**Pole**

**type** JménoTypu = **array** [DolníMez **..** HorníMez] **of** TypPoložek

 - dolní mez a horní mez jsou celočíselné konstanty

 - typ položek libovolný, ale všechny položky jsou stejného typu

**const** Max = 10;

**type** T = **array** [1..Max] **of** integer;

**var** P: T;

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| prvek pole | P[1] | P[2] | P[3] | P[4] | P[5] | P[6] | P[7] | P[8] | P[9] | P[10] |
| hodnota prvku pole | **30** | **5** | **8** | **51** | **24** | **9** | **12** | **7** | **65** | **11** |

**type** TR = **array** [1..5] **of** real;

**var** PR, QR: TR; {dvě pole stejného typu}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| prvek pole | PR[1] | PR[2] | PR[3] | PR[4] | PR[5] |
| hodnota prvku pole | **1,2** | **0,5** | **8,02** | **4,5** | **7,4** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| prvek pole | QR[1] | QR[2] | QR[3] | QR[4] | QR[5] |
| hodnota prvku pole | **32,4** | **12,9** | **17,0** | **41,2** | **3,8** |

Zkratka – deklarace proměnné bez pojmenování typu, definice nového typu je součástí deklarace proměnných:

 **var** PR, QR: **array** [1..5] **of** real;

**Indexování** P[3] := 8; read(PR[4]);

S proměnnou typu pole lze bez indexování provádět jedinou operaci, a to dosazení mezi dvěma proměnnými téhož typu: QR:=PR; { Hodnoty prvků pole QR se vloží do pole PR}

Pole se může indexovat proměnnou (obecně dokonce libovolným celočíselným výrazem) → možnost v cyklu vykonat jistou akci se všemi složkami pole.

Typicky se využívá for-cyklus a indexování prvků pole jeho řídicí proměnnou:

**for** I:=1 **to** Max **do** read(P[I]); {načtení pole po složkách}

**for** I:=1 **to** Max **do** write (P[I]); {tisk pole po složkách, na řádek vytisknuty všechny prvky pole}

 {30 5 8 51 24 9 12 7 65 11 }

**for** I:=1 **to** Max **do** writeln (P[I]); {tisk pole po složkách, na každý řádek jeden prvek pole}

 {30

 5

 8

 **.**

 **.**

 11 }

…

S:=0;

**for** I:=1 **to** Max **do** S:=S+P[I]; {součet všech prvků pole}

writeln(S) { tisk součtu všech prvků pole} { 222}

Pozor na **hodnotu indexu mimo deklarovaný rozsah** – reakce podle nastavení přepínače $R:{$R+} provádí se kontrola při výpočtu →**běhová chyba**

přetečení indexu (Range check error)

{$R-} kontrola se neprovádí → kratší a rychlejší přeložený kód, ale **logická chyba** (chybný výpočet bez upozornění, může třeba dojít ke změně jiné proměnné). Standardní výchozí nastavení: bez kontrol *Doporučení:* dokud není program odladěn, vždy zapínat!

**Vícerozměrné pole**

**type** Matice = **array** [1..3, 1..4] **of** integer;

**var** M: Matice;

…

M[2,3] := 3145;

- počet indexů není omezen (v praxi max. 3)

- více indexů → pomalejší přístup k prvku (počítá se adresa)

- zkratka za „pole polí“

**type** Matice = **array** [1..3] **of array** [1..4] **of** integer;

…

M[2][3] := 3145;

{toto je rovněž povolený korektní zápis, ekvivalentní význam}

**Vyhledávání v poli**

Úkol: zjistit, zda se v poli nachází daná hodnota a kde (pokud je tam vícekrát, chceme první výskyt)

Algoritmus: jeden sekvenční průchod polem – časová složitost O(*N*)

**var** A: **array** [1..N] **of** T;

X: T; {hledaná hodnota}

**1. for-cyklus**

J := 0;

**for** I := 1 **to** N **do**

**if** A[I] = X **then** J := I;

**if** J = 0 **then** … { X v poli A není }

**else** … { X nalezen – A[J] }

Jednoduché, ale nešikovné (cyklus pokračuje i po nalezení X)

**2. for-cyklus s výskokem**

J := 0;

**for** I := 1 **to** N **do**

**if** A[I] = X **then begin** J: = I; break **end**;

**if** J = 0 **then** … { X v poli A není }

**else** … { X nalezen – A[J] }

Obdobné řešení, ale cyklus zbytečně nepokračuje po nalezení X.

