Porovnání klasických konstrukcí – jména

Jména

-max. délka (Fortran= 6, Fortran90, ANSI C = 31, Cobol 30, C++ neom., ale omezeno implementací; ADA, Java = neom.)

-použitelnost spojovacích znaků nedovolují Fortran77, Pascal, ostatní ano

-case sensitivity (C++,C, Java ano, ostatní ne). Nevýhodou je zhoršení čitelnosti = jména vypadají stejně , ale mají různý význam.

Speciální slova

–Klíčová slova = v určitém kontextu mají speciální význam

–Předdefinovaná slova = identifikátory speciálního významu, které lze předefinovat (např vše z balíku java.lang – String, Object, Systém…)

–Rezervovaná slova = nemohou být použita jako uživatelem definovaná jména (např.abstract, boolean, break, …, if, …, while)

Proměnné = abstrakce paměťových míst

Formálně = 6tice atributů

(jméno, adresa, hodnota, typ, doba existence, rozsah platnosti)

Způsob deklarace: explicitní / implicitní

Porovnání klasických konstrukcí – jména

- Jméno – nemají je všechny proměnné
- Adresa – místo v paměti (během doby výpočtu či místa v programu se může měnit)
- Alias – dvě proměnné sdílí ve stejné době stejné místo (špatnost)
  - Pointery
  - Referenční proměnné
  - Variantní záznamy (Pascal)
  - Uniony (C, C++)
  - Fortran (EQUIVALENCE)
  - Parametry podprogramů
- Typ – určuje množinu hodnot a operací
- Hodnota – obsah přiděleného místa v paměti

- L hodnota = adresa proměnné
- R hodnota = hodnota proměnné
- Binding = vazba proměnné k atributu

Porovnání klasických konstrukcí – jména a typy

- **Statická vazba** (jména s typem / s adresou)
  navázání se provede před dobou výpočtu a po celou exekuci se nemění
  Vazba s typem určena buď explicitní deklarací nebo implicitní deklarací
- **Dynamická vazba** (jména s typem / s adresou)
  nastane během výpočtu nebo se může při exekuci měnit
  - Dynamická vazba s typem
    specifikována přiřazováním (např. Lisp)
    výhoda – flexibilita (např. generické jednotky)
    nevýhoda- vysoké náklady + obtížná detekce chyb při překladu
  - Vazba s pamětí (nastane alokací z volné paměti, končí dealokací)
    doba existence proměnné (lifetime) je čas, po který je vázána na určité paměťové místo.

Kategorie proměnných podle doby existence (lifetime)

- Statické = navázání na paměť před exekucí a nemění se po celou exekuci
  Fortran 77, C static
  výhody: efektivní – přímé adresování, podpr. senzitivní na historii
  nevýhody: bez rekurze

Porovnání klasických konstrukcí – jména a typy

- Dynamické

  V zásobníku = přidělení paměti při zpracování deklarací. Pro skalární proměnnou jsou kromě adresy přiděleny atributy staticky (lokální prom. C, Pascalu).

  výhody: rekurze, nevýhody: režie s alokací/dealokací, ztrácí historickou informaci, neefektivní přístup na proměnné (nepřímé adresy)

  Explicitní na haldě = přidělení / uvolnění direktivou v programu během výpočtu. Zpřístupnění pointery nebo odkazy (objekty ovládané new/delete v C++, objekty Javy)

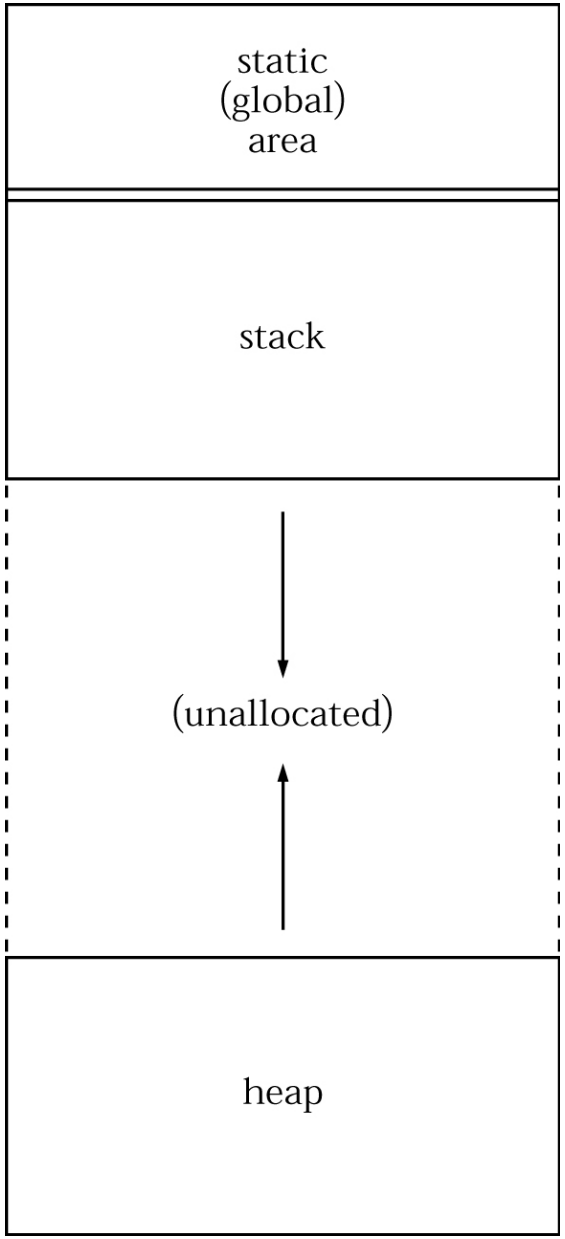
  výhody: umožňují plně dynamické přidělování paměti,

  nevýhody: neefektivní a nespolehlivé (zejm. při slabším typovém systému)

  Implicitní přidělování na haldě = alokace/dealokace způsobena přiřazením (proměnné APL,

  Výhody: flexibilita, nevýhody: neefektivní – všechny atributy jsou dynamické špatná detekce chyb

Většina jazyků používá kombinace -


static
(global)
area
stack
(unallocated)
heap

Porovnání klasických konstrukcí – typy

Typová kontrola je aktivita zabezpečující, že operandy operátorů jsou kompatibilních typů

Kompatibilní typy jsou takové, které jsou buď legální pro daný operátor, nebo jazyk dovoluje implicitní konverzi pomocí překladačem generovaných instrukcí na legální typ (automatická konverze = coercion)

Při statické vazbě s typem je možná statická typová kontrola

Při dynamické vazbě s typem je nutná dynamická typová kontrola.

