíselné dělení, i má hodnotu 55
  i:=i div 10; -- celočíselné dělení, i má hodnotu 5
  i:=i mod 2; -- zbytek po dělení, i má hodnotu 1
  j:=-i; -- unární operace, j má hodnotu -1
  end aritmetika;
```

2.5 Přetypování

Znalce jiných programovacích jazyků možná překvapí, že není dovoleno přiřazení mezi proměnnými znaménkového `signed` typu (datový typ `integer`) a bezznaménkového (`unsigned`) typu (datový typ `natural`). Například konstrukce

```
var
  i: integer;
  x: natural;...
  i:=x;
```

je nepřípustná a překladač v místě přiřazení vyhlásí chybu "nekompatibilní operandy". Toto omezení je v jazyce proto, že přiřazování mezi signed a unsigned typy zpravidla není logicky zdůvodnitelné a svědčí spíše o chybném návrhu programu. Pokud z nějakého důvodu přesto potřebujete toto přiřazení provést, použijte přetypování:

```
x:=natural(i);
```

které říká, že se výraz (zde `i`) má převést na typ natural a poté přiřadit do `x` (zde typu `natural`). Uvědomte si, že pokud hodnota `i` bude záporná, program při provádění v tomto místě skončí chybou, protože číslo nejde ani s přetypováním přiřadit do proměnné, do které by se "nevešlo".

> Poznámka
> Tři tečky označují v příkladech v této učebnici vynechaný kód, který není podstatný pro vysvětlení problematiky.

2.6 Reálná čísla

Flex má k dispozici také reálná čísla, nutná pro nejrůznější výpočty. Reálná čísla jsou zastoupena typem `real`. S reálnými čísly lze provádět stejné aritmetické operace jako s celými, pouze místo celočíselného dělení (`div`) je dělení (`/`) a operace zbytek po dělení není definována.

Přímý zápis reálného čísla se od celého čísla liší tím, že povinně obsahuje desetinnou tečku nebo exponent. Exponent je číslo uvedené písmenem `E` (nebo `e`), které říká, o kolik míst je potřeba posunout desetinnou tečku doprava (kladný exponent) nebo doleva (záporný exponent). Smyslem zápisu čísla s exponentem (tzv. semilogaritmický tvar) je zjednodušit zápis velkých čísel.

Příklady zápisů reálných čísel:

```
var
  r: real;...
  r:=3.1415926   -- pí
  r:=1e6         -- milión
  r:=1000000.0   -- milión
  r:=1.0e6       -- milión
  r:=1.23456e2   -- totéž co 123.456
  r:=1.23456e5   -- totéž co 123456.0
  r:=1.23456e8   -- totéž co 123456000.0
  r:=1.23456e-1  -- totéž co 0.123456
  r:=1.23456e-6  -- totéž co 0.00000123456
  r:=123.456e3   -- totéž co 123456.0
```

2.7 Příkaz if, relační a logické operace

Příkaz `if`, jinak též příkaz pro větvení programu nebo podmíněný příkaz je konstrukce, která zajistí provedení části programu jen při splnění stanovené podmínky. Ve své nejjednodušší podobě vypadá takto:

```
if a>b then
  b:=a;
  end if;
```

a slovy by se dal jeho význam popsat jako "Pokud platí podmínka `a` je větší než `b`, proveď příkaz `b:=a`".

V příkladu je uveden jediný příkaz, ve skutečnosti je ale možné uvést libovolné množství libovolných příkazů včetně dalších příkazů `if`, například:

```
if a>b then
  b:=a;
  c:=2*a;
  d:=x+y+z;
  if d>10 then  -- vnořený příkaz if
    r:=r/7.25;
    end if;
  end if;
```

Později uvidíme, že stejné pravidlo platí pro všechny ostatní strukturované příkazy ve Flexu.

Podmínka v příkazu `if` (stejně jako podmínka v kterémkoliv dalším příkazu) může obsahovat relační operátory:

operátor	název	příklad
>	větší než	`x > y`
<	menší než	`x < y`
=	je stejné	`x = y`
<>	je různé	`x <> y`
>=	větší nebo stejné	`x >= y`
<=	menší nebo stejné	`x <= y`

Tabulka č. 3. Relační operátory

a dále je možné podmínky sdružovat pomocí logických operátorů:

operátor	název	příklad	popis
`and`	logické and (disjunkce)	`x>0 and x<10`	provede se, pokud jsou splněny obě podmínky
`or`	logické or (konjunkce)	`x >0 or y>0`	provede se, pokud je splněna alespoň jedna z podmínek
`not`	negace	`not x>0`	provede se, pokud není splněna podmínka

Tabulka č. 4. Logické operátory

Stejně jako u aritmetických operátorů je možné použít závorky.

Použití operátorů v příkazu `if` pak vypadá třeba takto:

```
if (a>0 and a<10) or (b>0 and b<10) then
  b:=a;
  end if;
```

Oproti jiným jazykům je neobvyklé (avšak zdůvodnitelné větší odolností proti chybám z přehlédnutí) to, že není definována vzájemná priorita operátorů `and` a `or`. Důsledkem toho je, že ve výrazech obsahujících současně `and` i `or` musí být použity závorky:

```
(a=0 and b=0) or c=0
```

je přípustný zápis, zatímco v případě:

```
a=0 and b=0 or c=0
```

vyhlásí překladač chybu.

2.8 Další varianty příkazu if

Čistě teoreticky by pro veškeré větvení programu stačil výše popsaný tvar příkazu `if`, pro pohodlí programátorů jsou ale k dispozici jeho dvě další podoby.

```
if a>b
  then b:=a
  else a:=b
  end if;
```

znamená "Pokud platí podmínka `a>b` pak proveď příkaz `b:=a` jinak proveď `a:=b`".

Poslední forma příkazu `if` zavádí zkratku `elsif`:

```
if a=0 then x:=b
elsif a=1 then x:=c
elsif a=2 then x:=d
elsif a=3 then x:=e
else x:=0
end if;
```

Ekvivalentní zápis bez `elsif` by vypadal takto:

```
if a=0
 then x:=b
 else if a=1
      then x:=c
      else if a=2
           then x:=d
           else if a=3
                then x:=e
                else x:=0
                end if;
           end if;
      end if;
 end if;
```

Vidíme, že každé použití zkratky `elsif` místo `else if` znamená úsporu jednoho `end if`, což znamená celkové zpřehlednění programu.

2.9 Příkaz loop

Poslední ze základních druhů příkazů je příkaz cyklu, neboli konstrukce, která zajistí opakované provádění části programu. Ve své nejjednodušší podobě vypadá takto:

```
i:=0;
while i<10 loop
  i:=i+1;
  end loop;
```

a slovy by se dal jeho význam popsat jako "Dokud platí podmínka `i<10` prováděj příkaz `i:=i+1`". Je zřejmé, že podmínka dříve nebo později platit přestane – uvnitř cyklu se k proměnné `i` přičítá jednička, neboli jinými slovy po každém průchodu cyklem je `i` zvýšeno o jedničku. V tomto příkladu je počáteční hodnota `i` nula, proto se tělo cyklu (příkazy mezi `loop` a `end loop`) provede desetkrát.

Pokud přičítání jedničky (nebo jakoukoliv jinou operaci, která má vliv na podmínku) zapomenete do těla cyklu naprogramovat, jako třeba zde

```
i:=0;
while i<10 loop
  end loop;
```

bude se cyklus provádět nekonečně dlouho, nebo přesněji řečeno dokud nevypnete počítač nebo dokud program ručním zásahem z task manageru neukončíte. O "nekonečném" cyklu se zmiňujeme proto, že jde o jednu z nejběžnějších programátorských chyb a raději dopředu počítejte s tím, že se vám jej "povede" naprogramovat v nejrůznějších podobách mnohokrát. Někdy je ovšem nekonečný cyklus žádoucí, pak stačí použít formu:

```
loop...
  end loop;
```

2.10 Cyklus until

Kromě cyklu s podmínkou na začátku (označujeme jej také jako cyklus `while`) existuje cyklus s podmínkou na konci (cyklus `until`).

```
i:=0;
loop
  i:=i+1;
  until i>=10;
```

Od předchozího cyklu se liší jednak umístěním podmínky, jednak tím, že cyklus skončí až bude podmínka poprvé splněna.

2.11 Cyklus for

Pokud předem víte nebo umíte vypočítat kolikrát se má cyklus provést, můžete použít cyklus `for`.

```
for i in 1..10 loop...
  end loop;
```

Cyklus `for` se typicky používá pro práci s poli, jak je ukázáno v následujících kapitolách.

2.12 Pole

Dosud jsme se zabývali jednoduchými datovými typy (číselnými a znakovými). Jako nadstavba těchto typů existují typy složené (nebo též strukturované), které se skládají z několika jednoduchých typů. Prvním z nich je typ pole.

Pole si lze představit na příkladu aplikace internetového obchodu. Zákazník si vybírá různé zboží, které přidává do nákupního košíku. Nákupní košík není nic jiného, než několik proměnných typu přirozené číslo, do kterých se ukládají katalogová čísla vybraných položek zboží. Řešení bez použití pole by bylo poněkud těžkopádné:

```
var
  kosik_polozka_1: natural;
  kosik_polozka_2: natural;
  kosik_polozka_3: natural;
  pocet_polozek: natural:= 0;
  vybrane_zbozi: natural;...
  -- někde výše bylo do proměnné vybrane_zbozi uloženo katalogové
  -- číslo zboží, teď jde už jen o to jej přidat do košíku

  if pocet_polozek=0 then
    pocet_polozek:=pocet_polozek+1;
    kosik_polozka_1:=vybrane_zbozi;

  elsif pocet_polozek=1 then
    pocet_polozek:=pocet_polozek+1;
    kosik_polozka_2:=vybrane_zbozi;

  elsif pocet_polozek=2 then
```

```
  pocet_polozek:=pocet_polozek+1;
  kosik_polozka_3:=vybrane_zbozi;

else
  -- ohlásit chybu, že se další zboží do košíku už nevejde
  end if;
```

Navíc by se dalo rozumně naprogramovat jen pro velice malý košík. Představte si jen tu práci, kdyby se do košíku mělo vejít až 100 položek zboží! A těch problémů, kdyby si zadavatel vzpomněl, že si přeje zvětšit obsah košíku na tisíc položek!

Naštěstí máme ve Flexu k dispozici pole. Pole si můžeme představit jako řadu stejně pojmenovaných proměnných, které se liší jen číslem. To sice platí pro proměnné `kosik_polozka_n` v předchozím příkladu také, ale pole poskytuje jednu podstatnou výhodu navíc: číslo položky nemusíme zadávat přímo v programu, ale můžeme jej vypočítat. Více než další teoretizování ukáže stejný příklad, ale upravený pro použití pole:

```
var
  kosik: array 1..3 of natural; -- pole proměnných typu natural (počet prvků
                                -- pole je roven velikosti košíku, typ prvku
                                -- pole je zvolen tak, aby se do něj dalo
                                -- uložit číslo zboží)

  pocet_polozek: natural:= 0;  -- kolik je v košíku uloženo položek zboží

  vybrane_zbozi: natural;      -- které zboží bylo vybráno...
  -- někde výše bylo do proměnné vybrane_zbozi uloženo katalogové
  -- číslo zboží, teď jde už jen o to jej přidat do košíku

  if pocet_polozek<3 then
    pocet_polozek:=pocet_polozek+1;
    kosik[pocet_polozek]:=vybrane_zbozi;
  else
    -- ohlásit chybu, že se další zboží do košíku už nevejde
    end if;
```

Vidíte, že přidání položky do košíku se scvrklo na dva příkazy. Navíc při požadavku zvětšení košíku stačí místo trojky napsat do programu třeba 1000 a nic jiného se už měnit nemusí.

2.13 Konstanty

Vraťme se k příkladu z poslední kapitoly. Hovoří se tam o tom, že zvětšit nákupní košík ze 3 na 1000 položek je otázka změny jednoho čísla, ale to není tak úplně pravda. Jistě vidíte, že číslo se v našem příkladu vyskytuje dvakrát (a ve skutečném programu třeba stokrát) a snadno se může stát, že jej programátor zapomene změnit na všech místech. Dobře a čistě napsaný program s takovou možností samozřejmě počítá a zajistí, aby k podobné chybě nemohlo dojít. V tomto případě patří k dobrému programátorskému stylu uvést konkrétní číslo jako pojmenovanou konstantu a v celém programu se odkazovat jen na její jméno, nikoliv konkrétní hodnotu.

Příklad s využitím konstanty vypadá takto:

```
const
  velikost_kosiku = 3; -- počet kusů zboží,
                       -- které se vejdou do nákupního košíku

var
  kosik: array 1..velikost_kosiku of natural;
  pocet_polozek: natural:= 0;
  vybrane_zbozi: natural;...
  -- někde výše bylo do proměnné vybrane_zbozi uloženo katalogové
  -- číslo zboží, teď jde už jen o to jej přidat do košíku

  if pocet_polozek<velikost_kosiku then
    pocet_polozek:=pocet_polozek+1;
    kosik[pocet_polozek]:=vybrane_zbozi;
  else
    -- ohlásit chybu, že se další zboží do košíku už nevejde
    end if;
```

Nyní už jsme dosáhli programátorského ideálu. Stačí změnit jediné číslo na jediném místě a zákazníkem požadovaná úprava programu je hotova.

2.14 Datové typy

Dosud jsme se s datovými typy setkávali jenom v souvislosti s proměnnými. Například proměnná `kosik` z předchozího příkladu byla datového typu "pole přirozených čísel". Představte si situaci, kdybychom měli nákupní košíky dva a obsah jednoho z nich chtěli celý zkopírovat do druhého – třeba proto, aby si uživatel mohl posledních pár položek nákupu rozmyslet a kliknutím na tlačítko se vrátil ke stavu košíku, jaký měl před chvílí. Z programátorského hlediska to není nic složitého, prostě by se starší kopie košíku zkopírovala zpět do aktuálního košíku. Jak to ale udělat? Primitivní řešení je vytvořit dvě proměnné a pěkně je položku po položce zkopírovat, asi takto:

```
var
  kosik: array 1..velikost_kosiku of natural;
  kopie: array 1..velikost_kosiku of natural;

for i in 1..velikost_kosiku loop
  kopie[i]:=kosik[i];
  end loop;
```

Příklad č. 23. řešení s poli různého typu

Řešení je sice funkční, ale – jak už to tak v programování bývá – přesto se nedá považovat za správné. Postupně si probereme jednotlivé kroky, jak a proč jej zlepšit. První detail, kterého si asi všimnete je zcela zbytečné dvojí opakování textu `array 1..velikost_kosiku of natural`. Odstranit toto opakování je přitom snadné, stačí datový typ nákupního košíku definovat a pojmenovat zvlášť a potom už se na něj odkazovat pouze jeho jménem.

Deklarace proměnných by potom vypadala takto:

```
type
  nakupni_kosik = 1..velikost_kosiku of natural;

var
  kosik: nakupni_kosik;
  kopie: nakupni_kosik;
```

Příklad č. 24. řešení s poli stejného typu

Ušetřili jsme opakovaný popis typu, což by mohlo být v opravdových programech, kde se taková situace opakuje mnohokrát, docela příjemné. Navíc jsme dosáhli jednoho příznivého vedlejšího efektu: pokud někdy v budoucnu definici nákupního košíku změníme, stačí ji přepsat na jediném místě, což kromě pohodlí odstraní i nepříjemnou možnost, že bychom na některou proměnnou zapomněli a omylem ji nechali v původním tvaru.

Ušetřit opakovaný popis typu je sice užitečné, ale není to zdaleka všechno. Pojmenováním nějakého datového typu jsme totiž vytvořili typ nový, pro který platí podobná pravidla jako pro již existující jednoduché typy. V tomto okamžiku je z nich nejzajímavější to, že proměnné stejného typu lze vzájemně přiřadit stejně jako třeba obyčejná přirozená čísla. Místo kopírování v cyklu složku po složce teď stačí napsat:

```
kopie:=kosik
```

čímž se zkopírují všechny složky proměnné `kosik` do odpovídajících složek proměnné `kopie` najednou, jediným příkazem. To by u příkladu 23 možné nebylo, protože tam jsou proměnné `kosik` a `kopie` definovány sice na pohled stejně, ale z hlediska programovacího jazyka jde o dva různé typy, které nemohou být vzájemně přiřazeny jako celek (přiřazení po složkách ovšem možné je, protože složky jsou stejného typu).

Kromě pohodlí jsme touto cestou dospěli k jedné významné vlastnosti Flexu – typové kontrole. Díky ní není přípustné přiřadit například do proměnné typu nákupní košík třeba proměnnou typu seznam osobních čísel zaměstnanců, i když jsou obě definovány jako pole přirozených čísel. Překladač Flexu na takový omyl přijde již při překladu a programátora upozorní, že má v programu chybu nebo nějakou ne zcela logickou konstrukci. Typová kontrola představuje značný pokrok proti programovacím jazykům druhé generace (například Fortran, C) které tuto možnost kontroly z principu neobsahovaly.

2.15 Záznamy

Stejný příklad s internetovým obchodem použijeme i pro vysvětlení druhého ze strukturovaných datových typů, záznamu. V příkladu máme jako identifikaci zboží uvedeno katalogové číslo zboží. Takto navržený program vcelku vyhovuje, ale co když je jako zboží uveden hřebík a zákazník si jich kupuje tisíc? Je jasné, že nebudeme zákazníka nutit aby si tisíckrát za sebou koupil jeden hřebík, ale že budeme potřebovat kromě katalogového čísla ke každé položce v košíku uložit ještě počet kusů. Pro tento účel se výborně hodí datový typ záznam.

```
const
  velikost_kosiku = 3; -- počet kusů zboží
                       -- které se vejdou do nákupního košíku

type
  polozka_nakupniho_kosiku =
    kat_cislo: natural;
    pocet_kusu: natural;
    end record;

  nakupni_kosik = 1..velikost_kosiku of polozka_nakupniho_kosiku;

var
  kosik: nakupni_kosik;
  pocet_polozek: natural:= 0;...
  -- do nákupního košíku chceme přidat deset kusů zboží kat.č. 12345678
  pocet_polozek:=pocet_polozek+1;
  kosik[pocet_polozek].pocet_kusu:=10;
  kosik[pocet_polozek].kat_cislo:=12345678;
```

V příkladu se pro názornost přidává zboží přímo, což je v reálné aplikaci samozřejmě nesmysl. Ve skutečnosti bychom postupovali spíše touto cestou (začátek příkladu je shodný, proto je uvedena jen část počínaje deklarací proměnných):

```
var
  vybrane_zbozi: polozka_nakupniho_kosiku;
  kosik: nakupni_kosik;
  pocet_polozek: natural:= 0;...
  -- zjistit, jaké zboží a kolik kusů se má přidat do košíku
  vyber_zbozi(vybrane_zbozi); -- Volání procedury vyber_zbozi.
                              -- Jejími detaily se nebudeme zabývat,
                              -- stačí vědět, že naplní proměnnou
                              -- jejíž jméno je uvedeno jako parametr
                              -- číslem zboží a počtem kusů.


  pocet_polozek:=pocet_polozek+1;
  kosik[pocet_polozek]:=vybrane_zbozi;
```

3. Užitečné drobnosti

V předchozí kapitole jsme probrali základní datové typy a příkazy. Než se pustíme do dalších programových konstrukcí, ukážeme několik užitečných drobností a zjednodušení, které byly v zájmu srozumitelnosti předchozí kapitoly vynechány.

3.1 Zkrácené přiřazení

V příkladech v předchozí kapitole byl několikrát použit příkaz pro zvětšení hodnoty proměnné o jedničku:

```
a:=a+1;
```

Pokud jej rozebereme detailně, znamená "vypočítej výraz `a+1` a výsledek ulož do proměnné `a`". Konstrukce, ve které se jméno proměnné vyskytuje jak na levé straně přiřazovacího příkazu, tak na začátku jeho pravé strany, je v praxi velice častá. Konkrétní proměnná může být navíc určena i dosti složitým výpočtem, třeba nějak takto:

```
citac[poradi[x,y],rovina[a],polozka[(n+1) mod 7]]:=
citac[poradi[x,y],rovina[a],polozka[(n+1) mod 7]]+1;
```

což je nepřehledné a snadno se udělá chyba, která se pak velice těžko hledá. Flex proto zavádí tzv. zkrácený přiřazovací příkaz, který dovoluje konstrukci typu:

```
a:=a+1;
```

zapsat ve formě:

```
a+1;
```

Zkrácené přiřazení se dá použít nejen pro přičtení jedničky (nebo obecně přičtení čehokoliv), ale všechny aritmetické operace. Příklad přípustných zápisů:

```
a+100;
a+12-b;
citac[poradi[x,y],rovina[a],polozka[(n+1) mod 7]+1
```

3.2 Omezení rozsahu čísla

Již jsme se zmiňovali o technikách a postupech vhodných pro omezení počtu chyb v programech. Jedna z osvědčených metod je určení dolní a horní hranice čísla pomocí klíčového slova `range`:

```
type
  t_cas = record
    hodiny: natural range 0..23;
    minuty: natural range 0..59;
    sekundy: natural range 0..59;
    end record;
```

V příkladu je definován datový typ `t_cas` pro uložení časového údaje. Pokud tento typ použijete ve svém programu, bude se při každém přiřazení automaticky kontrolovat zda je přiřazovaný údaj v povoleném rozsahu. Pokud tedy například napíšete

```
var
  cas: t_cas;

begin
  cas.hodiny:=32;
```

překladač již v okamžiku překladu programu zjistí, že je něco v nepořádku a ohlásí chybu. Bez omezení pomocí `range` by zůstal překlep neodhalen a projevil by se až při testování hotového programu, v horším případě třeba u zákazníka o pár let později.

Ani když chyba není odhalena již při překladu (protože špatné číslo nezadáváte přímo, ale určujete jej výpočtem), nezůstáváte bez ochrany. Při každém přiřazení se automaticky kontroluje správnost rozsahu a při nepřípustné hodnotě je vyvolána výjimka (výjimky jsou popsány v referenční příručce, (viz kapitolu 9), což zatím znamená, že program skončí s chybovým hlášením.

Tuto kontrolu za běhu je ovšem možné vypnout (viz referenční příručka), aby například kriticky důležitý program nehavaroval kvůli banální chybě při vcelku nepodstatném zobrazování aktuálního času. Pro normální použití se ale doporučuje všechny kontroly ponechat zapnuté

3.3 Agregát

V předchozí kapitole jsme bez vysvětlení několikrát použili agregát. Agregátem je jazyková konstrukce pro přiřazení hodnoty strukturované proměnné, například poli:

