Porovnání klasických konstrukcí – jména

Jména (identifikátory)

-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)
- Aliasy – 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

Kategorie proměnných podle vazby s typem a s paměťovým místem

- **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.

Porovnání klasických konstrukcí – jména a typy

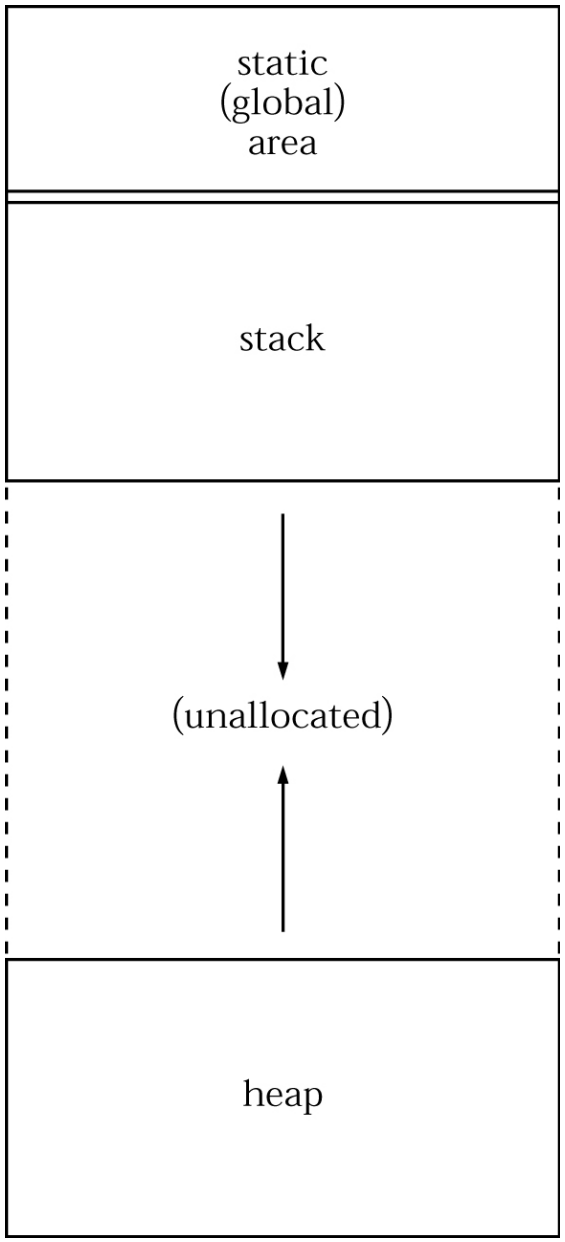
Kategorie proměnných podle doby existence (lifetime)

- 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
- Dynamické
  V zásobníku = přidělení paměti při exekuci 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
  Výhody: flexibilita, nevýhody: neefektivní – všechny atributy jsou dynamické špatná detekce chyb

Většina jazyků používá kombinace – násl. obr.ukazuje rozdělění pam.prostoru


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 = anglicky coercion)

Při statické vazbě s typem je možná statická typová kontrola

Při dynamické vazbě s typem je nutná dynamická typová kontrola.

Programovací jazyk má silný typový systém, pokud typová kontrola odhalí veškeré typové chyby

Problémy s typovou konverzí:

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

b) Při rozšiřování **I** $\rightarrow$ **R** (možná ztráta přesnosti)

Porovnání klasických konstrukcí – typy

Konkrétní jazyky (které mají/nemají silný typový systém):

- Fortran77

  nemá z důvodů parametrů, příkazu Equivalence

- Pascal

  není plně silný protože dovoluje variantní záznamy

- C, C++

  nemá z důvodů lze obejít typovou kontrolu parametrů, uniony nejsou kontrolovány

- ADA, Java

  téměř jsou –

Pravidla pro **coerci** (implicitně prováděnou konverzi) výrazně oslabují silný typový systém

Porovnání klasických konstrukcí – typy

Kompatibilita typů se určuje na základě:

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

dobře implementovatelná, silně restriktivní

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

flexibilnější, hůře implementovatelné

Pascal a C (kromě záznamů) používá strukturální,

ADA, Java - jmennou

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, pak ještě globálnější. . . Uplatní se pokud jazyk dovolí vnořování prog.jednotek
- 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 . . .20

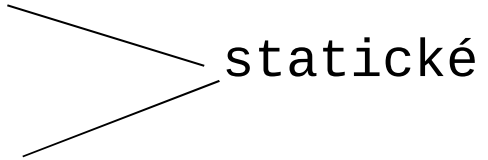
Pokud by používala dynamický, pak tiskne . . .30