Použitelné jen v implementacích, které mají „break“.

**3. while-cyklus**

I := 1;

**while** (I <= N) **and** (A[I] <> X) **do** inc(I);

**if** I > N **then** … { X v poli A není }

**else** … { X nalezen – A[I] }

Vše řeší vhodně zvolená složená podmínka, ale POZOR!

Není-li prvek X v poli A obsažen, může nastat běhová chyba

přetečení indexu, neboť v podmínce se testuje A[I] pro I=N+1.

Korektní řešení v případě zkráceného vyhodnocování {$B-}.

Dva způsoby **vyhodnocování logických výrazů**

- volba režimu přepínačem $B:

{$B+} úplné vyhodnocování – nejprve všechny podvýrazy (norma)

{$B-} zkrácené vyhodnocování – zleva doprava dokud není

rozhodnuto o výsledku (implicitní nastavení v TP)

 **4. cyklus řízený proměnou typu boolean**

**var** Dalsi: boolean; {zda zpracovávat další prvek}

…

I := 1;

Dalsi := true;

**while** Dalsi **do**

**if** A[I] = X **then**

**begin** Dalsi := false;

… { X nalezen – A[I] }

**end**

**else if** I = N **then**

**begin** Dalsi := false;

… { X v poli A není }

**end**

**else** inc(I);

**5. vyhledávání pomocí zarážky** → zjednodušení podmínky ve while-cyklu

**var** A: **array** [1..N+1] **of** T;

{prvek A[N+1] pro uložení zarážky}

X: T; {hledaná hodnota}

A[N+1] := X; {uložení zarážky}

I := 1;

**while** A[I] <> X **do** inc(I);

**if** I > N **then** … { X v poli A není }

**else** … { X nalezen – A[I] }

Hodnota X je v poli A vždy nalezena – pokud ale I>N, tak až na místě zarážky, tzn. původně v poli A[1..N] nebyla.

**Binární vyhledávání** - vyhledávání v uspořádaném poli

Algoritmus: metoda půlení intervalů – časová složitost O(log *N*)

**var** A: **array** [1..N] ofT; {A[1]<A[2]<…<A[N]}

X: T; {hledaná hodnota}

i,j,k: integer; {pomocné indexy}

…

i := 1; j := N; {meze zkoumaného úseku}

**repeat**

k := (i+j) **div** 2; {index prostředního prvku}

**if** X > A[k] **then** i := k+1

**else** j := k-1

**until** (A[k] = X) **or** (i > j);

**if** X = A[k] **then** … { X nalezen – A[k] }

**else** … { X v poli A není }

**Procedury**

Deklarace procedury:

**procedure** JménoProcedury;

<lokální deklarace> … stejné jako oblast deklarací v programu, ale jen v rámci procedury

**begin**

 <tělo = složený příkaz> … stejné jako tělo programu, ale jen v rámci procedury

**end**;

**procedure** Cara; {vypíše čáru z 50 hvězdiček}

**const** N=50;

**var** I: integer;

**begin**

 **for** I:=1 **to** N **do** write(‘\*’);

writeln

**end**;

Volání procedury: jednoduchý příkaz

**begin**

…; {několikrát volám proceduru Cara uvnitř těla programu}

Cara; {\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*}

…;

Cara; {\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*}

…;

**End.**

Sémantika:

- odskok na provedení příkazů uvedených v těle procedury

- poté návrat za místo volání (podle uložené návratové adresy)

Použití:

- **určitá činnost se vykonává v programu opakovaně vícekrát**

**- dekompozice programu na menší logické celky**

**Lokální deklarace**

**(Deklarace proměnných uvnitř procedury, proto platí jen pro tuto proceduru, ne jinde v programu)**

- v maximální míře využívat (jinak strukturování kódu do procedur ztrácí smysl)

- není viditelná vně procedury (v hlavním programu ani třeba v jiné proceduře)

- procedura je samostatný prostor jmen (nevadí jiné použití stejného identifikátoru mimo proceduru)

- zastiňuje stejnojmennou globální deklaraci (ta je pak v proceduře nedostupná) {globální proměnné deklarovány na začátku
 programu, platí v průběhu celého programu}

- nezastíněné globální identifikátory lze v proceduře používat (ale děláme to jen tehdy, když k tomu máme dobrý důvod)

- lokálně v proceduře lze deklarovat nejen proměnné, typy, konstanty a návěstí, ale také další procedury a funkce (ty pak lze volat jen uvnitř této procedury)

- proměnné deklarované lokálně v proceduře nemají definovanou počáteční hodnotu (nejsou automaticky vynulovány)

**Viditelnost**

**program** A;

 …

 **procedure** B;

 …

 **procedure** C;

 **begin** {C}

 … ←lze volat B (globální) a C (rekurzívně)

 **end**; {C}

 **procedure** D;

 **begin** {D}

 … ←lze volat B (globální), C (předcházející) a D (rekurzívně)

 **end**; {D}

 …

 **begin** {B}

 … ← lze volat C a D (lokální) a B (rekurzívně sama sebe)

 **end**; {B}

…

**begin** {A}

…←lze volat B, není viditelná C ani D

**end**. {A}

**Parametry (proměnné v hlavičce procedury)**

- nástroj pro předávání vstupních hodnot do procedury a výstupních, tzn. výsledků, ven z procedury (jinak by procedura dělala pořád úplně stejnou akci)

Při deklaraci procedury: v hlavičce seznam formálních parametrů (nadefinujeme proměnné)

 - parametry mohou být libovolného typu (i strukturovaného)

Při volání procedury: v hlavičce seznam skutečných parametrů → musí jich být stejný počet a stejného typu a ve stejném pořadí, jako byly formální parametry

**procedure** Cara(N: byte; Z: char); { při volaní → Cara(číslo, ’znak’) → dosazeny skutečné parametry }

{vypíše čáru z N znaků Z}

**var** I: integer;

**begin**

**for** I:=1 **to** N **do** write(Z);

writeln

**end**; { konec deklarace procedury }

**begin** { tělo programu }

…

…

 Volání: Cara(33, ’=’); {=================================}

 Cara(40, ’.’); {………………………………….}

…

…

**end**.

***Upřesnění syntaxe:***

- formální parametry se oddělují středníkem

- více formálních parametrů téhož typu lze oddělit čárkami a typ zapsat jen jednou (podobně jako v deklaracích proměnných),

např.: **procedure** P(A, B, C: **real;** M, N: **integer);**

- je nutné uvést vždy jméno typu (identifikátor), nikoli jeho definici → je-li parametrem např. pole, musí být příslušný typ předem definován a pojmenován pomocí **type,** nelze psát např.:  **procedure** P(X: **array**[1..10] **of** integer); **NELZE!!!**

- skutečné parametry se oddělují čárkami

- na místě skutečných parametrů nemusí být jen konstanty, ale mohou to být jakékoliv výrazy příslušného typu

***Upřesnění sémantiky:***

- výrazy zapsané na místě skutečných parametrů se vyhodnotí a jejich hodnota se dosadí do odpovídajících formálních parametrů

- formální parametry jsou tedy vlastně další lokální proměnné, jejichž počáteční hodnota je stanovena v okamžiku volání (dále se mohou v proceduře libovolně měnit, navenek se to nijak neprojeví, po skončení výpočtu procedury zaniknou)

→**předávání parametrů HODNOTOU** - pouze vstupní!