```
var
  x: array 1..5 of natural;...
  -- naplnění pole bez použití agregátu
  x[1]:=10;
  x[2]:=20;
  x[3]:=30;
  x[4]:=40;
  x[5]:=50;

  -- naplnění pole pomocí agregátu
  x:=[10,20,30,40,50];
```

Podobné zjednodušení lze použít i pro záznam

```
var
  cas: t_cas; -- datový typ z předchozí kapitoly...
  -- z nějakého důvodu potřebujeme dosadit čas "těsně před půlnocí"
  -- naplnění záznamu bez použití agregátu
  cas.hodiny:=23;
  cas.minuty:=59;
  cas.sekundy:=59;

  -- naplnění záznamu pomocí agregátu
  cas:=[23,59,59];
```

Zdálo by se, že agregát není nic jiného, než zjednodušená forma zápisu pro pohodlí programátora. Ve skutečnosti ale má ještě jednu důležitou úlohu – je to další z forem ochrany proti chybám a opomenutím programátora. Představte si situaci, kdy v hotovém programu z nějakého důvodu uděláte změnu, datový typ `t_cas` doplníte ještě o tisíciny sekundy

```
t_cas = record
   hodiny: natural range 0..23; -- přípustné číslo hodiny v časovém údaji
   minuty: natural range 0..59; -- přípustné číslo minuty v časovém údaji
   sekundy: natural range 0..59; -- přípustné číslo sekundy v časovém údaji
   tisiciny: natural range 0..999; -- přípustné číslo tisiciny v časovém údaji
   end record;
```

ale zapomenete je dopsat na některá místa programu. Bez použití agregátu by mohla snadno vzniknout těžko odhalitelná chyba v chování programu, protože překladač by neměl informaci o tom, že chcete naplnit celý záznam a ve složce `tisiciny` by nechal jakési nedefinované číslo. Agregát je naopak přímo určen pro naplnění celého záznamu, takže by při překladu nastala chyba a překladač by vás "donutil" chybějící číslo doplnit:

```
cas:=[23,59,59,999];
```

Podobně by se zachoval i agregát pole při zvětšení počtu prvků pole.

3.4 Implicitní hodnoty

V příkladech jsme použili, i když bez bližšího vysvětlení, implicitní hodnoty proměnných ve tvaru

```
var
  i: natural:= 10;

begin...
```

což je totéž jako bychom použili proměnnou bez implicitní hodnoty a hned za příslušný begin napsali odpovídající přiřazení

```
var
  i: natural;

begin
  i:=10;...
```

S implicitními hodnotami se ale dají dělat i zajímavější věci. Zkusme například modifikovat výše definovaný záznam `t_cas` takto:

```
type
 t_cas = record
  hodiny: natural range 0..23:= 23;
  minuty: natural range 0..59:= 59:
  sekundy: natural range 0..59:= 59;
  end record;
```

Tím jsme přiřadili každé složce její implicitní hodnotu a deklarace proměnné

```
var
 cas: t_cas;
```

tak bude automaticky obsahovat inicializaci uvedenou v definici typu, to jest bude se chovat stejně jako kdybychom napsali

```
var
 cas: t_cas:= [23,59,59];
```

3.5 Zpřísněná typová kontrola

Flex patří mezi jazyky se silnou typovou kontrolu a mezi nimi k těm "přísnějším". Dokonce je vybaven jazykovými prostředky pro předepsání ještě striktnější kontroly, než je stanoveno implicitně. K čemu je to dobré?

Možná si vzpomenete na havárii americké výzkumné sondy Mars Polar Lander za nějakých 165 miliónů dolarů. Vyšetřování údajně ukázalo, že nehoda byla způsobena tím, že obslužný podprogram pro výškoměr vracel výšku ve stopách, zatímco podprogram pro řízení sestupu (napsaný jiným týmem programátorů) fungoval v metrech. Nikdo si nesouladu jednotek nevšiml a při skutečném sestupu byl přistávací manévr zahájen v příliš malé výšce a sonda se rozbila o povrch Marsu.

Těžko říct, jak to bylo doopravdy, ale faktem je, že moderní jazyky poskytují prostředky, které na podobný omyl automaticky upozorní. Ve Flexu je pro tento účel k dispozici klíčové slovo `protected`, které zajistí, že datový typ nebude kompatibilní s žádným jiným typem.

```
type
 metry          = protected real;
 feet           = protected real;

var
 vyska          = metry;   -- výška v metrech
 altitude       = feet;    -- výška ve stopách

begin
 vyska:=125.5;    -- přípustné
 altitude:=vyska; -- nepřípustné, překladač zde ohlásí chybu
```

Zpřísněná typová kontrola podobný omyl automaticky hlídá a upozorní programátora, že něco není s návrhem programu v pořádku.

3.6 Předčasné ukončení cyklu

Uvnitř každého cyklu je možné použít příkaz `break` pro okamžité opuštění cyklu.

```
loop
 if stisknuta_klavesa=Esc then break end if;
 zobraz_aktualni_stav;
 end loop;
```

Procedura `zobraz_aktualni_stav` se neustále opakovaně volá v nekonečném cyklu, který je možné přerušit stisknutím klávesy (**Esc**).

> Poznámka
> Pokud byste na základě analogie s C nebo Borland Pascalem hledali na tomto místě také "obrácený" příkaz continue, hledáte marně. Nejde totiž o komplementární příkaz, ale příkaz logicky náležející k příkazům prováděným uvnitř cyklu, který fakticky nemá s cyklem cokoliv společného. Neprovedení zbytku příkazů ve vnitřku cyklu dosáhnete pomocí obecně dostupných příkazů (`if`, `raise`a dalších).

3.7 Víceřádkové komentáře

V předchozích příkladech používáme jednořádkové komentáře které začínají dvojicí znaků `--`, po kterých se napsaný text až do konce řádku při překladu ignoruje. Kromě těchto komentářů je možné ve použít variantu s textem vloženým mezi složené závorky `{}`. Protože mezi složenými závorkami může ležet i znak pro konec řádku, říkáme tomuto typu komentáře víceřádkový. Na rozdíl od jednořádkového musí být víceřádkový komentář vždy ukončen, jinak bude celý program až do konce považován za komentář.

Víceřádkové komentáře se mohou vnořovat, což je užitečné například v případě, kdy chcete pomocí komentáře dočasně zneplatnit část zdrojového textu, který již nějaké (jedno- i víceřádkové) komentáře obsahuje.

```
-- toto je komentář

{ víceřádkový
komentář }

{ komentář { s vnořeným } komentářem }

{ ještě jeden příklad
-- složitější { kombinace
} -- komentářů
různého druhu }

A:= { komentář lze vkládat kamkoliv mezi lexikální elementy programu } 10;
A: { ale toto není dobře, protože:= je jediný lexikální element } = 10;
```

4. Procedury

S procedurami jsme se již letmo setkali v předchozím textu. Bylo to volání knihovních procedur jako třeba `write_line_txt`. Pokud jste se s procedurami dosud nesetkali v jiném programovacím jazyce, asi jste již na tomto příkladu intuitivně pochopili k čemu je procedura dobrá: je to část programu, která má vlastní jméno, která "něco užitečného" dělá a která se dá z jiného místa programu zavolat. Procedura také zpravidla mívá jeden nebo více parametrů, pomocí kterých se předávají hodnoty, které mají být procedurou zpracovány. Části zdrojového textu ve které je procedura popsána se říká deklarace procedury.

4.1 Procedura bez parametrů

Nejjednodušší procedura je procedura bez parametrů. V praxi se příliš často nepoužívá, protože postrádá nejzajímavější vlastnost procedury – zobecnění algoritmu pro různé vstupní hodnoty předávané právě pomocí parametrů. Přesto najde uplatnění, třeba v případech podobných tomu z následujícího příkladu.

Chcete vypisovat datum a čas kdy byl váš program spuštěn a kdy skončil. To je požadavek běžný a snadno řešitelný a bez použití procedur by mohl být vyřešen nějak takto:

```
program vypis =

with
  support;

begin
  write_txt('Program spuštěn ');
  write_txt('dne ');
  write_txt(get_date);
  write_txt(' v ');
  write_txt(get_time);
  write_txt(' hod.');
  new_line;... -- tady probíhá vlastní program, pro nás teď nezajímavý

  write_txt('Program ukončen ');
  write_txt('dne ');
  write_txt(get_date);
  write_txt(' v ');
  write_txt(get_time);
  write_txt(' hod.');
  new_line;
  end vypis;
```

Příklad č. 45. výpis, první verze

Jistě jste si všimli, že se mnoho řádků v programu opakuje, navíc se kód podružného významu plete do těla programu ve kterém jde o něco zcela jiného, musí zde být definovány proměnné atd. Přesunutí opakujícího se kódu do procedury přinese významné zlepšení:

```
program vypis =

with
  support;

procedure zobraz_cas =

begin
  write_txt('dne ');
  write_txt(get_date);
  write_txt(' v ');
  write_txt(get_time);
  write_txt(' hod.');
  new_line;
  end zobraz_cas;


begin
  write_txt('Program spuštěn ');
  zobraz_cas; -- volání procedury pro zobrazení času... -- tady probíhá vlastní program, pro nás teď nezajímavý

  write_txt('Program ukončen ');
  zobraz_cas; -- volání procedury pro zobrazení času
  end vypis;
```

Příklad č. 46. výpis, druhá verze

V druhé verzi příkladu zmizely zbytečně se opakující řádky a hlavní program se zjednodušil. Navíc jsme získali jednu podstatnou výhodu: při případné budoucí úpravě formátu vypisovaného data a času bude stačit upravit pouze tuto proceduru a nebudeme muset v celém programu vyhledávat a upravovat úseky kódu pro zobrazení data a času.

4.2 Parametry

Asi vás napadlo, že zobrazení času v předchozím příkladu je napsané poněkud neohrabaně. Kousek je rozepsán přímo v kódu, zbytek "doražen" voláním procedury. Nabízí se čistší řešení – do procedury přesunout i zobrazení úvodního textu. Text je ovšem nutné proceduře nějak předat, protože na začátku i na konci programu je jiný a obecně může být libovolný v každém dalším případě, kdy bude tato procedura volána. Pro tyto účely existuje možnost předání parametru do procedury. Nová verze s využitím parametru bude vypadat takto:

```
program vypis =

with
  support;

procedure zobraz_cas (udalost: text) =

begin
  write_txt(udalost);
```

```
  write_txt('dne ');
  write_txt(get_date);
  write_txt(' v ');
  write_txt(get_time);
  write_txt(' hod.');
  new_line;
  end zobraz_cas;


begin
  zobraz_cas('Program spuštěn'); -- volání procedury pro zobrazení času... -- tady probíhá vlastní program, pro nás teď nezajím

  zobraz_cas('Program ukončen'); -- volání procedury pro zobrazení času
  end vypis;
```

Příklad č. 47. výpis, třetí verze

Parametru uvedenému v hlavičce procedury se říká formální parametr. Určuje jméno parametru pod kterým bude jeho hodnota dostupná uvnitř procedury, jeho datový typ a ještě některé další vlastnosti, které budou vysvětleny v následujících odstavcích.

Parametru uvedenému v příkazu volání procedury se říká skutečný parametr. Určuje jaká hodnota (výraz, proměnná) bude předána proceduře ke zpracování.

V příkladu má proceduřa `zobraz_cas` definován jeden formální parametr pojmenovaný `text`. Při prvním volání této procedury se jako hodnota parametru dosadí text 'Program spuštěn' a procedura proběhne tak, jako kdyby bylo všude v jejím těle na místo jména `text` uvedeno přímo 'Program spuštěn'. Obdobně při druhém volání se dosadí text 'Program ukončen' a v proceduře se místo formálního parametru [text] použije jeho skutečná hodnota 'Program ukončen'.

4.3 Mód parametru

V předchozím odstavci je formální parametr [text] definován jako vstupní parametr. Znamená to, že je možné pomocí něj předat data do procedury, ale uvnitř procedury se na parametr hledí jako na konstatnu – může být použit ve výrazu, ale nesmí se nijak měnit.

Existuje ještě druhý režim parametru – vstupně výstupní. Data předaná jako vstupně výstupní parametr je možné v proceduře změnit, tj. do tohoto parametru lze uvnitř procedury přiřadit nějakou hodnotu. Použití si ukážeme na příkladu procedury která vypíše zadaný text na obrazovku a zároveň počítá, kolik řádků textu vypsala.