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();`



- 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;     // i je statická urč.při překladu
const int j = 20 * 20 + i;// j  „      -------      „
int f(int p) {
    const int k = p + 1; // k je dynamická
    return k + I + j;
        }
```

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

Porovnání klasických konstrukcí – typy

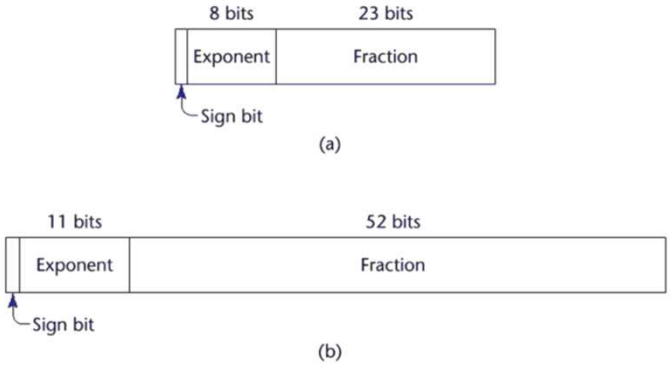
Typ: Definuje kolekci datových objektů a operací na nich proveditelných

- 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;) – !!!pozor na konverze!!!

v jazycích pro vědecké 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 –kódování ASCII (128 znaků), UNICODE (16 bitů) Java, Python i C#

Porovnání klasických konstrukcí – typy

Ordinální (zobrazitelné=přečíslitelné do integer). Patří sem:

- primitivní mimo float
- definované uživatelem (pro čitelnost a spolehlivost programu). Zahrnují:
  - **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 jejich rodiče (např. type RYCHLOST = 1 .. 5)

Výhody:ordinálních typů jsou 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,Python), časově náročná implementace

Porovnání klasických konstrukcí – typy

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

V jazycích odlišnosti:

jaké mohou být typy indexů ?

C, Fortran, Java celočíselné, ADA,Pascal ordinální

? Způsob alokace

1. Statická pole (= pevné délky)
   ukládaná do statické oblasti paměti (Fortran77), globální pole Pascalu, 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,

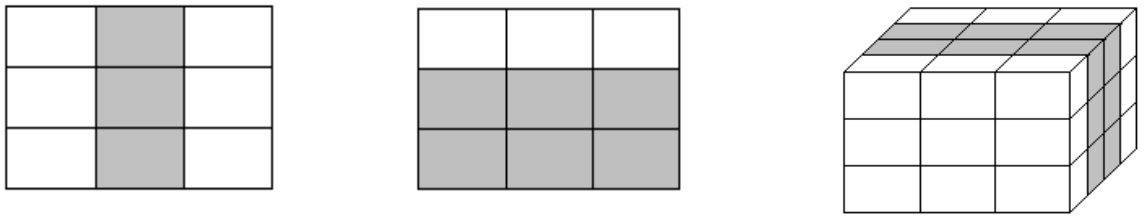
-ostatní neomezeně.

? Operace na polích

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

-Fortran90 dovoluje– 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)

Obr.Příklady řezů a výřezů

Př. priřazení MAT = CUBE(1:3,1:3, 2)

Porovnání klasických konstrukcí – typy

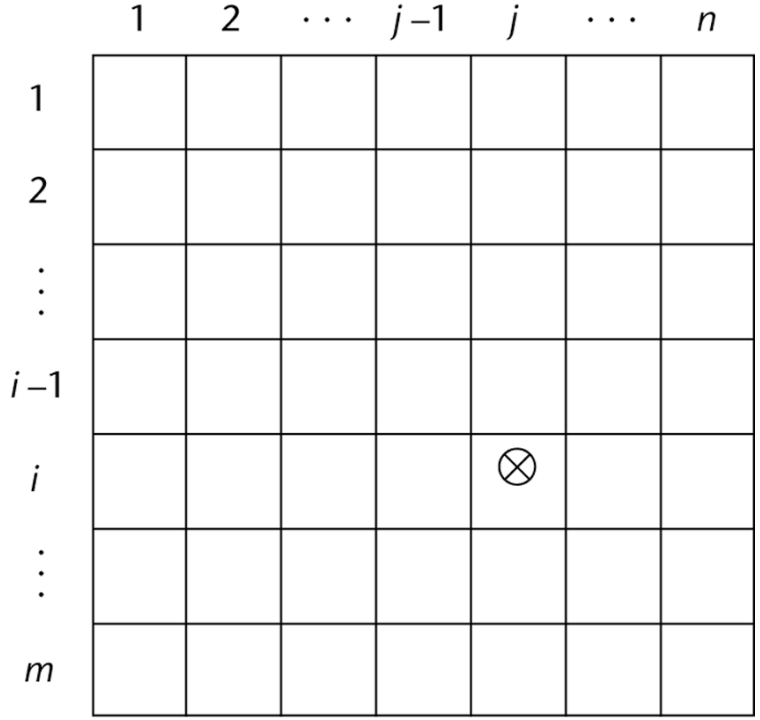
Přístupová fce pro jednorozměrné pole má tvar

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

+ ((k-lower_bound) * element_size)

Přístupová fce pro vícerozměrná pole (řazení po sloupcích / řádcích)

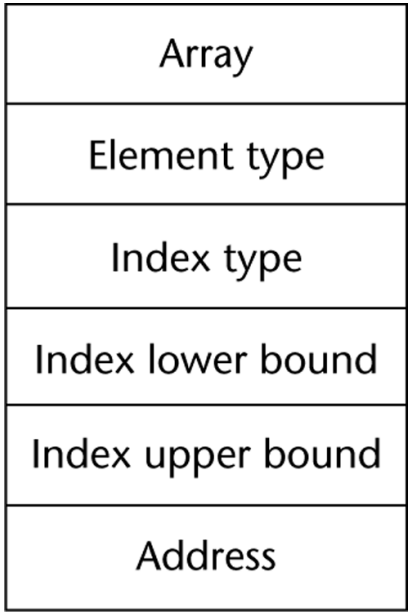
location (a[i,j]) = address of a [row_lb,col_lb] + (((i - row_lb) * n) + (j - col_lb)) * element_size // lb = lower bound



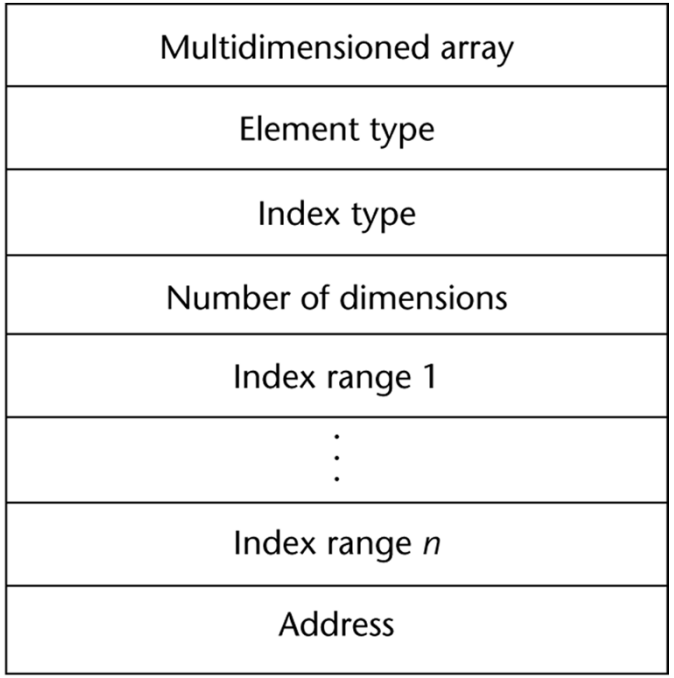
Porovnání klasických konstrukcí – typy

Co o poli potřebuje vědět překladač´je tzv. deskriptor pole

jednorozměrné



vícerozměrné



Porovnání klasických konstrukcí – typy

Asociativní pole

Perl,Python,PHP mají asociativní pole = neuspořádaná kolekce dvojic (klíč, hodnota), nemají indexy

- **Př. Perl: 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

Př free unionu v Pascalu, za case nemusí být rozlišovací položka (RV v předešlém slidu), je dovoleno uvést pouze typ

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

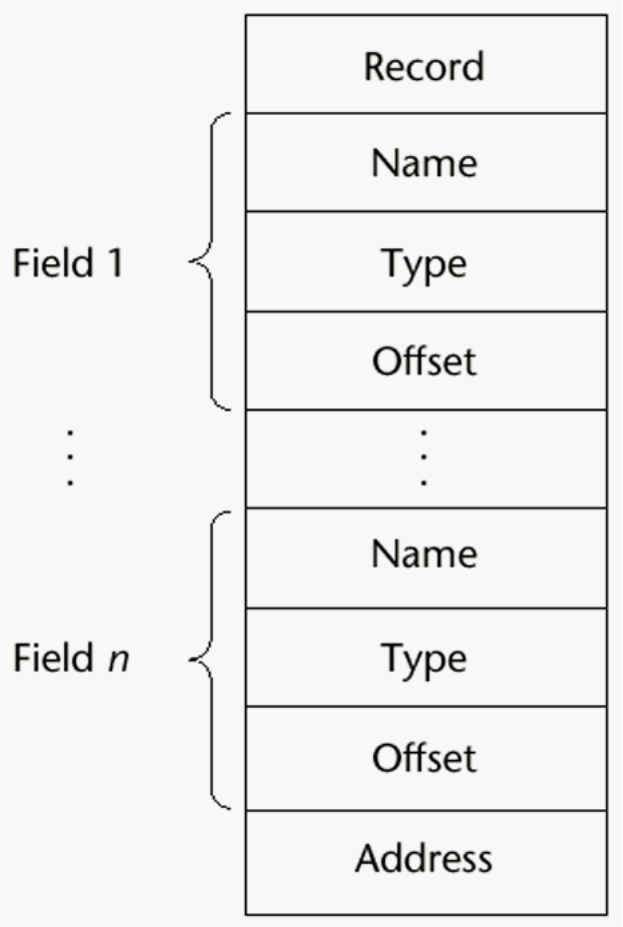
Důsledek – Pascal protože dovoluje free uniony, taknemůže mít garbage collector

- 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, Python 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**

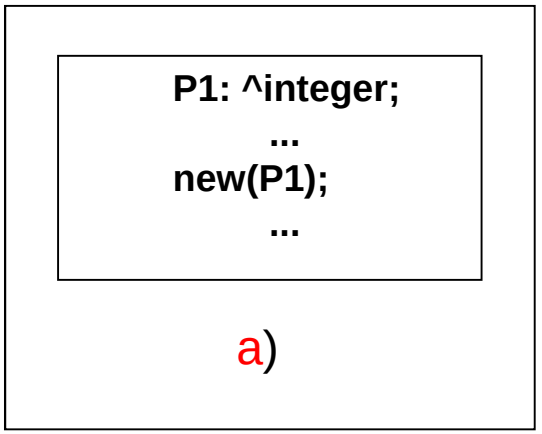
- dereference tvaru jménopointeru^.položka

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



- 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 je ukazatel na float
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  teprve tohle je správně*/
    return 0;
}
```

