Procedurálne programovanie

slido slido.com
#1021046
PrPr – P12

Ján Zelenka

Ústav Informatiky

Slovenská akadémia vied

Obsah prednášky

slido slido.com
#1021046
PrPr – P12

1. Zhrnutie

Spätná väzba: *https://forms.gle/6q5D2G6UwrtimXEx9*

Nadväzujúce predmety na FIIT

slido slido.com
#1021046
PrPr – P12

- Objektovo-orientované programovanie
  - prevažne programovanie v jazyku Java
- Paralelné programovanie
- Aplikačné programovanie v C++

- Algoritmy a dátové štruktúry

Hodnotenie študentov

slido slido.com
#1021046
PrPr – P12

Projekt:	32 bodov (14 + 18)
Počítačové testy:	15 bodov
Aktivita:	6 body
Záverečný test (skúška):	47 bodov
Spolu:	max. 100 bodov
Záverečné hodnotenie:	min. 56 bodov

Podmienky absolvovania - Výučba

slido slido.com
#1021046
PrPr – P12

Získanie zápočtu z cvičení (max. 53 bodov):

- nenulový počet bodov študent môže získať len za časti projektu odovzdané najneskôr v stanovených termínoch požadovaným spôsobom
- aktívna účasť na cvičeniach
  - účasť je povinná na každom cvičení
- vypracovanie **oboch** projektov v akceptovateľnej kvalite, odovzdanie podľa harmonogramu a získanie **minimálneho počtu** bodov
- absolvovanie počítačového testu a získanie min. počtu bodov
- celkovo spolu **minimálne 27 bodov.**

Podmienky absolvovania - Skúška

slido slido.com
#1021046
PrPr – P12

Podmienky na vykonanie skúšky:

- získanie min. 27b počas semestra
- **dva termíny**: riadny a opravný.
  - ak si študent chce zlepšiť známku z riadneho termínu, pred opravným termínom sa musí vzdať hodnotenia z riadneho.
- Maximálne bodové hodnotenie za skúšku – **47 bodov**

Absolvovanie predmetu:

- podľa stupnice STU (aspoň 56% z celkového hodnotenia)

Skúška

slido slido.com
#1021046
PrPr – P12

Riadny termín

- utorok **9.1.2024**
- **beh 3: 13:30-16:15**
- miestnosť: **A400** a iné

Opravný termín

- streda **31.1.2024**
- **beh 1: 8:30 – 11:15**
- miestnosť: **A400** (-1.61, kap.100)

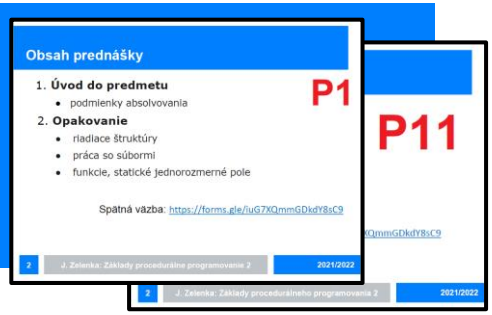
Referencie

slido slido.com
#1021046
PrPr – P12

- HEROUT, P. Učebnice jazyka C: 1. díl. České Budějovice : Nakladatelství KOPP, 2005. 271 p. ISBN 80-7232-220-6.
- HEROUT, P. Učebnice jazyka C: 2. díl. České Budějovice : Nakladatelství KOPP, 2000. 236 p. ISBN 80-85828-50-2.
- BOU EZZEDDINE, A. -- TVAROŽEK, J. Programovanie v jazyku C v riešených príkladoch (1). Bratislava: Vydavateľstvo SPEKTRUM STU, 2018. 233 p. ISBN 978-80-227-4865-0.
- KERNIGHAN, B W. -- RITCHIE, D M. Programovací jazyk C. Bratislava : Alfa, 2019. 249 p.
- http://www.cplusplus.com/reference/clibrary/
- https://en.cppreference.com/w/c

Zhrnutie

Premenné



Základné typy:

- znak – char
- celé číslo – int, short, long, long long
- desatinné číslo – float, double, long double

```
int x;
char c = 'C';
float x, y, z;
const float PI = 3.14;
```

Modifikátor:

- const – read-only premenná
- volatile – "nepredvídateľná" premenná (kompilátor ju nevie optimalizovať)

Pamäťové triedy:

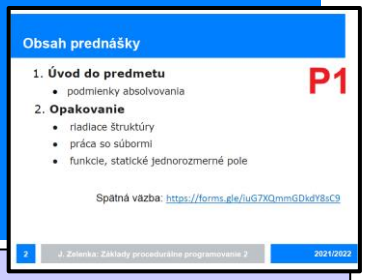
- auto (implicitne lokálne premenné, v stacku)
- register – premenná by mala byť uložená v registroch
- static – premenná je platná mimo definovaného bloku (napr. globálna premenná)
- extern – premenná je definovaná v inom súbore (module)

Pretypovanie:

- (typ) x

```
[pamatova_trieda] [modifikator] [unsigned] typ meno;
```

Operátory




```
//unarne operatory
+a //0+a
-a //0-a
!a //logicke NOT
~a //bitove NOT
++a //inkrementacia a (a=a+1)
--a //dekrementacia a (a=a-1)
a++ //vrati a potom inkrementacia
a-- //vrati a potom dekrementacia
(typ)a //pretypovanie a na typ
&a //adresa a
sizeof(a) //velkost a v bytoch
```

```
//ternarny operator
x ? a : b;

//priradenie
a = b
a *= b
```

*, /, %, +, -,
<<. >>, &, ^, |

```
//binarne operatory
a * b // nasobenie
a / b // delenie
a % b // zvysok po deleni
a + b // scitanie
a - b // odcitanie
a << b // posun bitov dolava
a >> b // posun bitov doprava
a < b // mensi
a <= b // mensi alebo rovny
a > b // vacsi
a >= b // vacsi alebo rovny
a == b // rovny
a != b // rozny
a & b // bitove AND
a ^ b // bitove XOR
a | b // bitove OR
a && b // logicke AND
a || b // logicke OR
```

Podmienky


```
if(a) b;

if(a) {b; c;}

if(a) { b; }else{ c; }

if(a){b;
}else if(c){ d;
}else{e;}
```

```
switch(a){
     case b: c;
     case d: case e: f;
     default: g;
}
```

```
switch(a){
     case b: c; break;
     case d: case e: f; break;
     default: g;
}
```

Cykly

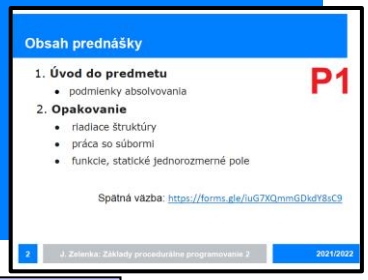

```
for (inicializacny_vyraz; podmienka_vykonania; inkrementalny_vyraz){
   //telo cyklu...prikazy;
}
```

```
while (podmienka_vykonania){
    //telo cyklu...prikazy
    //zvycajne zmena riadiacej premennej
}
```

```
do {
   // telo cyklu...prikazy
   // zvycajne zmena riadiacej premennej
}while (podmienka_vykonania)
```

- **`break`** – ukončenie cyklu a pokračovanie za cyklom
- **`continue`** – ukončenie tela cyklu a pokračovanie ďalšou iteráciou

Funkcie




```
typ/void meno([argumenty...]){ [return vysledok;]}
```

typ návratovej hodnoty (void - procedura)

(unikátne) meno funkcie

typy a mená argumentov funkcie (void ak nemá)

implementácia funkcie uzavretá v {}

hodnota, ktorá sa vráti volajúcemu funkcie (nie procedúry)

```
int f(){int i = 0;} i++;

//prototyp
typ funkcia(argumenty);
```

premenná deklarovaná vo vnútri funkcii zaniká po skončení danej funkcie

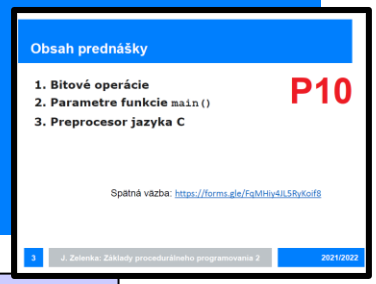
umiestnený pred deklaráciou/prvou referenciou na danú funkciou (zvyčajne pred funkciu main)

Funkcie


```
//predanie premenej y funkcii f cez argument x (hodnotou)
void f (typ x); f(y);

//predanie pola/retazca funkcii f cez argument x (ukazovatelom)
void f (typ *x); f(pole);

//predanie strukturu funkcii f cez argument x (ukazovatelom)
void f (typ *x); f(&struktura);

//predanie premenej y funkcii f cez argument x (ukazovatelom)
void f (typ *x); f(&y);
```

predanie premennej pomocou ukazovateľa umožňuje zmeniť premennú, ktorá "nepatrí" funkcii

Funkcia main




```
int main(int argc, int char *argv[]){return int;}
```

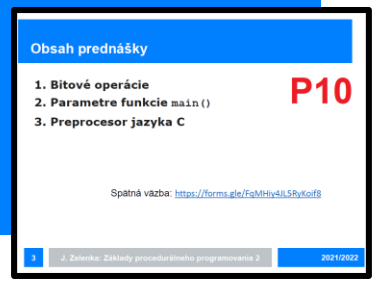
počet vstupných argumentov

vstupné argumenty (pozor až druhý prvok)

výstupný stav programu (pre OS)

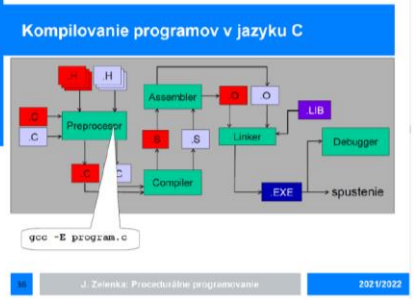
```
//vstupne argumenty programu
./program ahoj 1
./program "ahoj 1"
```

Direktívy predprocesora




```
#include <kniznica_C.h>
#include "vlastny.h"
#define meno_makra text
#if konst
#ifdef meno_makra
```

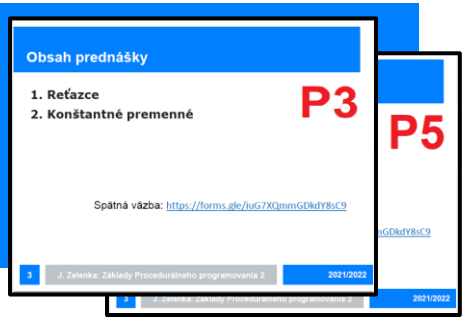
Činnosť preprocesora



- spracováva zdrojový text **PRED** kompilátorom
- zamieňa text, napr. identifikátory konštánt za číselné hodnoty
- vypustí zo zdrojového textu všetky komentáre
- všetky odkazované hlavičkové súbory sa vložia do zdrojového súboru
- prevádza podmienený preklad
- nekontroluje syntakticú správnosť programu
- riadok, ktorý má spracovávať preprocesor sa začína znakom **#**