```
program parametry =

with
  support;

-- vypsat zadaný text a do zadané proměnné přičíst jedničku
procedure vypis (udaj: in text; pocet: in out integer) =
begin
  pocet+1;
  write_line_txt(udaj);
  end vypis;
```

```
var
  -- deklarujeme proměnnou pro počítání zobrazených řádků
  -- (do začátku samozřejmě musí být vynulovaná)
  pocet: integer:= 0;

begin
  vypis('A',pocet);
  vypis('B',pocet);
  vypis('C',pocet);
  end parametry;
```

Příklad č. 48. mód parametru

Procedura `vypis` v příkladu má dva parametry, jeden vstupní a druhý vstupně výstupní. Vstupní parametr je označen klíčovým slovem `in`, vstupně výstupní se označuje klíčovými slovy `in out`. Klíčové slovo pro označení módu parametru smí být vynecháno, v takovém případě se parametr chápe jako vstupní (jinými slovy mód parametru je implicitně `in`).

Všimněte si odlišnosti v použití parametrů při volání procedury. Skutečný parametr odpovídající vstupnímu formálnímu parametru může být jakýkoliv výraz, jehož výsledkem je hodnota příslušného typu. Naproti tomu za vstupně výstupní formální parametr se může jako skutečný parametr dosadit pouze proměnná. Při volání procedury se hodnota vstupního parametru vypočte a do procedury se pouze předá výsledek, vstupně výstupní parametr se předá "tak jak je" (viz poznámku) a procedura jeho hodnotu používá a mění podle libosti – nebo spíše podle toho jak je napsaná.

> Poznámka
> Ujištění pro znalce starodávného jazyka Algol 60: ne, opravdu nemáme na mysli předání jménem:-)

4.4 Funkce

Funkce, vlastně procedura, která předává pod svým jménem jednu hodnotu zpět do volajícího programu – přibližně tak, jak jsme zvyklí u funkcí v matematice. Vše nejlépe objasní příklad:

```
program funkce =

procedure prumer (a: integer; b: integer) return integer =
begin
  result:=(a+b) div 2;
  end prumer;

var
  x: integer;

begin
  x:=prumer(3,5); -- do x se přiřadí 4
  x:=prumer(3,5)+5; -- do x se přiřadí 9
  x:=prumer(2,prumer(3,5)); -- do x se přiřadí 3
  end funkce;
```

Příklad č. 49. funkce

V příkladu je deklarována procedura `prumer`, která má dva vstupní celočíselné parametry (typu `integer`) a funkční hodnotu také typu `integer` (formálně vzato jde vlastně o třetí parametr se speciálním módem). Proměnná, ve které je uložena funkční hodnota (a do které také musíte funkční hodnotu přiřadit, má-li být funkcí vrácena) je zastoupena klíčovým slovem `result`.

Funkci můžete použít všude tam, kde je přípustné použití konstanty resp. výrazu stejného typu. Funkce jsou vhodný nástroj pro matematické výpočty, protože je možné je přímo používat ve výrazech bez nutnosti ukládat mezivýsledky do zvláštních proměnných.

Poznámka

V Flexu se automaticky předpokládá, že funkce vrací v rámci jednoho výrazu při stejných parametrech vždy stejný výsledek. Jinak řečeno to znamená, že například výraz `ziskej_cislo(x)+ziskej_cislo(x)` překladač považuje za plně ekvivalentní výrazu `2*ziskej_cislo(x)` a podle toho také provádí případné optimalizace výrazů. Oblíbená "céčková" konstrukce `getbyte(soubor)+256*getbyte(soubor)` je tedy ve Flexu chybná!

5. Užitečné drobnosti pro psaní procedur

5.1 Implicitní parametry

Představte si, že máte proceduru deklarovanou jako

```
otevri_okno(velikost_vodorovne: integer;
        velikost_svisle: integer;
        barva_textu: barva;
        barva_pozadi: barvy);
```

a skoro pokaždé když ji voláte chcete černý text na bílém pozadí. Neustálé opakování příkazu

```
otevri_okno(100,75,cerna,bila);
```

ve kterém se mění pouze první dvě čísla je poněkud nudné. Pro tento případ máte k dispozici implicitní parametry, díky kterým můžete neustále opakovanou hodnotu uvést přímo do deklarace parametru a při volání ji prostě vynechat:

```
otevri_okno(velikost_vodorovne: integer;
        velikost_svisle: integer;
        barva_textu: barva:=cerna;
        barva_pozadi: barvy:=bila);
```

Normální volání potom vypadá třeba takto:

```
otevri_okno(100,75);
```

a barvy uvedete pouze když se liší od implicitních:

```
otevri_okno(100,75,zelena);
otevri_okno(100,75,zelena,ruzova);
otevri_okno(100,75,,ruzova);
```

Všimněte si dvou čárek za sebou v posledním případě. Nebyl uveden předposlední parametr, ale čárky oddělující parametry musely v tomto případě zůstat, aby překladač poznal, který parametr jste vynechali. Pro úplnost je nutné se zmínit, že je možné uvést i nepovinné čárky za posledním skutečným parametrem:

```
otevri_okno(100,75,zelena,);
```

5.2 Klíčové parametry

Širším využíváním implicitních parametrů se snadno dostanete do situace podobné následující:

```
otevri_okno(velikost_vodorovne: integer;
        velikost_svisle: integer;
        vzor_pozadi: vzory:=kosticky;
        ramecek: ramecky:=dvojity;
        rohy_ramecku: rohy:=ostre;
        barva_ramecku: barvy:=fialova;
        nadpis_ramecku: nadpisy:='';
        barva_nadpisu: barvy:=cervena;
        barva_textu: barvy:=cerna;
        barva_pozadi: barvy:=bila);
```

Při volání pak potřebujete změnit jen poslední z implicitních parametrů. Postupem známým z předchozího odstavce byste se dostali k zápisu

```
otevri_okno(100,75,,,,,,,,zelena);
```

Poznáte zpětně na první pohled ze zdrojového textu, jestli jste chtěli zelený text nebo zelené pozadí? Znamenalo by to, že budete muset pracně počítat čárky a když později přidáte do procedury další implicitní parametr, budete možná muset do každého volání procedury jednu čárku připsat, jinak bude program dělat něco jiného než jste zamýšleli. Flex má pro tuto situaci řešení v podobě klíčových parametrů. S jejich použitím by předchozí příkaz vypadal takto:

```
otevri_okno(100,75,for barva_pozadi use zelena);
```

Volání "s čárkami" (terminologií Flexu s použitím pozičních parametrů) a s klíčovými parametry lze kombinovat

```
otevri_okno(100,75,kosticky,,ostre,
        for barva_textu use oranzova,
        for barva_pozadi use zelena);
```

jenom je nutné vzít v úvahu, že když ve volání procedury použijete klíčový parametr, následující parametry mohou být opět pouze klíčové.

5.3 Předčasné opuštění procedury

Občas se stane, že již někde uprostřed procedury zjistíte, že nemá cenu ji dále provádět. Mohou být špatná data nebo už jste vypočítali co bylo potřeba a pokračování je zbytečné. Pro tyto účely je ve Flexu k dispozici příkaz `return`, který způsobí okamžitý návrat z procedury zpět do volajícího programu, stejně jako kdyby procedura došla až ke svému `end`

```
procedure vypocet (x: integer) return integer =
begin
  -- v případě nepřípustné hodnoty skončíme
  if x<0 or x>521 then
    result:=0; -- tohle se má vrátit při chybě
    return;
    end if;
```

```
    -- normální výpočet, pro nás teď nezajímavý...
    end vypocet;
```

Povšimněte si, že klíčové slovo `return` je použito dvakrát ve zcela rozdílných významech. V prvním případě je to již známá konstrukce funkce, kdy `return` uvádí typ návratové hodnoty, v druhém případě jde o příkaz způsobující okamžité ukončení provádění procedury. Obě využití `return` spolu jinak nemají nic společného, jde o pouhou "recyklaci" klíčových slov, která mívají ve Flexu několik různých významů, pokud nehrozí nebezpečí záměny.

6. Třídy a objektové programování

Objektové programování je jeden z možných způsobů strukturování programu. Jeho základním principem je výhradně nepřímý přístup k datům – s daty se manipuluje pouze pomocí vyhrazených metod (což jsou vlastně procedury, jenom se jim tak neříká) a programátor tak při práci s daty nemusí (vlastně "nesmí") znát jejich vnitřní strukturu. Tento princip je vyjádřen termínem zapouzdření, jedním ze tří magických slůvek, do kterých se obvykle vlastnosti objektů shrnují.

Objektově je možné programovat prakticky v libovolném programovacím jazyce, například v dosud představené podmnožině Flexu. Data by byla definována vždy jako datový typ record a metody by byly realizovány pomocí procedur. Zbytek by záležel na správném návrhu programu a kázni programátora. Bylo by to ovšem pracné a i zbytečné, protože ve Flexu jsou k dispozici výrazové prostředky určené přímo pro objektové programování.

S objektovým programováním se spojuje dědičnost, druhé z magických slůvek. Dědičnost je vlastně jazyková konstrukce (či přesněji řečeno celá skupina konstrukcí a pravidel v jazyce), která umožňuje z jednoho objektu ("předka") odvodit jiný ("potomka"), který má všechny vlastnosti předka ("zdědí je"). Při tom mohou být přidány nové vlastnosti, které zděděné vlastnosti rozšiřují nebo některé z nich nahrazují novými. Ve skutečnosti nemá dědičnost s principem objektové struktury programu nic společného, dá se v jazyce definovat zcela nezávisle na objektovém přístupu k programování a je možné ji využívat i bez objektů a naopak objektově programovat bez dědičnosti. Zkombinováním principu dědičnosti a zapouzdření však vznikne daleko mocnější nástroj než při jejich osamoceném použití, proto asi byla zařazena do základních vlastností objektového programování.

Logickým důsledkem vypracování myšlenky dědičnosti k dokonalosti je vlastnost nazývaná třetím magickým slůvkem polymorfismus. Jde pouze o to, že potomek může být použit všude tam, kde smí být použit jeho předek; pokud ovšem má oproti předkovi některé vlastnosti změněné, bude se chovat odlišně. Obdobně jako dědičnost není polymorfismus nezbytným atributem objektového programování, ale v kombinaci s ním dále obohacuje výrazové prostředky programovacího jazyka a vytváří tak moderní programovací nástroj nové generace.

Později uvidíme, že podobně zajímavých a důležitých vlastností existuje ještě několik, bohužel však na ně nezbylo žádné magické slůvko a neměly tak to štěstí aby se dostaly přímo do "definice" objektového programování.

6.1 Třídy

Název objektové programování (též objektově orientované programování, OOP) byl zvolen poněkud nešťastně, protože zavádí druhý význam pro termín objekt, který je odedávna vyhrazen pro proměnné uložené v paměti počítače. Budeme se držet této dávné terminologie a pro "objekty" objektového programování budeme používat modernější termín třída (class), jak jej ostatně používá i řada dalších programovacích jazyků. Termín objekt (object, memory object) tak zůstává k dispozici pro proměnné, nebo přesněji a formálněji řečeno instance jakéhokoliv datového typu (včetně třídy) v paměti počítače.

Třídě odpovídá jazyková konstrukce na první pohled velmi podobná definici datového typu záznam:

```
class obdelnik =
  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku
  end obdelnik;
```

Stejně jako u záznamu představují `x` a y složky, ke kterým je možné přistupovat úplně stejně jako ke složkám záznamu. Potud je možné používat záznam i třídu shodně (ještě s výjimkou agregátů, které pro třídy neexistují). Třída má ale na rozdíl od záznamu navíc jednu zcela zásadní vlastnost – lze ji doplnit o metody pro práci s jednotlivými složkami tohoto "záznamu":

```
class obdelnik =
  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku

  -- metoda pro výpočet plochy obdélníku
  static plocha return integer =
  begin
    result:=x*y;
    end plocha;

  end obdelnik;
```

Uvedená třída `obdelnik` tak představuje datový typ do kterého je možné uložit rozměry obdélníku a který je navíc svázán s metodou pro výpočet jeho plochy. Pro zápis metody platí podobná pravidla jako pro zápis procedury, pouze místo klíčového slova `procedure` se použije klíčové slovo `static`. Volání této metody je ale poněkud odlišné, jak ukáže další příklad:

```
var
  obd: obdelnik; -- nejdříve si vytvoříme proměnnou typu obdelnik,
                 -- neboli odborně řečeno jednu instanci
                 -- třídy obdelnik
begin
  obd.x:=10;
  obd.y:=20;
  write_line_int(obd.plocha); -- volání metody plocha
```

Možná jste si všimli jistého rozporu mezi úvodem a příkladem. Říkáme, že data mají být ve třídě zapouzdřená, ale ke složkám jako `obd.x` přistupujeme přímo. Máte pravdu, je to sice technicky možné, ale není to příliš čisté. Napravíme to v další verzi příkladu:

```
program tridy =

with
  support;

class obdelnik =
  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku

  -- metoda pro nastavení rozměru obdélníku
  static rozmer (sirka: integer; vyska: integer) =
```

```
  begin
    x:=sirka;
    y:=vyska;
    end rozmer;

  -- metoda pro výpočet plochy obdélníku
  static plocha return integer =
  begin
    result:=x*y;
    end plocha;

  end obdelnik;

var
  obd: obdelnik; -- instance třídy obdelnik

begin
  obd.rozmer(10,20);
  write_line_int(obd.plocha);
  end tridy;
```

Příklad č. 64. třídy

Takhle je to již správné a čisté a zdá se, že problém zapouzdření je již zcela objasněn. Ale opak je pravdou – to zajímavější teprve přijde. Představte si, že třídu `obdelnik` již používáte na mnoha místech v rozsáhlém programu a často při tom využíváte metodu `plocha` (naproti tomu velikosti obdélníku měníte jen málokdy). Všechno funguje, ale zjistili jste, že násobení trvá neúměrně dlouho a výpočet plochy tak celý program značně zpomaluje. Díky zapouzdření odstraníte problém velice snadno na pár řádcích, aniž byste se museli starat o použití třídy v celém programu. Násobení prostě uděláte předem již při zadávání rozměru a při zjišťování plochy jen přeberete hotový výsledek. Upravená třída pak může vypadat třeba takto:

```
class obdelnik =
  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku
    s: integer; -- předvypočítaná plocha obdélníku

  -- metoda pro nastavení rozměru obdélníku
  static rozmer (sirka: integer; vyska: integer) =
  begin
    x:=sirka;
    y:=vyska;
    s:=x*y;  -- předvypočítat plochu
    end rozmer;

  -- metoda pro výpočet plochy obdélníku
  static plocha return integer =
  begin
    result:=s; -- jenom převzít již vypočtenou plochu
    end plocha;
```

```
end obdelnik;
```

Příklad č. 65. jiná varianta předchozí třídy

Použití zustává naprosto stejné, nikde jinde v programu se nemuselo nic měnit ale běh programu se zrychlil. Pokud si místo násobení představíte nějaký opravdu složitý výpočet, urychlení programu může být veliké.

6.2 Omezení viditelnosti

V předchozí kapitole jsme ukázali jednu z velkých výhod využití zapouzdření – snadnou modifikaci chování nebo jen způsobu implementace třídy na jednom jediném místě. Zároveň jsme ale naznačili problém s možným nedodržením konvencí. Co když nějaký z programátorů kteří na programu pracují porušil pravidla a dosazuje třeba rozměry obdélníku přímo? Budeme muset při posledním vylepšení třídy hledat všechna taková místa a opravit je nebo do nich doplnit výpočet plochy? Naštěstí nikoliv, protože Flex obsahuje prostředky pomocí kterých je možné omezit viditelnost složek třídy z jiných míst programu.

Třídu rozdělíme klíčovým slovem `private` na dvě části: vše nad slovem `private` je volně k dispozici programátorům, pod slovem `private` je určeno pouze pro vnitřní potřeby třídy.

```
class obdelnik =

  -- hlavička metody pro nastavení rozměru obdélníku
  static rozmer (sirka: integer; vyska: integer);

  -- hlavička metody pro výpočet plochy obdélníku
  static plocha return integer;

private

  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku
    s: integer; -- předvypočítaná plocha obdélníku

  -- tělo metody pro nastavení rozměru obdélníku
  static rozmer =
  begin
    x:=sirka;
    y:=vyska;
    s:=x*y;  -- předvypočítat plochu
    end rozmer;

  -- tělo metody pro výpočet plochy obdélníku
  static plocha =
  begin
    result:=s; -- jenom převzít již vypočtenou plochu
    end plocha;
```

```
end obdelnik;
```

Příklad č. 66. omezení viditelnosti

Takto upravená třída již nedovolí nikomu napsat například

```
var
  obd: obdelnik;

begin
  obd.x:=10;
  obd.y:=20;
```

Na řádku `obd.x:=10;` vyhlásí překladač chybu a odmítne program přeložit, dokud jej nepřepíšete do tvaru

```
var
  obd: obdelnik;

begin
  obd.rozmer(10,20);
```

Nyní si můžete být jisti, že nikde v programu není nedovolená konstrukce použita a můžete třídu v klidu modifikovat.

6.3 Dědičnost

V úvodu zmíněný princip dědičnosti je ve svém základu vlastně velice jednoduchý, jak je nejlépe vidět na příkladu.

```
class obdelnik =

  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku

  -- metoda pro nastavení rozměru obdélníku
  static rozmer (sirka: integer; vyska: integer) =
  begin
    x:=sirka;
    y:=vyska;
    end rozmer;

  -- metoda pro výpočet plochy obdélníku
  static plocha return integer =
  begin
    result:=x*y;
    end plocha;

  end obdelnik;
```

```
class kvadr =
  extend obdelnik;

  var
    z: integer; -- hloubka kvádru

  -- metoda pro nastavení rozměru kvádru
  static rozmer (sirka: integer; vyska: integer; hloubka: integer) =
  begin
    x:=sirka;
    y:=vyska;
    z:=hloubka;
    end rozmer;

  -- metoda pro výpočet povrchu kvádru
  static plocha return integer =
  begin
    result:=2*(x*y+y*z+x*z);
    end plocha;

  -- metoda pro výpočet objemu kvádru
  static objem return integer =
  begin
    -- objem kvádru je plocha základny krát hloubka
    result:=x*y*z;
    end objem;

  end kvadr;
```

Příklad č. 69. dědičnost

V příkladu je nejdříve zopakována definice třídy `obdelnik` z příkladu 64. Pro lepší přehlednost byla zvolena starší verze této třídy, ale příklad by stejně dobře fungoval i s následujícími verzemi.

Následně je odvozena třída `kvadr`. Její definice začíná klauzulí `extend obdelnik;`, která říká, že třída bude odvozena z třídy `obdelnik`. Všimněte si, že odvození nebylo zvoleno náhodně, ale je založeno na logické souvislosti mezi obdélníkem a kvádrem. Kvádr se dá chápat jako nadstavba nad obdélníkem – vznikne doplněním obdélníka o další vlastnost, o třetí rozměr.

Třída `kvadr` ovšem nemůže zůstat zcela shodná s třídou `obdelnik`, protože kvádr má o jeden rozměr více. Doplníme proto složku `z`, ve které bude uložena hloubka kvádru. Dále musíme nově nadefinovat metodu `rozmer`, která teď bude mít tři parametry (šířku,výšku a hloubku kvádru). Metoda `plocha` bude bez parametrů stejně jako u obdélníku, ale bude počítat povrch celého kvádru. Nakonec doplníme zcela novou metodu `objem`, která vypočte objem kvádru jako plochu základny krát hloubku.

Kdybychom odvozovali dále, mohli bychom z obdélníka odvodit třeba třídu `jehlan` a pak bychom dostali zárodek něčeho, čemu se v objektovém programování říká hierarchie: jeden předek (`obdelnik`) ma dva potomky (`kvadr`, `jehlan`) které rozdílně implementují metodu `objem`. Pokus o další pokračování odvozováním nových objektů by ale asi brzy odhalil slabinu v samém kořenu hierarchie: jak z obdélníku odvodit třeba kouli? Ukazuje se, že kořen hierarchie byl zvolen poněkud nešťastně, `obdelnik` má už příliš mnoho konkrétních vlastností než aby bylo možné prostým doplňováním nových vytvořit popis řady geometrických těles. Možným řešením by bylo zvolit jako kořen například třídu popisující jakýsi abstraktní geometrický tvar nebo třeba bod v prostoru – to by už záleželo na analýze problému a na předvídavosti programátora.

6.4 Polymorfismus

Mějme proceduru, jejímž parametrem je třída `obdelnik` z příkladu 69:

```
-- Procedura pro výpočet hmotnosti tělesa vyrobeného ze železného plechu.
-- Parametr sila je sila plechu, vysledek je hmotnost výrobku.
procedure hmotnost(vyrobek: in class obdelnik;
                   sila: in integer:= 1)
                   return integer =
begin
  -- Pro zjednodušení počítáme s hmotností 7 gramů na krychlový
  -- centimetr, snad to nikoho neurazí. Výsledek je v bůhvíčem, snad
  -- v miligramech, ale na tom teď nezáleží.
  result:=vyrobek.plocha * sila * 7;
  end hmotnost;
```

Procedura je definována pro třídu `obdelnik`. Na tom není nic divného, ale teď přijde půvab polymorfismu: kromě obdélníku je použitelná pro kteréhokoliv potomka! Schválně zkusme vypočítat