Pointer je neinicializovaný, hodnota x se přiřadí do neznáma

```
main()
{ int x,*p;
    x = 10; p=x;          /* p =&x  tohle je správně*/
    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 prohlásil: “The introduction of pointers into high level languages has been a step backward” (flexibility $\leftrightarrow$ 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 $\Rightarrow$ dangling reference nemůže vzniknout.

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

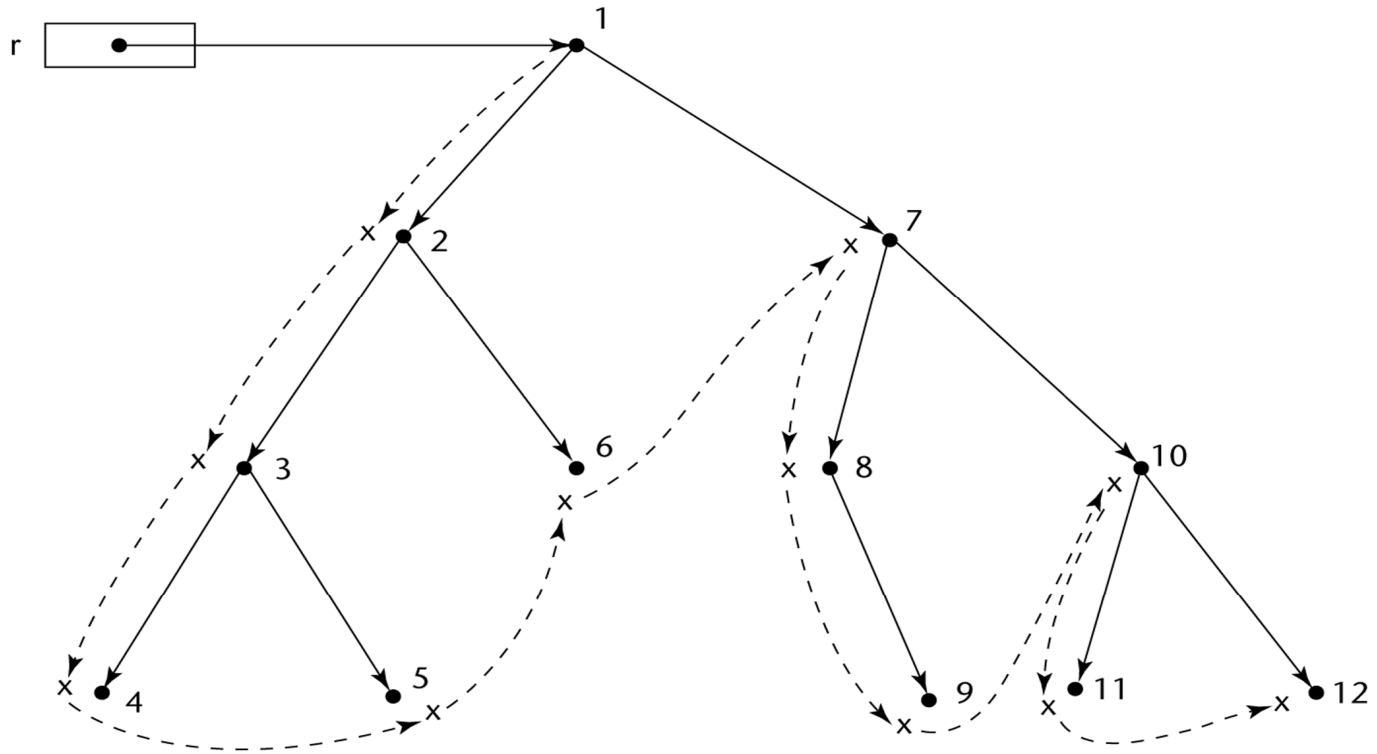
C#:

- 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⇒ji vrátí do volných)

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



Průběh označování 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); je dovoleno v C
6. Je dovoleno přetěžování operátorů?
7. Je alternativnost zápisu (Java, C) C=C+1; C+=1; C++; ++C mají tentýž efekt, což neprospívá čitelnosti

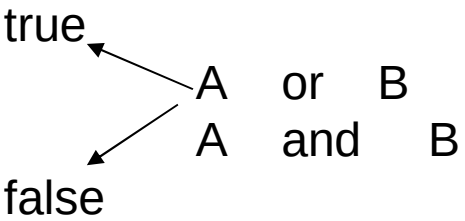
Porovnání klasických konstrukcí – výrazy a příkazy

```
#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 operandů jazyky neurčují*/
 printf("%d\n",i);  printf("%d\n",x);

 y=2; i=3;
 z = f(&i) + (i=y+i);   /* ?pořadi vyhodnoceni a tedy výsledek se mohou lišit*/
 printf("%d\n",z); printf("%d\n",i);
 getchar();
 return 0;
}            /*BC vyhodnocuje nejdříve fci, ale MicrosoftC vyhodnocuje zprava doleva*/

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 **i=1, j=2, k=3**
if (i == 2 & ++j == 3) k=4; ?jaký výsledek **i=1, j=3, k=2**

Porovnání klasických konstrukcí – výrazy a příkazy

Využití podmíněných výrazů

v přiřazovacích příkazech C jazyků, Javy `k = ( j == 0) ? j+1 : j-1 ;`

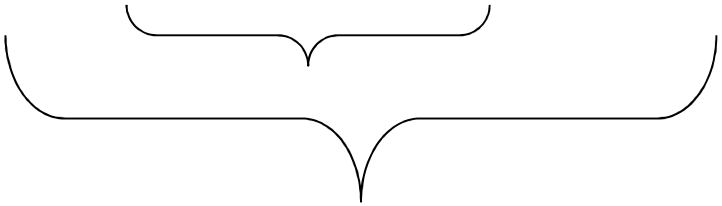
(pozn. v C jazycích, Javě produkuje přiřazení hodnotu, může být ve výrazu.)

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_list n : statement n;
end;
```

Alternativa else / others. „Návěští“ ordinálního typu

C jazyky, Java

```
switch (expression) {
    case constant_expr1 : statements;
            ....
    case constant_expr n : statements n;
    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

Cykly

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
```

nebezpečí záměny tečky a čárky

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

Java vyžaduje explicitní deklaraci parametru cyklu

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

Rozporný příkaz skoku

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

Výhody - snadno implementovatelný
- efektivně implementovatelný

Formy návěští:

číslo: Pascal

číslo Fortran

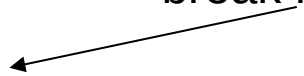
identifikátor: Cjazyky

<<identifikátor>> ADA

proměnná PL/1

Java nemá skok GOTO, částečně jej nahrazuje break, ten ukončí blok s nav

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



Porovnání klasických konstrukcí – výrazy a příkazy

Zásada: Používej skoků co nejméně

Př. Katastrofický důsledek snahy po obecnosti (zavest promennou navesti v PL1)