29 | J. Zelenka: Základy procedurálneho programovania 2 | 2021/2022

Vstup/Výstup - Konzola




```
//Znak
getchar()
putchar(znak)

//Retazec
gets(retazec)
fgets(retazec, velkost, stdin)
puts("Ahoj")

//Formatovane data
scanf("%d", &x)
printf("%d", x)
```

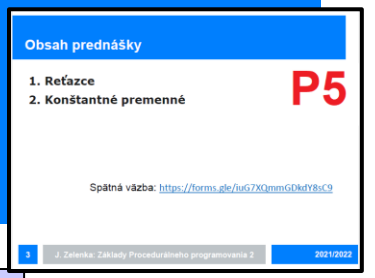
nebezpečná, odstránená v C11

& nie je potrebné pri poliach

- Vstupno-výstupné buffre treba vyprázdniť (nový riadok pri vstupe, ukončovací znak pri výstupe)
- Bezpečné funkcie umožňujú špecifikovať dĺžku vstupného reťazca (pretečenie pamäte)

```
#include <stdio.h>
```

Vstup/Výstup - Súbor

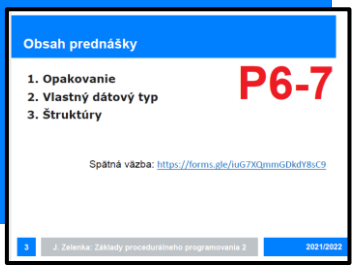



```
fopen(subor, mod+typ)
fclose(p_subor)

//Kurzor
ftell(p_subor)
fseek(p_subor, offset, origin)

//Pomocne funkcie
feof(p_subor)
rename(povodne_meno, nove_meno)
remove(subor)

//Nacitanie znaku
fgetc(p_subor)
fputc(znak, p_subor)
```

SEEK_SET (začiatok súboru),
SEEK_CUR (aktuálna pozícia)
SEEK_END (koniec súboru)

```
#include <stdio.h>

//Nacitanie retazca
fgets(retazec, n, p_subor)
fputc(retazec, p_subor)

//Formatovane data
fscanf(p_subor, format, […])
fprintf(p_subor, format, […])

//Binarny subor
fread(void *ptr,sizeof(prvok),
pocet, p_subor)

fwrite(void *ptr,sizeof(prvok),
pocet, p_subor)
```

Štruktúry

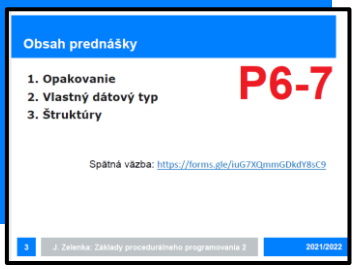



```
//definicia
struct struktura{
    typ x;
    typ y;
}
```

```
//definicia prvku spajaneho zoznamu
struct polozka{
    typ x;
    typ y;
    polozka *next;
}
```

```
//deklaracia premennych
struct struktura premenna;
struct struktura *p_struktura;

//pristup k prvkom
premenna.x
p_struktura->x
```

Vymenovaný typ & Union

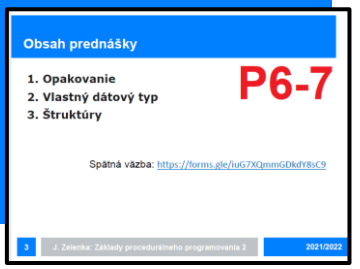



```
//definicia
union uMeno{
    typ x;
    typ y[10];
}

//deklaracia
union uMeno premenna;

//pristup k prvkom
premenna.x
```

položky unionu sa prekrývajú (vyhradí sa pamäť o veľkosti najväčšieho prvku)

```
//definicia
enum bool{
    false, true
};

//deklaracia
enum bool premenna;

//pristup k prvkom
premenna = true

if (premenna == false)
```

Vlastný dátový typ



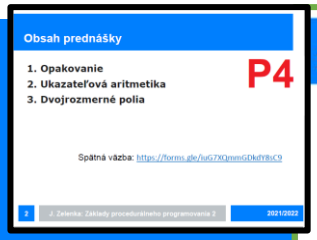

```
//definicia
typedef float *P_FLOAT;

typedef struct struktura{
    typ x;
    typ y;
} STRUKTURA;

typedef enum bool{
    false, true
} BOOL;

//deklaracia
P_FLOAT pf;
STRUKTURA s;
```

Ukazovateľ


```
//deklaracia
typ *p;
void *v; //iba priradenie (dlzka je neznama)
struct typ *p_s;
typ pole[]; //pole/retazec - ukazovatel na prvy prvok

//praca s ukazovatelom
p //adresa
*p //hodnota ulozena na tejto adrese
p_s->a
&premenna //adresa kde je ulozena premenna
*(typ*)v
```

ukazovateľ je premenná, do ktorej ukladáme adresu

Pole



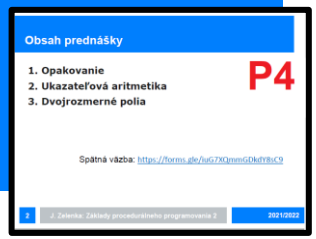

```
//deklaracia
#define N 10
typ pole[N];
typ pole[N] = {x, y, z}; //+ inicializacia
typ pole[] = {x, y, z}; //dlzka podla pociatocnych hodnot

//rozmery
pole[int]
pole[int][int]

//pristup
pole[int]
*(pole + int) //==pole[int]
&pole[int] //adresa prvku na pozicii int
pole + int //==&pole[int]
```

veľkosť poľa sa po deklarácii nedá zmeniť

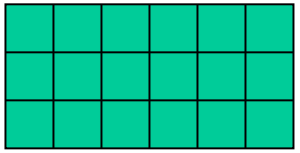
prvky sú indexované od 0 (posledný má index N-1)

prvky sú uložené na súvislých pamäťových blokoch

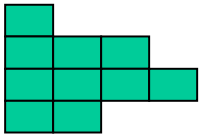
Porovnanie: charakteristika



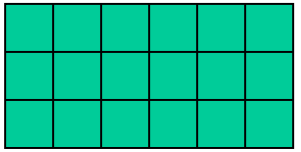
- **`xa`** (**`int xa[2][3]`**): pravoúhle pole



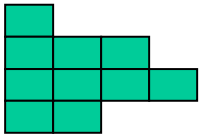
- **`xb`** (**`int *xb[2]`**): "zubaté" pole



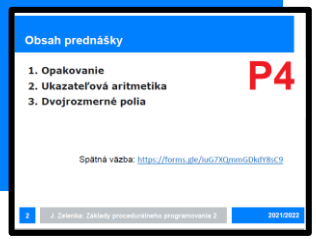
- **`xc`** (**`int (*xc)[3]`**): pravoúhle pole



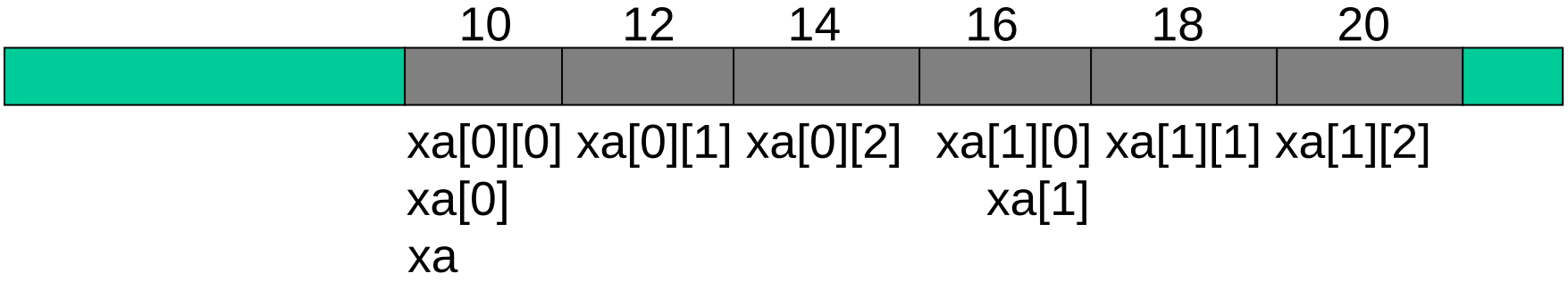
- **`xd`** (**`int **xd`**): "zubaté" pole



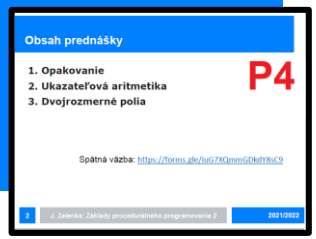
Porovnanie: pamäťové nároky



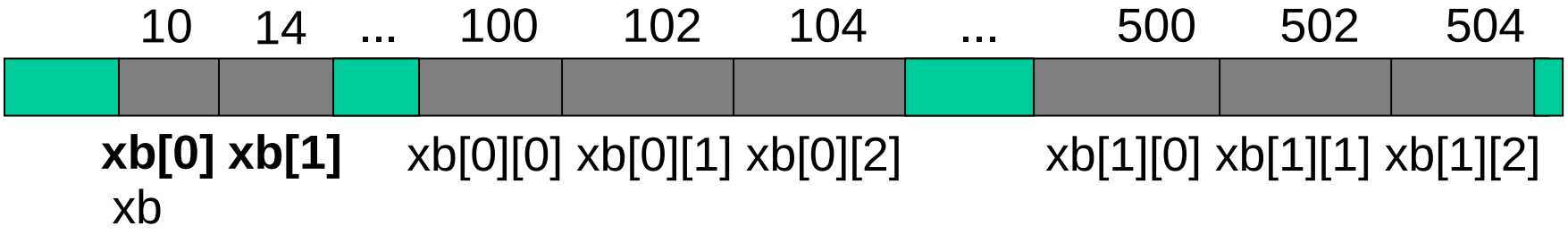
- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
  - Výhoda - nie je potrebná alokácia ďalších smerníkov
  - Nevýhoda – potreba veľkého bloku pamäte, problém v prípade poľa veľkého rozsahu



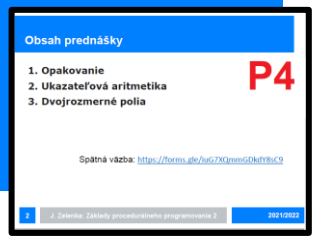
Porovnanie: pamäťové nároky



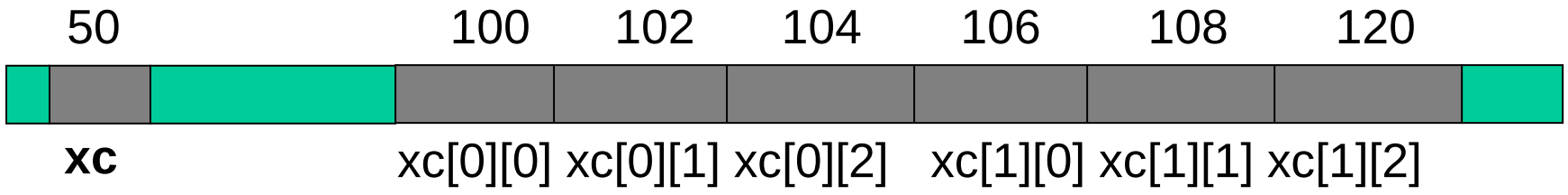
- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
- **`xb`** (**`int *xb[2]`**): naviac pamäť pre 2 ukazovatele (počet riadkov **`xb[0]`**, **`xb[1]`**)
  - riadky nie sú v pamäti uložené v celku



