PROGRAMOVÉ STRUKTURY: POROVNÁNÍ

Proměnné, typový systém, datové typy, výrazy a příkazy, OOP

KIV/PGS

© Karel Ježek & Martin Zíma

Porovnání klasických konstrukcí

- Jména (identifikátory)
  - maximální délka
    - 6 – Fortran, Fortran90
    - 30 – Cobol
    - 31 – ANSI C
    - neomezeně
      - C++ – omezeno implementací
      - ADA, Java
  - case-sensitivity
    - ano – C, C++, Java
    - nevýhodou je zhoršení čitelnosti
      - jména vypadají stejně, ale mají různý význam

© Karel Ježek & Martin Zíma

Porovnání klasických konstrukcí (2)

- Speciální slova
  - klíčová slova
    - v určitém kontextu mají speciální význam
  - předdefinovaná slova
    - identifikátory speciálního významu
    - lze je předefinovat
      - např. vše z balíku `java.lang` – `String`, `Object`, `System`, ...
  - rezervovaná slova
    - nemohou být použita jako uživatelem definovaná jména
      - např. `abstract`, `boolean`, `break`, `if`, `while`, ...

Porovnání klasických konstrukcí (3)

- Proměnné
  - abstrakce paměťových míst
  - formálně se jedná o 6tici atributů
    - (jméno, adresa, hodnota, typ, doba existence, rozsah platnosti)
  - způsob deklarace
    - explicitní
    - implicitní

Proměnné

- Jméno
  - nemají je všechny proměnné
- Adresa
  - místo v paměti
  - během doby výpočtu či místa v programu se může měnit
- 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

 © Karel Ježek & Martin Zíma

Aliasy

- Dvě proměnné sdílí ve stejné době stejné místo
  - pointery
  - referenční proměnné
  - variantní záznamy (Pascal)
  - uniony (C, C++)
  - equivalence (Fortran)
  - parametry podprogramů

 © Karel Ježek & Martin Zíma

Kategorie proměnných

- S typem vazby 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 přiřazením (např. Lisp)
    - výhoda – flexibilita (např. generické jednotky)
    - nevýhoda – vysoké náklady a 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 (2)

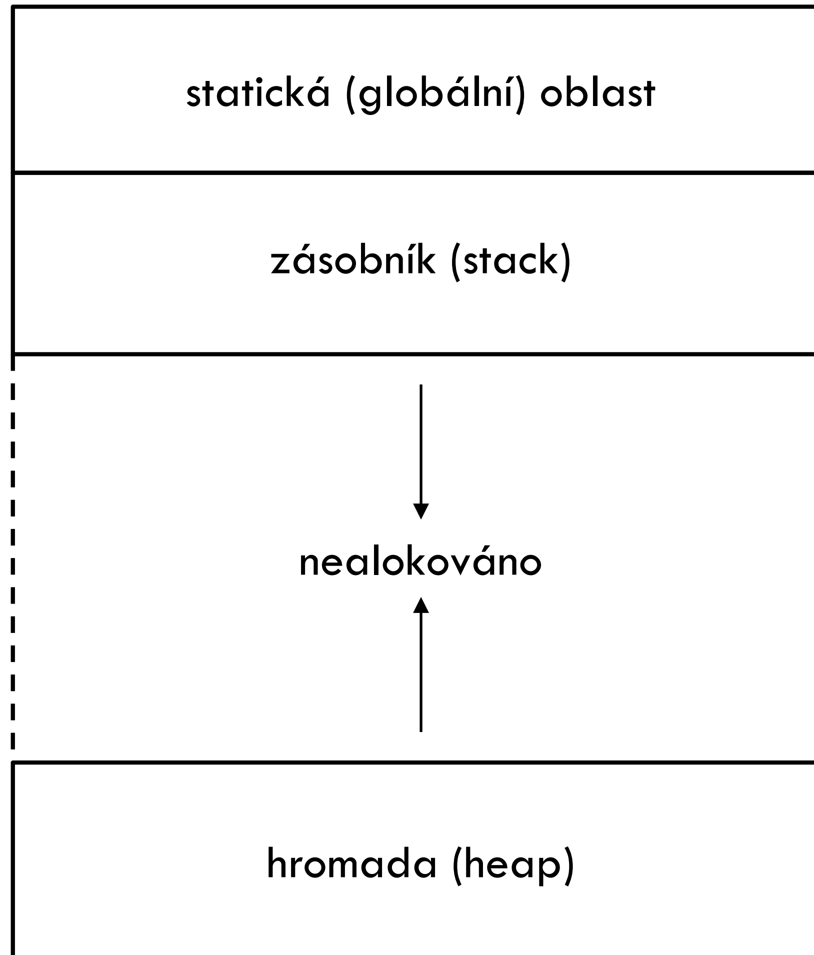
- Podle doby existence (lifetime)
  - statické
    - navázání na paměť před exekucí a nemění se po celou exekuci
    - Fortran 77, C static
    - výhody: efektivní – přímé adresování, podprogram sensitivní na historii
    - nevýhody: bez rekurze
  - dynamické
    - v zásobníku
      - přidělení paměti při exekuci zpracování deklarací
      - pro skalární proměnnou jsou kromě adresy přiděleny atributy staticky (lokální proměnné v C, Pascal)
      - výhody: rekurze
      - nevýhody
        - režie s alokací/dealokací
        - ztrácí historickou informaci
        - neefektivní přístup na proměnné (nepřímé adresy)

 © Karel Ježek & Martin Zíma

Kategorie proměnných (3)

- Dynamické (pokračování)
  - 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
      - zejména 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

Rozdělení paměťového prostoru



Typový systém

- Typová kontrola je aktivita zabezpečující, že operandy operátorů jsou kompatibilních typů
- Kompatibilní typy jsou takové
  - budou legální pro daný operátor
  - 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 datová kontrola
- Programovací jazyk má **silný typový systém**, pokud typová kontrola odhalí veškeré typové chyby
- Problémy s typovou konverzí
  - při zužování – možná neproveditelnost zaokrouhlení či ořezání
  - při rozšiřování – možná ztráta přesnosti

Typový systém (2)

- Konkrétní jazyky, které silný typový systém nemají
  - slabě typované jazyky
    - Fortran77 – parametry, příkaz EQUIVALENCE
    - Pascal – dovoluje variantní záznamy
    - C, C++ parametry, uniony nejsou kontrolovány
- Konkrétní jazyky, které silný typový systém mají
  - silně typované jazyky
    - Java – téměř je
    - ADA
    - Python
- Pravidla pro coerci výrazně oslabují silný typový systém

Typový systém (3)