Programovací jazyk má <u>silný typový systém</u>, pokud typová kontrola odhalí veškeré typové chyby

Problémy s typovou konverzí:

a) Při zužování **R** $\rightarrow$ **I** (rounding/truncation),

b) Při rozšiřování **I** $\rightarrow$ **R**

(možná ztráta přesnosti)

Porovnání klasických konstrukcí – typy

Konkrétní jazyky (silný typový systém):

- Fortran77

  není ← parametry, Equivalence
- Pascal

  není ← variantní záznamy
- C, C++

  není ← lze obejít typovou kontrolu parametrů, uniony nejsou kontrolovány
- ADA, Java

  téměř jsou –

Pravidla pro coerci výrazně oslabují silný typový systém

Porovnání klasických konstrukcí – typy

Kompatibilita typů

Jmenná kompatibilita – dvě proměnné jsou kompatibilních typů, pokud jsou uvedeny v téže deklaraci, nebo v deklaracích používajících stejného jména typu

dobře implementovatelná, silně restriktivní

Strukturální kompatibilita – dvě proměnné jsou kompatibilní mají-li jejich typy identickou strukturu

flexibilnější, hůře implementovatelné

Pascal a C (kromě záznamů),

ADA, Java - jmenná

Porovnání klasických konstrukcí – jména a typy

Rozsah platnosti (scope) proměnné je částí programového textu, ve kterém je proměnná viditelná. Pravidla viditelnosti určují, jak jsou jména asociována s proměnnými

Rozsah existence (lifetime) je čas, po který je proměnná vázána na určité paměťové místo

Statický (lexikální) rozsah platnosti

- Určen programovým textem
- K určení asociace jméno – proměnná je třeba nalézt deklaraci
- Vyhledávání: nejprve lokální deklarace, pak globálnější rozsahová jednotka, . . .
- Proměnné mohou být zakryty (slepé skvrny)
- C++, ADA, Java dovolují i přístup k zakrytým proměnným (Třída.proměnná)
- Prostředkem k vytváření rozsahových jednotek jsou bloky

Dynamický rozsah platnosti

- Založen na posloupnosti volání programových jednotek (namísto hlediska statického tvaru programového textu, řídí se průchodem výpočtu programem)
- Proměnné jsou propojeny s deklaracemi řetězcem vyvolaných podprogramů

Porovnání klasických konstrukcí – jména a typy

```
Př. MAIN
      deklarace x
      SUB 1
            deklarace x
                        …
            call SUB 2
            …
      END SUB 1
      SUB 2
            …
            odkaz na x  // je x  z MAIN nebo  ze SUB1 ?
            …
      END SUB 2
      …
      CALL  SUB 1
      …
      END MAIN
```

Porovnání klasických konstrukcí – jména a typy

Rozsah platnosti (scope) a rozsah existence (lifetime) jsou různé pojmy. Jméno může existovat a přitom být nepřístupné.

Referenční prostředí jsou jména všech proměnných viditelných v daném místě programu

V jazycích se statickým rozsahem platnosti jsou referenčním prostředím jména lokálních proměnných a nezakrytých proměnných obklopujících jednotek

V jazycích s dynamickým rozsahem platnosti jsou referenčním prostředím jména lokálních proměnných a nezakrytých proměnných aktivních jednotek

Porovnání klasických konstrukcí – jména a typy

```
public class Scope
  { public static int x = 20;
    public static void f()
    { System.out.println(x); }
    public static void main(String[] args)
    { int x = 30;
      f();
    }
  }
```

Java používá statický scope, takže tiskne . . .

Pokud by používala dynamický, pak tiskne . . .

Dynamický používá originální LISP, VBScript, Javascript, Perl (starší verze)

Porovnání klasických konstrukcí – jména a typy

Konstanty (mají fixní hodnotu po dobu trvání jejich existence v programu, nemají atribut adresa = na jejich umístění nelze v programu odkazovat) :

- Určené v době překladu– př.Javy:
  `static final int zero = 0;`
- Určené v době zavádění programu
  `static final Date now = new Date();`

statické

- Dynamické konstanty: -v C# konstanty definované readonly
  -v Javě: každé non-static final přiřazení v konstruktoru.
  -v C:

```
#include <stdio.h>
const int i = 10;                    // ?
const int j = 20 * 20 + i;           // ?
int f(int p) {
    const int k = p + 1;             // ?
    return k + I + j;
}
```

- Literály = konstanty, které nemají jméno
- Manifestová konstanta = jméno pro literál

Porovnání klasických konstrukcí – typy

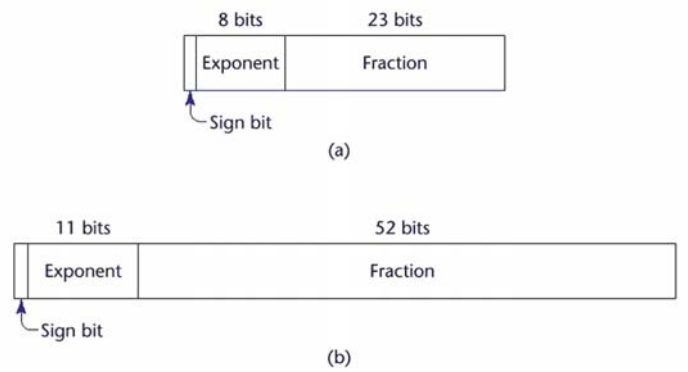
Typ: Definuje kolekci datových objektů a operací na nich proveditelných

- primitivní = jejich definice nevyužívá jiných typů
- složené

Integer – reflektují harwareové možnosti počítače, aproximace celých čísel

Floating Point – obvykle reflektují hardware, aproximace reálných čísel (např. ADA type delka is digits 12 range 0.0 ..300000.0;)

v jazycích pro V V výpočty zaváděn min. ve dvou formách



Decimal -pracují s přesným počtem cifer (finanční aplikace)

Boolean –obvykle implementovány bytově, lze i bitově. V C je nahražen ***int*** *0 / nenula*

Character –určeno kódováním ASCII (128 znaků), UNICODE (16 bitů) viz Java i C#

Porovnání klasických konstrukcí – typy

Ordinální (zobrazitelné/přečíslitelné do integer)
- primitivní mimo float
- definované uživatelem (pro čitelnost a spolehlivost programu)
  - **vyjmenované typy** =uživatel vyjmenuje posloupnost hodnot typu,
    Implementují se jako seznam pojmenovaných integer konstant,
    např Pascal, ADA, C++
    type BARVA = (BILA, ZLUTA, CERVENA, CERNA) ;
    C# př. enum dny {pon, ut, str, ctvr, pat, sob, ned};
    Java je má od 5.0. př. public enum Barva (BILA,ZLUTA,CERVENA,CERNA);
    V nejjednodušší podobě lze chápat také jako seznam integer
    Realizovány ale jako typ třída Enum. Možnost konstruktorů, metod, …
    ale uživatel nemůže vytvářet potomky Enum
    enum je klíčové slovo.
  - **typ interval** =souvislá část ordinálního typu. Implementují se jako
    typ rodiče (např. type RYCHLOST = 1 .. 5)

Výhody: čitelnost, bezpečnost

Porovnání klasických konstrukcí – typy

String – hodnotou je sekvence znaků

- Pascal, C, C++ = neprimitivní typ, pole znaků
- Java má typ, String class – hodnotou jsou konstantní řetězce, StringBuffer class – lze měnit hodnoty a indexovat, je podobné jako znaková pole
- ADA, Fortran90, Basic, Snobol = spíše primitivní typ, množství operací
- délka řetězců:
  - statická (Fortran90, ADA, Cobol, String class Javy), efektivní implementace
  - limitovaná dynamická (C, C++ indikují konec znakem null)
  - dynamická (Snobol4, Perl), časově náročná implementace

Porovnání klasických konstrukcí – typy

Array – agregát homogenních prvků, identifikovatelných pozicí relativní k prvému prvku

? typ indexů

C, Fortran, Java celočíselné, ADA,Pascal ordinální

? Způsob alokace

1. Statická pole (= pevné délky)
   ukládaná do statické oblasti paměti (Fortran77), globální pole Pascal, C.
   Meze indexů jsou konstantní
2. Statická pole ukládaná do zásobníku
   (Pascal lokální, C lokální mimo static)
3. Dynamická v zásobníku . (ADA)
   (= délku určují hodnoty proměnných). Flexibilní
4. Dynamická na haldě (Fortran90, Java, Perl).