```
var
  obd: obdelnik;
  kva: kvadr;

begin
  -- nastavit rozměr obdélníku
  obd.rozmer(10,20);
  -- nastavit rozměr kvádru
  kva.rozmer(10,20,30);
  -- na obrazovku vypsat hmotnost obdélníku
  write_line_int(hmotnost(obd));
  -- na obrazovku vypsat hmotnost kvádru
  write_line_int(hmotnost(kva));
```

a přeložený program spustíme:

```
1400
1400
```

Na prvním řádku se podle očekávání vypíše číslo 1400, ale na druhém řádku je místo správných 15400 také jenom 1400. Kde je problém?

Procedura je napsaná pro třídu `obdelnik` a tudíž volá metody třídy `obdelnik` tak jak byly známy v době překladu, čili staticky. Jaký div, když jsou deklarovány jako `static`!

Aby program splňoval intuitivní požadavek volat třídu té metody která je doopravdy předána, musí se trochu upravit definice třídy. Z hlediska programátora to znamená pouze náhradu klíčového slova `static` slovem `virtual` a určité (ale logické) omezení, že u předefinovaná virtuální metoda má automaticky stejné parametry jako odpovídající metoda předka. Na příkladu by to vypadalo asi takto:

```
program polymorfismus =

with
  support;

class obdelnik =

  var
    x: integer; -- šířka obdélníku
```

```
  y: integer; -- výška obdélníku

 -- metoda pro nastavení rozměru obdélníku
 static rozmer (sirka: integer; vyska: integer) =
 begin
  x:=sirka;
  y:=vyska;
  end rozmer;

 -- metoda pro výpočet plochy obdélníku
 virtual plocha return integer =
 begin
  result:=x*y;
  end plocha;

 end obdelnik;


class kvadr =
 extend obdelnik;

 var
  z: integer; -- hloubka kvádru

 -- metoda pro nastavení rozměru kvádru
 static rozmer (sirka: integer; vyska: integer; hloubka: integer) =
 begin
  x:=sirka;
  y:=vyska;
  z:=hloubka;
  end rozmer;

 -- metoda pro výpočet povrchu kvádru
 override plocha =
 begin
  result:=2*(x*y+y*z+x*z);
  end plocha;

 -- metoda pro výpočet objemu kvádru
 static objem return integer =
 begin
  -- objem kvádru je plocha základny krát hloubka
  result:=x*y*{this:ancestor.plocha*}z;
  end objem;

 end kvadr;



-- Procedura pro výpočet hmotnosti tělesa vyrobeného ze železného plechu.
-- Parametr sila je sila plechu, vysledek je hmotnost výrobku.
```

```
procedure hmotnost(vyrobek: in class obdelnik; sila: in integer:= 1)
return integer =begin
  -- Pro zjednodušení počítáme s hmotností 7 gramů na krychlový
  -- centimetr, snad to nikoho neurazí. Výsledek je v bůhvíčem, možná
  -- v miligramech, ale na tom teď nezáleží.
  result:=vyrobek.plocha * sila * 7;
  end hmotnost;


var
  obd: obdelnik;
  kva: kvadr;
  m_obd: integer;
  m_kva: integer;

begin
  obd.rozmer(10,20);
  kva.rozmer(10,20,30);
  write_line_int(hmotnost(obd));
  write_line_int(hmotnost(kva));
  end polymorfismus;
```

Příklad č. 72. polymorfismus

Nyní již vše funguje tak jak má: první řádek vypíše hmotnost plechového obdélníku a druhý správnou hmotnost kvádru:

```
1400
15400
```

Pokud vás zajímá jak je to uděláno "uvnitř", statická metoda je vlastně obyčejná procedura, jenom s trochu odlišnou syntaxí. Virtuální metoda je odkaz na proceduru, který se vypočítá a do třídy dosadí v okamžiku vzniku instance třídy. Volání virtuální metody je vlastně volání jakési neviditelné procedury která teprve na základě tohoto odkazu vybere a zavolá správnou proceduru.

A ještě něco: naučili jsme se dvě nová klíčová slova (resp. jejich nové použití). V proceduře nestačilo uvést, že mód parametru je `in`, ale museli jsme explicitně napsat, že procedura může akceptovat i potomky uvedeného typu (třídy): `in class`. A ve třídě `kvadr` bylo nutné uvést, že redefinujeme metodu `plocha`, čili použít klíčové slovo `override` následované jménem metody bez uvedení parametrů. Pokud bychom totiž napsali `virtual plocha return integer`, nejednalo by se o redefinici virtuální metody předka, ale novou definici úplně jiné metody, pouze shodně přejmenované – a měli bychom zpátky problém z minulého příkladu.

7. Užitečné drobnosti v třídách

7.1 Abstraktní třídy a metody

Někdy není vhodné začínat hierarchii tříd hned nějakou konkrétní, "použitelnou" třídou, ale je lepší nadefinovat si nějakou prázdnou třídu, ze které se postupně odvozují další. Pro tento účel existuje ve Flexu možnost definovat abstraktní metodu a abstraktní třídu. Ukážeme, jak by se jejich využitím dal upravit příklad 72:

```
class abstract geometricky_objekt =

  virtual abstract plocha return integer;

  end geometricky_objekt;


class obdelnik =
  extend geometricky_objekt;

  var
    x: integer; -- šířka obdélníku
    y: integer; -- výška obdélníku

  -- metoda pro nastavení rozměru obdélníku
  static rozmer (sirka: integer; vyska: integer) =
  begin
    x:=sirka;
    y:=vyska;
    end rozmer;

  -- metoda pro výpočet plochy obdélníku
  override plocha =
  begin
    result:=x*y;
    end plocha;

  end obdelnik;
```

Třída `kvadr` a další neuvedené části programu zůstávají beze změny.

Klíčové slovo `abstract` u jména třídy znamená, že třída se nesmí přímo použít a že je určena pouze jako základ pro odvozování dalších tříd. Obdobně `abstract` u metody říká, že jde pouze o vzor a že musí být nahrazena (`override`) nějakou skutečnou metodou aby mohla být použita.

8. Tipy pro správné programování

8.1 Když to funguje, tak je to správně?

Samotný fakt, že program funguje ještě neznamená, že je správně. V každém programu, snad kromě zcela krátkých triviálních prográmků, jsou chyby které se objeví třeba až dlouho po jeho úspěšném otestování a nasazení do provozu. Některé zvlášť zákeřné chyby se objeví až třeba při provedení drobné změny v programu. Praxe ukazuje, že při používání standardizovaných postupů a dodržení "čistoty stylu" je v programech takových skrytých chyb méně.

V této kapitole uvádíme některá doporučení, která vedou k čistšímu programování. Není nutné, abyste se všemi doporučeními přesně řídili, ale pokud s některým z nich nesouhlasíte nebo je pro vás nepoužitelné, vypracujte si vlastní metodiku a tou se řiďte. Odměnou vám bude kód s menším množstvím chyb a snazší údržbou.

8.2 Kontrola zadávaných hodnot

Kontrolujte si všechny vstupní hodnoty, ať už je zadá uživatel přímo z klávesnice, načtete je ze souboru nebo vám je předá jiná aplikace. Vypadá hloupě, když necháte uživatele zadat nějaké datum, on napíše třeba 32.února a váš program si toho nevšimne a bude pak dělat nesmyslné věci nebo dokonce "spadne".

Dobře napsaný program si ověří každý vstupní údaj, který je vůbec možné alespoň částečně zkontrolovat. Takovémuto ověřování říkají programátoři formální kontroly.

8.3 Komentování zdrojového textu

Pište do svých programů komentáře. Poznamenejte si, k čemu je dobrá která proměnná, co znamená který parametr procedury či metody a co vlastně dělá procedura sama.

Obvykle je zbytečné psát do komentářů co technicky děláte, třeba že zkoušíte, zda je proměnná `X` větší než nula, ale určitě bude zajímavé vědět, proč je vlastně v programu takový test potřeba.

Je vhodné se řídit pravidlem, že každá deklarace a každý příkaz (nebo skupina příkazů) musí být uvozena komentářem.

Komentáře pište jakoby pro někoho jiného, kdo po vás bude program číst. Hodí se to, i když jste vlci samotáři, protože za pět let budete do svého starého programu nevěřícně zírat, co že jste to tehdy napsali a bez komentářů vám možná nezbude, než jej místo drobné opravy napsat celý znova.

8.4 Nešetřete zbytečně pamětí

Pokud v programu nebo proceduře potřebujete použít pro tři různé účely celočíselnou proměnnou, deklarujte si prostě tři různé celočíselné proměnné a nepokoušejte se vystačit s jedinou. Těch pár ušetřených bytů paměti nikomu nepomůže a vy si ušetříte problémy s přehledností a srozumitelností, s hledáním chyb a se zavlečením nečekaných chyb při pozdějších úpravách programu.

Poznámka

O těch několik "vyplýtvaných" bytů stejně nakonec nepřijdete, protože dobrý překladač tuto situaci umí rozpoznat a optimalizaci využití paměti udělá v případě potřeby za vás.

8.5 Programujte stejné věci stejně

V každém programu existuje řada podobných akcí, například prohledání pole, ošetření chyby ve vstupních datech a podobně. Všechny se vyskytují v různých modifikacích, což často svádí k tomu řešit jednoduché případy téhož základního typu jinak než složité. Často se také stává, že jsou dokonce i prakticky stejné akce řešeny pokaždé jinak – příkladem může být prohledávání pole jednou odpředu, podruhé odzadu a potřetí metodou půlení intervalu aniž by pro to byl konkrétní důvod, jiné pojmenování řídící proměnné cyklu nebo náhodná volba cyklů for/break, while nebo until. Rozhodněte se pro jeden způsob řešení a pak jej používejte všude, kde je to jenom možné. Například ve zmíněné úloze prohledávání pole si můžete stanovit, že vždy použijete cyklus while, řídící proměnnou nazvete `i`, rozsah odvodíte od pojmenovaných mezí indexu pole a prohledávat budete od začátku ke konci. Tento postup pak striktně dodržíte i když by v jednotlivých případech odlišné řešení bylo o řádek kratší nebo přineslo urychlení programu o 0.001 %.

8.6 Nepište příliš hutný kód

Některým programátorům dělá radost, když se jim podaří udělat deset různých věcí v jediném řádku programu, ale ve skutečnosti to není nic chvályhodného. Když po nich bude později někdo program luštit, bude muset stále přemýšlet, co vlastně chtěl programátor doopravdy udělat, co je důležitý úmysl a co neškodný vedlejší efekt. Častý programátorský zlozvyk je například vracení kódu výsledku operace funkční hodnotou, které pak vede k používání zkratek typu

```
if not otevri_soubor('C:\data\vypis.txt')
  then... -- chyba
  else... -- zpracování souboru
  end if;
```

Ne, že by tento postup nefungoval, dokonce je leckdy doporučovaný jako základní programátorský obrat. Problém nastane, když budete chtít přidat další podmínky a další akce, resp. mít všechny obdobné úseky programu v souladu s předchozím doporučením psané stejným stylem. V takovém případě rychle roste složitost podmínky a ztrácí se přehled o tom, v jakém pořadí mají být které akce provedeny. Obecnější a proto doporučené řešení je použít zvláštní proměnou pro výsledek operace a podle té se potom rozhodovat.