```
B1: BEGIN; DCL L LABEL;
    ...
B2: BEGIN; DCL X,Y FLOAT;
            ...
L1: Y=Y+X;
            ...
L=L1;
END;
    ...
GOTO L; --zde existuje promenna L, ale hodnota L1 jiz neexistuje, jsme mimo B1
END;
```

Porovnání klasických konstrukcí – podprogramy

Procedury a Funkce jsou nejstarší formou abstrakce – abstrakce procesů

(Java a C# nemají klasické 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 podprogramy 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

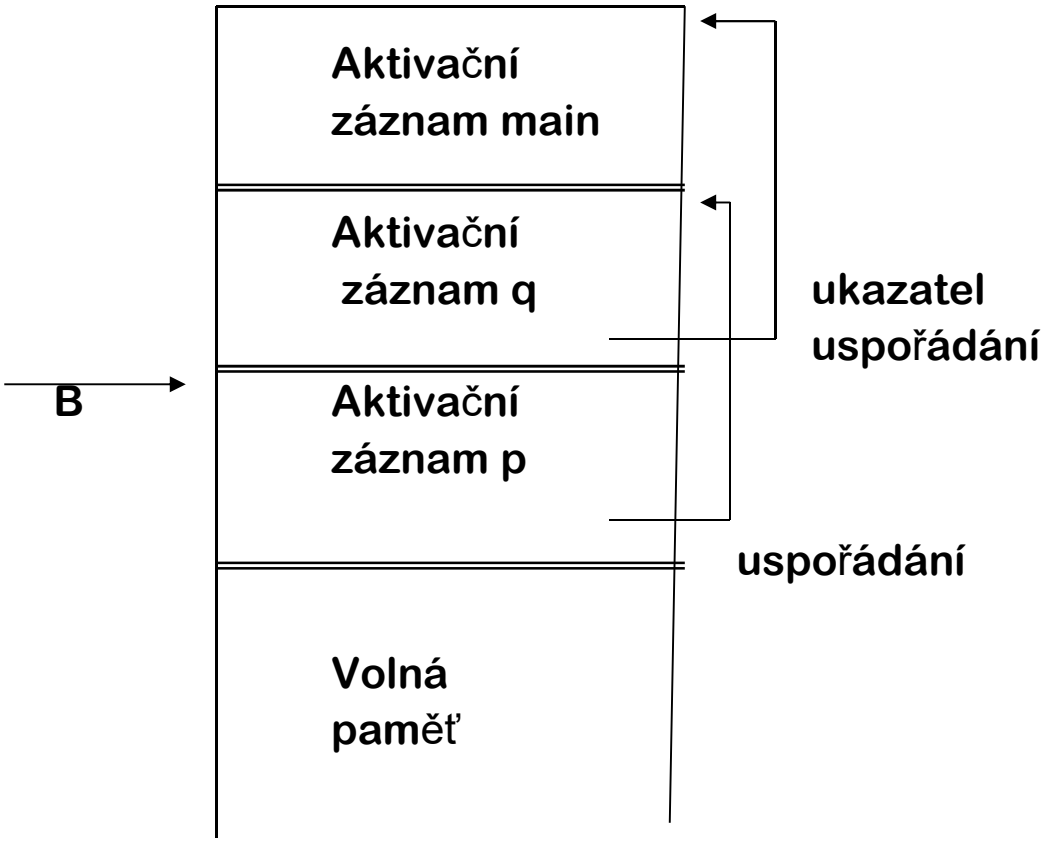
-Aktuální AZ je přístupný prostřednictvím ukazatele (nazvěme ho B) na jeho bázi

-Po skončení rozsahové jednotky je její AZ odstraněn ze zásobníku dle ukazatele uspořádání AZ

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):

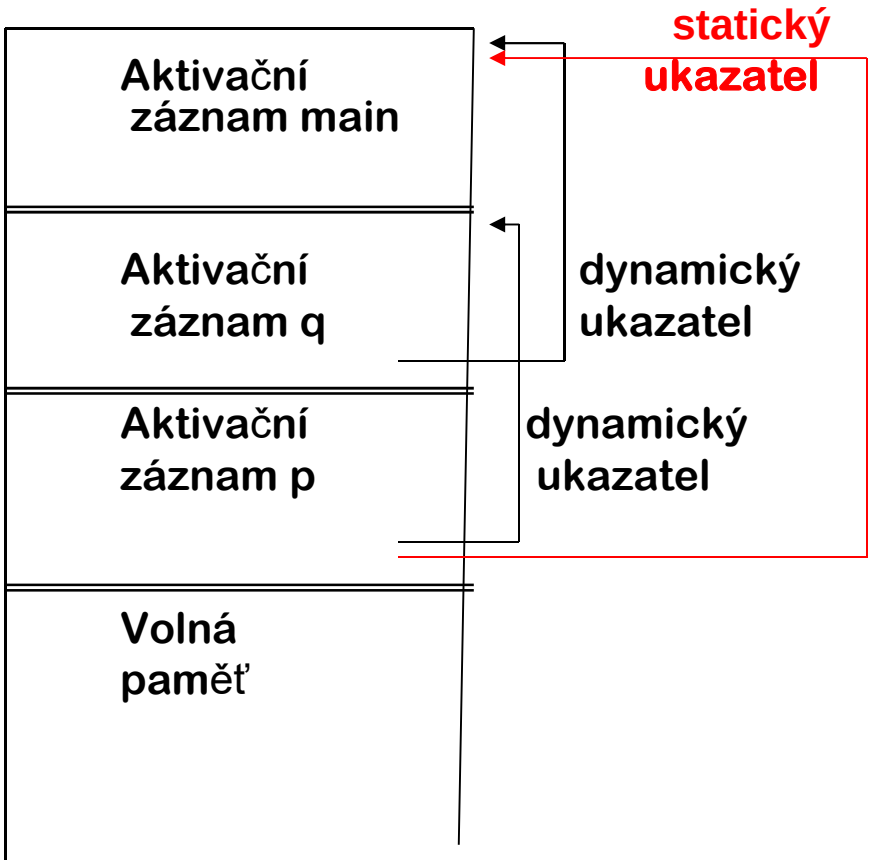
1. **(dynamický) Na rušení AZ opuštěných rozsahových jednotek (viz výše)**
2. **(statický) pro přístup k nelokálním proměnným**

Porovnání klasických konstrukcí – podprogramy

Př. Uvažujme možnost vnořování podpr

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

  q(2);
  return 0;
}
```

B



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

V C, C++ má statický řetěz délku 1, není proto nutný. 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 (Java). 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.

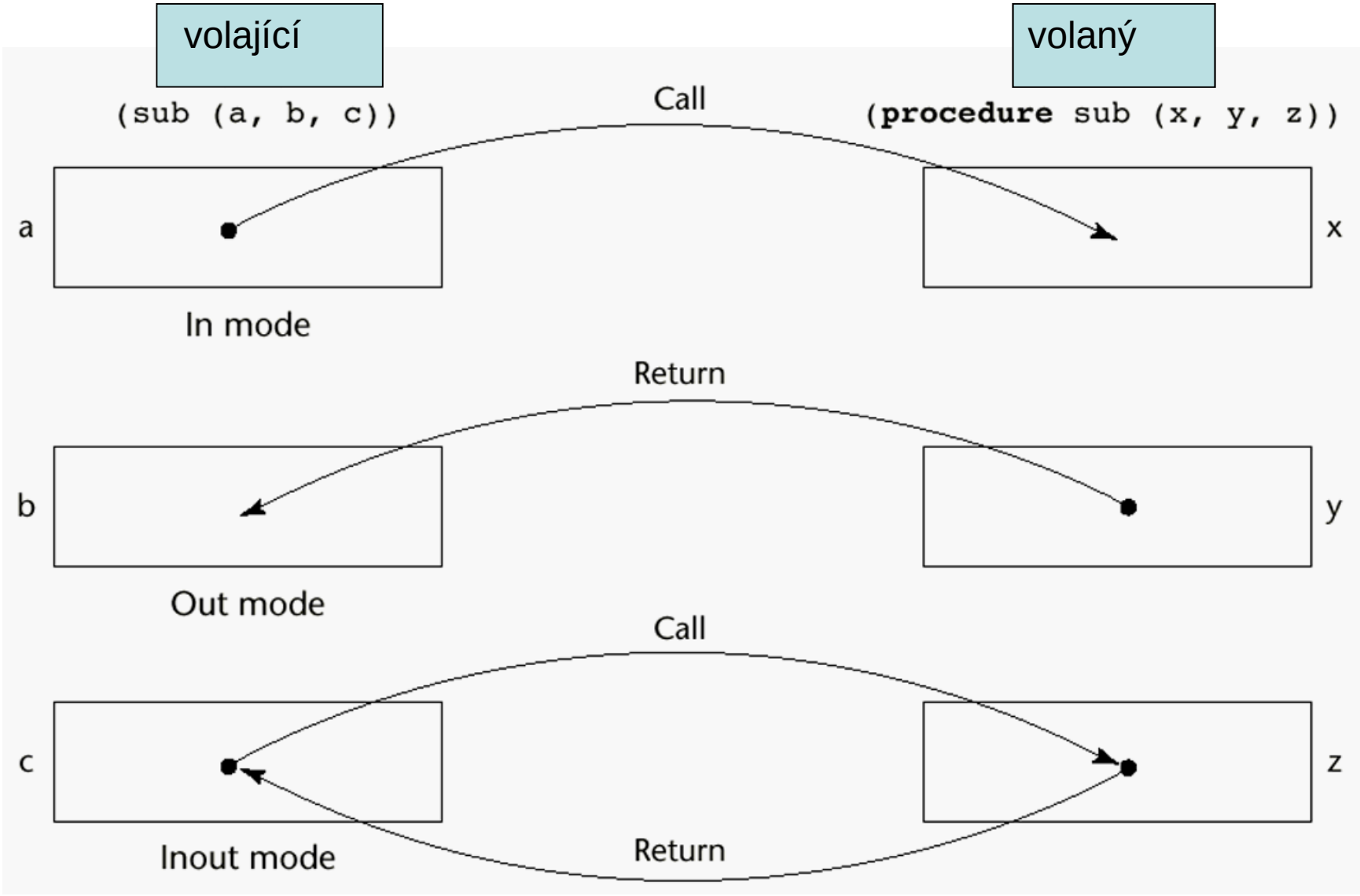
  podprogram Sub(a , b) **;**

  . . .