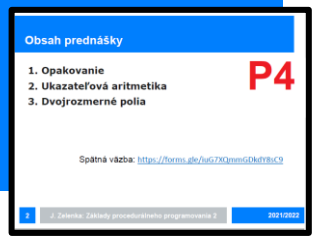
Porovnanie: pamäťové nároky



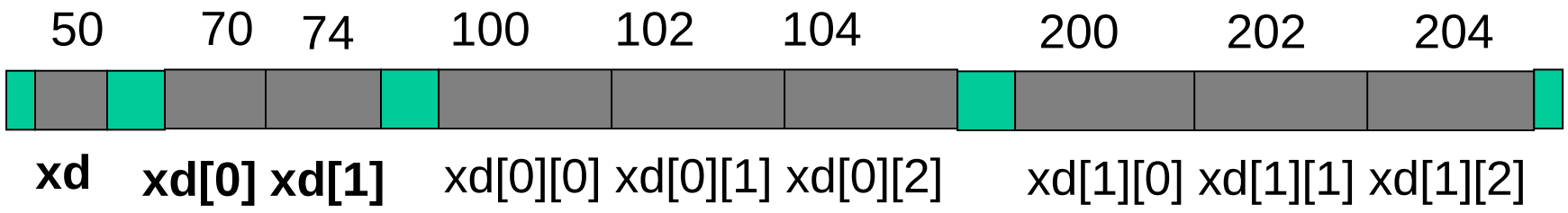
- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
- **`xb`** (**`int *xb[2]`**): naviac pamäť pre 2 ukazovatele (počet riadkov **`xb[0]`**, **`xb[1]`**)
- **`xc`** (**`int (*xc)[3]`**): naviac pamäť pre 1 ukazovateľ na typ **`int`**



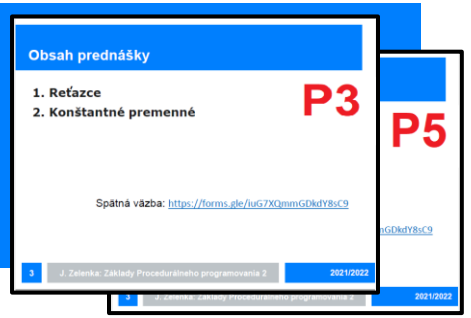
Porovnanie: pamäťové nároky



- **`xa`** (**`int xa[2][3]`**): pamäťovo najvýhodnejšia
- **`xb`** (**`int *xb[2]`**): naviac pamäť pre 2 ukazovatele (počet riadkov **`xb[0]`**, **`xb[1]`**)
- **`xc`** (**`int (*xc)[3]`**): naviac pamäť pre 1 ukazovateľ na typ **`int`**
- **`xd`** (**`int **xd`**): naviac 3 ukazovatele (pre **`xd`** a riadky), najpomalší prístup k prvkom
  - riadky nie sú v pamäti uložené v celku



Reťazce a znaky



- znak – 'A'
- reťazec – "Ahoj"
- ukončovací znak – '\0'
- reťazec je pole znakov

```
#include <string.h>
strlen(retazec);

strcpy(a, b);
strcat(a, b);
strcmp(a, b);
```

bezpečnejšie

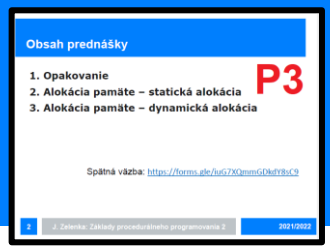
```
strncpy(a, b, n);
strncat(a, b, n);
strncmp(a, b, n);
```

```
//deklaracia
char retazec = "Ahoj";
char retazec[5] = {'A','h','o','j',0};

//pristup k znakom retazca
for(int i=0; retazec[i]; i++){}
```

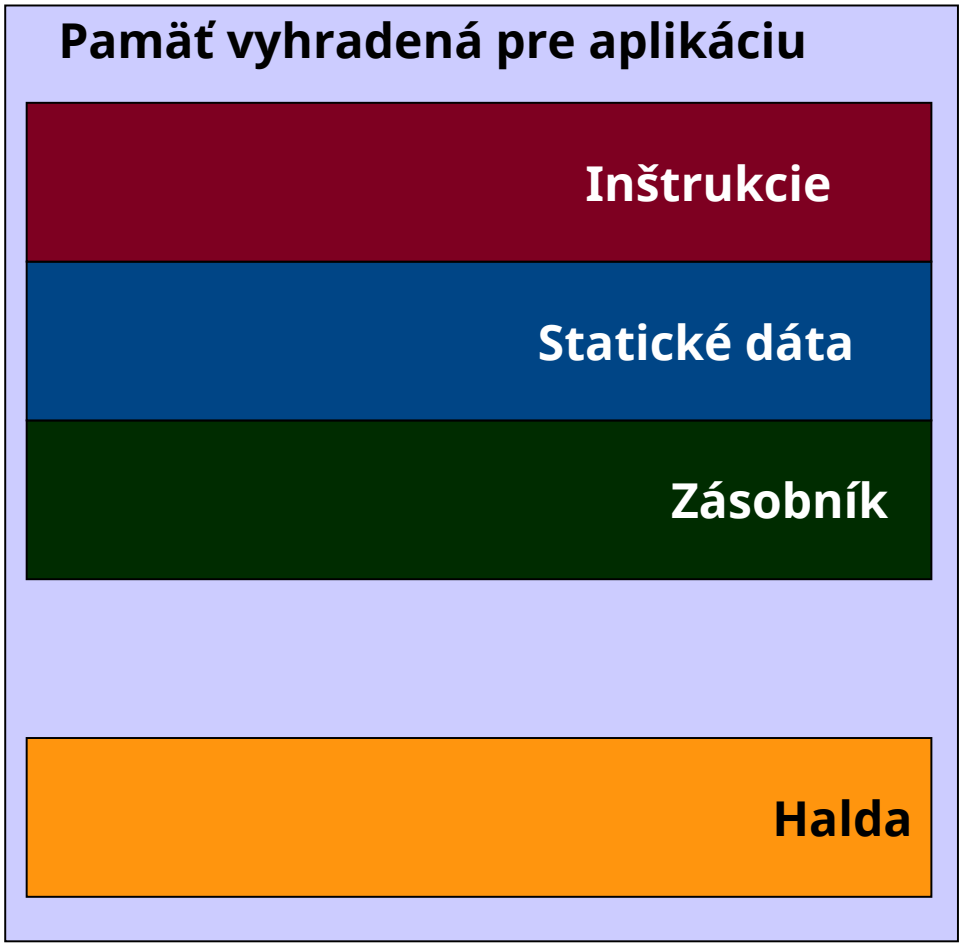
'\0' sa vyhodnotí ako false

```
#include <ctype.h>

tolower(char)
toupper(char)
isalpha(char)
islower(char)
isupper(char)
isnumber(char)
isblank(char)
```

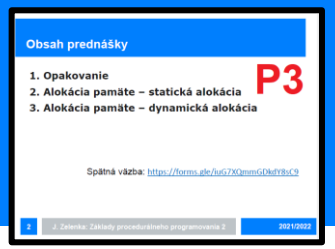
Organizácia pamäte



- Inštrukcie
  - Asemblerovský kód aplikácie
- Statické dáta (static)
  - Globálne premenné
- Zásobník (stack)
  - Volania funkcií (iné pre každú funkciu)
  - Lokálne premenné
- Halda, hromada (heap)
  - Dynamicky alokované premenné (malloc, free)
  - Iné pri každom spustení



Alokácia pamäte



Vyhradenie pamäťového priestoru

- **Statická alokácia**
  - trvalé alokovanie miesta v dátovej oblasti
  - **Životnosť**: od spustenia po koniec programu
  - riadi operačný systém
- **Dynamická alokácia**
  - pamäťové nároky vznikajú a zanikajú počas behu programu
  - **Životnosť**: od alokovania po uvoľnenie pamäte!
  - riadi programátor

Dynamická alokácia pamäte


```
//Alokacia pamate
malloc()

//dynamicka alokacia premennej
typ *x;
x = malloc(sizeof(typ));

//dynamicka alokacia pola/retazca
typ *pole;
pole = malloc(sizeof(typ)*velkost);

//dynamicka alokacia struktury
struct typ *p_struct;
p_struct = malloc(sizeof( struck typ));
```

```
#include <stdlib.h>
```

```
//Uvolnenie pamate
free(ukazovatel);

//Realokacia
realloc(p, velkost);
```