- Kompatibilita typů se určuje na základě
  - jmenné kompatibility
    - dvě proměnné jsou kompatibilních typů
      - pokud jsou uvedeny v téže deklaraci
      - v deklaracích používají stejného jména typu
    - dobře implementovatelné, silně restriktivní
    - jazyky: ADA, Java
  - strukturální kompatibility
    - dvě proměnné jsou kompatibilní, mají-li jejich typy identickou strukturu
    - flexibilnější, hůře implementovatelné
    - jazyky: Pascal, C (kromě záznamů)

 © Karel Ježek & Martin Zíma

Rozsah platnosti a existence

- 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 programových jednotek
  - proměnné mohou být zakryty (slepé skvrny)

Rozsah platnosti a existence (2)

- Statický (lexikální) rozsah platnosti (pokračování)
  - ADA, C++ a Java dovolují přístup k zakrytým proměnným
  - např. `Trida.promenna`
  - 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

 © Karel Ježek & Martin Zíma

Rozsah platnosti a existence (3)


```
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
```

Rozsah platnosti a existence (4)

- 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

Rozsah platnosti a existence (5)


```
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 scope používá originální LISP, VBScript, Javascript, Perl (starší verze)

Konstanty

- Mají fixní hodnotu po dobu trvání jejich existence v programu
- Nemají atribut adresa
  - na jejich umístění nelze v programu odkazovat
- Statické určené v době
  - překladu – např. Java

```
static final int zero = 0;
```

  - zavádění programu

```
static final Date now = new Date();
```

Konstanty (2)

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

```
#include <stdio.h>
const int i= 10;             // statická při překladu
const int j = 20* 20 + i; // statická při překladu

int f(int p) {
  const int k = p + 1;     // dynamická
  return k + i + j;
}
```

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

Datové typy

- Definuje kolekci datových objektů a operací na nich proveditelných
  - primitivní
    - jejich definice nevyužívá jiných typů
  - složené
- `Integer`
  - reflektují hardwarové možnosti počítače
  - aproximace celých čísel
- Floating Point
  - obvykle reflektují hardware
  - aproximace reálných čísel
  - v jazycích pro vědecké výpočty zaváděn min. ve dvou formách
    - `Single` (4B) – znamínkový bit + 8 bitů exponent + 23 bitů mantisa
    - `Double` (8B) – znamínkový bit + 11 bitů exponent + 52 bitů mantisa

 © Karel Ježek & Martin Zíma

Datové typy (2)

- Decimal
  - pracují s přesným počtem cifer
  - např. finanční aplikace
- Boolean
  - obvykle implementovaný bytově, lze i bitově
  - v C je nahržen int s hodnotami 0 a nenula
- Character
  - kódování
    - ASCII (128 znaků)
    - UNICODE (16 bitů) – Java, Python, C#

Datové typy (3)

- Ordinální
  - tj. zobrazitelné, přečíslitelné do `Integer`
  - zahrnují
    - primitivní typy mimo Floating Point
    - vyjmenované uživatelem
      - používají se pro čitelnost a spolehlivost programu
      - zahrnují
        - vyjmenované typy
        - typ interval
  - výhody: čitelnost, bezpečnost

Datové typy (4)

- Vyjmenované typy
  - uživatel vyjmenuje posloupnost hodnot typu
  - implementují se jako seznam pojmenovaných `Integer` konstant
  - ADA, C++, Pascal

    ```
    type BARVA = (BILA, ZLUTA, CERVENA, CERNA);
    ```
  - C#

    ```
    enum dny{pon, ut, str, ctvr, pat, sob, ned};
    ```
  - Java (od verze 1.5)

    ```
    public enum Barva (BILA,ZLUTA,CERVENA,CERNA);
    ```
    - v nejjednodušší podobě lze chápat jako seznam `Integer`ů
    - realizovaný ale jako třída `Enum`
      - možnost konstruktorů, metod, …
      - uživatel nemůže vytvářet potomky `Enum`
    - `enum` je klíčové slovo

Datové typy (5)

- Typ interval
  - souvislá část ordinálního typu
  - implementují se jako typ jejich rodiče
  - např. `type RYCHLOST = 1 .. 5`

String

- Hodnotou je sekvence znaků
- Pascal, C, C++
  - neprimitivní typ, pole znaků
- Java
  - `String class` – hodnotou jsou konstantní řetězce
  - `StringBuffer class`
    - lze měnit hodnoty a indexovat
    - podobné jako znakové pole
- ADA, Fortran90, Basic, Snobol
  - spíše primitivní typ, množství operací
- Délka řetězců
  - statická (Fortan90, ADA, Cobol, String class Javy) – efektivní implementace
  - limitovaná dynamická (C, C++) – konec indikují znakem `null`
  - dynamická (Snobol4, Perl, Python) – časově náročná implementace

Array

- Agregát homogenních prvků, identifikovatelných pozicí relativní k prvému prvku
- Typy indexů
  - celočíselné – C, Fortran, Java
  - ordinální – ADA, Pascal
- Způsob alokace
  - statická = pevné délky
    - ukládána do statické paměti (Fortran77, globální – Pascal, C)
    - ukládána do zásobníku (lokální – Pascal, C (mimo static))
      - meze indexů jsou konstantní
  - dynamická v zásobníku (ADA)
    - délku určují hodnoty proměnných, flexibilní
  - dynamická na haldě (Fortran90, Java, Perl)

Array (2)

- Přístupová funkce pro jednorozměrné pole má tvar

```
location(vector[k]) =
  address(vector[lower_bound]) +
  ((k-lower_bound) * element_size)
```

- Přístupová funkce pro vícerozměrná pole (řazení po sloupcích / řádcích) má tvar

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

  - lb znamená lower bound

Array (3)

- Co o poli potřebuje vědět překladač je tzv. **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

 © Karel Ježek & Martin Zíma

Asociativní pole

- Neuspořádaná kolekce dvojic (klíč, hodnota)
  - nemají indexy
  - Perl, Python
- Příklad v Perlu
  - jména začínají `%`
  - literály jsou odděleny závorkami

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

  - zpřístupnění je pomocí složených závorek s klíčem

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

  - prvky lze odstranit pomocí `delete`

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

Record

- Záznam
- Heterogenní agregát datových prvků, které jsou zpřístupněny jménem
  - kartézský součin v prostoru položek
  - odkazování na položky
    - Cobol: `OF` notace
    - ostatní: . notace
  - příklad v C:

```
struct {int i; char ch;} v1,v2,v3;
```

    - operace
      - přiřazení – pro identické typy
      - inicializace, porovnávání

Record (2)

- Uniony
  - typy, jejichž proměnné mohou obsahovat v různých okamžicích výpočtu hodnoty různých typů
  - příklad v C:

```
union u_type {int i; char ch;} v1,v2,v3;
```

  - příklad v Pascalu:

```
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);
```

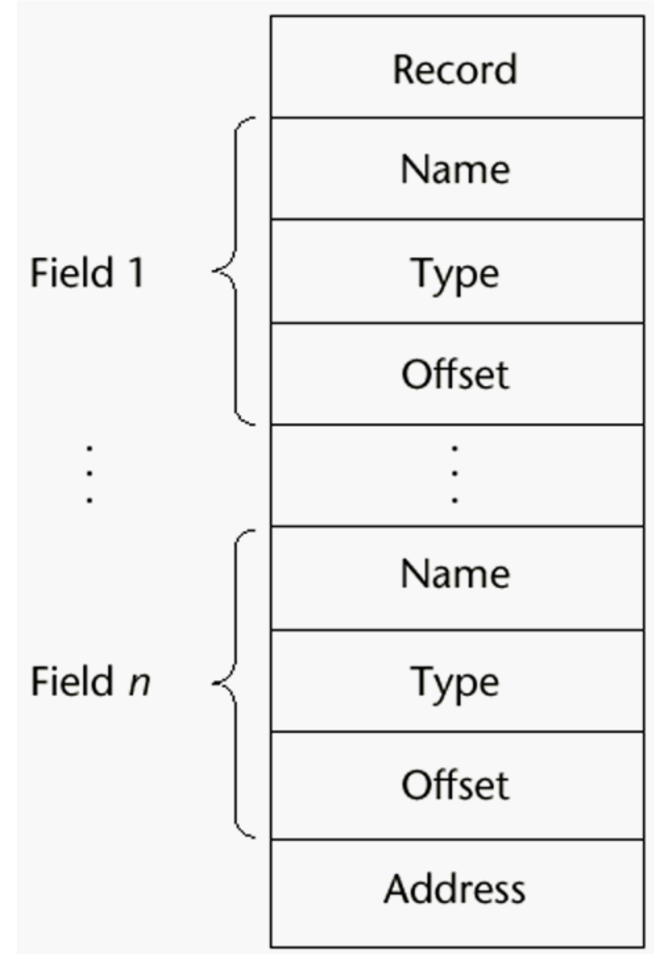
  - řádně se přeloží a vypíše nesmyls

 © Karel Ježek & Martin Zíma

Record (3)

- Přístup k prvkům záznamu je mnohem rychlejší než k prvku pole

  ```
  Location (record.field_i) =
  address(record.field_1) + Σ offset_i
  ```
  - i = 1 až n - 1



Set

- Proměnné mohou mít hodnotu neuspořádané kolekce hodnot ordinálního typu
  - Pascal

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

  - Java, Python
    - mají třídu pro množinové operace
  - ADA, C
    - nevedou
  - implementace
    - bitovými řetězci, používají logické operace
    - vysšší efektivita než pole, ale menší pružnost
      - omezovaný počet prvků

 © Karel Ježek & Martin Zíma

Pointer

- Nabývá hodnot paměťového místa a nil (či null)
- Použití pro
  - nepřímé adresování normálních proměnných
  - dynamické přidělování paměti
- Operace s ukazateli
  - přiřazení, dereference
- Špatnost ukazatelů
  - *dangling* (neurčená hodnota) *pointers*
  - ztracené proměnné

 © Karel Ježek & Martin Zíma

Pointer (2)

- 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);
    ```

  - implementace `dispose` znemožňující *dangling* není možná

ztracená paměť

 © Karel Ježek & Martin Zíma

Pointer (3)

- C, C++
  - `*` je operátor dereference
  - `&` je operátor produkující adresu proměnné
    - `j = *ptr` přiřadí `j` hodnotu umístěnou v `ptr`
  - pointer bývá položkou záznamu
    - `*p.položka` nebo `p->položka`
  - 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 přenášek funkce jako parametry
  - dangling a ztraceným pointerům nelze nijak zabránit

 © Karel Ježek & Martin Zíma

Pointer (4)

- 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

Pointer (5)

- ADA
  - pouze proměnné z haldy (`access type`)
  - dereference (`pointer.jmeno_polozky`)
  - 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 ↔ safety

 © Karel Ježek & Martin Zíma

Pointer (6)

- 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 deklarovány implicitně ⇒ dangling reference nemůže vzniknout
  - paměť haldy je uvolněna *garbage collectorem*, poté, co systém detekuje, že již není používána

© Karel Ježek & Martin Zíma

Pointer (7)

- C#
  - má pointery tvaru

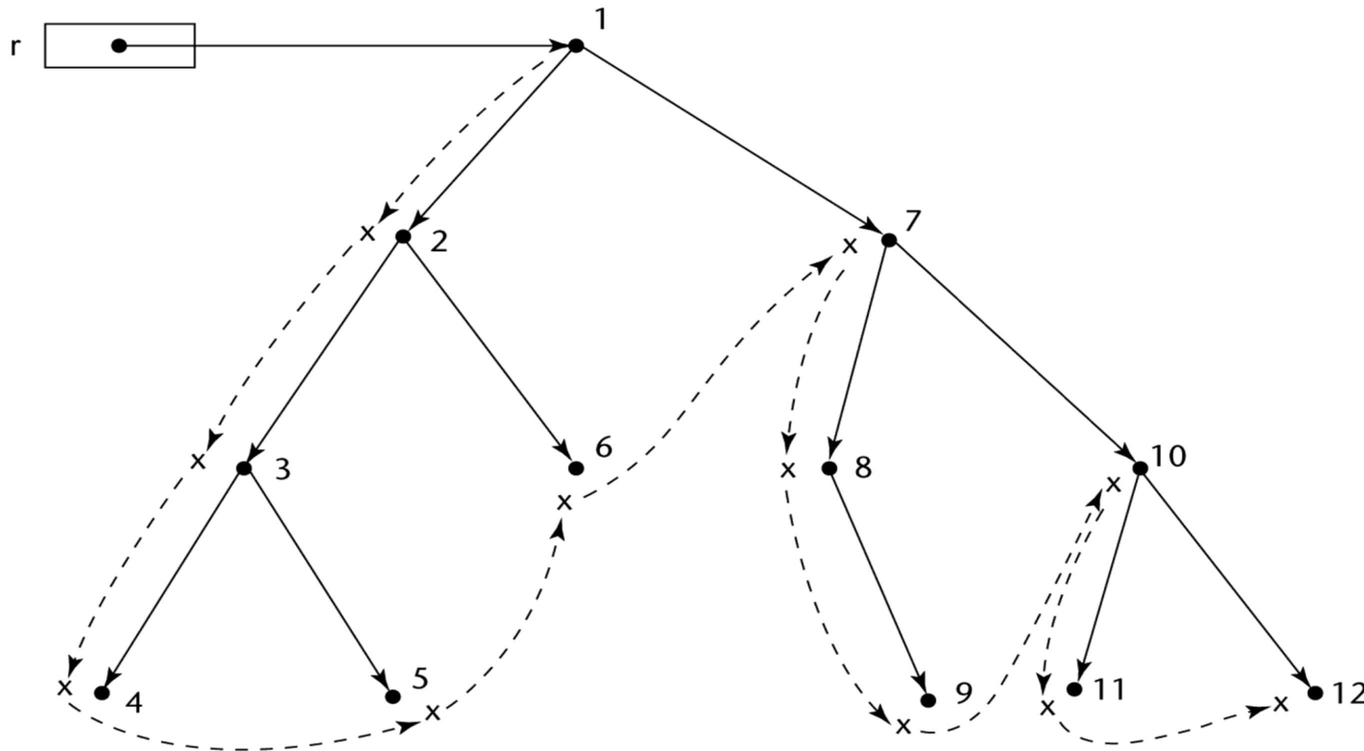
    `referent-type *identifikator`
    - type může být i `void`
  - metody pracující s pointerem musí mít modifikátor `unsafe`
  - referenční proměnné má také

Manažování haldy

- Čítačem odkazů
  - každá buňka je vybavena čítačem
  - když má hodnotu nula, vrátí ji do volných
- Garbage collectorem
  - když nemá, prohledá všechny buňky a haldu zdrcne

Manažování haldy (2)

- Průběh označování paměťových míst čističem



Výrazy a příkazy

- Výrazy
  - aritmetické
  - logické
- Ovlivnění vyhodnocení?
  - precedence operátorů?
  - asociativita operátorů?
  - arita operátorů?
  - pořadí vyhodnocení operátorů?
  - je omezován vedlejší efekt na operandech?

    `X = f(&i)+(i = i + 2);` - je dovoleno v C
  - je dovoleno přetěžování operátorů?
  - je alternativnost zápisu (Java, C)

    `C = C + 1;  C += 1;  C++;  ++C;`
  - mají tentýž efekt, což neprospívá čitelnosti

Výrazy a příkazy (2)