Porovnání klasických konstrukcí – typy

? Počet indexů

-Fortran77 - do 7,

-C, C++ ,Java jen 1 ale prvky mohou být pole,

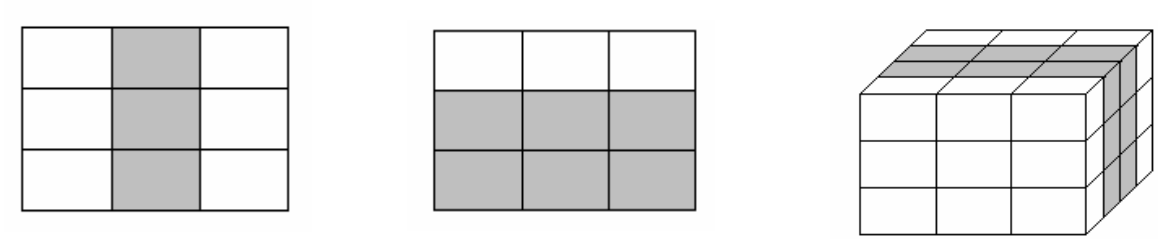
-ostatní neomezeně.

? Operace na polích

-běžně přiřazování a test rovnosti / nerovnosti polí, někdy inicializace

-Fortran90 – maticové násobení, řezy, výřezy

INTEGER MAT(1:3, 1:3), CUBE(1:3, 1:3, 1:4)



MAT(1:3,2) MAT(2:3,1:3) CUBE(1:3,1:3,2:3)

MAT = CUBE(1:3,1:3, 2)

Porovnání klasických konstrukcí – typy

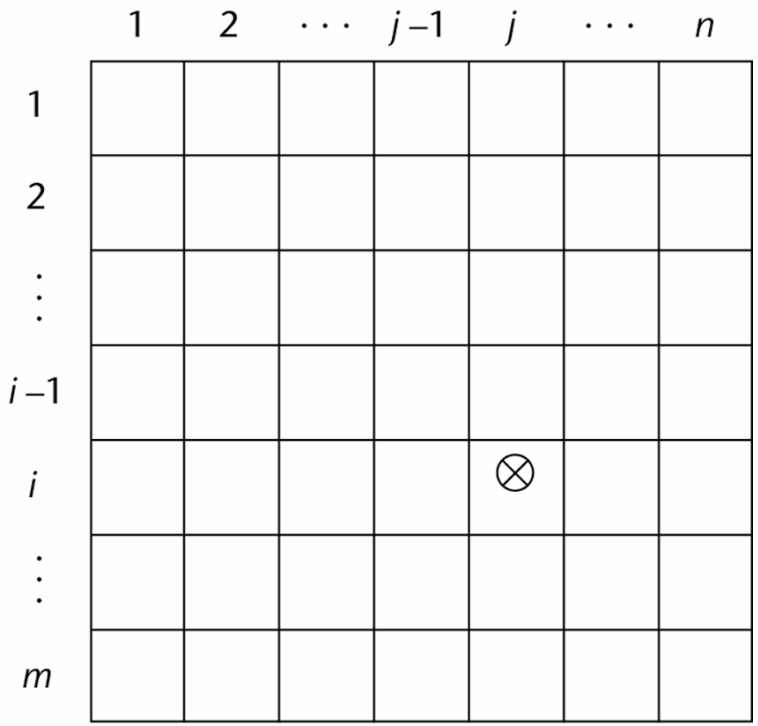
Přístupová fce pro jednorozměrné pole

location(vector[k]) = address (vector[lower_bound])

+ ((k-lower_bound) * element_size)

Přístupová fce pro vícerozměrná pole j1

location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size



j1 řazení po sloupcích / po řádcích

jezek_ka; 19.1.2006

Porovnání klasických konstrukcí – typy

Co o poli potřebuje vědět překladač´= deskriptor pole

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Porovnání klasických konstrukcí – typy

-Perl – asociativní pole = neuspořádaná kolekce dvojic (klíč, hodnota), nemají indexy

- Jména začínají `%`; literály jsou odděleny závorkami

```
%cao_temps = ("Mon" => 77, "Tue" => 79,
  "Wed" => 65, …);
```

- Zpřístupnění je pomocí slož.závorek s klíčem

```
$cao_temps{"Wed"} = 83;
```

  – Prvky lze odstranit pomocí `delete`

```
delete $cao_temps{"Tue"};
```

Porovnání klasických konstrukcí – a typy

Record –(záznam) možně heterogenní agregát datových prvků, které jsou zpřístupněny jménem (kartezský součin v prostoru typů položek)
odkazování na položky OF notací Cobol, ostatní „.“ notací
Př C **struct {int i; char ch;} v1,v2,v3;**

Operace -přiřazení (pro identické typy), inicializace, porovnání
Uniony – typy, jejichž proměnné mohou obsahovat v různých okamžicích výpočtu hodnoty různých typů.

Př. C **union u_type {int i; char ch;} v1,v2,v3; /* free union- nekontroluje typ*/**
Př.Pascal

```
type R = record
        ...
        case RV : boolean of          /*discriminated union*/
                false : (i : integer);
                true : (ch : char)
        end;
var V : R; ...
V.RV := false; V.i := 2; V.RV := true; write(V.ch);
```

řádně se přeloží a vypíše nesmysl. %

Porovnání klasických konstrukcí – typy

```
type R = record
        ...
        case boolean of                    /* to je free union */
                false : (P: pointer);
                true : (i: integer);
        end;
```