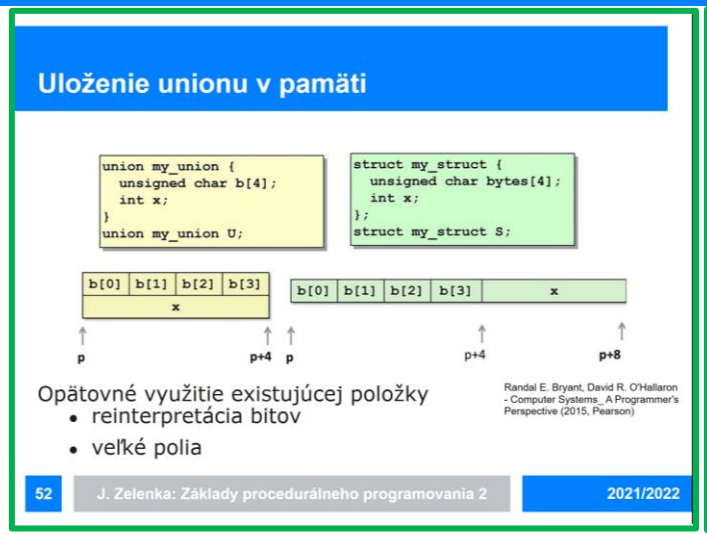
Rodostrom

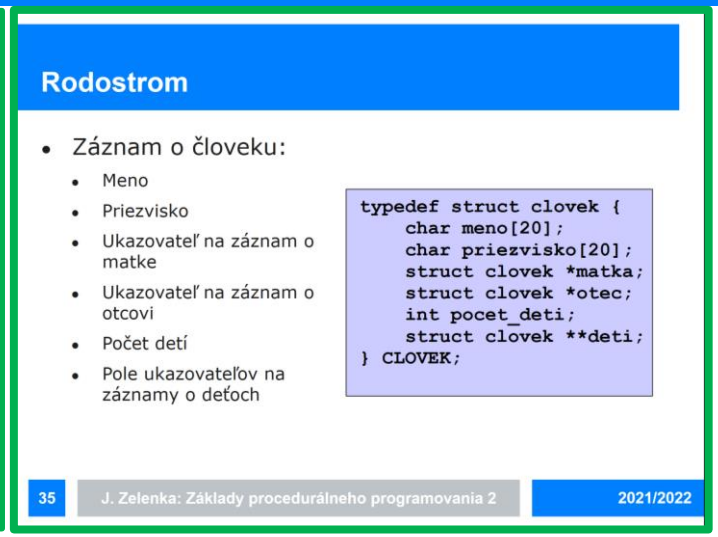
slido slido.com
#1021046
PrPr – P12

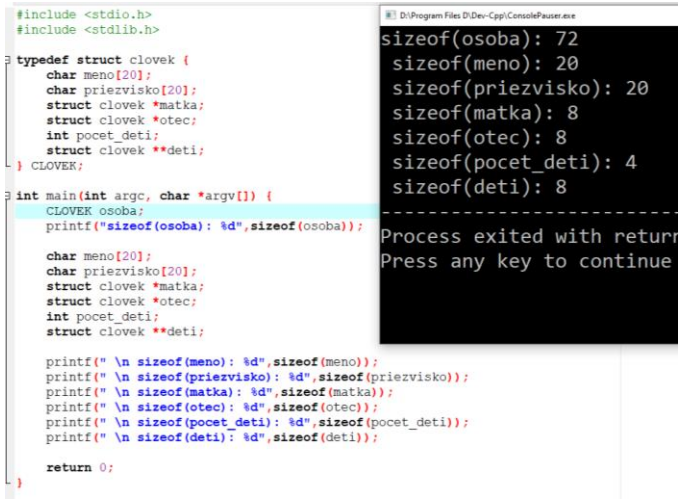
```
typedef struct clovek {
    char meno[20];
    char priezvisko[20];
    struct clovek *matka;
    struct clovek *otec;
    int pocet_deti;
    struct clovek **deti;
} CLOVEK;
```

Rodostrom - alokácia pamäte

slido slido.com
#1021046
PrPr – P12

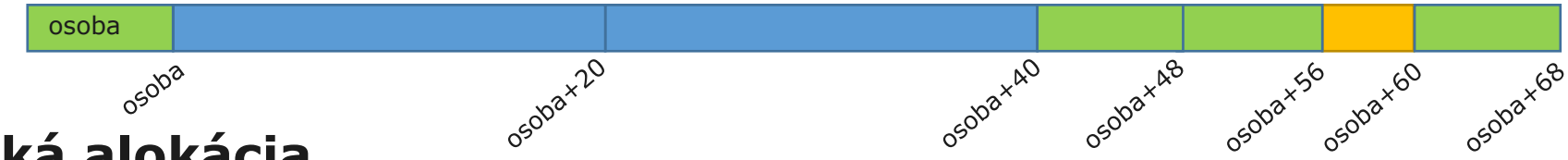






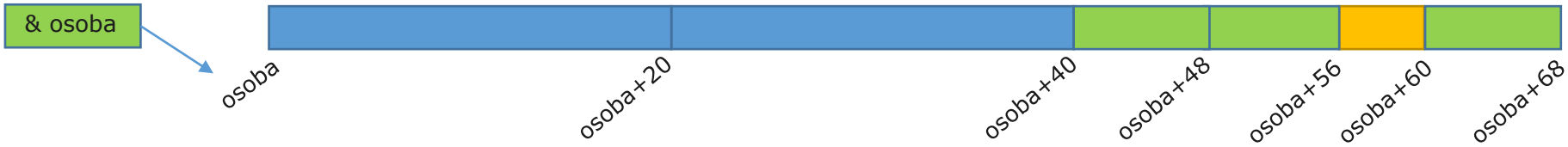
- **Statická alokácia**

  CLOVEK osoba;



- **Dynamická alokácia**

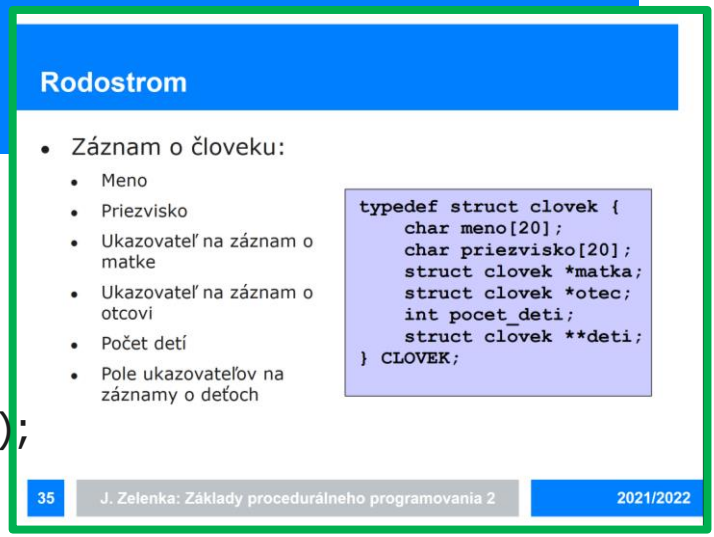
  CLOVEK *osoba; osoba=(CLOVEK*)malloc(sizeof(CLOVEK));

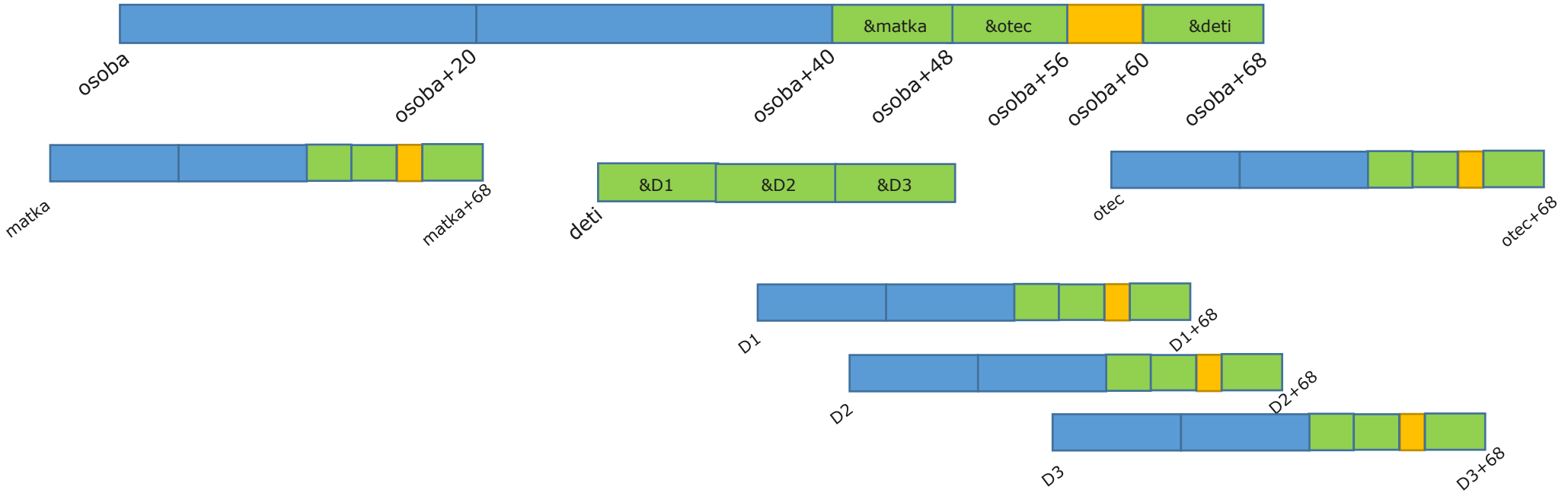


Rodostrom - alokácia pamäte

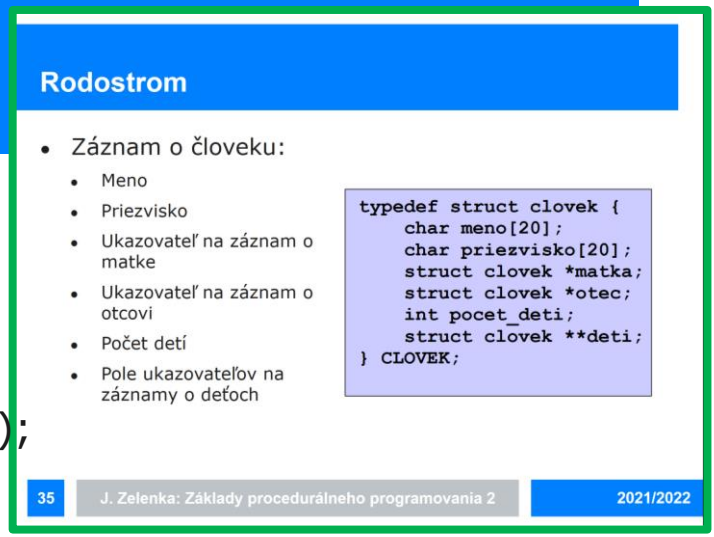
CLOVEK osoba;

```
osoba.matka = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.otec = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.pocet_deti =3;
osoba.deti = (CLOVEK**)malloc(osoba.pocet_deti *sizeof(CLOVEK*));
for (i=0; i< osoba.pocet_deti; i++)
      osoba.deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
```



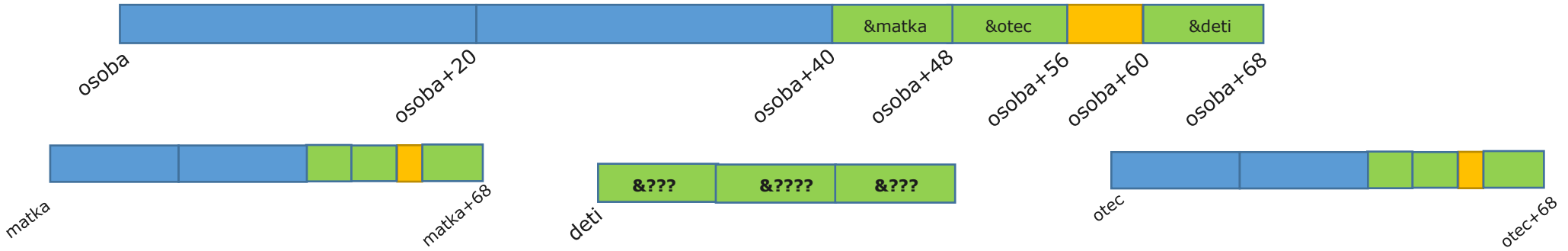


Rodostrom - uvolnenie pamäte

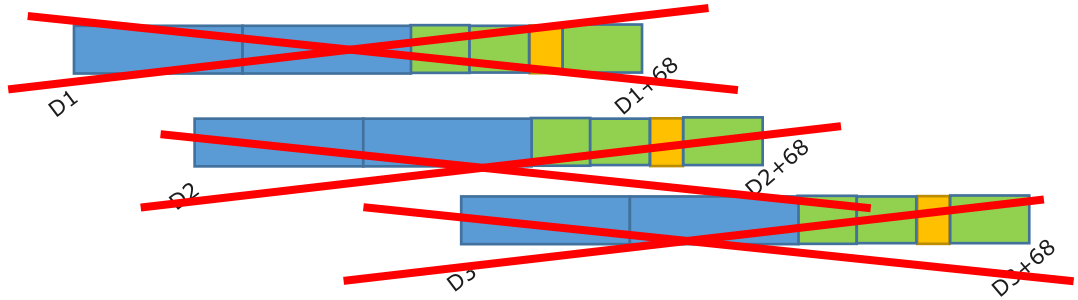


CLOVEK osoba;