```
#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 funkci, ale Microsoft C vyhodnocuje zprava doleva */

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

Výrazy a příkazy (3)

- Pozor na záměnu u C a C++
  - `if(x = y)` – přiřazení, ale produkuje logickou hodnotu
  - `if(x == y)` – porovnání
    - proto C#, Java dovolí za `if` pouze logický výraz
- Logické výrazy nabízí možnost zkráceného vyhodnocení
  - A and B bude false, jestliže A je false
  - A or B bude true, jestliže A je true
- ADA
  - if A and then B then S1else S2 end if;
  - if A or else B then S1 else S2 end if;
- Java
  - např. pro `i = 1; j = 2; k = 3;`

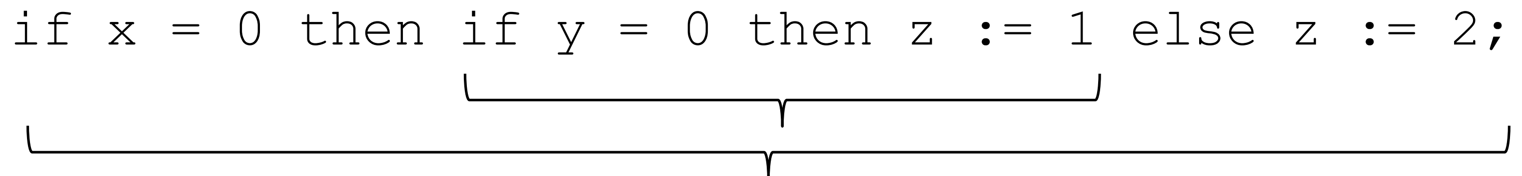
    `if (i == 2 && ++j == 3) k = 4;`
    - jaké hodnoty budou mít proměnné `i, j, k`?

Využití podmíněných výrazů

- V přiřazovacích příkazech C jazyků, Java

```
k = (j == 0) ? j + 1 : j - 1;
```

  - přiřazení produkuje v C jazycích a Javě hodnotu
    - může být ve výrazu
- V řídících příkazech
  - neúplný a úplný podmíněný příkaz
  - problém "*dangling-else*" při vnořovaném `if`



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

Vícenásobný selektor - přepínač

- ADA, Pascal

```
case expression of
  constant_list1:
    statement1;
  …
  constant_listn:
    statementn;
end;
```

  - lze přidat alternativu `others/else`
  - návěští jsou ordinálního typu

- C jazyky, Java

```
switch(expression) {
  case constant_expr1:
    statements1;
  …
  case constant_exprn:
    statementsn;
  default: statements;
}
```

  - alternativy separuje `break` (C, C++, Java) nebo také `goto` (C#)
  - návěští je výraz typu `int`, `short`, `char` a `byte`
    - u C# navíc `enum` a `string`

Cykly

- Primitivní tvar `loop … end loop;`
- Cyklus logicky řízený pretestem
  - ADA, Pascal `while podmínka do;`
  - C jazyky, Java `while podmínka příkaz;`
- Cyklus logicky řízený posttestem
  - Pascal `repeat … until podmínka;`
  - C jazyky, Java `do {…} while (podmínka);`

 © Karel Ježek & Martin Zíma

Cykly (2)

- Cyklus for
  - s parametrem cyklu, krokem, iniciální a koncovou hodnotou
  - Pascal
    - `for variable := init to final do statement;`
      - po skončení cyklu není hodnota `variable` definována
  - ADA
    - obdobně, ale `variable` je implicitně deklarovanou proměnnou cyklu, vně neexistuje
  - Java
    - vyžaduje explicitní deklaraci parametru cyklu
  - C++, C#, Java

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

Cykly (3)

- 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?

(Rozporný) příkaz skoku

- Nevýhody
  - znepřehledňuje program
  - je nebezpečný
  - znemožňuje formální verifikaci programu
- Výhody
  - snadno a efektivně implementovatelný
- Formy návěští
  - Pascal `číslo:`
  - Fortran `číslo`
  - C jazyky `identifikátor:`
  - ADA `<<identifikátor>>`
  - PL/1 `proměnná`
  - Java
    - nemá skok `GOTO`, částečně je nahrazuje `break`

 © Karel Ježek & Martin Zíma

(Rozporný) příkaz skoku (2)

- Zásada: používej skoků co nejméně
- Příklad katastrofického důsledku snahy po obecnosti
  - zavedení proměnné typu návěští v PL/1

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

  - v příkazu `GOTO` existuje proměnná `L`, ale hodnota `L1` již neexistuje
    - jsme mimo blok `B1`

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

Podprogramy (2)

- Pojmy
  - definice podprogramu
  - záhlaví podprogramu
  - tělo podprogramu
  - formální parametry
  - skutečné parametry
  - korespondence formálních a skutečných parametrů
    - jmenná

      `jmenopp(jmenoformalniho=jmenoskutecneho, …)`
    - poziční

      `jmenopp(jmenoskutecneho, jmenoskutecneho, …)`
  - default (předběžné) hodnoty parametrů

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?