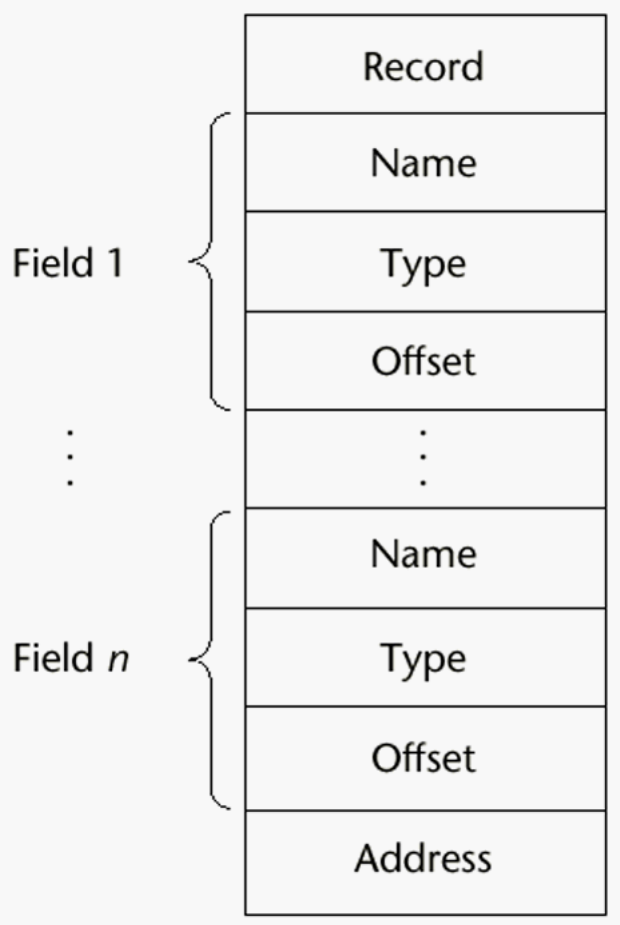
Důsledek – Pascal nemůže mít garbage collector

- ADA má proto povinný diskriminant a pokud se přiřazuje diskriminující položce, musí se přiřadit celému záznamu - bezpečné
- Java nemá typ záznam, má třídy

Porovnání klasických konstrukcí – typy

Přístup k prvkům záznamu je mnohem rychlejší než k prvku pole

$$\text{Location (record.field_i)} = \text{address (record.field_1)} + \sum_{i=1}^{n-1} \text{offset_i}$$



Porovnání klasických konstrukcí – typy

Set – typ, jehož proměnné mohou mít hodnotu neuspořádané kolekce hodnot ordinálního typu

Zavedeny v Pascalu

*type NejakyTyp = Set of OrdinalniTyp (*třeba char*);*

Java má třídu pro set operace, Ada a C je nevedou

Implementace – bitovými řetězci, používají logické operace

Vyšší efektivita než pole ale menší pružnost (omezovaný počet prvků)

Porovnání klasických konstrukcí – typy

Pointer - typ nabývající hodnot paměťového místa a nil (někde null)

Použití pro: -nepřímé adresování normálních proměnných

-dynamické přidělování paměti

Operace s ukazateli: -přiřazení, dereference

Špatnost ukazatelů = dangling (neurčená hodnota) pointers a ztracené proměnné

Pascal – má jen pointery na proměnné z heapu

alokované pomocí **new** a uvolňované **dispose**

- dereference tvaru jménopointeru^.položka

new(P1); P2 := P1; dispose(P1); P2 je teď dangling pointer

Problém ztracených proměnných

new(P1); .... new(P1); * b)

V místě a) i b) je ztracená paměť

```
P1: ^integer;
...
new(P1);
...
```

a)

- implementace dispose znemožňující dangling není možná

Porovnání klasických konstrukcí – typy

C, C++ * je operátor dereference

& je operátor produkující adresu proměnné

`j = *ptr` přiřadí j hodnotu umístěnou v `ptr`

Pointer bývá položkou záznamu, pak tvar: *p.položka, nebo p → polozka

Zpřístupnění pointerem mají vlastnosti adres v JSI (flexibilní a nebezpečné)

Pointerová aritmetika pointer + index

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` je ekvivalentní `stuff[5]` nebo `p[5]`

`*(p+i)` je ekvivalentní `stuff[i]` nebo `p[i]`

Pointer může ukazovat na funkci (umožňuje přenášet fce jako parametry)

Dangling a ztraceným pointerům nelze nijak zabránit

Porovnání klasických konstrukcí – typy

Nebezpečnost pointerů v C:

```
main()
{ int x, *p;
    x = 10;         /* p =&x */
    *p = x;
    return 0;
}
```

Pointer je neinicializovaný, hodnota x se přiřadí do neznáma

```
main()
{ int x,*p;
    x = 10; p=x;            /* p =&x */
    printf("%d",*p);
    return 0;
}
```

Pointer je neinicializovaný, tiskne neznámou hodnotu

Porovnání klasických konstrukcí – typy

ADA – pouze proměnné z haldy (access type)

– dereference (pointer . jméno_položky)

– dangling významně potlačeno (automatické uvolňování místa na haldě, jakmile se výpočet dostane mimo rozsah platnosti ukazatele)

Hoare: "The introduction of pointers into high level languages has been a step backward" (flexibility ↔ safety)

Java:

– nemá pointery

– má referenční proměnné (ukazují na objekty místo do paměti)

– referenčním proměnným může být přiřazen odkaz na různé instance třídy. Instance Java tříd jsou dealokovány implicitně ⇒ dangling reference nemůže vzniknout.

– Paměť haldy je uvolněna garbage collectorem, poté co systém detekuje, že již není používána.

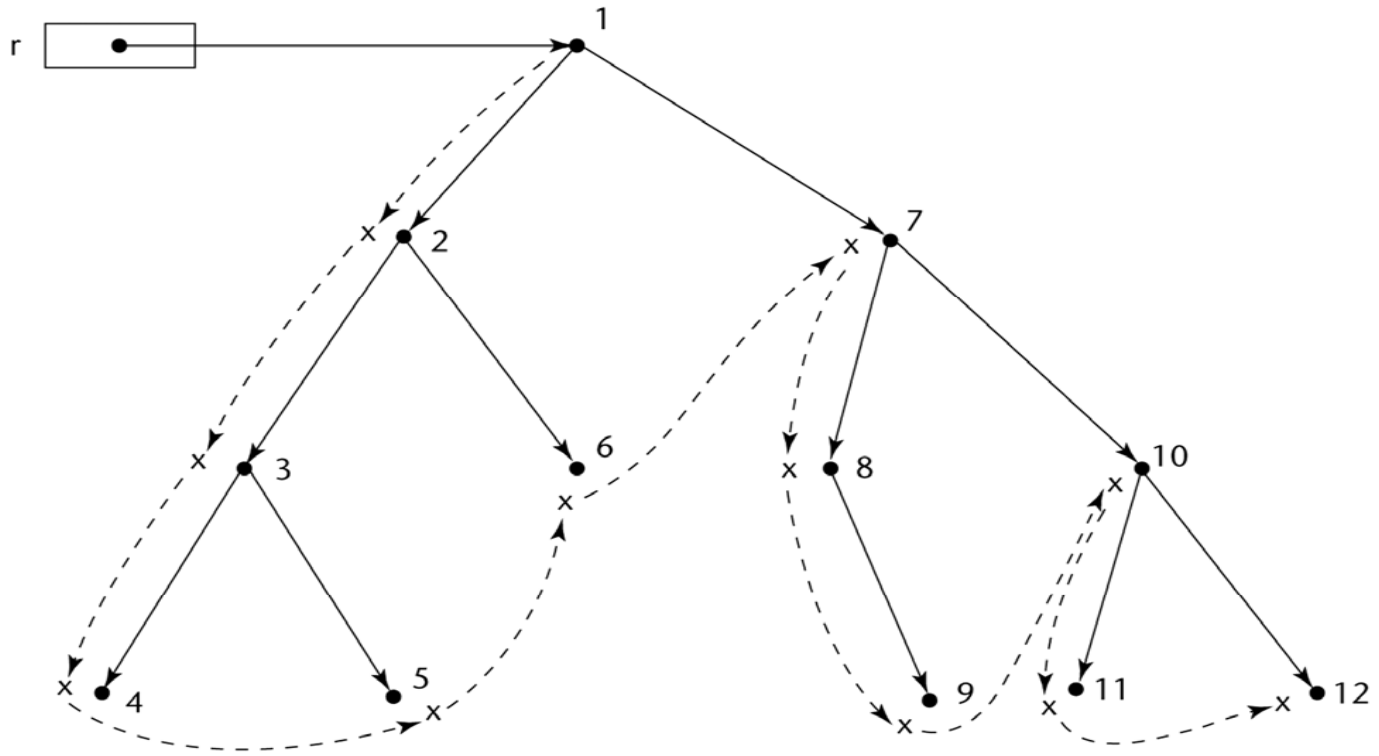
C#:

- má pointry tvaru *referent-type * identifikátor* type může být i void
- metody pracující s pointerem musí mít modifikátor *unsafe*
- referenční proměnné má také

Porovnání klasických konstrukcí – **Manažování haldy**:

-Čítačem odkazů (každá buňka vybavena čítačem, když 0⇒vrátí do volných)

-Garbage collectorem (když nemá, prohledá všechny buňky a haldu zdrcne)



Průběh označování paměťových míst čističem

Porovnání klasických konstrukcí – výrazy a příkazy

Výrazy - aritmetické
-logické

Ovlivnění vyhodnocení:

1. Precedence operátorů?
2. Asociativita operátorů?
3. Arita operátorů?
4. Pořadí vyhodnocení operandů?
5. Je omezován vedlejší efekt na operandech?
   X=f(&i)+(i=i+2); v C
6. Je dovoleno přetěžování operátorů?
7. Alternativnost zápisu (Java, C) C=C+1; C+=1; C++; ++C

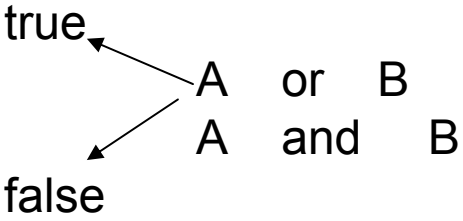
Porovnání klasických konstrukcí – výrazy a příkazy

```
#include <stdio.h>
int f(int *a);

int main()
{int x,z;
 int y=2; int i=3;
 /*C, C++, Java přiřazení produkuje výslednou hodnotu použitelnou jako operand*/
 x = (i=y+i) + f(&i);   /* ?pořadi vyhodnoceni*/
 printf("%d\n",i);  printf("%d\n",x);

 y=2; i=3;
 z = f(&i) + (i=y+i);   /* ?pořadi vyhodnoceni*/
 printf("%d\n",z); printf("%d\n",i);
 getchar();
 return 0;
}

int f(int *i)
{int x;
 *i = *i * *i;
 return *i;
}
```

Porovnání klasických konstrukcí – výrazy a příkazy

! u C, C++ na záměnu **if (x = y) …** /*je přiřazením ale produkuje log. hodnotu*/
se zápisem **if (x == y) …**
Proto C#, Java dovolují za **if** pouze logický výraz

Logické výrazy nabízí možnost zkráceného vyhodnocení



ADA

if A and then B then S1 else S2 end if;
if A or else B then S1 else S2 end if;

zkrácené:

if A and then B then S1 else S2 end if;
if A or else B then S1 else S2 end if;

Java

např. pro **(i=1;j=2;k=3;):**

if (i == 2 && ++j == 3) k=4; ?jaký výsledek
if (i == 2 & ++j == 3) k=4; ?jaký výsledek

Porovnání klasických konstrukcí – výrazy a příkazy

Využití podmíněných výrazů

- **v přiřazovacích příkazech C jazyků, Javy** `k = ( j == 0) ? j+1 : j-1 ;`
  - **(pozn. v C jazycích, Javě produkuje přiřazení hodnotu, může být ve výrazu.)**
- **v řídících příkazech**
  - **neúplný a úplný podmíněný příkaz**
  - **problém „dangling else“ při vnořovaném „if“**

if x=0 then if y=0 then z:=1 else z:=2;

řešení: Pascal, Java – else patří k nejbližšímu nespárovanému if
ADA – párování if … end if

Porovnání klasických konstrukcí – výrazy a příkazy

Příkaz vícenásobného selektoru - přepínač

Pascal, ADA:

```
case expression of
    constant_list : statement1;
            ....
    constant_listn : statementn;
end;
```

Alternativa else / others. „Návěští“ ordinálního typu

C jazyky, Java

```
switch (expression) {
    case constant_expr1 : statements;
            ....
    case constant_exprn : statementsn;
    default : statements
}
```

Alternativy u **C, C++, Java** separuje „break“, u **C#** také goto.

„Návěští“ je výraz typu **int, short, char, byte** u **C#** i **enum, string**.

Porovnání klasických konstrukcí – výrazy a příkazy

<u>Cykly</u>

loop … end loop**;** primitivní tvar

while *podmínka* do …; cyklus logicky řízený pretestemPascal,ADA
while (*podmínka*) *příkaz*; Cjazyky, Java

repeat … until *podmínka*; cyklus logicky řízený posttestem Pascal
do { příkazy ; } while (podmínka); Cjazyky, Java

for … ; s parametrem cyklu, krokem,iniciální a koncovou hodnotou

Fortran zavedl:

```
      DO 10 I = 1, 5
      ...
   10 CONTINUE
```

Pascal:

for *variable* := *init* **to** *final* **do** *statement*;

po skončení cyklu není hodnota *variable* definována

ADA obdobně, ale *variable* je implic. deklarovanou proměnnou cyklu, vně neexistuje

Porovnání klasických konstrukcí – výrazy a příkazy

C++, C#, Java

```
for (int count = 0; count < fin; count++) { ...      } ;
        1.                    2.        4.        3.
```

Lze deklarovat čítač přímo v cyklu, pak

-v C++ platí až do konce funkce

-v Javě platí jen do konce cyklu

Co charakterizuje cykly:

- Jakého typu mohou být parametr a meze cyklu?
- Kolikrát se vyhodnocují meze a krok?
- Kdy je prováděna kontrola ukončení cyklu?
- Lze uvnitř cyklu přiřadit hodnotu parametru cyklu?
- Jaká je hodnota parametru po skončení cyklu?
- Je přípustné skočit do cyklu?
- Je přípustné vyskočit z cyklu?