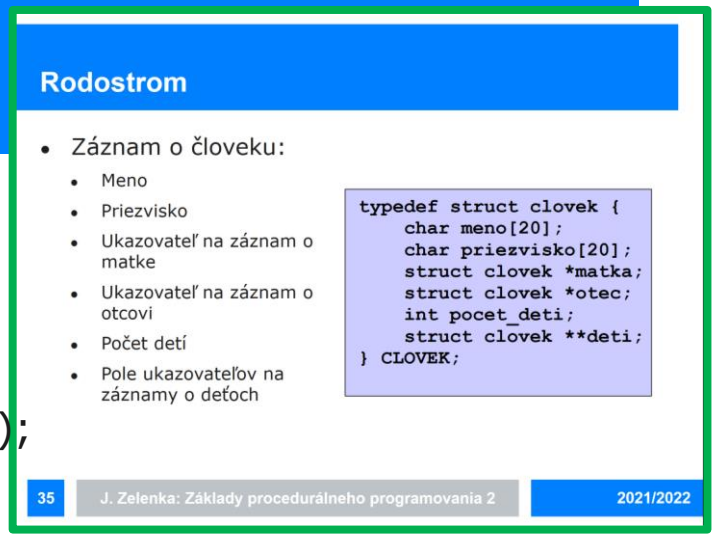
```
osoba.matka = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.otec = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.pocet_deti =3;
osoba.deti = (CLOVEK**)malloc(osoba.pocet_deti *sizeof(CLOVEK*));
for (i=0; i< osoba.pocet_deti; i++)
    osoba.deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
```




```
for (i=0; i<osoba.pocet_deti; i++)
    free(osoba.deti[i]);
```

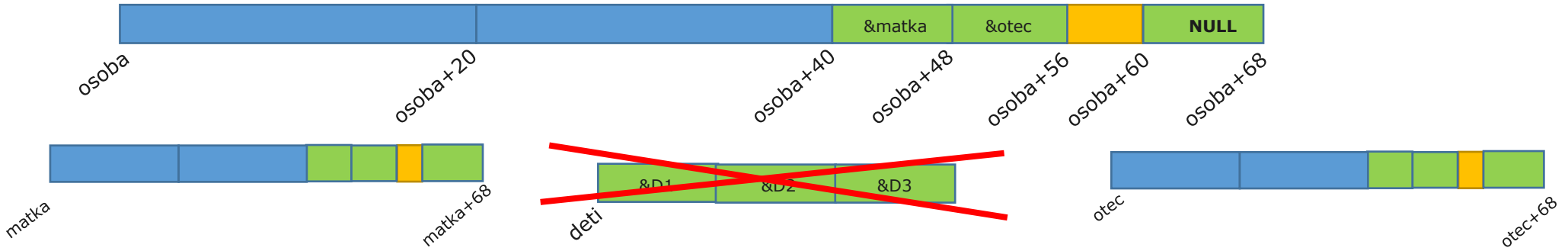


Rodostrom - uvolnenie pamäte



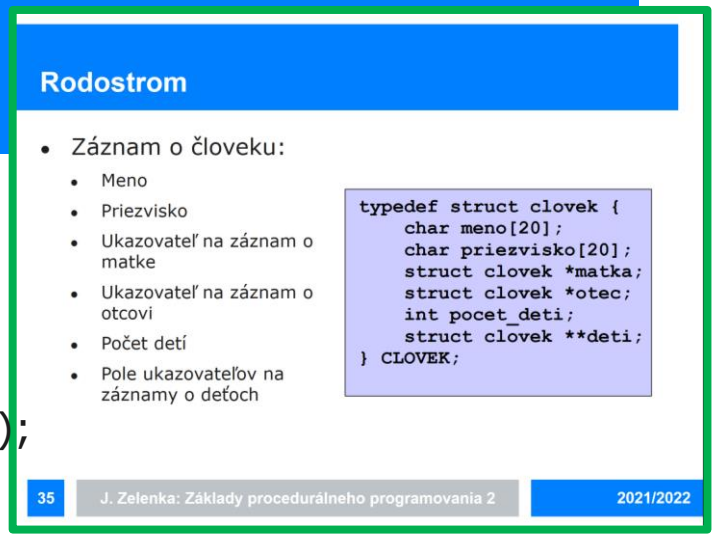
CLOVEK osoba;

```
osoba.matka = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.otec = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.pocet_deti =3;
osoba.deti = (CLOVEK**)malloc(osoba.pocet_deti *sizeof(CLOVEK*));
for (i=0; i< osoba.pocet_deti; i++)
     osoba.deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
```



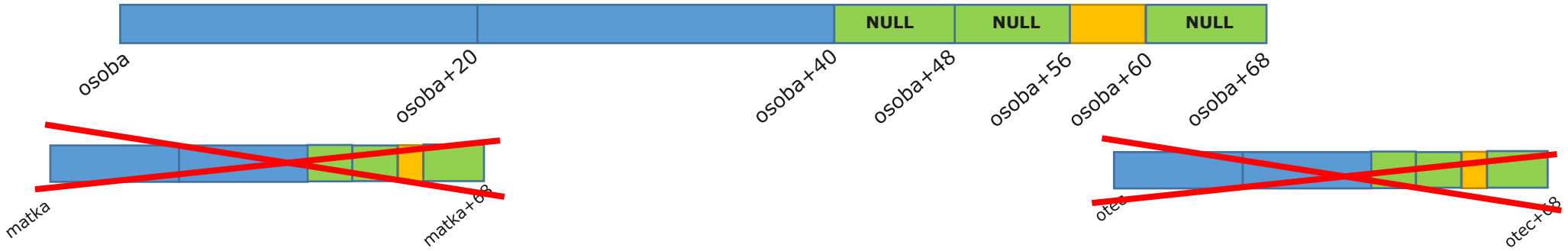

```
for (i=0; i<osoba.pocet_deti; i++)
     free(osoba.deti[i]);
free(osoba.deti);  osoba.deti = NULL; osoba.pocet_deti=0;
```

Rodostrom - uvolnenie pamäte



CLOVEK osoba;

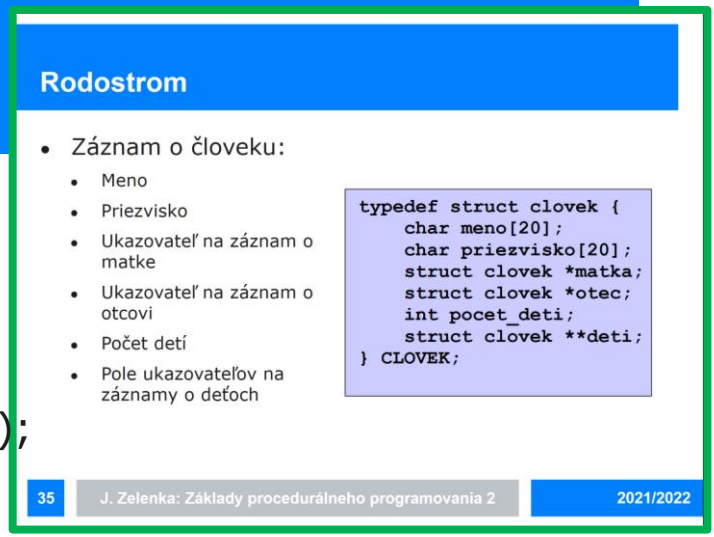
```
osoba.matka = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.otec = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.pocet_deti =3;
osoba.deti = (CLOVEK**)malloc(osoba.pocet_deti *sizeof(CLOVEK*));
for (i=0; i< osoba.pocet_deti; i++)
      osoba.deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
```




```
for (i=0; i<osoba.pocet_deti; i++)
      free(osoba.deti[i]);
free(osoba.deti);  osoba.deti = NULL; osoba.pocet_deti=0;
```

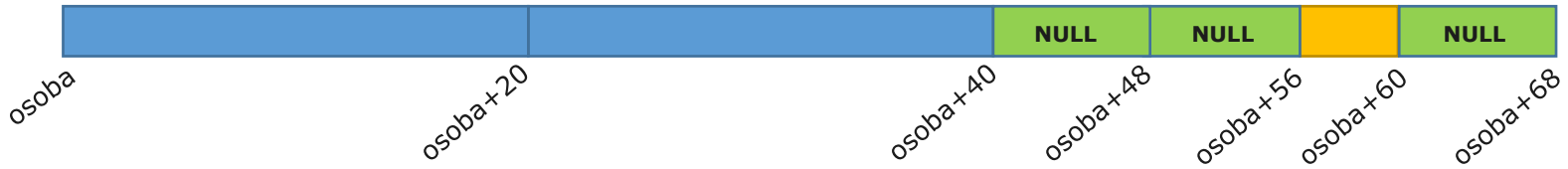
free(osoba.otec); osoba.otec = NULL; free(osoba.matka); osoba.matka = NULL;

Rodostrom - uvolnenie pamäte



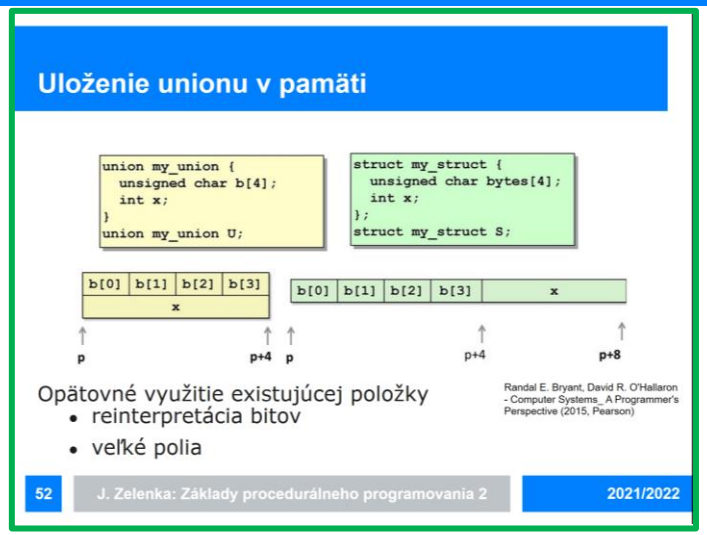
CLOVEK osoba;