Umístění lokálních proměnných

- Dynamicky v zásobníku
  - umožní rekurzivní volání a úsporu paměti
  - potřebuje čas pro alokaci a uvolnění, nezachovává historii, musí adresovat nepřímo
  - Pascal, ADA výhradně dynamicky, Java, C většinou
- Dynamicky na haldě
  - Smalltalk
- Staticky
  - opačné vlastnosti
  - Fortran90 většinou, C `static` lokální proměnné

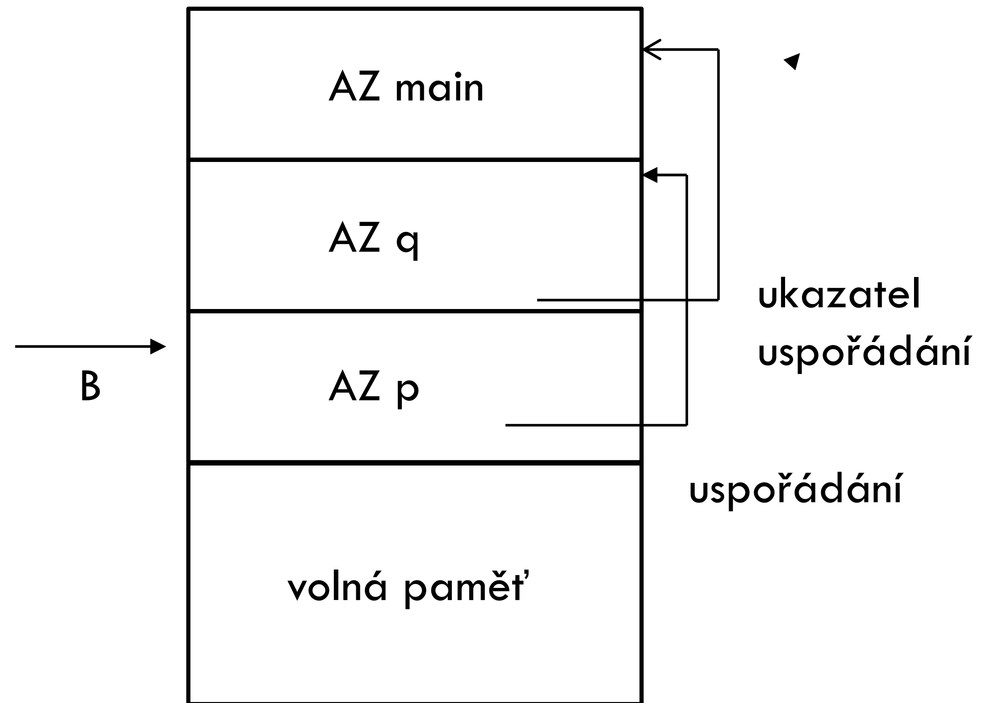
Aktivační záznamy

- Uložena lokální data podprogramů spolu s dalšími údaji
  - místo pro lokální proměnné
  - místo pro předávané parametry
  - místo pro funkční hodnotu u funkcí
  - návratová adresa
  - informace o uspořádání informačních 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

Aktivační záznamy (2)

- Příklad v 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;
}
```

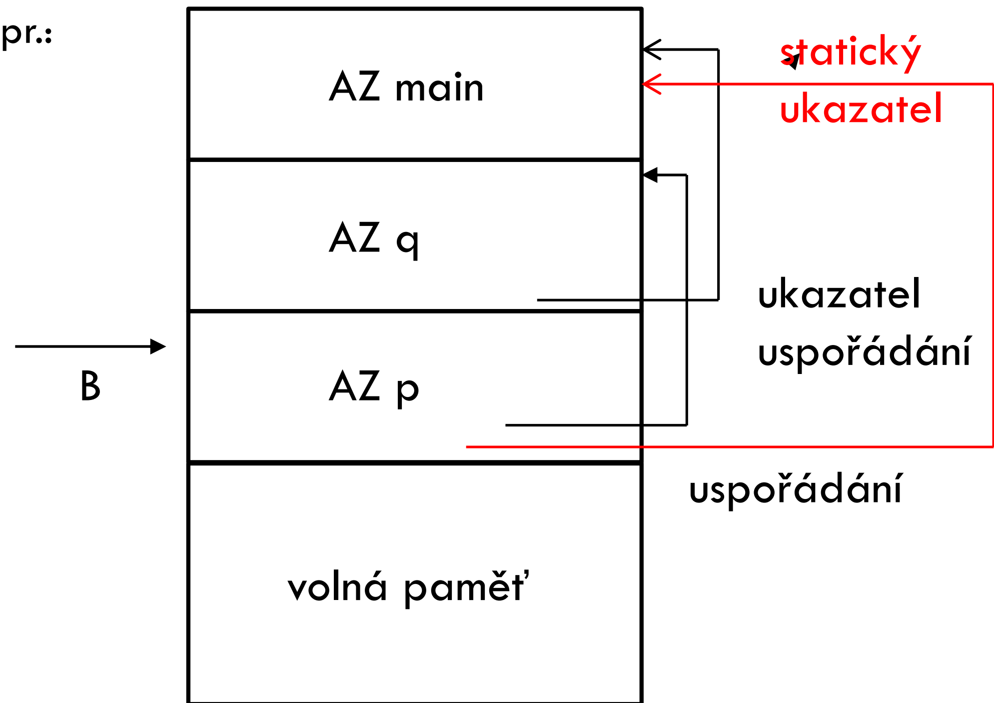


- Jazyky se statickým rozsahem platnosti proměnných a vnořováním podprogramů vyžadují dva typy ukazatelů uspořádání (řetězců ukazatelů)
  1. (dynamický) na zrušení AZ opuštěných rozsahových jednotek (viz výše)
  2. (statický) pro přístup k nelokálním proměnným

Aktivační záznamy (3)

- 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;
}
```



- K přistupování na proměnnou `x` z funkce `p` je nutno použít statický ukazatel
  - C, C++ - má statický řetěz délku 1, není proto nutný
  - ADA, Pascal – statický řetěz může nabývat libovolné délky

Aktivační záznamy (4)

- Použití řetězce dynamických ukazatelů k přístupu 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
- C jazyky, 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 program, 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