  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 vůbec

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

Jaké je výpočtové prostředí sub2 po jeho vyvolání v sub4 ? Existuje více možností

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

Př.ADA

```
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  "*";
```

Př.C++

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

Porovnání klasických konstrukcí – podprogramy

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

C++ obecný tvar:

template<class parameters> definice funkce, která může obsahovat class parametry

Example:

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

Instalace je možná pro libovolný typ, který má gefinováno > , např. integer

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

1) efect je následující

C++ template funkce je instalována implicitně když bu\´d je použita v příkazu vyvolání nebo je použita s & operatorem

```
 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;
```

OOP

- **Objektový polymorfismus** v jazycích Java, Object Pascal a C++
- Objektové konstrukce v programovacích jazycích
- Jak jsou dědičnost a polymorfismus implementovány v překladači
- Příklady využití polymorfismu
- Násobná dědičnost a rozhraní

Př. 1JavaZvirata v OOPOstatni

```
class Animal {
    String  type ;
    Animal() {                  type = "animal  ";}
    String sounds() {           return "not known";}
    void prnt() {               System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                     type = "dog   "; }
    String sounds() {           return "haf";   }
}
class Cat extends Animal {
    Cat() {             type = "cat   ";   }
    String sounds() {       return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();
} } //konec. Co se tiskne? Dle očekávání kočka mnouka, pes steka
```

Objektové prostředky Pascalu

Př.2PascalZvířata Obdobný příklad v Pascalu

```
program Zvirata;
{$APPTYPE CONSOLE}
uses  SysUtils;
type
    Uk_Zvire = ^Zvire;
    Zvire = object
            Druh    : String;
            procedure Inicializuj;
            function Zvuky: String;
            procedure Tisk;
           end;
    Uk_Pes = ^Pes;
    Pes =   object(Zvire)
            procedure Inicializuj;
            function Zvuky: String;
           end;
    Uk_Kocka = ^Kocka;
    Kocka = object(Zvire)
            procedure Inicializuj;
            function Zvuky: String;
           end;
```

```
{-----------Implementace metod-------------------------}
    procedure Zvire.Inicializuj;
    begin  Druh := 'Zvire  '
    end;
    function Zvire.Zvuky: String;
    begin  Zvuky := 'nezname'
    end;
    procedure Zvire.Tisk;
    begin  writeln(Druh, Zvuky);
    end;

    procedure Pes.Inicializuj;
    begin  Druh := 'Pes   '
    end;
    function Pes.Zvuky: String;
    begin  Zvuky := 'steka'
    end;

    procedure Kocka.Inicializuj;
    begin  Druh := 'Kocka   '
    end;
    function Kocka.Zvuky: String;
    begin  Zvuky := 'mnouka'
    end;
```

```
{----------------Deklarace objektuu--------------------}
    var
      U1, U2, U3: Uk_Zvire;
      Nezname: Zvire;
      Micka: Kocka;
      Filipes: Pes;

{---------------Hlavni program-------------------------}
begin
    Filipes.Inicializuj;   Filipes.Tisk;    {  !!!???   }
    new(U1); U1^.Inicializuj; U1^.Tisk;
    Micka.Inicializuj; writeln(Micka.Druh, Micka.Zvuky);
    readln;
end.
```

Konec Př.1ObjectPascalZvířata. Co se tiskne? Pes zde vydává neznámé zvuky

Lze zařídit stejné chování i Java programu?

Umí se stejně chovat i Java? Co se tiskne?

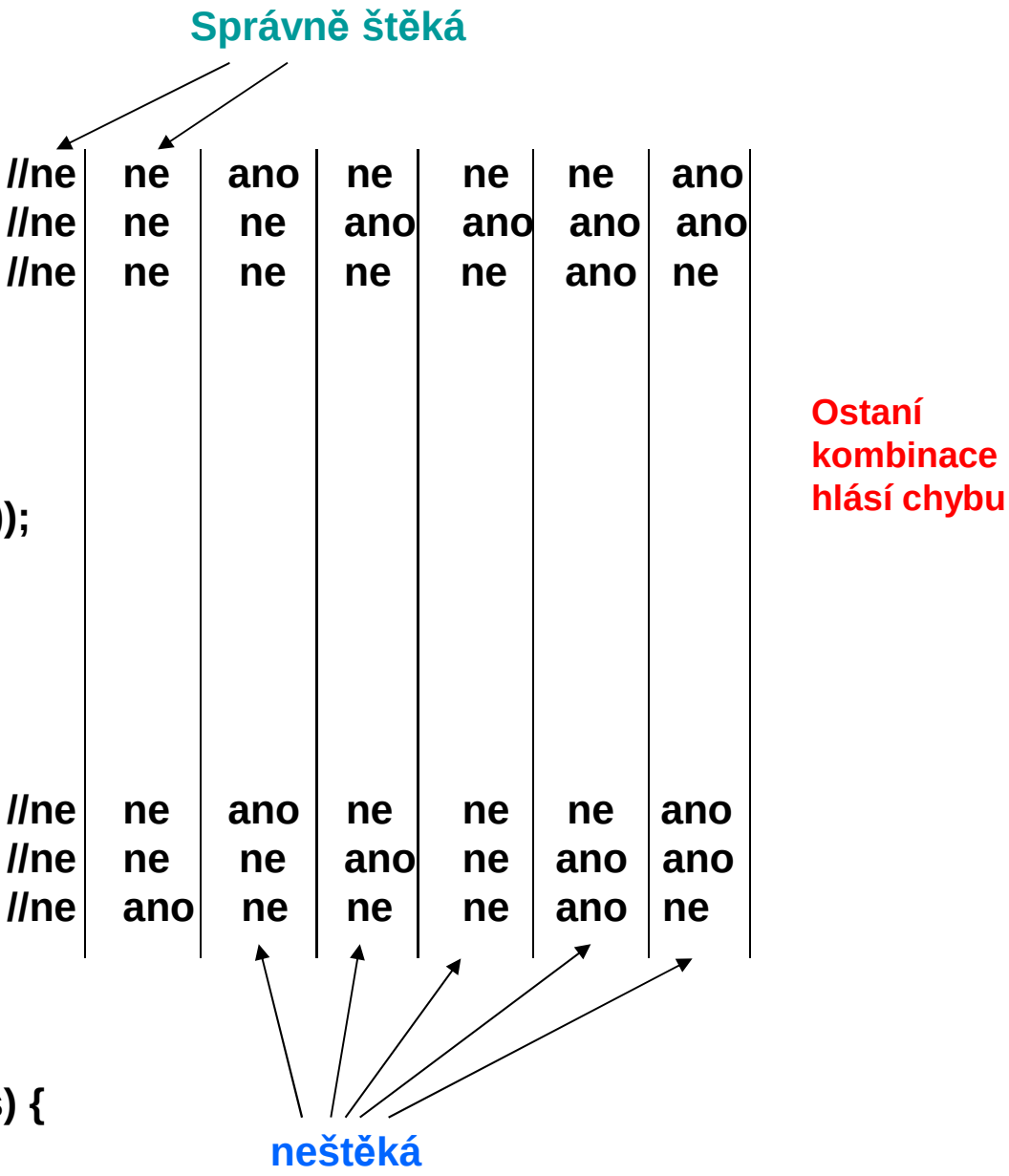
```
class Animal {      //program AnimS.java
    String  type ;
    Animal() {                              type = "animal  ";}
    static String sounds() {                return "not known";}
    void prnt() {      System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                                 type = "dog   "; }
    static String sounds() {                return "haf";   }
}
class Cat extends Animal {
    Cat() {                                 type = "cat   ";   }
    static String sounds() {                return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();            //vsechno se ozyva not known
} }
```

```
class Animal { //                    program AnimF.java ?co to ted udela?
    String  type ;
    Animal() {                       type = "animal  ";}
    final String sounds() {      return "not known";}
    void prnt() {        System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {              type = "dog   "; }
    final String sounds() {                  return "haf";   }
}
class Cat extends Animal {
    Cat() {              type = "cat   ";   }
    final String sounds() {       return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();
} }          //zase vsichni jsou not known
```

```
class Animal { //      soubor AnimF.java  ?co to ted udela?
    String  type ;
    Animal() {                          type = "animal  ";}
   private final String sounds() {          return "not known";}
    void prnt() {        System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {              type = "dog   "; }
 private final String sounds() {             return "haf";   }
}
class Cat extends Animal {
    Cat() {              type = "cat   ";   }
 private final String sounds() {       return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();
} }          //opet vsichni davaji zvuky not known
```