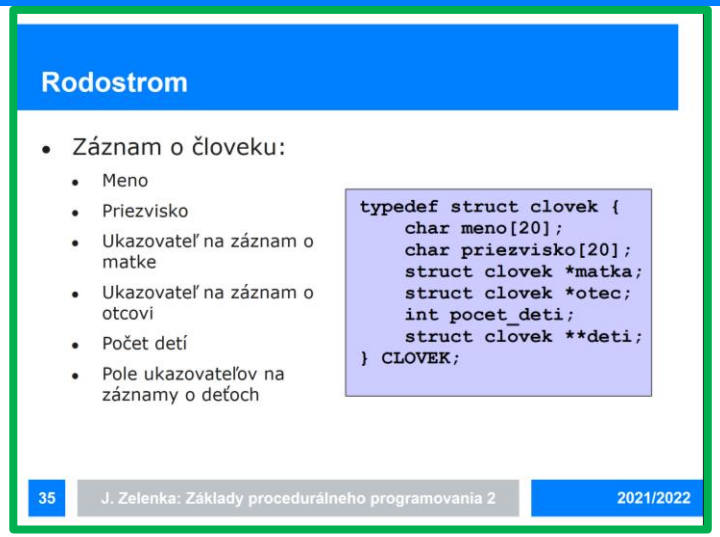
```
osoba.matka = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.otec = (CLOVEK*)malloc(sizeof(CLOVEK));
osoba.pocet_deti =3;
osoba.deti = (CLOVEK**)malloc(osoba.pocet_deti *sizeof(CLOVEK*));
for (i=0; i< osoba.pocet_deti; i++)
     osoba.deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
```

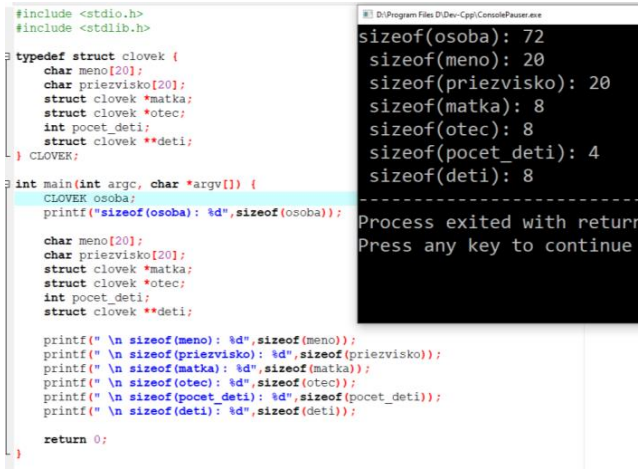



```
for (i=0; i<osoba.pocet_deti; i++)
     free(osoba.deti[i]);
free(osoba.deti);  osoba.deti = NULL; osoba.pocet_deti = 0;
free(osoba.otec);  osoba.otec = NULL;  free(osoba.matka);  osoba.matka = NULL;
```

Rodostrom - alokácia pamäte pole

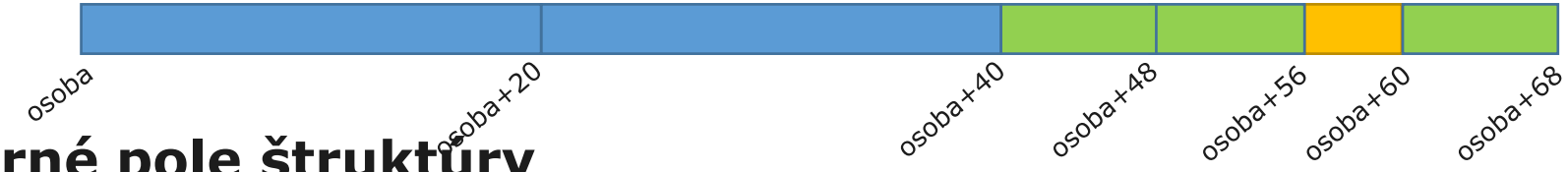






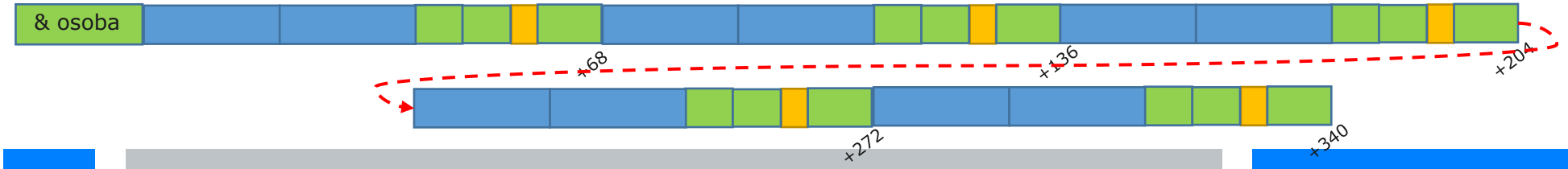
- **Statická alokácia**

  CLOVEK osoba;



- **Jednorozmerné pole štruktúry**

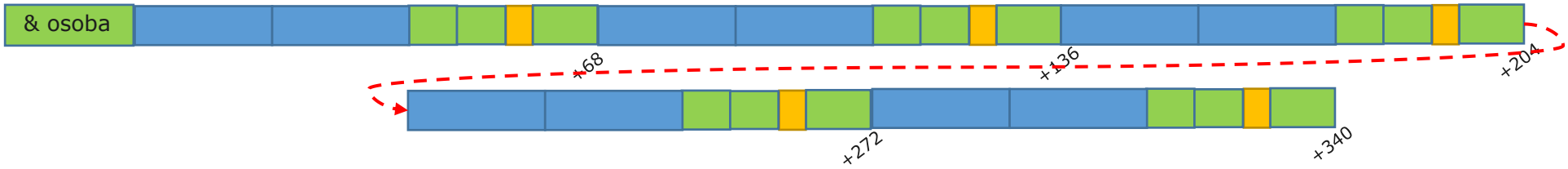
  CLOVEK osoba[5];

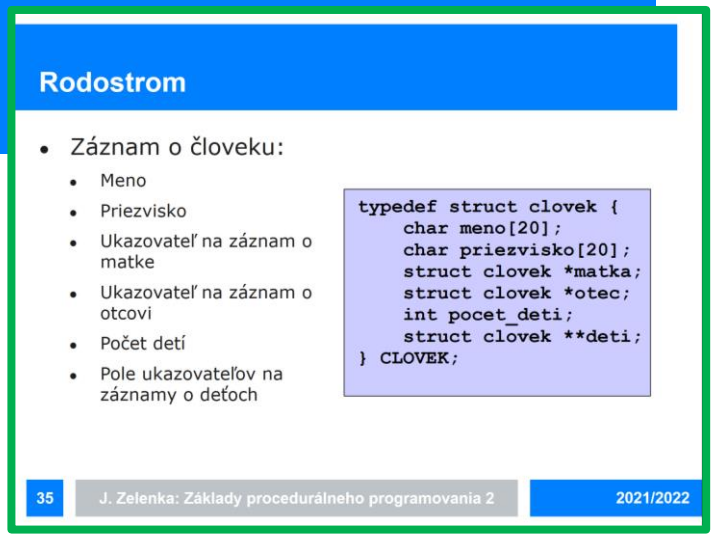


Rodostrom - alokácia pamäte pole

CLOVEK osoba[5];

```
for(j=0; j<5; j++){
    osoba[j].matka = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].otec = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].pocet_deti = 3;
    osoba[j].deti = (CLOVEK**)malloc(osoba[j].pocet_deti *sizeof(CLOVEK*));
    for (i=0; i< osoba[j].pocet_deti; i++)
        osoba[j].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
}
```

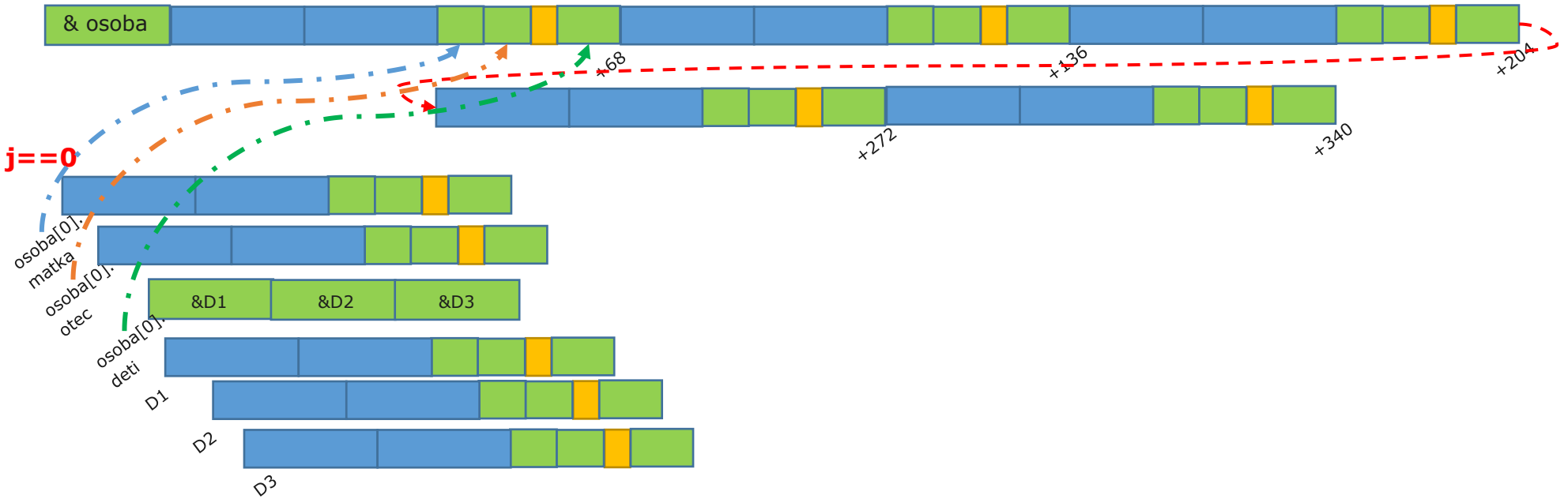




Rodostrom - alokácia pamäte pole

CLOVEK osoba[5];

```
for(j=0; j<5; j++){
    osoba[j].matka = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].otec = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].pocet_deti = 3;
    osoba[j].deti = (CLOVEK**)malloc(osoba[j].pocet_deti *sizeof(CLOVEK*));
    for (i=0; i< osoba[j].pocet_deti; i++)
        osoba[j].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
}
```



Rodostrom

- Záznam o človeku:
  - Meno
  - Priezvisko
  - Ukazovateľ na záznam o matke
  - Ukazovateľ na záznam o otcovi
  - Počet detí
  - Pole ukazovateľov na záznamy o deťoch