Způsoby předávání parametrů

- Hodnotou (in mode)
  - obvykle předáním hodnoty do parametru (lokální proměnné) programu (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

Způsoby předávání parametrů (2)

- Odkazem (in out 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

 © Karel Ježek & Martin Zíma

Způsoby předávání parametrů (3)

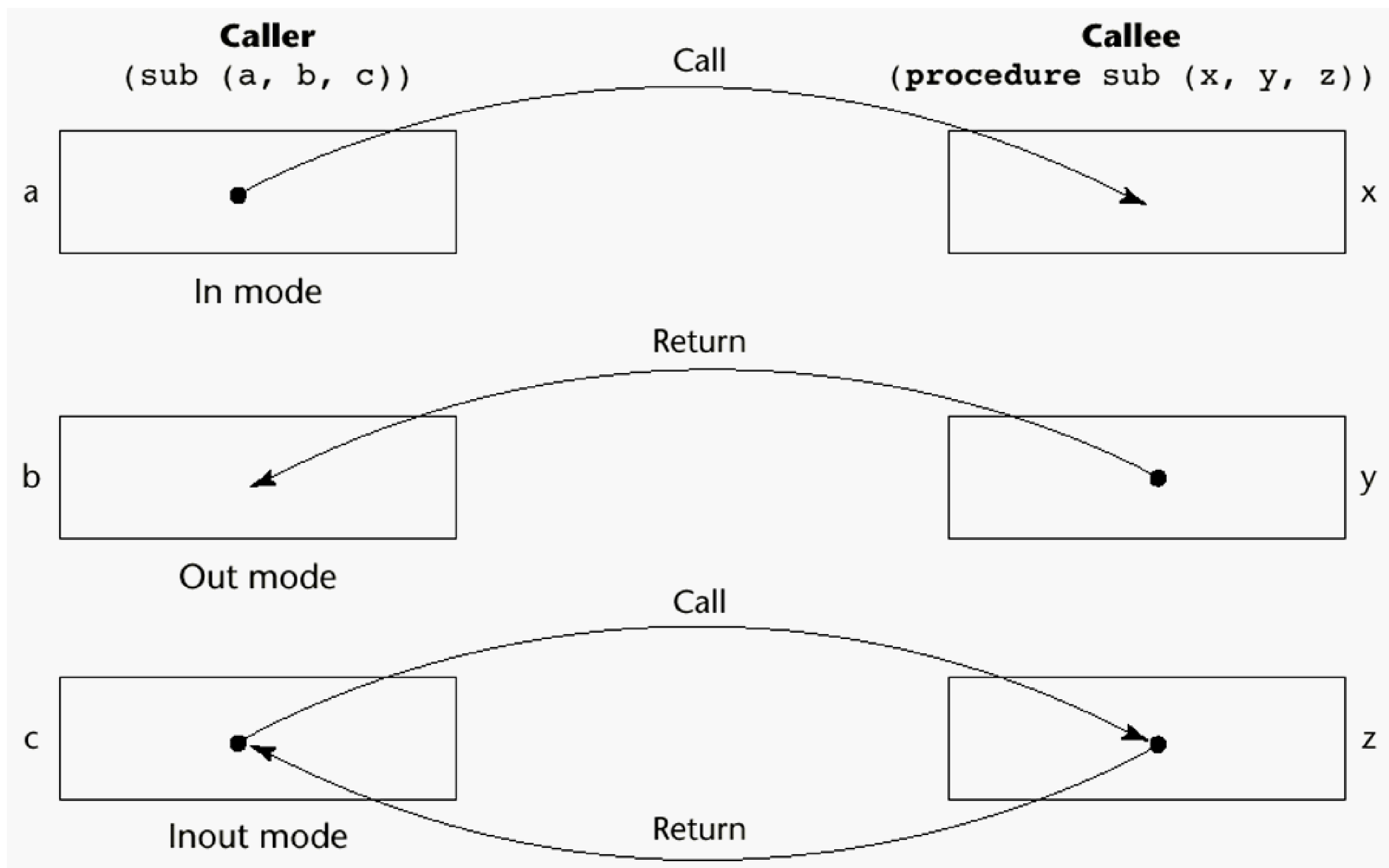
- Předání vícerozměrného pole
  - je-li podprogram separátně překládán, potřebuje znát velikost pole
  - C, C++
    - má pole polí ukládané po řádcích
    - údaje pro mapovací funkci požadují zadání počtu sloupců v definici funkce

```
void fce(int matice [][10]) {…}
```

    - výsledná nepružnost vede k preferenci použití pointerů na pole
  - Java, podobně ADA
    - má jednorozměrné pole s prvky typu pole
    - každý objekt pole dědí `length` atribut
    - lze proto deklarovat pružně

```
float fce(float matice [][]) {…}
```

Způsoby předávání parametrů (4)



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
  - mělká vazba
    - platné je prostředí volajícího podprogramu (`sub4`)
    - používá SNOBOL
  - hluboká vazba
    - platí prostředí, kde je definován volaný podprogram (`sub1`)
    - používají blokově strukturované jazyky
  - ad hoc vazba
    - platí prostředí příkazu volání, který předává podprogram jako parametr (`sub3`)
    - nepoužívá se

© Karel Ježek & Martin Zíma

Přetěžovaný podprogram

- ADA, C++, Java – má stejné jméno s jiným podprogramem a existuje ve stejném prostředí platnosti
  - poskytuje **ad hoc polymorfismus**
- ADA, C++ – dovolují i přetěžování operátorů
- Příklad 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 "*";
```

- Příklad C++

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

  - tzv. funkční prototyp

 © Karel Ježek & Martin Zíma

Generické podprogramy

- Dovolují pracovat s parametry různých typů
  - poskytují parametrický polymorfismus
- C++ obecný tvar
  - `template<class parameters>` definice funkce, která může obsahovat `class` parametry
- Příklad:

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

Generické podprogramy (2)

- Instalace je možná pro libovolný typ, který má definováno >
  - např. Integer

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

- Efekt je následující
  - C++ template funkce je instalována implicitně
    - když je použita v příkazu vyvolání
    - nebo je použita s & operátorem

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

OOP

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

Program Anim.java


```
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"; }
}
```

Program Anim.java (2)


```
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();
      }
}
```

- Co se tiskne?
  - dle očekávání, kočka mňouká, pes štěká

Program Zviratka.pas


```
program Zvirata;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
     Uk_Zvire = ^Zvire;
     Zvire = object
              Druh      : String;
              procedure Inicializuj;
              function Zvuky: String;
              procedure Tisk;
             end;
```

Program Zviratka.pas (2)