Ještě potřebujeme nějaký mechanismus, jak předávat výsledky výpočtu z procedury ven

→ **předávání parametrů ODKAZEM** - vstupní i výstupní

***Syntaxe:***

- v deklaraci procedury jsou formální parametry předávané odkazem označeny
 klíčovým slovem **var** (platí vždy do nejbližšího středníku), např.:

**procedure** P(**A: real;** **var B, C: real;** **M: integer;** **var N: integer);**

{**parametry A, M jsou předávány hodnotou**, **parametry B, C, N odkazem**}

- při volání procedury na místě skutečných parametrů musí být proměnné

 **Předávání hodnotou** – formální parametr je lokální proměnná, skutečný parametr může být výraz, jeho hodnota se předá v okamžiku volání → **pouze vstupní parametr**. Po skončení procedury hodnota zapomenuta (zanikne).

 **Předávání odkazem** - skutečným parametrem musí být proměnná, předává se její adresa, hodnota se nekopíruje → vstupní i výstupní parametr

Odkazem volaný parametr je vhodné použít i v případech, kdy má být proceduře předána hodnota proměnné, která je velmi rozsáhlá (například pole). Při volání odkazem je totiž proceduře předána pouze adresa skutečného parametru, odpadá vytvoření jeho kopie. Potom je však nutno zajistit, aby procedura nezpůsobila nechtěnou změnu hodnoty parametru.

Program VolaniHodnotou;

**var** j:integer;

 **Procedure** A(i:integer);

 **begin**

 i:=i+1; write(i)

 **end;**

**begin**

j:=1;

write(j);A(j);writeln(j)

**end**. { vytiskne **1 2 1** } { hodnota **2** byla po skončení procedury zapomenuta, a proto je **j** opět rovno **1**}

Program VolaniOdkazem;

**var** j:integer;

 **Procedure** A(var i:integer);

 **begin**

 i:=i+1; write(i)

 **end;**

**begin**

j:=1;

write(j);A(j);writeln(j)

**end**. { vytiskne **1 2 2** } {V proměnné **j** zůstává hodnota **2** i po skončení procedury}

**procedure** zamen(a:integer); { **volání hodnotou**, do lokální proměnné **a** se zkopíruje hodnota globální proměnné **a** }

 **begin**

 a:=5; {lokální proměnná **a}**

 **end**; { konec deklarace procedury}

 **begin** { tělo programu }

 a:=1; { globální proměnná **a**}

 zamen(a);

 writeln(a); { vypíše **1**, protože v proceduře jsem manipuloval s lokální proměnnou, její hodnota zanikla po skončení
 procedury ( leží v paměti jinde než globální proměnná **a)**}

 **end.**

 **procedure** zamen(var a:integer); { **volání odkazem**}

  **begin**

 a:=5;

 **end;**

  **begin**

 a:=1;

 zamen(a);

 writeln(a); { vypíše **5**, protože při volání odkazem pracuji i v proceduře s proměnnou, která ukazuje na stejné místo

 v paměti jako globální proměnná}

 **end.**

V následující ukázce programu je pro výpočet přirozené mocniny reálného čísla deklarována procedura *Mocnina*. Vstupní data jsou jí předávána v parametrech *Zaklad* (umocňované číslo) a *Exponent* (mocnitel), výsledek výpočtu je vyvážen parametrem *Vysledek*. V algoritmu výpočtu je využito platnosti vztahu *x*2*n* = (*x*2)*n*.