Porovnání klasických konstrukcí – výrazy a příkazy

<u>Rozporný příkaz skoku</u>

Nevýhody - znepřehledňuje program
- je nebezpečný
- znemožňuje formální verifikaci programu

Výhody - snadno implementovatelný
- efektivně implementovatelný

Formy návěští:

číslo: Pascal

číslo Fortran

identifikátor: Cjazyky

<<identifikátor>> ADA

proměnná PL/1

Java nemá skok GOTO, částečně jej nahrazuje break

```
nav1:   {
            nav2:   {       break nav2;
                    }
        }
```

Porovnání klasických konstrukcí – výrazy a příkazy

<u>Zásada: Používej skoků co nejméně</u>

Př. Katastrofický důsledek snahy po obecnosti

```
B1: BEGIN; DCL L LABEL;
    ...
B2: BEGIN; DCL X,Y FLOAT;
            ...
L1: Y=Y+X;
            ...
L=L1;
END;
    ...
GOTO L; ...
END;
```

Porovnání klasických konstrukcí – podprogramy

Procedury a Funkce jsou nejstarší formou abstrakce – abstrakce procesů

(Java a C# nemají funkce, ale metody mohou mít libovolný typ)

Základní charakteristiky:

- Podprogram má jeden vstupní bod
- Volající je během exekuce volaného podprogramu pozastaven
- Po skončení běhu podprogramu se výpočet vrací do místa, kde byl podprogram vyvolán

Pojmy:

- Definice podprogramu
- Záhlaví podprogramu
- Tělo podprogramu
- Formální parametry
- Skutečné parametry
- Korespondence formálních a skutečných parametrů
  - jmenná vyvolání: jménopp(jménoformálního jménoskutečného, …
  - poziční vyvolání: jménopp(jménoskutečného, jménoskutečného…
- Default (předběžné) hodnoty parametrů

Porovnání klasických konstrukcí – podprogramy

Kritéria hodnocení podprogramů:

- Způsob předávání parametrů
- Možnost typové kontroly parametrů
- Jsou lokální proměnné umisťovány staticky nebo dynamicky?
- Jaké je platné prostředí pro předávané parametry, které jsou typu podprogram
- Je povoleno vnořování podprogramů
- Mohou být podprogramy přetíženy (různé pp mají stejné jméno)
- Mohou být podprogramy generické
- Je dovolena separátní kompilace podprogramů

Ad umístění lokálních proměnných:

-dynamicky v zásobníku

umožní rekurzivní volání a úsporu paměti

potřebuje čas pro alokaci a uvolnění, nezachová historii, musí adresovat nepřímo

(Pascal, Ada výhradně dynamicky, Java, C většinou)

-dynamicky na haldě (Smalltalk)

-staticky, pak opačné vlastnosti (Fortran90 většinou, C static lokální proměnné)

Porovnání klasických konstrukcí – podprogramy

Lokální data podprogramů jsou spolu s dalšími údaji uloženy v AKTIVAČNÍM ZÁZNAMU

Místo pro lokální proměnné
Místo pro předávané parametry
(Místo pro funkční hodnotu u funkcí)
Návratová adresa
Informace o uspořádání aktivačních záznamů
Místo pro dočasné proměnné při vyhodnocování výrazů

Aktivační záznamy většiny jazyků jsou umístěny v zásobníku.

- Umožňuje vnořování rozsahových jednotek
- Umožňuje rekurzivní vyvolání

-Aktuální AZ je přístupný prostřednictvím ukazatele (nazvěme ho B) na jeho bázi

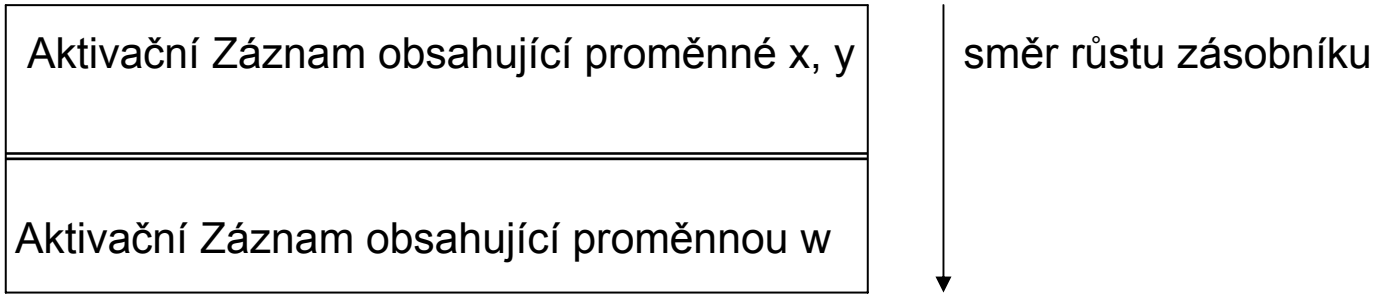
-Po skončení rozsahové jednotky je tato odstraněna ze zásobníku dle ukazatele uspořádání AZ

Porovnání klasických konstrukcí – podprogramy

Př.

```
{ int x; int y;
        { int z; //vnořené
        }      //rozsahové
        { int w; //jednotky
        // místo 1
        }
}
```