```
Uk_Pes = ^Pes;
Pes =   object(Zvire)
         procedure Inicializuj;
         function Zvuky: String;
        end;

Uk_Kocka = ^Kocka;
Kocka = object(Zvire)
         procedure Inicializuj;
         function Zvuky: String;
        end;
```

Program Zviratka.pas (3)


```
{-----------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;
```

Program Zviratka.pas (4)


```
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;
```

Program Zviratka.pas (5)


```
{----------Deklarace objektu--------}
     var
        U1: Uk_Zvire;
        Nezname: Zvire;
        Micka: Kocka;
        Filipes: Pes;
```

Program Zviratka.pas (6)


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

- Co se tiskne?
  - pes zde vydává neznámé zvuky
  - lze zařídit stejné chování v Java programu?

Program AnimS.java


```
class Animal {
        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"; }
}
```

 © Karel Ježek & Martin Zíma

Program AnimS.java (2)


```
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();
	}
}
```

- Co to tiskne?
  - vše se ozývá `not known`

Program AnimF.java


```
class Animal {
        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"; }
}
```

Program AnimF.java (2)


```
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();
      }
}
```

- Co se tiskne?
  - zase všichni jsou `not known`

Program AnimPF.java


```
class Animal {
        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"; }
}
```

Program AnimPF.java (2)


```
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();
      }
}
```

- Co se tiskne?
  - zase všichni jsou `not known`

Kombinace static, private a final


```
class Animal {                                    správně štěká
  String type;
  Animal() { type = "animal   "; }
// static                                 ne  ne ano  ne  ne  ne ano
// private                                ne  ne  ne ano ano ano ano
// final                                  ne  ne  ne  ne  ne ano  ne
  String sounds() { return "not known"; }
  void prnt() {…}                                   ostatní
}                                                  kombinace
class Dog extends Animal {                        hlásí chybu
  Dog() { type = "dog    "; }
// static                                 ne  ne ano  ne  ne  ne ano
// private                                ne  ne  ne ano  ne ano ano
// final                                  ne ano  ne  ne  ne ano  ne
  String sounds() { return "haf"; }
}                                                     neštěká
```

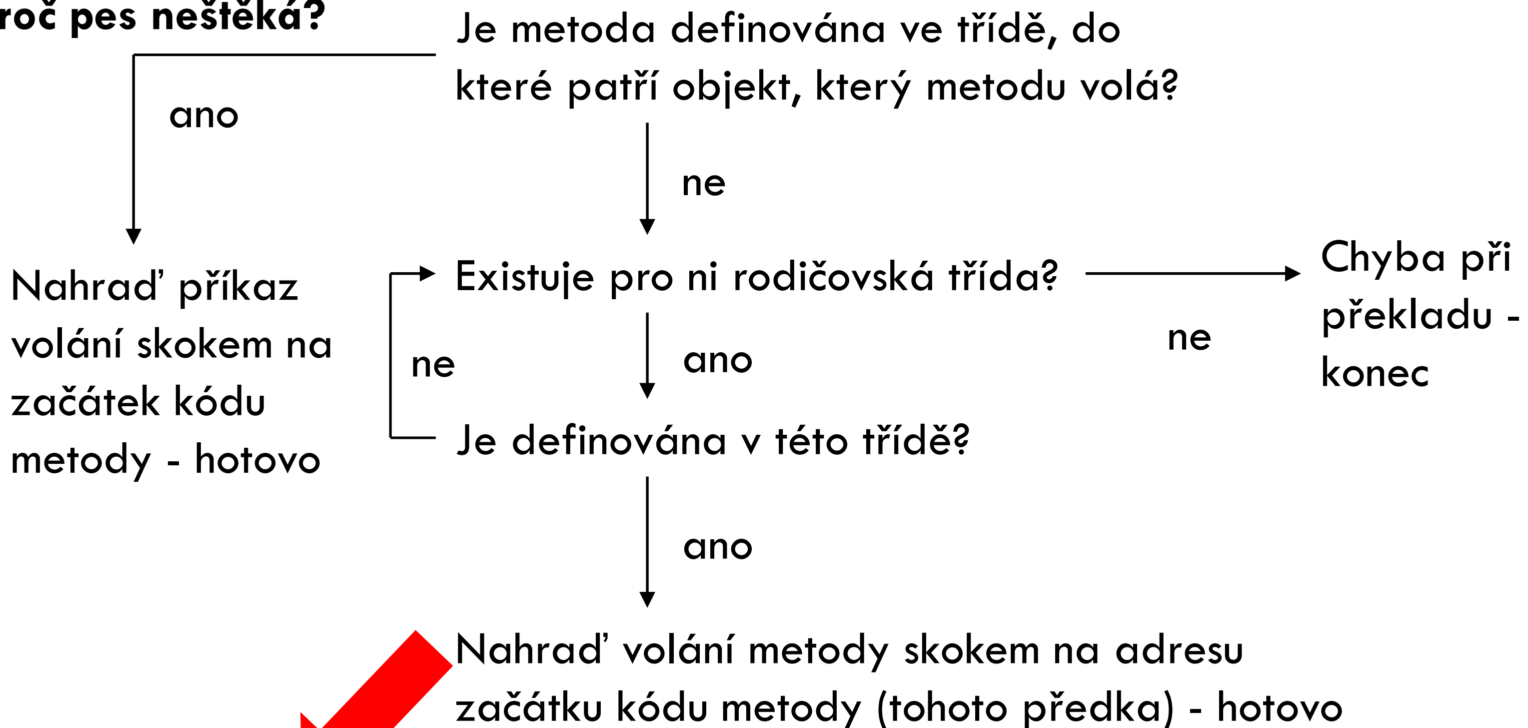
Řešení v Borland Pascalu