Na dalším obrázku máte kombinace privat, static, final a výsledné chování

```
class Animal {
   public String  type ;
   public Animal() {
      type = "animal  "; }
//static
//private
//final
   String sounds() {
     return "not known";
   }
   public void prnt() {
     System.out.println(type + sounds());
} }
class Dog extends Animal {
   public Dog() {
      type = "dog   ";
   }
//static
//private
//final
     String sounds() {return "haf"; }
}
public class AnimX {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   filipes.prnt();
   notknown.prnt();
```



Př.Zvirata1x Správné řešení aby pes stekal v Borland Pascal

```
Zvire =  object
           Druh    : String;
           constructor Inicializuj;                {!!! Inicializuj musí být konstruktor}
           function Zvuky: String; virtual;        {!!! Zvuky musí být virtual}
           procedure Tisk;
         end;
. . .
Pes =   object(Zvire)
           constructor Inicializuj;                {!!!}
           function Zvuky: String; virtual;        {!!!}
        end;
. . .
Filipes.Inicializuj;
  Filipes.Tisk;                          { !!!??? }

  new(U1);
  U1^.Inicializuj;
  U1^.Tisk; . . .
```

řešení Object Pascalem (Pascal z Delphi)

```
Př.Zvirata1
Zvire =  class
           Druh    : String;
           constructor Inicializuj;                  {!!!}
           function Zvuky: String; virtual;          {!!!}
           procedure Tisk;
         end;
Pes =   class(Zvire)
           constructor Inicializuj;                  {!!!}
           function Zvuky: String; override;         {!!!u potomka je overide místo virtual}
        end;
. . .
Filipes := Pes.Inicializuj;
  Filipes.Tisk;                                      { !!!??? }

  new(U1);
  U1^:= Zvire.Inicializuj;
  U1^.Tisk;

  Micka := Kocka.Inicializuj;
  writeln(Micka.Druh, Micka.Zvuky);  . . .
```

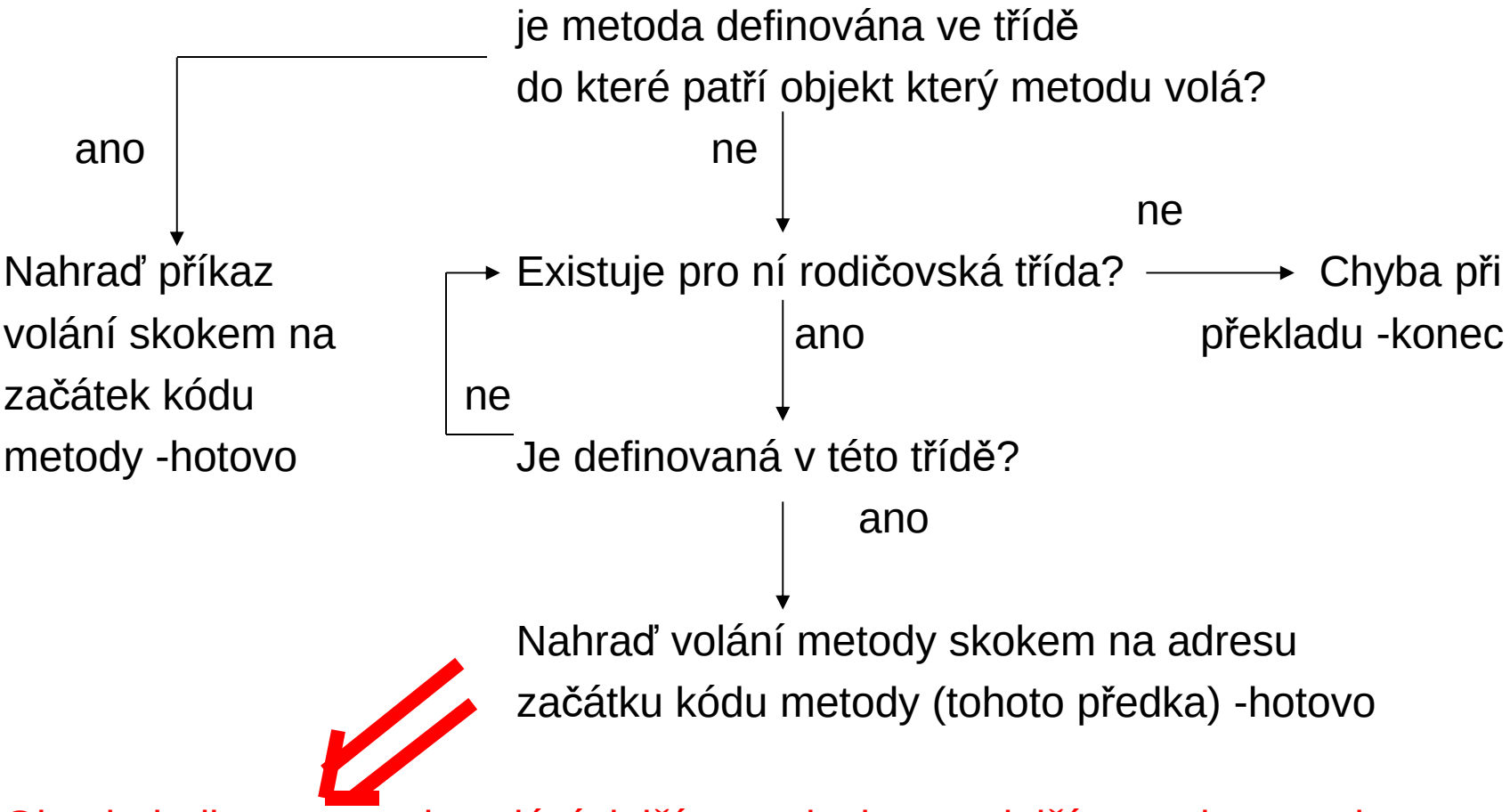
Zápis v C++

Př.3CZvirata (pozn. C++ nema funkcni hodnotu string)

```
class Zvire {public: char Druh[20];  void Inicializuj();   int Zvuky();   void Tisk(); };
class Pes:public Zvire {public:   void Inicializuj();   int Zvuky(); };
class Kocka:public Zvire {public:   void Inicializuj();   int Zvuky(); };
void Zvire::Inicializuj() {   strcpy(Druh, "zvire");}
int Zvire::Zvuky() {   return 10;}
void Pes::Inicializuj() {   strcpy(Druh, "pes");}
int Pes::Zvuky() {   return 11;}
void Kocka::Inicializuj() {   strcpy(Druh, "kocka");}
int Kocka::Zvuky() {   return 12;}
void Zvire::Tisk() {cout << Druh;
    if (Zvuky()==10) cout << " nezname";
    if (Zvuky()==11) cout << " steka";
    if (Zvuky()==12) cout << " mnouka";
    cout << "\n";
}
main()
{
  Zvire Nezname;
  Pes Filipes;
  Kocka Micka;
  Filipes.Inicializuj(); Filipes.Tisk();     //tiskne pes neznam
  Micka.Inicializuj(); Micka.Tisk();       //tiskne kocka neznam
  getchar();
  return 0;
}
```

```
Př.4CZvirata
class Zvire {
public:
   char Druh[20];
   void Inicializuj();
   virtual int Zvuky() { return 10; };
   void Tisk() ;
};
class Pes:public Zvire {
public:
   void Inicializuj();
   int Zvuky() { return 11; };
};
//.  .  . Tady je to stejne jako predesle
int main()
{  Zvire Nezname;
   Pes Filipes;
   Kocka Micka;
   Filipes.Inicializuj(); Filipes.Tisk();     //tiskne pes steka
   Micka.Inicializuj(); Micka.Tisk();         //tiskne kocka mnouka
   getchar();
   return 0;
}
```

Dědičnost a statická (brzká) vazba =proč pes neštěká



Obsahuje-li tato metoda volání další metody, je tato další metoda metodou předka, i když potomek má metodu, která ji překrývá – viz Zvirata