{======================================================================}

program VypocetMocniny;

var X: Real; { vstupní data programu - základ }

 N: Word; { a exponent mocniny }

 procedure Mocnina (Zaklad: Real; Exponent: Word; var Vysledek: Real);

 begin { Mocnina }

 Vysledek := 1; { inicializace proměnné Vysledek }

 while Exponent > 0 do

 begin

 while not Odd(Exponent) do { úprava podle x^(2\*n) = (x^2)^n }

 begin

 Zaklad := Sqr(Zaklad);

 Exponent := Exponent div 2

 end;

 Vysledek := Vysledek \* Zaklad; { aktualizace mezivýsledku }

 Exponent := Exponent - 1 { a exponentu }

 end

 end; { Mocnina }

begin { program }

 Writeln; { tisk titulku }

 Writeln('Výpočet mocniny');

 Writeln;

 Write('základ: '); { načtení základu a exponentu }

 Readln(X);

 Write('exponent: ');

 Readln(N);

 Mocnina(X, N, X); { volání procedury }

 Writeln('výsledek: ', X) { tisk výsledku }

end. { program }

|  |  |
| --- | --- |
|  |  |

***Sémantika:***

- vyhodnotí se adresy skutečných parametrů a předají se do odpovídajících formálních parametrů

- formální parametry jsou tedy vlastně dočasné lokální odkazy na globálně existující proměnné

- jakékoliv manipulace s formálním parametrem v těle procedury se prostřednictvím odkazu- fakticky provádějí s příslušnou globální proměnnou → procedura v parametru dostane vstupní hodnotu, může ho měnit a může v něm zanechat výstupní hodnotu (po skončení výpočtu procedury sice parametr zanikne – ale to je jen ten lokální odkaz, nikoli vlastní proměnná!)

***Použití:***

- má-li parametr sloužit jako výstupní (nebo vstupní i výstupní)

- je-li parametr rozsáhlou strukturou (vytváření lokální kopie při předávání parametru hodnotou by bylo náročné paměťově i časově)

*Příklady:*

**program** R;

**var** Pole: **array**[1..2] **of** char;

 I: integer;

 **procedure** P (**var** C: char);

 **begin** I:=2; writeln(C) **end**;

**begin**

Pole[1]:=‘A’; Pole[2]:=‘B’; I:=1;

P(Pole[I])

**end**. Co vypíše? {**A** adresa parametru je vyhodnocena v okamžiku volání procedury!}

**procedure** Z (**var** A, B: integer);

**begin**

A:=5; B:=13; write(A)

**end**; Co vypíše? {**většinou 5**, ale při zavolání procedury s oběma stejnými parametry,
 např. **Z(X,X)**, **vytiskne 13**}

Pokud bychom měli stejně pojmenovanou lokální a globální proměnnou, tak **v hlavním programu platí vždy globální proměnná** a **v proceduře vždy lokální pproměnná.** Ukažme si to na příkladu:

 **var** x: integer; { **globální proměnná**}

  **procedure** nastav;

 **var** x: integer; { **lokální proměnná**}

 **begin**

 x:=100;

 writeln('V procedure vítezi lokalni **x**, protoze x=',x);

 **end;** { v tomto okamžiku zaniká lokální proměnná}

 **begin**

 x:=5;

 nastav; { **volání procedury s lokální proměnnou x**} { vytiskne **100**} {zaniká lokální **x** po skončení procedury}

 writeln('V hlavním programu vitezi globalni **x**, protoze x=',x); { vytiskne **5**} {**platí globální x**}

 readln;

 **end.**

**Funkce**

Deklarace funkce:

**function** JménoFunkce (formální parametry): TypVýsledku;

<lokální deklarace> … stejné jako oblast deklarací v programu

**begin**

<tělo = složený příkaz> … stejné jako tělo programu

**end**;

- mezi příkazy v těle funkce definování návratové hodnoty ve tvaru: JménoFunkce := Výraz

- JménoFunkce se v těle funkce jeví jako lokální proměnná typu TypVýsledku, do níž **lze pouze dosazovat** a tím se definuje návratová hodnota (výsledek)

- **dosazovat lze i opakovaně - platí poslední dosazená hodnota.**

- TypVýsledku může být (alespoň v TP) číselný, znakový, string, boolean *(nebo ukazatel )*, nesmí být strukturovaný

Jinak pro funkce platí vše jako pro procedury – lokalita identifikátorů, viditelnost, způsoby předávání parametrů.

**Je poměrně obvyklé, že funkce nepoužívají parametry předávané odkazem – výsledek práce se předává ven jako funkční hodnota**.