```
otevri_soubor('C:\data\vypis.txt',vysledek);
if not vysledek
  then... -- chyba
  else... -- zpracování souboru
  end if;
```

8.7 Neoptimalizujte

Nesnažte se zbytečně program optimalizovat. Jednak každá optimalizace zbytečně zhoršuje srozumitelnost programu a tím i zvyšuje pravděpodobnost chyby, jednak se vám stejně většinou nepodaří odhadnout, co vlastně program nejvíc brzdí. Na hledání úzkých míst bude dost času po odladění programu a dost možná, že ani takové zásahy nakonec nebudou potřeba.

Pro rychlost běhu programu je klíčový správný návrh koncepce programu, volba vhodných algoritmů s přijatelnou složitostí a další úkony, které si musíte rozmyslet ještě před programováním. Sebelepším programováním toho, až na řídké výjimky, již mnoho nezachráníte.

8.8 Využívejte datové typy

Datové typy jsou v programovacím jazyce proto, aby za vás hlídaly omyly, nedostatky v logice programu, překlepy a nedokonale provedené změny a opravy. Využívejte je a neřešte celý program s jediným typem čísla. Když chcete uložit do proměnné den v týdnu, nedeklarujte ji jako integer (vzpomenete si vůbec za půl roku, jestli nula představuje pondělí nebo neděli nebo jestli vlastně nezačínáte jedničkou?), ale použijte třeba range 1..7, nebo ještě lépe výčtový typ. V posledním případě je zaručeno, že do této proměnné nepůjde přiřadit nic jiného než den v týdnu a že všechny omyly tak budou automaticky odhaleny.

Nezapomínejte ani na protected typy, viz kapitolu 3.5.

8.9 Využívejte konstanty

Každé číslo pevně zadané v programu je potenciální hrozbou pro případné rozšiřování nebo modifikování programu. Zkušenosti ukazují, že je vhodné všechna čísla, která nějakým způsobem definují vlastnosti programu (například maximální povolený počet znaků příjmení, počáteční velikost okna na obrazovce) pojmenovat a uvést na začátku zdrojového textu programu jako konstanty. Ve zbytku programu se na tyto konstanty odkazujete jen jejich jménem. Takže pokud například původně máte velikost okna 300 bodů a později se ukáže, že 325 by bylo lepší, stačí změnit toto číslo na jediném místě a nemusíte hledat a zkoumat výskyt každého čísla 300 (a pro jistotu i 299 a 301) v celém programu.

8.10 Návrh programu metodou shora dolů

I když to teorie nedoporučuje, návrh programu se běžně provádí současně s programováním. Pro nepříliš složité aplikace to může být i nejrychlejší postup. Přesto je i v tomto případě vhodné dodržovat některá pravidla, z nichž princip návrhu shora dolů považujeme za klíčový a při tom často porušovaný.

Návrh shora dolů (stále mluvíme o situaci kdy programátor sám navrhuje koncepci programu a současně jej programuje) spočívá v tom, že si napřed napíšeme kostru programu která "nic nedělá" a postupně doplňujeme jeho funkce. Nejprve nahrubo, například tak, že napíšeme prázdné procedury (třídy) pro výběr akce z menu, zpracování dat a zobrazení výsledků na obrazovce. Nebojte se napsat si dočasné procedury či metody, "prototypy", třeba výběr z menu, který nevolá menu, ale napevno vybere nějaký kód akce. Teprve když je program pohromadě, můžete se pustit o úroveň níž a řešit problémy detailněji. A až úplně nakonec si necháte piplání se s bity a pixely.

Doporučujeme si o metodách navrhování programu resp. projektování přečíst něco z nepřeberného množství literatury, kde se dozvíte daleko víc, než je možné shrnout v základní učebnici programovacího jazyka.

8.11 Pozor na vedlejší jevy

Každá procedura by měla pracovat pouze s proměnnými (nebo obecněji objekty) které ji předáte jako parametry, u metod samozřejmě též s příslušnou instancí třídy. Situace, kdy procedura mění i jiné proměnné, než které jí byly předány jako parametry, se nazývá vedlejší jev a považuje se obvykle za nevhodnou programátorskou praktiku. U větších programů doporučujeme vyhnout se vedlejším jevům i za cenu, že nějakou proměnnou budete uvádět ve volání mnoha procedur. Chyby vzniklé jako vedlejší jev při volání procedur jsou velmi těžko odhalitelné a jejich hledání vás může připravit o spoustu času.

Zvlášť pečlivě byste měli o vedlejších jevech uvažovat u funkcí. Funkce by se měly používat přibližně v "matematickém" smyslu, čili jako procedury, které mají několik vstupních parametrů, ze kterých vypočtou nějakou hodnotu, již pak předají prostřednictvím návratové hodnoty. Vedlejší jevy u funkcí jsou dosti nepřehledné, nehledě na to, že překladač automaticky předpokládá, že žádné takové jevy funkce nemá a na základě tohoto předpokladu automaticky stanovuje pořadí výpočtu funkcí ve výrazu.

8.12 Linie výpočtu

Sledujte "linii výpočtu" a nenechte se při programování rozptylovat večdlejšími nebo výjimečnými operacemi.

Když při psaní procedury narazíte na místo kde by mohla vzniknout chyba, přímo na tomto místě napište postup pro ošetření chyby spojený s opuštěním procedury (příkazem `return`). Pokud to není možné, rozdělte proceduru na několik jednodušších, abyste mohli tento způsob ošetřování chyb dodržet.

8.13 Testování programu

Testování zdánlivě hotového programu je nudná, ale zcela nezbytná práce. Abyste si ji ulehčili, doporučujeme dodržet několik základních pravidel:

- Již při psaní třídy, metody či procedury si uvědomte, co přesně má dělat a pokud je to možné, otestujte ji samostatně, například v programu napsaném speciálně pro tento účel.
- Doplňte do programu automatické testovací funkce, které budou dostupné například po nastavení hodnoty nějaké konstanty. V testovacím režimu můžete například zobrazovat vnitřní proměnné programu, vypisovat jména otevíraných souborů nebo odkazů. To vše pomůže odhalit potenciální chyby.
- Pořiďte si kvalitní, nejlépe reálná, testovací data. Pokud váš program pracuje například se jmény a adresami (pozor na platné zákony!), udělejte si testovací soubor z nějakého "telefonního seznamu na CD" nebo adresáře firem. Pomocí vymyšlených údajů jen těžko dosáhnete reálné variability jmen a adres a třeba věrohodného statistického rozložení jednotlivých písmen v abecedním třídění.

8.14 Jmenné konvence

U větších projektů se vyplatí si stanovit konvence pro pojmenovávání proměnných, typů, tříd, metod, procedur, konstant a dalších entit, protože to velmi přispívá k přehlednosti programu. Můžete si například

stanovit, že jména typů budou začínat písmenem t (kterým nebude smět začínat žádné jiné jméno), jména pointerů na typ nebo třídu písmenem p a tak dále. Možností je neomezeně, každá však pomůže v lepší orientaci v programu, zejména je-li program delší a pracuje-li na něm více programátorů.

8.15 "Rodiny" chyb

Když se vám konečně podaří najít a opravit chybu v programu, není ještě všechna práce hotová. Je velice vhodné udělat ještě několik dalších opatření:

- Pokuste se najít chyby stejného typu v celém programu, případně i v knihovnách a v ostatních vašich programech.
- Zamyslete se, jak zabránit opakování stejného typu chyby.
- Upravte program tak, aby se takováto chyba již nedala znova udělat.

Může se stát, že se v nějaké části programu (ve třídě, v proceduře) často nacházejí nové a nové chyby. Taková situace zpravidla indikuje nějaký koncepční nedostatek. Nejlepší řešení bývá takový zdroj chyb smazat a nově navrhnout a naprogramovat.

8.16 Ladicí výpisy

Ladicí výpisy jsou starodávný, ale přesto stále účinný prostředek pro hledání chyb. Pokud program dělá něco co nemá a vy nevíte proč, nebojte se připsat pár řádek jen pro účely testování. Zkuste si třeba jinou cestou vypočítat co by měly obsahovat vytypované proměnné a výsledek vypsat společně se skutečným obsahem proměnných na obrazovku. Není to žádná začátečnická pomůcka za kterou byste se museli stydět, naopak velké aplikace často obsahují celé rozsáhlé podsystémy pro vlastní testování, které se dají například pomocí parametrů při spouštění programu aktivovat.

8.17 Grafická úprava zdrojových textů

Grafická úprava neboli formátování zdrojových textů programu je velice důležitá pro orientaci v programu. Sám text programu je z formálního hlediska zcela dostačující, i kdyby byl celý program napsaný do jediného řádku sto tisíc znaků dlouhého, přesto se doporučuje stanovit si nějaká pravidla a řídit se jimi. Použité formátování by mělo zejména postihnout logické členění programu. Je vhodné, když je na první pohled patrné, kde který složený příkaz začíná a kde končí, jaké má části a kde jsou v něm obsaženy vnořené příkazy a podobně.

"Firemní" doporučení

Níže uvádíme několik nejdůležitějších pravidel formátování zdrojových textů, kterými se řídí vývojový tým Flexu. Doporučujeme též prostudovat si pro inspiraci zdrojové texty knihoven SMPL dodaných společně s překladačem.

Normalizovaná šířku textu je 100 znaků. Pokud je řádek zdrojového textu delší, musí být zalomen tak, aby nevybočil přes hranici 100 znaků. Výjimka je možná ve zdůvodněných případech, například pokud má část programu (typicky agregát pole nebo záznamu) podobu tabulky, kde by zalomení naopak přehlednost zhoršilo.

Druhý a další řádek delšího příkazu (včetně koncového `end` složeného příkazu) se odsazují o dva znaky doprava.

```
if promenna>10 then
 prikaz_1;
 prikaz_2;
 end if;
```

Pokračování vnořených příkazů se odsazuje vždy o další dva znaky. Části `then`, `else`, `when` se odsazují jakoby šlo o samostatné příkazy.

```
if promenna>10 then
 prikaz_1;
 prikaz_2;
 if promenna>20
  then
    prikaz_3;
    prikaz_4;
  else
    prikaz_5;
    prikaz_6;
    end if;
  end if;
```

Deklarace vnořené ve složitějších deklaracích typu (`record`, `enum`, `class`, parametry procedury) se odsazují o čtyři znaky od počátku deklarace.

```
type
 t_record    = record
  slozka_1: integer;
  slozka_2: integer;
  end record;
```

Význačné body v deklaracích, jako jsou **:** a = se píší na 20. sloupec. Toto neplatí pro = v deklaraci procedury, modulu, třídy.