Dědičnost a dynamická (pozdní) vazba= pak pes štěká

- Realizovaná pomocí virtuálních metod
- Při překladu se vytváří pro každou třídu tzv. datový segment, obsahující:
  - údaj o velikosti instance a datových složkách
  - údaj o předkovi třídy
  - ukazatele na tabulku metod s pozdní vazbou (**V**irtual **M**ethod **T**able)
- Před prvým voláním virtuální metody musí být provedena (příp. implicitně) speciální inicializační metoda – constructor (v Javě přímo stvoří objekt)
- Constructor vytvoří spojení (při běhu programu) mezi instancí volající konstruktor a VMT. Součástí instance je místo pro ukazatel na VMT třídy, ke které instance patří. Constructor také v případech kdy je objekt na haldě ho přímo vytvoří (přidělí mu místo -tzv. Class Instance Record). Objekty tvořené klasickou deklarací (bez new …) jsou umístěny v RunTime zásobníku, alokaci jejich CIR tam zajistí překladač.
- Volání virtuální metody je realizováno nepřímým skokem přes VMT
- Pokud není znám typ instance (objektu) při překladu (viz případ kdy ukazatel na objekt typu předka lze použít k odkazu na objekt typu potomka), umožní VMT polymorfní chování tzv **objektový polymorfismus**

Dědičnost a dynamická (pozdní) vazba

Př CPP zápis.

class A { public: void f( ); virtual void g( ); double x, y; } ;

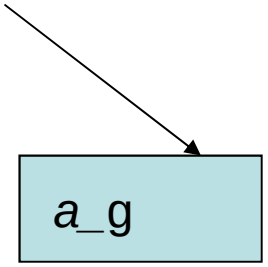
class B: public A { public: void f( ); virtual void h( ); **void g( )**; double z; };

Alokované místo (CIR) objektu *a* třídy A

Místo pro x

místo pro y

VMT ukazatel



VMT pro a

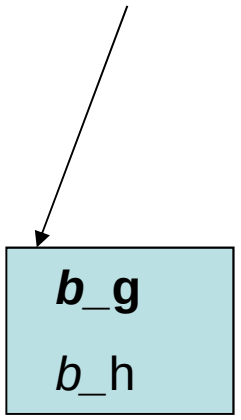
Alokované místo (CIR) objektu *b* třídy B

Místo pro x

místo pro y

místo pro z

VMT ukazatel



VMT pro b

Pozn.: a_f , b_f jsou statické, skok na jejich začátek se zařídí při překladu

Dědičnost a dynamická (pozdní) vazba

Př CPP zápis. Zde g z třídy A není ve třídě B překrytá, takže objekt b dědí a_g

class A { public: void f( ); virtual void g( ); double x, y; } ;

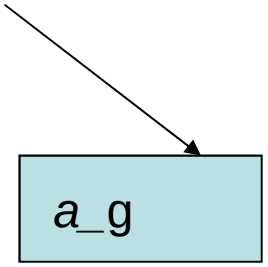
class B: public A { public: void f( ); virtual void h( ); double z; };

Alokované místo objektu *a* třídy A

Místo pro x

místo pro y

VMT ukazatel



VMT pro a

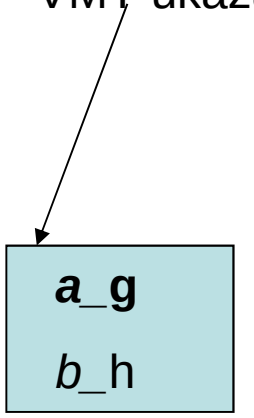
Alokované místo objektu *b* třídy B

Místo pro x

místo pro y

místo pro z

VMT ukazatel



VMT pro b

OOP konstrukce C++

Třídy odvozeny ze struct C

class jméno třídy {

private: privátní položky a metody *používány jen metodami této třídy*

protected: chráněné " **„** *používány metodami této třídy a potomky*

public: veřejné " **„** *používány volně, tvoří interface třídy*

}

- **data members = datové členy (elementy)**
- **member functions = členské funkce**
- **Třídy lze tvořit z class (1.možnost definice tříd) jsou člemy implicitně private**
- **“ “ “ struct (2.možnost definice tříd) jsou členy implicitně public**
- **“ z union (3.možnost definice tříd) jsou implicitně public a nedá se to změnit**
- **instance třídy mohou byt -statické (netvoří se při rekurzi ani pomocí new)**
  -dynamické v haldě (příkazem new)
  -dynamické v zásobníku (možnost vzniku při rekurzi)
- **datové elementy mohou být také dynamické (vznik new / odstranění delete)**
- **funkční elementy lze definovat dvojím způsobem:**
  - **hlavička fce uvedena v definici třídy a vlastní deklarace fce je mimo třídu**
  - **celé v definici třídy (včetně těla fce, nazývají se pak inlined)**

OOP konstrukce C++

- **Konstruktory pojmenovány shodně s třídou**
  - **inicializují datové elementy**
  - **Jsou implicitně volatelné**
  - **může jich být více pro třídu**

- **Destruktory pojmenovány ~jméno třídy**
  - **implicitně volatelné**
  - **C++ nemá čistič paměti**

- **Řízení dědičnosti**
  - **private/public děděné elementy v potomkovi**
  - **možnost násobné dědičnosti**
  - **konstrukcí friend**

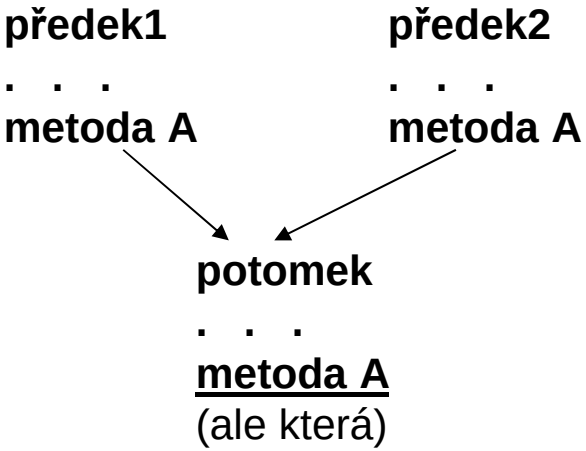
```
class potomek:    [virtual][private/public] předek1
                  [[,virtual][private/public] předek2
                                  . . .
                  ]
        { data a metody
        friend přitel
        };
```

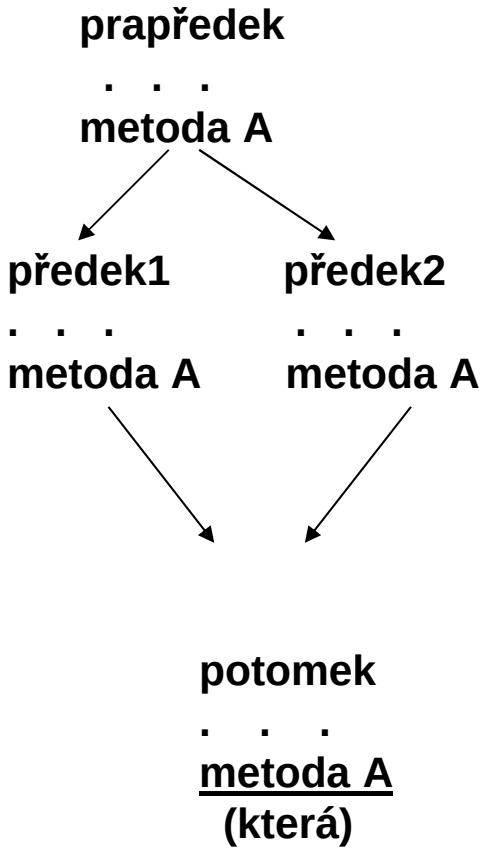
OOP konstrukce C++

přístup v rodiči	modifikace přístupu v potomkovi: public	private
public	public	private
protected	protected	private
private	nepřístupný	nepřístupný

```
Př.     CLASS Předek1 {  PUBLIC: int p11;
                         PROTECTED: int p12;
                         PRIVATE: int p13;
        };
        CLASS Předek2 {  PUBLIC: int p21;
                         PROTECTED: int p22;
                         PRIVATE: int p23;
        };
        CLASS Potomek: PUBLIC Předek1, PRIVATE Předek2; {
                PUBLIC: int pot1;
                PROTECTED: int pot2;
                PRIVATE: int pot3;
        };
// Jaké datové elementy má Potomek? Public pot1, p11. Protected pot2, p12. Private
     pot3, p21, p22
```

OOP konstrukce C++ (problém násobné dědičnosti)





Řešení:

-napsat plným jménem (jméno předka :: jméno)

- virtual předek1, virtual předek2 } **slovem virtual dáme informace, že se má uplatnit jen jeden**

Objektové vlastnosti C# (pro informaci)

- Používá **class** i **struct**
- Pro dědění při definici tříd používá CPP syntax

  public class NovaTrida : RodicovskaTrida { . . . }
- V podtřídě lze nahradit metodu zděděnou od rodiče zápisem

  new definiceMetody;

  ta pak zakryje děděnou metodu stejného jména.