Volání funkce: JménoFunkce (skutečné parametry)

- výraz (může být součástí většího výrazu)

*Příklad:*

**function** Max3 (A, B, C: integer): integer; {vybírá maximum ze tří zadaných celých čísel}

**var** D: integer;

**begin**

**if** A>B **then** D:=A **else** D:=B;

**if** C>D **then** D:=C;

Max3:=D

**end**;

*Pozor na chybu:*

**function** Max3 (A, B, C: integer): integer; {vybírá maximum ze tří zadaných celých čísel}

**begin**

**if** A>B **then** Max3:=A **else** Max3:=B;

**if** C>Max3 **then** Max3:=C;

**end**;

Max3 – v pořádku (lze dosazovat opakovaně)

Max3 – chyba – překladač chápe jako rekurzivní volání funkce (s chybným počtem parametrů)!

*Příklad:* přestupný rok

**function** Prestupny (Rok: integer): boolean;

{zjištění, zda je rok přestupný – podle Gregoriánského kalendáře platného od roku 1582}

**begin**

Prestupny := (Rok **mod** 4 = 0) **and** (Rok **mod** 100 <> 0) **or** (Rok **mod** 400 = 0)

**end**;

**var** R: integer;

…

**if** Prestupny(R) **then writeln(‘Rok ’,R,’ je prestupny.’)**

 **else writeln(‘Rok ’,R,’ neni prestupny.’)**

Rok - formální parametr funkce - nadefinování vstupní proměnné

R - skutečný parametr funkce -hodnota), která je stejného typu jako formální parametr

Napíšeme si funkci, která nám sečte dvě zadaná čísla.

 **function** secti (a,b:Integer):integer;

 **begin**

 secti:=a+b;

 **end;**

 **begin**

 writeln('Vysledek je : ',secti(2,5));

 readln;

 **end.**

Všimněme si, že s funkcí pracujeme jako s běžnou proměnnou a můžeme ji dát např. přímo do příkazu **write**. Taky jsme mohli nejdříve návratovou hodnotu funkce přiřadit k proměnné (stejného typu ovšem) a teprve tuto proměnnou použít v příkazu **write**.

**Způsoby komunikace podprogramu s okolím:**

1. parametry předávané hodnotou – pouze vstupní

2. parametry předávané odkazem – vstupní i výstupní

3. návratová hodnota funkce – pouze výstupní

(jen jedna hodnota jednoduchého typu)

4. globální proměnné – vstupní i výstupní

(používat jen ve výjimečných a odůvodněných případech)

*Příklad:* určení NSD Eukleidovým algoritmem

**procedure** NSD (A, B: integer; **var** C: integer);

**begin**

 **while** A<>B **do**

 **if** A>B **then** A:=A-B **else** B:=B-A;

 C:=A

**end**; { Volání: NSD(33,15,X); }

**function** NSD (A, B: integer): integer;

**begin**

 **while** A<>B **do**

 **if** A>B **then** A:=A-B **else** B:=B-A;

 NSD:=A

**end**; {Volání: X:=NSD(33,15);}

**procedure** NSD (A, B: integer);

**begin**

 **while** A<>B **do**

 **if** A>B **then** A:=A-B **else** B:=B-A;

 X:=A

**end**; { Volání: NSD(33,15); }

Poslední uvedený případ nepoužívat! – Procedura vkládá výsledek výpočtu do pevně zvolené globální proměnné.

**Inicializované proměnné deklarované v proceduře**

- jsou to lokální identifikátory viditelné pouze v rámci procedury

- alokují se však staticky → nezanikají po skončení výpočtu procedury, udržují si svou poslední hodnotu do příštího volání

- inicializace se provádí jen jednou na začátku výpočtu programu

**program** P;

 **procedure** S;

 **const** I: byte = 1;

 **begin** inc(I); write(I) **end**; {procedura **inc(I)** přičítá k  **I** číslo **1**. }

**begin**

S; S; S

**end**. Co vypíše? {234}