```
Zvire =  object
            Druh    : String;
            constructor Inicializuj;          {!!! Inicializuj musí byt konstructor}
            function Zvuky: String; virtual;  {!!! Zvuky musi byt 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 v Object Pascalu


```
Zvire =  class
            Druh     : String;
            constructor Inicializuj;          {!!! Inicializuj musí byt konstructor}
            function Zvuky: String; virtual;  {!!! Zvuky musi byt virtual}
            procedure Tisk;
          end;
 ...
 Pes =    class(Zvire)
            constructor Inicializuj;            {!!!}
            function Zvuky: String; override;   {!!! u potomka je override}
         end;
 ...
 Filipes.Inicializuj;
 Filipes.Tisk;

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

Dědičnost a statická (brzká) vazba



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

Dědičnost a dynamická (pozdní) vazba

- V tomto případě 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
    - *Virtual **M**ethod **T**able*
- Před prvým voláním virtuální metody musí být provedena (případně implicitně) speciální inicializační metoda
  - **constructor** (v Javě přímo stvoří objekt)

Dědičnost a dynamická vazba (2)

- **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. ***C**lass **I**nstance **R**ecord*
  - 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 vazba (3)

- Příklad v C++

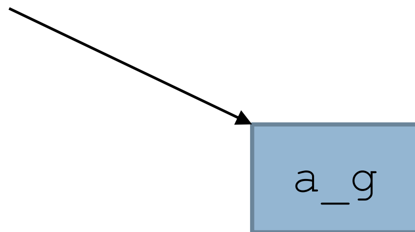
```
class A { public:void f(); virtual void g(); double x, y; } ;
class B: public A { public: void f(); virtual void h(); void g(); double z; };
```

Alokované místo (CIR) objektu `a` třídy `A`

místo pro `x`

místo pro `y`

VMT ukazatel



VMT pro `a`

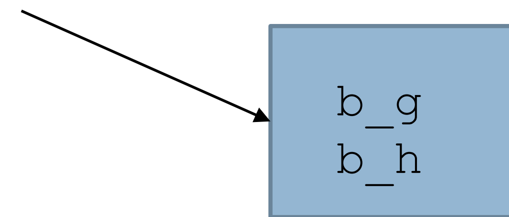
Alokované místo (CIR) objektu `b` třídy `B`

místo pro `x`

místo pro `y`

místo pro `z`

VMT ukazatel



VMT pro `b`

- poznámka: `a_f`, `b_f` jsou statické, skok na jejich začátek se zařídí při překladu

Dědičnost a dynamická vazba (4)

- Příklad v C++
  - zde `g` z třídy `A` není ve třídě `B` překrytá, takže objekt `b` dědí `a_g`

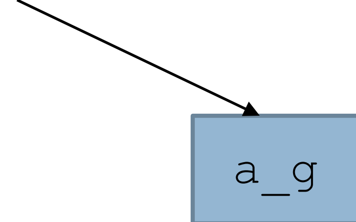
```
class A { public:void f(); virtual void g(); double x, y; } ;
class B: public A { public: void f(); virtual void h(); double z; };
```

<u>Alokované místo objektu `a` třídy `A`</u>

místo pro `x`

místo pro `y`

VMT ukazatel



VMT pro `a`

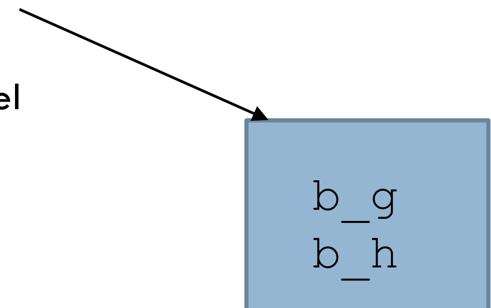
<u>Alokované místo (CIR) objektu `b` třídy `B`</u>

místo pro `x`

místo pro `y`

místo pro `z`

VMT ukazatel



VMT pro `b`

OOP konstrukce v C++

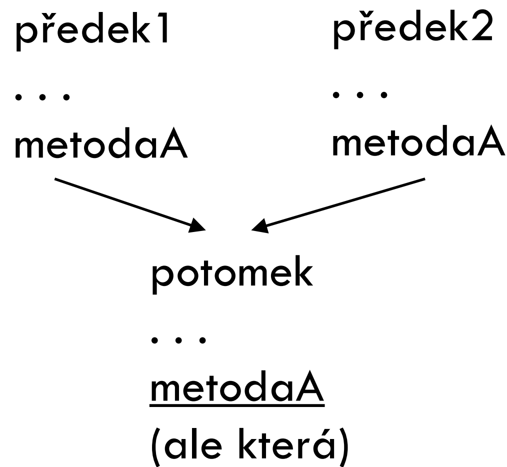
- Příklad

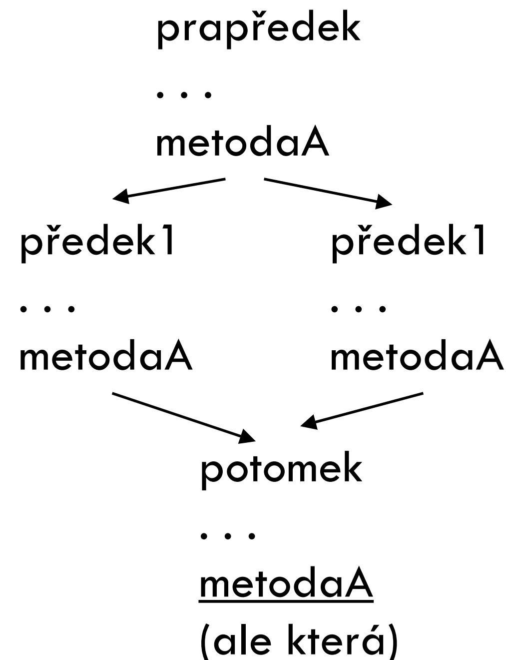
```
CLASS Predek1 {
  PUBLIC:    int p11;
  PROTECTED: int p12;
  PRIVATE:   int p13;
};
CLASS Predek2 {
  PUBLIC: int p21;
  PROTECTED: int p22;
  PRIVATE: int p23;
};
CLASS Potomek: PUBLIC Predek1, PRIVATE Predek2; {
  PUBLIC:    int pot1;
  PROTECTED: int pot2;
  PRIVATE:   int pot3;
};
```

Přístup v rodiči	Modifikace přístupu k potomkovi	
	public	**private**
public	public	private
protected	protected	private
private	nepřístupný	nepřístupný

- Jaké datové elementy má `Potomek`?
  - `PUBLIC pot1, p11`
  - `PROTECTED pot2, p12`
  - `PRIVATE pot3, p21, p22`

Násobná dědičnost v C++





- Řešení
  - napsat plným jménem (`jméno předka::jméno`)
  - `virtual předek1,` `virtual předek2` } slovem `virtual` dáme informace, že se má uplatnit jen jeden

Objektové vlastnosti C#

- Používá `class` i `struct`
- Pro dědění při definici tříd používá C++ syntax

```
public class NovaTrida:RodicovskaTrida {};
```

- V podtřídě lze nahradit metodu zděděnou od rodiče

```
new definiceMetody;
```

  - ta pak zakryje děděnou metodu stejného jména
- Metodu z rodiče lze ale přesto volat

```
base.vykresli();
```

- Dynamická vazba je u metody rodičovské třídy povinně označena `virtual`
  - u metod odvozených tříd povinně označena `override`

Objektové vlastnosti C# (2)

- Příklad

```
public classObrazec {
  public virtual void Vykresli() { . . . }
  . . .
}
public classKruh : Obrazec {
  public override void Vykresli() { . . . }
  . . .
}
public classCtverec : 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