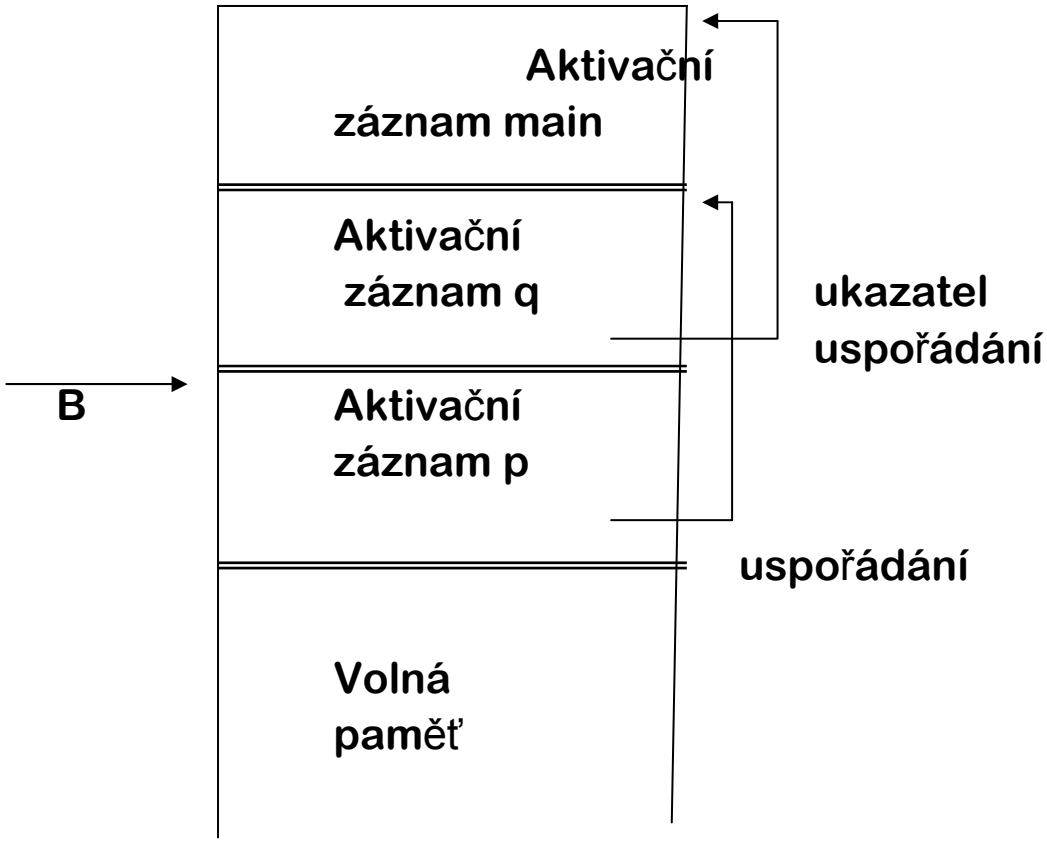
Zásobník v místě1



Porovnání klasických konstrukcí – podprogramy

Př. v jazyce C

```
int x;
void p( int y)
{ int i = x;
  char c; ...
}
void q ( int a)
{ int x;
  p(1);
     ukazatel
}
main()
{ q(2);
  return 0;
}
```



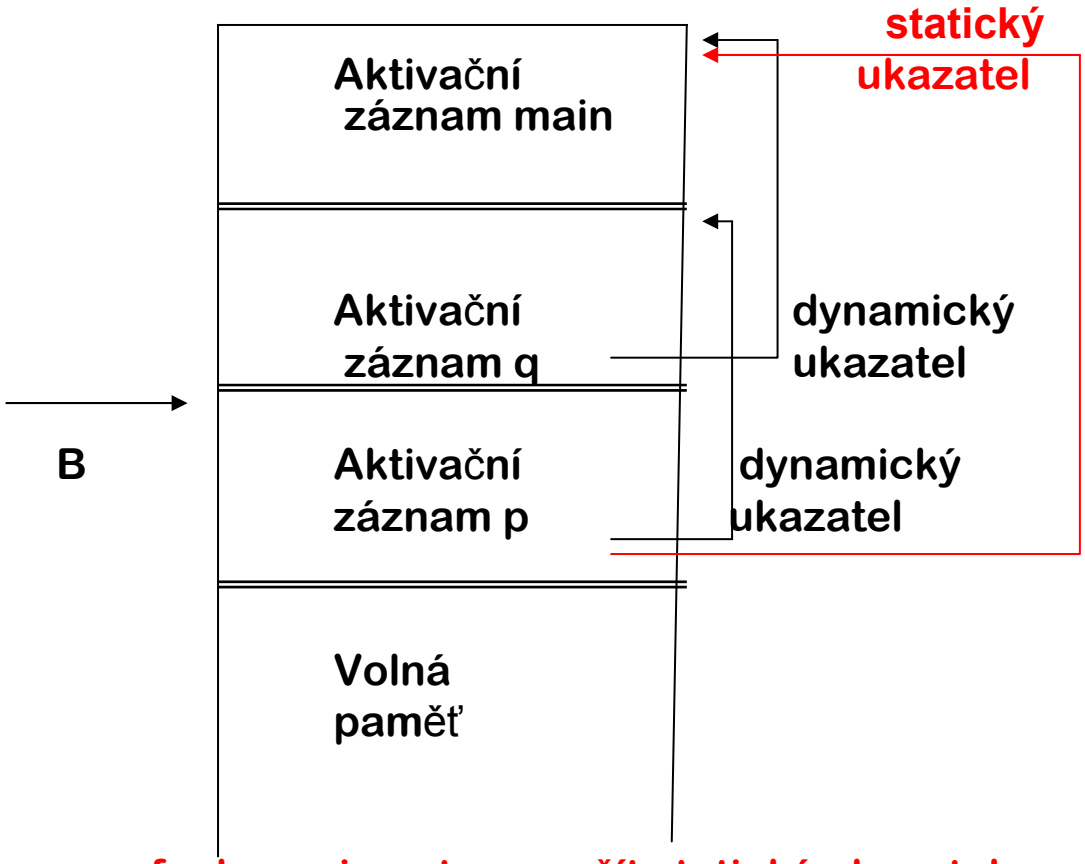
Jazyky se statickým rozsahem platnosti proměnných a vnořováním podprogramů vyžadují dva typy ukazatelů uspořádání (řetězců ukazatelů):

1. **(dynamický) Na rušení AZ opuštěných rozsahových jednotek (viz výše)**
2. **(statický) pro přístup k nelokálním proměnným**

Porovnání klasických konstrukcí – podprogramy

Př. v jazyce C

```
int x;

void p( int y)
{ int i = x;
  char c; ...
}
void q ( int a)
{ int x;
  p(1);
}
main()
{ q(2);
  return 0;
}
```



K přistupování na proměnnou x z funkce p je nutno použít statický ukazatel

V C, C++, Javě má statický řetěz délku 1, v Adě, Pascalu může nabývat libovolné délky

Porovnání klasických konstrukcí – podprogramy

- Použití řetězce dynamických ukazatelů k přístupu k nelokálním proměnným způsobí, že nelokální proměnné budou zpřístupněny podle dynamické úrovně AZ
- Použití řetězce statických ukazatelů způsobí, že nelokální proměnné budou zpřístupněny podle lexikálního tvaru programu
- Cjazyky, Java mají buď globální (static) proměnné, které jsou přístupné přímo, nebo lokální patřící aktuálnímu objektu / vrcholovému AZ, které jsou přístupné přes „this“ pointer
- Jazyky s vnořovanými podprogramy při odkazu na proměnnou, která je o n úrovní globálnější než-li aktuálně prováděný podprogram, musí sestoupit do příslušného AZ o n úrovní statického řetězce.
- Úroveň vnoření L rozsahových jednotek, potřebnou velikost AZ a offset F proměnných v AZ vůči jeho počátku zaznamenává překladač. (L, F) je dvojice, která reprezentuje adresu proměnné.

Porovnání klasických konstrukcí – podprogramy

Způsoby předávání parametrů:

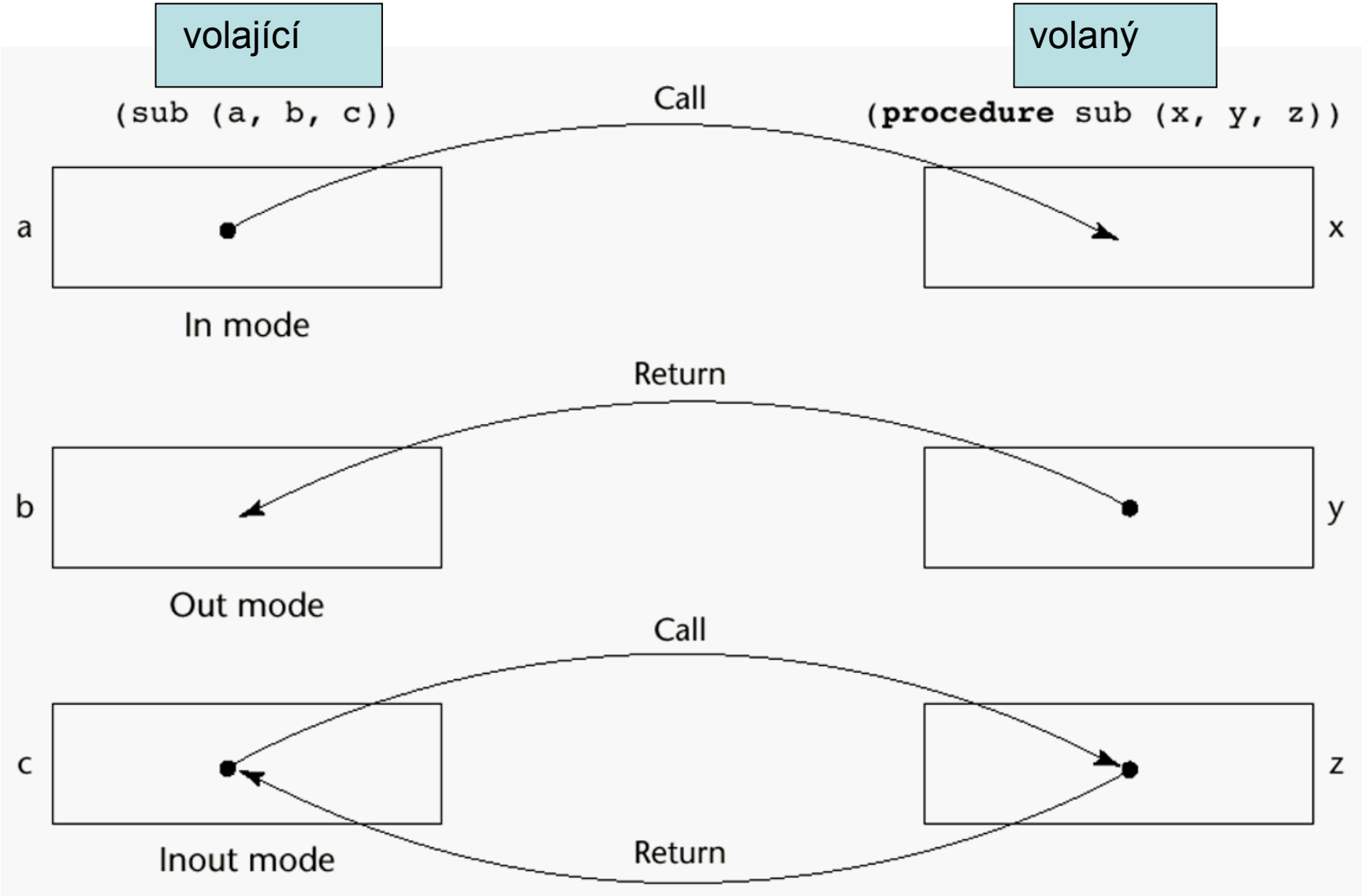
- Hodnotou (in mode), obvykle předáním hodnoty do parametru (lokální proměnné) podprogramu. Vyžaduje více paměti, zdržuje přesouváním
- Výsledkem (out mode), do místa volání je předána při návratu z podprogramu lokální hodnota. Vyžaduje dodatečné místo i čas na přesun
- Hodnotou výsledkem (in out mode), kopíruje do podprogramu i při návratu do místa volání. Stejné nevýhody jako předešlé
- Odkazem (in ouit mode), předá se přístupová cesta. Předání je rychlé, nepotřebuje další paměť. Parametr se musí adresovat nepřímo, může způsobit synonyma.

```
podprogram Sub( a ,  b) ;
. . .
Sub( x , x) ;          . . .
```

- Jménem (in out mode), simuluje textovou substituci formálního parametru skutečným. Neefektivní implementace, umožňuje neprůhledné triky
- Předání vícerozměrného pole – je-li podprogram separátně překládán, potřebuje znát velikost pole. C, C++ má pole polí ukládané po řádcích, Údaje pro mapovací fci požadují zadání počtu sloupců např. void fce (int matice [ ] [10]) {…} v definici funkce. Výsledná nepružnost vede k preferenci použití pointerů na pole.

  Java má jednorozměrná pole s prvky opět pole. Každý objekt pole dědí length atribut. Lze proto deklarovat pružně float fce (float matice [ ] [ ]) { …} Podobně ADA

Porovnání klasických konstrukcí – podprogramy



Porovnání klasických konstrukcí – podprogramy

- Podprogramy jako parametry, C, C++ dovolují předávat jen pointery na funkce, Ada nedovoluje

```
sub1 {
        sub2 {
        }
        sub3 {
                call sub4 (sub2)
        }
        sub4 (subformalni)
                call subformalni
        }
        call sub3
}
```

Jaké je výpočtové prostředí sub2 po jeho vyvolání v sub4 ?

1. Mělká vazba – platné je prostředí volajícího podprogramu (sub4)
2. Hluboká vazba – platí prostředí, kde je definován volaný podprogram /sub1)
3. Ad hoc vazba – platí prostředí příkazu volání, který předává podprogram jako parametr (sub3)

Blokově strukturrované jazyky používají 2, SNOBOL užívá 1, 3 se neužívá

Porovnání klasických konstrukcí – podprogramy

Přetěžovaný podprogram je takový, který má stejné jméno s jiným podprogramem a existuje ve stejném prostředí platnosti (C++, Ada, Java). Poskytuje Ad hoc polymorfismus

C++, ADA dovolují i přetěžování operátorů

```
function "*"(A, B: INT_VECTOR_TYPE) return INTEGER is
    S: INTEGER := 0;
begin
    for I in A'RANGE loop
            S:= S + A(I) * B(I);
    end loop;
    return S;
end  "*";
```

C++

```
int  operator *(const vector &a, const vector &b);  //function prototype
```

Porovnání klasických konstrukcí – podprogramy

Generické podprogramy dovolují pracovat s parametry různých typů. Poskytují parametrický polymorfismus

C++ obecný tvar:

template<class parameters> function definition that may include the class parameters

Example:

```
template <class Typf>
Typf max(Typf first, Typf second) {
return first > second ? first : second;
}
```

instantiation is possible for any type where > is defined, e.g. integer

```
int max(int first, int second)
 {return first > second ? first : second;}
```
1)

1) the effect is as this

C++ template function is instantiated implicitly, either when the function is named in a call or when its address is taken with the & operator

Let
```
int a, b, c;
char d, e, f;
...
c = max(a, b);
```

Porovnání klasických konstrukcí – podprogramy

ADA generické funkce:

```
generic
    GENERIC FORMAL PARAMETERS
function NAME (PARAMETERS)  return TYPE;
function NAME (PARAMETERS) return TYPE  is
    DECLARATIONS
begin
STATEMENTS

end  NAME
```

```
generic
    type ITEM is (<>);
function MAXIMUM(X, Y: ITEM) return ITEM;
function MAXIMUM(X,Y: ITEM return ITEM is
begin
    if X > Y then return X;
            else return Y;
    end if;
end MAXIMUM;
```