```
type
 integer        = signed 32;

var
 promenna_1: integer;
```

Komentáře, které se píšou vpravo od deklarace (např. u parametru nebo proměnných) začínají na 50. sloupci.

9. A co dál

V této učebnici jsme probrali základ Flexu, který už stačí k běžnému programování, vlastně k napsání jakéhokoliv programu. Flex má ale daleko více možností, než bylo rozumné představit v základní učebnici. Pro další studium doporučujeme referenční příručku (Language Reference Manual (LRM)), ve které jsou přesně popsány všechny vlastnosti Flexu. Z LRM se dozvíte o zcela nových příkazech, funkcích a vlastnostech Flexu, ale také zjistíte, že popisy v této učebnici byly zjednodušené a že byly v zájmu srozumitelnosti vypuštěny různé speciální nebo méně používané vlastnosti.

LRM je psán přesným, ale poněkud strohým a pro někoho hůře stravitelným slohem, bohužel asi nezbytným pro příručky tohoto druhu. Abychom vám usnadnili orientaci, přinášíme na závěr stručný přehled témat, která v LRM najdete.

9.1 Výjimky

Výjimka je programová konstrukce pro snazší ošetření chyb v programu. Na konci každé procedury (metody, bloku) můžete uvést, co se má provést, pokud v proceduře (metodě, bloku) dojde k chybě (ošetření výjimky). Pokud výjimku neošetříte, procedura skončí a stejná výjimka vznikne v tom místě, odkud byla procedura volána a celá situace se opakuje, teď už o jednu úroveň výše. Tomuto postupu se říká šíření výjimky. Výjimka se šíří tak dlouho, dokud nenarazí na ošetření výjimky. Pokud není ošetřena nikde, skončí celý program, což ovšem nesvědčí o zrovna pečlivé práci programátora. Výjimka může vzniknout za situací definovaných jazykem (například dělení nulou, pokus o přístup k neexistujícímu prvku pole) nebo může být vyvolána přímo programátorem.

Klíčová slova: `message`, `raise`, `catch`.

9.2 Paralelní procesy a synchronizace

Flex obsahuje prostředky pro programování paralelních procesů (multithreading) včetně mechanismů pro synchronizaci a předávání dat mezi jednotlivými procesy.

Klíčová slova: `task`.

9.3 Zprávy

Flex přímo podporuje předávání zpráv mezi procesy pomocí specializovaných prostředků jazyka. Obsahuje správy front zpráv a další vlastnosti včetně synchronizace, čekání na zprávu, provádění akce v závislosti na přijaté zprávě.

Klíčová slova: `message`, `queue`, `send`, `accept`.

9.4 Atributy

Každá entita ve Flexu (datový typ, proměnná, procedura, program apod.) má řadu různých vlastností, které jsou v programu dostupné ve formě pojmenovaných atributů. Atributem je například velikost proměnné, typ proměnné, počet prvků pole a množství dalších.

Klíčová slova: `attribute`.

9.5 Podmíněný překlad

Během překladu programu ve Flexu se zpracovávají metapříkazy, pomocí kterých se dá zvolit zda se určitá část programu má nebo nemá přeložit.

Klíčová slova: `#if`, `#else`, `#end if`, `#declared`.

9.6 Textové substituce

Textová substituce je mechanismus pro automatické vkládání předem definovaného a pojmenovaného bloku textu na určená místa zdrojového textu. Jejím cílem je ušetřit práci s opisováním dlouhých opakujících se úseků zdrojového textu a zlepšit přehlednost programu.

Textová substituce je přibližně totéž, čemu se v jazyce C říká makro. Pojem makro je však ve Flexu vyhrazen komplexnější konstrukci, viz níže.

Klíčová slova: `#template`, `#expand`.

9.7 Makra

Makro je procedura ve Flexu, která se provádí během překladu. Pomocí maker je možné realizovat řadu jinak nedostupných vlastností, například vypočítávat tabulky konstant nebo zařadit přímo do zdrojového kódu scripty nebo programy ve vlastním specializovaném jazyce.

Klíčová slova: `macro`.

9.8 Definice vlastních operátorů

Ve Flexu lze změnit funkci existujících operátorů pro zvolený datový typ.

Klíčová slova: `overload`.

9.9 Přetížené procedury

Různé procedury můžete pojmenovat stejně, pokud se liší v počtu nebo typech parametrů. Která z nich má být skutečně provedena, se rozliší podle skutečných parametrů.

Klíčová slova: `overload`.

Přílohy

Příloha A. Základní programátorské dovednosti

V této příloze jsou popsány nejběžnější jednoduché programátorské obraty a postupy. Každý z nich je dokumentován příslušným příkladem. Příklady jsou psány vždy jako kompletní programy, aby bylo možné si je vyzkoušet a modifikovat.

A.1 Zvětšení hodnoty proměnné o jedničku

```
program priklad =
var
  i: natural;

begin
  -- klasický postup
  i:=i+1;

  -- využití zkráceného přiřazení
  i+1;
  end priklad;
```

A.2 Záměna dvou proměnných

```
program priklad =
var
  a: natural;
  b: natural;
  pomocna: natural;

begin
  pomocna:=a;
  a:=b;
  b:=pomocna;
  end priklad;
```

Program zamění obsah proměnných `a` a `b`. Pro záměnu je nutné použít třetí pomocnou proměnnou, do které se dočasně uloží obsah jedné ze zaměňovaných proměnných. Je jasné, že všechny tři proměnné musí být stejného typu, aby bylo možné provést přiřazení.

Zajímavost

Programátoři, kteří začali programovat před padesáti a více lety vědí, že záměnu proměnných lze provést i bez použití pomocné proměnné, pomocí výpočtu binární nonekvivalence (`a xor b; b xor a; a xor b`). Dnes tato metoda nemá význam, protože počítač má paměti dost a naopak je zbytečné jej zdržovat prováděním výpočtů.

A.3 Vložení prvku do pole

```
program priklad =

var
  pocet: natural;           -- počet platných prvků v poli
  a: array 1..10 of natural; -- pole prvků
  n: natural;               -- sem chceme přidat nový prvek
  x: natural;               -- tuto hodnotu chceme do pole přidat

begin
  -- pro názornost naplníme proměnné nějakými přípustnými hodnotami
  pocet:=8;
  a:=[1,2,3,4,5,6,7,8,0,0];
  n:=5;
  x:=100;

  -- prvky pole počínaje n-tým posuneme o jeden dozadu
  -- aby vzniklo místo pro přidávaný prvek
  for i in reverse n..pocet loop
    a[i+1]:=a[i];
    end loop;

  -- v cyklu jsme vlastně postupně provedli příkazy
  --   a[9]:=a[8];
  --   a[8]:=a[7];
  --   a[7]:=a[6];
  --   a[6]:=a[5];
  -- a pole "a" nyní vypadá takto
  -- a = [1,2,3,4,5,5,6,7,8,0];

  -- do n-tého prvku vložíme hodnotu x
  a[n]:=x;

  -- počet platných prvků pole se zvýšil o jeden
  pocet+1;

  -- toto je v poli "a" teď
  -- a = [1,2,3,4,100,5,6,7,8,0];
  -- pocet = 9
  end priklad;
```

Do pole `a` vkládáme jeden prvek. Algoritmus funguje pro každé `n` v rozsahu `1` až `pocet+1` (přičemž v případě `n=pocet+1` nejde o vkládání ale jednodušší přidání). Všimněte si, že při "dělání místa v poli" přesunujeme prvky počínaje posledním, aby se vzájemně nepřemazaly.

A.4 Odstranění prvku z pole

```
program priklad =
```

```
var
  pocet: natural;          -- počet platných prvků v poli
  a: array 1..10 of natural; -- pole prvků
  n: natural;              -- odsud chceme prvek odstranit

begin
  -- pro názornost naplníme proměnné nějakými přípustnými hodnotami
  pocet:=8;
  a:=[1,2,3,4,5,6,7,8,0,0];
  n:=5;

  -- prvky pole počínaje n+1 posuneme o jeden dopředu
  -- čímž bude odstraněn n-tý prvek
  for i in n+1..pocet loop
    a[i-1]:=a[i];
    end loop;

  -- v cyklu jsme vlastně postupně provedli příkazy
  --   a[5]:=a[6];
  --   a[6]:=a[7];
  --   a[7]:=a[8];

  -- počet platných prvků pole se snížil o jeden
  pocet-1;

  -- toto je v poli "a" teď
  -- a = [1,2,3,4,6,7,8,8,0,0];
  -- pocet = 7
  end priklad;
```

Všimněte si, že osmý prvek pole zůstal nezměněn. Je to obvyklý postup v případě, že platný počet prvků pole je dán nějakou jinou proměnnou. Nepoužité prvky pole mohou mít libovolnou hodnotu (někdy se též říká, že mají nedefinovanou hodnotu).

Pokud bychom z nějakého důvodu potřebovali aby byly nepoužité prvky vynulovány, museli bychom ještě před poslední příkaz (`pocet-1`) vložit řádek

```
a[pocet]:=0;
```

A.5 Vložení prvku do pole s využitím řezu pole

```
program priklad =

var
  pocet: natural;          -- počet platných prvků v poli
  a: array 1..10 of natural; -- pole prvků
  n: natural;              -- sem chceme přidat nový prvek
  x: natural;              -- tuto hodnotu chceme do pole přidat

begin
  -- pro názornost naplníme proměnné nějakými přípustnými hodnotami
  pocet:=8;
```

```
a:=[1,2,3,4,5,6,7,8,0,0];
n:=5;
x:=100;

-- prvky pole počínaje n-tým posuneme o jeden dozadu
-- aby vzniklo místo pro přidávaný prvek
a[n+1..pocet+1]:=a[n..pocet];

-- do n-tého prvku vložíme hodnotu x
a[n]:=x;

-- počet platných prvků pole se zvýšil o jeden
pocet+1;
end priklad;
```

Odstranění prvku by probíhalo obdobně. Výhoda použití řezu je kromě přehlednějšího zápisu v tom, že nemusíme přemýšlet, zda budeme prvky v poli přesunovat odzadu nebo odpředu.

A.6 Setřídění pole

```
program priklad =

const
  pocet = 10; -- počet prvků pole

var
  a: array 1..pocet of natural; -- pole prvků
  pom: natural; -- pomocná proměnná pro záměnu prvků

begin
  -- pro názornost naplníme proměnné nějakými přípustnými hodnotami
  a:=[1,4,7,0,5,2,3,9,8,6];

  -- opakovaně postupně zkoušíme dvojice sousedních prvků
  -- přičemž vynecháváme ty, které již nemohou být nesetříděné
  for x in reverse 1..pocet-1 loop
    for i in 1..x loop
      -- pokud se vyskytla anomálie setřídění...
      if a[i]>a[i+1] then
        --... odstraníme ji zaměněním prvků
        pom:=a[i];
        a[i]:=a[i+1];
        a[i+1]:=pom;
        end if;
      end loop;
    end loop;

  -- toto je v poli a teď
  -- a = [0,1,2,3,4,5,6,7,8,9];
  end priklad;
```

Třídění je běžný úkol, který se vyskytuje v nejrůznějších obměnách asi v každém větším programu. Zde je uvedeno nejjednodušší třídění, v literatuře zpravidla nazývané bubblesort. Z používaných algoritmů má nejhorší výpočetní složitost (každým prodloužením pole na dvojnásobek se doba třídění prodlouží čtyřnásobně), proto se používá jen pro zcela jednoduché případy, kdy počet prvků nepřekročí několik desítek. V náročnějších případech doporučujeme použít některý z algoritmů, které jsou dobře popsány v literatuře. Pokud je budete hledat na internetu, podívejte se na Shell sort, což je snadno naprogramovatelný algoritmus s přijatelnou složitostí a Quick sort, což je nejlepší možný obecný algoritmus. Za určitých předpokladů lze použít ještě účinnější speciální algoritmy, ale to je již téma přesahující rozsah této učebnice.