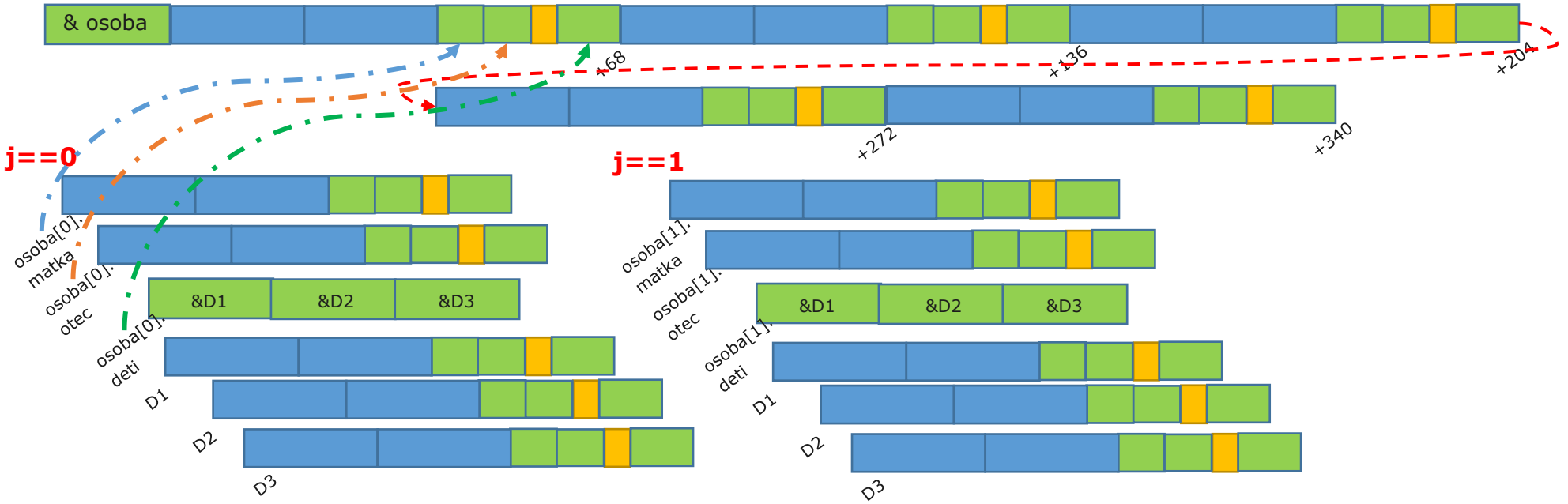
```
typedef struct clovek {
    char meno[20];
    char priezvisko[20];
    struct clovek *matka;
    struct clovek *otec;
    int pocet_deti;
    struct clovek **deti;
} CLOVEK;
```

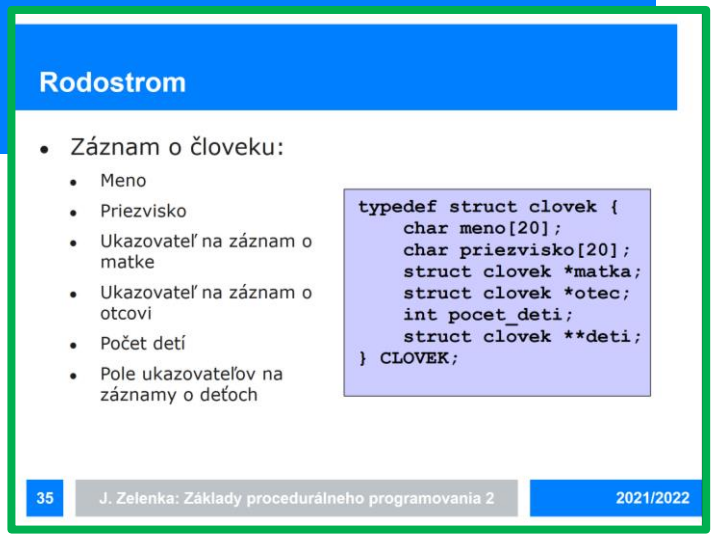
35 J. Zelenka: Základy procedurálneho programovania 2021/2022

Rodostrom - alokácia pamäte pole

CLOVEK osoba[5];

```
for(j=0; j<5; j++){
    osoba[j].matka = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].otec = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].pocet_deti = 3;
    osoba[j].deti = (CLOVEK**)malloc(osoba[j].pocet_deti *sizeof(CLOVEK*));
    for (i=0; i< osoba[j].pocet_deti; i++)
        osoba[j].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
}
```

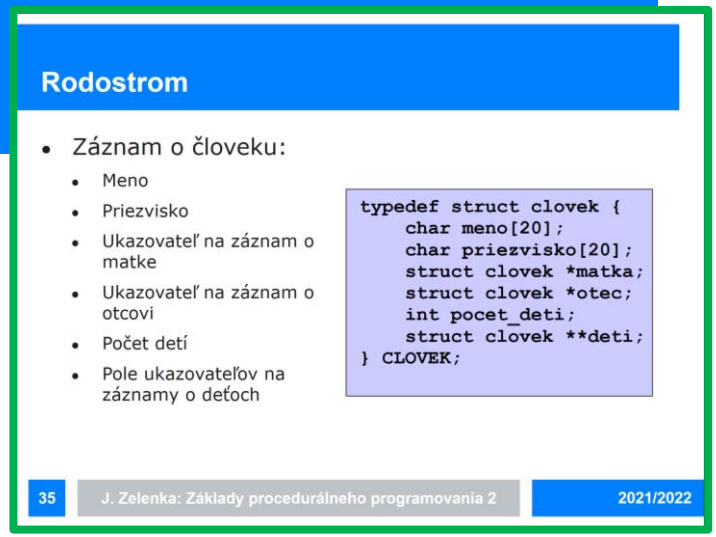


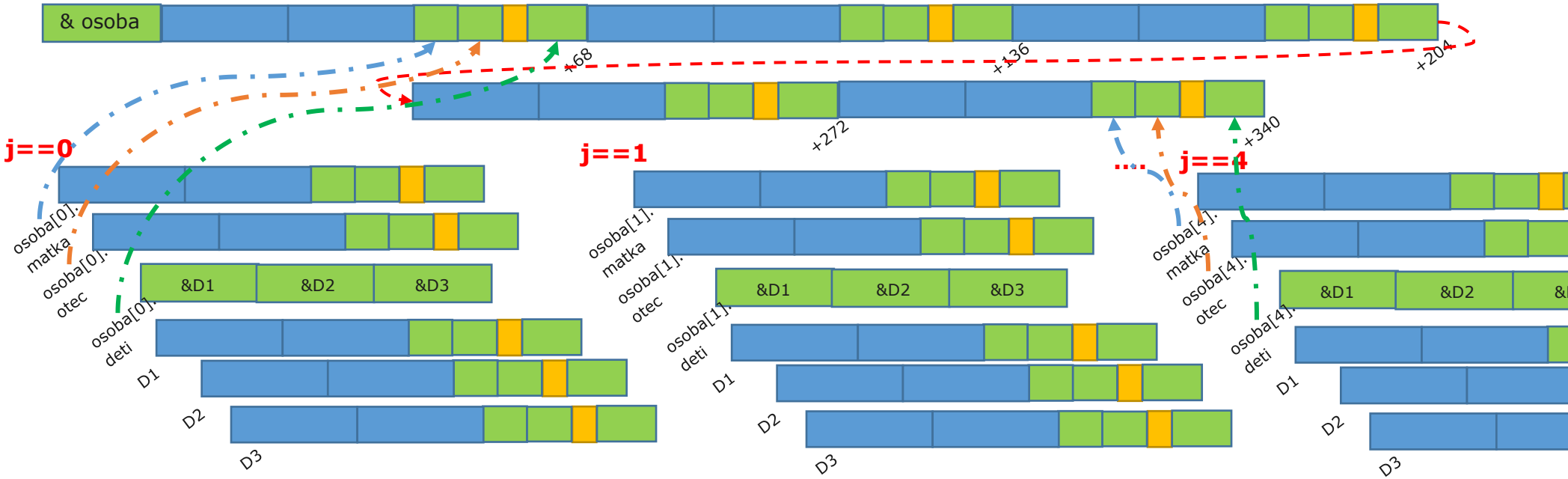


Rodostrom - alokácia pamäte pole

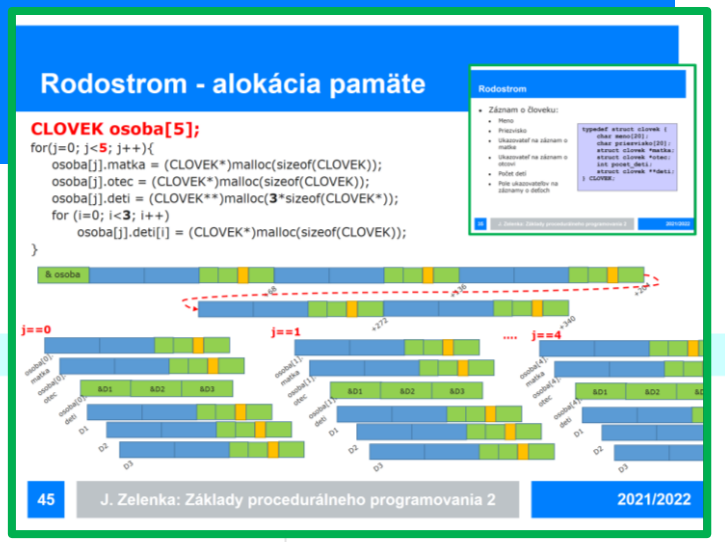
CLOVEK osoba[5];

```
for(j=0; j<5; j++){
    osoba[j].matka = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].otec = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].pocet_deti = 3;
    osoba[j].deti = (CLOVEK**)malloc(osoba[j].pocet_deti *sizeof(CLOVEK*));
    for (i=0; i< osoba[j].pocet_deti; i++)
        osoba[j].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
}
```