  Metodu z rodiče lze ale přesto volat pomocí zápisu např.

  base.vykresli( );
- Dynamická vazba je u metody rodičovské třídy povinně označena

  virtual

  a u metod odvozených tříd povinně označena

  override

  (převzato z Objekt Pascalu)

Př.

Objektové vlastnosti C# (pro informaci)

```
public class Obrazec {
    public virtual void Vykresli ( ) { . . . }
    . . .
}
public class Kruh : Obrazec {
    public override  void Vykresli ( ) { . . . }
    . . .
}
public class Ctverec : Obrazec {
    public override void Vykresli ( ) { . . . }
    . . .
}
```

- **Má abstraktní metody, např, abstract public void Vykresli( );**
  ty pak musí být implementovány ve všech potomcích a třídy s abstract metodou musí být označeny abstract
- **Kořenem všech tříd je Object jako u Javy**
- **Nestatické vnořené třídy nejsou zavedeny**

OOP konstrukce v jazyce ADA (pro informaci)

```
--Objektový typ je založen na konstrukci záznamu, slovo tagged udává, že to je třída
type  Datum  is  tagged
     record
          Den        : Den_Subtyp ;
          Mesic      : Mesic_Subtyp ;
          Rok        : Rok_Subtyp ;
     end  record ;

--Konstrukce potomka
type  Complet_Datum  is  new  Datum  with
     record
          Den_v_tydnu : Den_v_tydnu_Typ ;
     end  record ;

--Popis metod následuje za konstrukcí record
procedure  Display(D: Datum) is .  .  .  ;
procedure Display( CD : in  Complete_Datum )  is
  Display(Datum(CD)); . . . --Lze konvertovat objekty odvozeného typu na rodičovský typ
  Text_IO.Put(Day_Of_Week_Type'Image(CD.Day_Of_Week));
end  Display;
```

OOP konstrukce v jazyce ADA (pro informaci)

- **Zapouzdření realizuje pomocí *private části recordu***
- **Polymorfismus realizuje pomocí *Class-wide typu, ten* umožňuje odkazovat na celou rodinu typů**

```
D : Datum'Class := Datum'(6, Jul, 1415); --nutná inicializace upřesní typ ≡ místo v paměti
--D := Complete_Datum'(6, Jul, 1415, Mon) ; --pak neni již dovoleno, měnilo by typ
type Ptr is access Datum'Class ; -- pak může ukazovat na objekty obou typů

procedure Dynamic_Dispatching_Demo is
    A : array(1 .. 2) of Ptr ;
begin
    A(1) := new Datum'(6, Jul, 1415);
    A(2) := new Complete_Datum'(14, Jul, 1789, Wen) ;
    for I in A'Range loop
        Display( A(I).all ) ; --rozhodne se až při výpočtu
    end loop ;
end Dynamic_Dispatching_Demo ;

procedure Show(X : in Datum'Class) is –použitelné i jako typ parametru
begin
    Display(X) ; --rozhodne až při výpočtu
end Show; -- Použití v parametrech podprogramu:
```

OOP konstrukce v jazyce ADA Př. 8ZVÍŘATA (pro informaci)

```
with TEXT_IO; use TEXT_IO;

package ZVIRATA is
   type STRING10 is new STRING(1..10);
   type ZVIRE is tagged
    record DRUH: STRING10;
    end record;

    type UK_ZVIRE is access ZVIRE;
    procedure INICIALIZUJ(O: out ZVIRE);
    function ZVUKY(O:ZVIRE) return STRING10;
      procedure TISK(O:ZVIRE); --musi byt u zvirete

   type PES is new ZVIRE with record MAJITEL: STRING10; end record;
    type UK_PES is access PES;
    procedure INICIALIZUJ(O: out PES);
    function ZVUKY(O:PES) return STRING10;

end ZVIRATA;
```

```
package body ZVIRATA is

    procedure INICIALIZUJ(O: out ZVIRE) is
        begin O.DRUH := "zvire     ";
        end;

    procedure INICIALIZUJ(O: out PES) is
        begin O.DRUH := "pes       ";
        end;

    function ZVUKY(O:ZVIRE) return STRING10 is
        begin return "nezname   ";
        end;

    function ZVUKY(O:PES) return STRING10 is
        begin return "steka     ";
        end;

    procedure TISK(O:ZVIRE) is
        begin PUT_LINE(STRING(O.DRUH));
            PUT_LINE(STRING(ZVUKY(O)));
        end;

end ZVIRATA;
```

OOP konstrukce v jazyce ADA Př. 8ZVÍŘATA (pro informaci)

```
with ZVIRATA; use ZVIRATA;
procedure POK_ZV is

     UZ: UK_ZVIRE;
     UP: UK_PES;
     NEZNAME: ZVIRE;
     FILIPES: PES;

begin
   INICIALIZUJ(FILIPES);
   INICIALIZUJ(NEZNAME);
   TISK(NEZNAME);
   TISK(FILIPES);
   UP:=new PES;
   INICIALIZUJ(UP.all);
   TISK(UP.all);
end POK_ZV;
```

Co tiskne?

OOP konstrukce v jazyce ADA Př. 9ZVÍŘATA1 (pro informaci)

```
with TEXT_IO; use TEXT_IO;

package ZVIRATA1 is
   type STRING10 is new STRING(1..10);
   type ZVIRE is tagged
    record DRUH: STRING10;
    end record;

    type UK_ZVIRE is access ZVIRE;
    procedure INICIALIZUJ(O: out ZVIRE);
    function ZVUKY(O:ZVIRE) return STRING10;

    type PES is new ZVIRE with record MAJITEL: STRING10; end record;
    type UK_PES is access PES;
    procedure INICIALIZUJ(O: out PES);
    function ZVUKY(O:PES) return STRING10;

    procedure TISK(O:ZVIRE'CLASS);

end ZVIRATA1;
```

```
package body ZVIRATA1 is –Hl.program je opet POK_ZV.adb jako u 8ADAZVIRATA

    procedure INICIALIZUJ(O: out ZVIRE) is
        begin O.DRUH := "zvire     ";
        end;

    procedure INICIALIZUJ(O: out PES) is
        begin O.DRUH := "pes       ";
        end;

    function ZVUKY(O:ZVIRE) return STRING10 is
        begin return "nezname   ";
        end;

    function ZVUKY(O:PES) return STRING10 is
        begin return "steka     ";
        end;

    procedure TISK(O:ZVIRE'CLASS) is
        begin PUT_LINE(STRING(O.DRUH));
            PUT_LINE(STRING(ZVUKY(O)));
        end;

end ZVIRATA1;
```

OOP konstrukce v jazyce ADA (pro informaci)

Abstraktní typy a abstraktní podprogramy

```
package P is
        type Abstract_Datum is abstract tagged null record ;
        procedure Display(AD : Abstract_Datum) is abstract ;
end P ;
```

Abstraktní typ je použitelný jen k odvozování typů (ne k deklaraci)
Abstraktní procedura nemá tělo

```
 with P ; use P ;
package Q is
        . . .
type Ptr is access Abstract_Datum ;
type Datum is new Abstract_Datum with record . . . --D,M,R
type Complete_Datum is new Datum with record . . . --Den_v_tydnu
procedure Display(D : Datum) ; --v těle modulu bude její tělo
procedure Display(CD : Complete_Datum) ;-- “
end Q ;
```

Můžeme psát modul P se všemi jeho abstraktními procedurami před tím, než programujeme modul Q s detailním tvarem odvozených typů