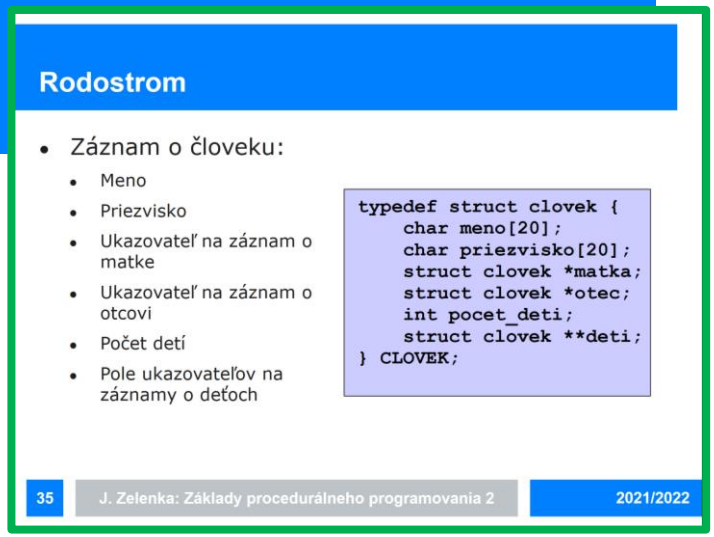
Rodostrom - alokácia pamäte pole



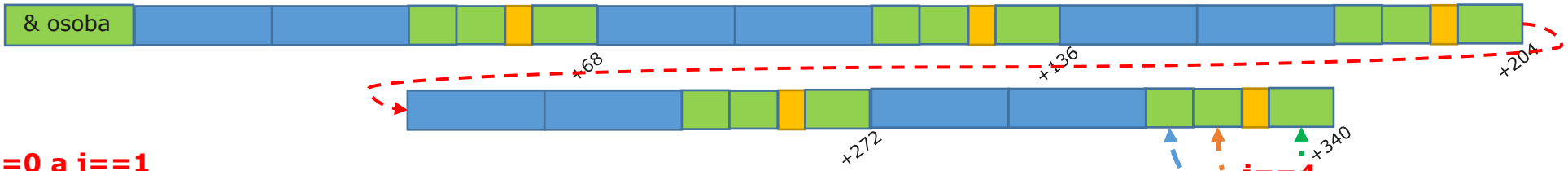

```
CLOVEK osoby[5];
strcpy(osoby[0].meno, "Jano");
strcpy(osoby[0].priezvisko, "Zelenka");
strcpy(osoby[1].meno, "Jozko");
strcpy(osoby[1].priezvisko, "Mrkvicka");

osoby[0].deti = (CLOVEK**)malloc(3*sizeof(CLOVEK*));
for (int i=0; i<3; i++)
       osoby[0].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));

for (int i=0; i<3; i++){
   strcpy(  osoby[0].deti[i]->meno, "detiJano1");
   strcpy(  osoby[0].deti[i]->priezvisko, "detiZelenka1");
}

for(int i=0; i<5; i++){
  printf("\n %d: %s %s",i+1, osoby[i].meno,osoby[i].priezvisko);
  if (i==0)
     for (int j=0; j<3; j++)
        printf("\n   %d.%d: %s %s",i+1,j+1, osoby[i].deti[j]->meno,osoby[i].deti[j]->priezvisko);
}
```

```
1: Jano Zelenka
  1.1: detiJano1 detiZelenka1
  1.2: detiJano1 detiZelenka1
  1.3: detiJano1 detiZelenka1
2: Jozko Mrkvicka
3: ☻
4: 4řd
5:
```

Rodostrom - uvolnenie pamäte pole

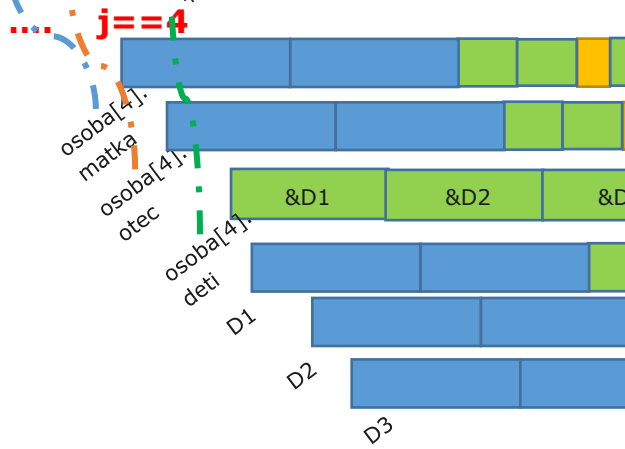



```
CLOVEK osoba[5];
for(j=0; j<5; j++){
    osoba[j].matka = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].otec = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].pocet_deti = 3;
    osoba[j].deti = (CLOVEK**)malloc(osoba[j].pocet_deti *sizeof(CLOVEK*));
    for (i=0; i< osoba[j].pocet_deti; i++)
        osoba[j].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
}
```

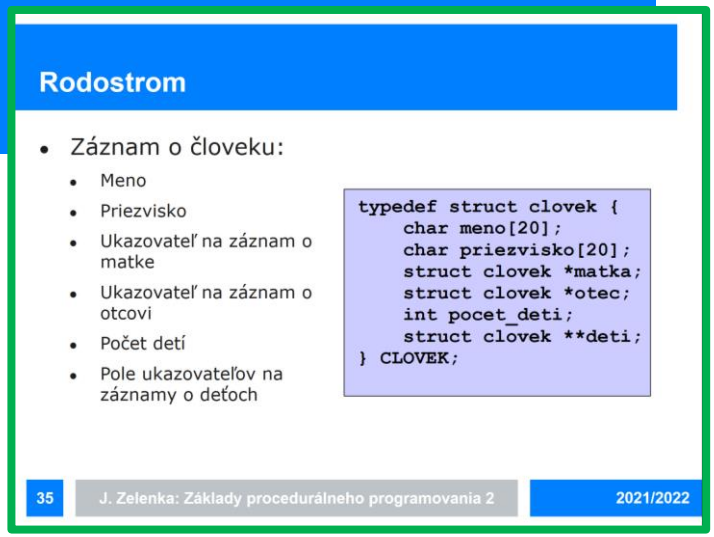


j==0 a j==1

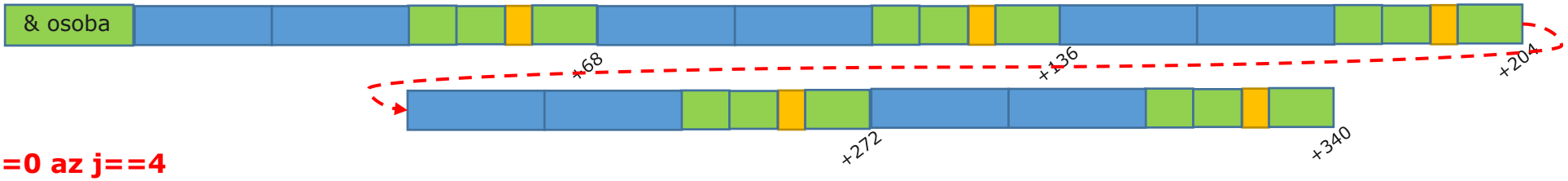
```
for(j=0; j<5; j++){
   for (i=0; i<pocet_deti; i++)  free(osoba[j].deti[i]);
   free(osoba[j].deti);  osoba[j].deti = NULL;
   pocet_deti = 0;
   free(osoba[j].otec);  osoba[j].otec = NULL;
   free(osoba[j].matka); osoba[j].matka = NULL;
}
```



Rodostrom - uvolnenie pamäte pole



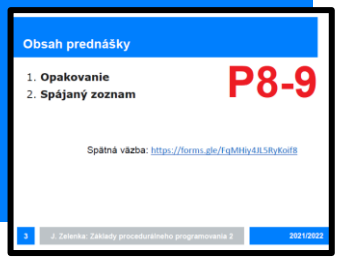

```
CLOVEK osoba[5];
for(j=0; j<5; j++){
    osoba[j].matka = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].otec = (CLOVEK*)malloc(sizeof(CLOVEK));
    osoba[j].pocet_deti = 3;
    osoba[j].deti = (CLOVEK**)malloc(osoba[j].pocet_deti *sizeof(CLOVEK*));
    for (i=0; i< osoba[j].pocet_deti; i++)
        osoba[j].deti[i] = (CLOVEK*)malloc(sizeof(CLOVEK));
}
```



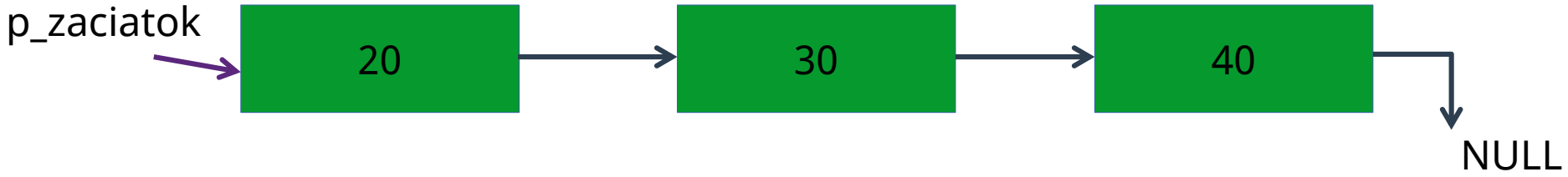
j==0 az j==4

```
for(j=0; j<5; j++){
  for (i=0; i<pocet_deti; i++)  free(osoba[j].deti[i]);
  free(osoba[j].deti);  osoba[j].deti = NULL;
  pocet_deti = 0;
  free(osoba[j].otec);  osoba[j].otec = NULL;
  free(osoba[j].matka); osoba[j].matka = NULL;
}
```

Spájaný zoznam



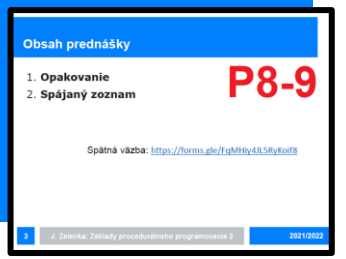
- **jednosmerný spajaný zoznam**



- **obojstranný spajaný zoznam**

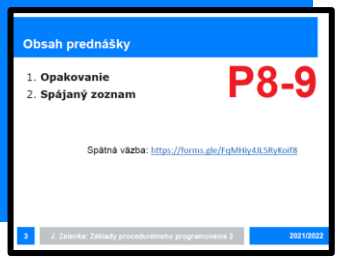
- **kruhový spajaný zoznam**

Spájaný zoznam a dynamické pole



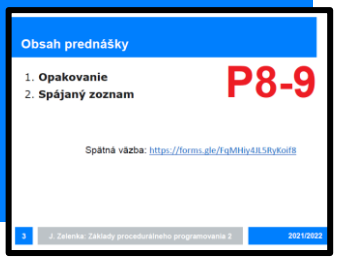
Spájaný zoznam	**Dynamické pole**
Dynamická veľkosť	Zväčšovanie je náročné
Vkladanie a mazanie je efektívne	Vkladanie a mazanie je neefektívne (zvyčajne treba posunúť prvky)
Nie je vhodný pre pristupovanie k prvkom podľa indexu (napr. triedenie)	**Prístup na i-ty prvok**
Pamäť je alokovaná dynamicky podľa potreby	Plytvanie pamäťou pri poloprázdnom poli

Výhody spájaného zoznamu



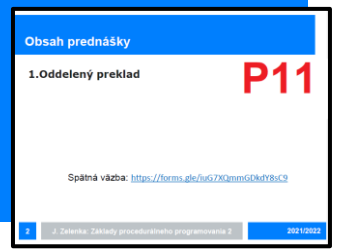
- Dynamická veľkosť/štruktúra
- Ľahké pridávanie a mazanie prvkov
  - vloženie nového prvku do poľa je drahé, pretože musíme vytvoriť miesto pre nový prvok, to znamená, že musíme posunúť existujúce prvky
    Napríklad majme pole, kde máme usporiadané ID-čka používateľov
    `id = [12, 28, 34, 50, 78, 91]`
    Ak chceme vložiť nového používateľa s ID 47 a chceme mať usporiadané pole, musíme posunúť všetky prvky za 34
- Mazanie je tiež drahé pri poliach
  -> čo treba spraviť pri mazaní prvku 28?

Nevýhody spájaného zoznamu

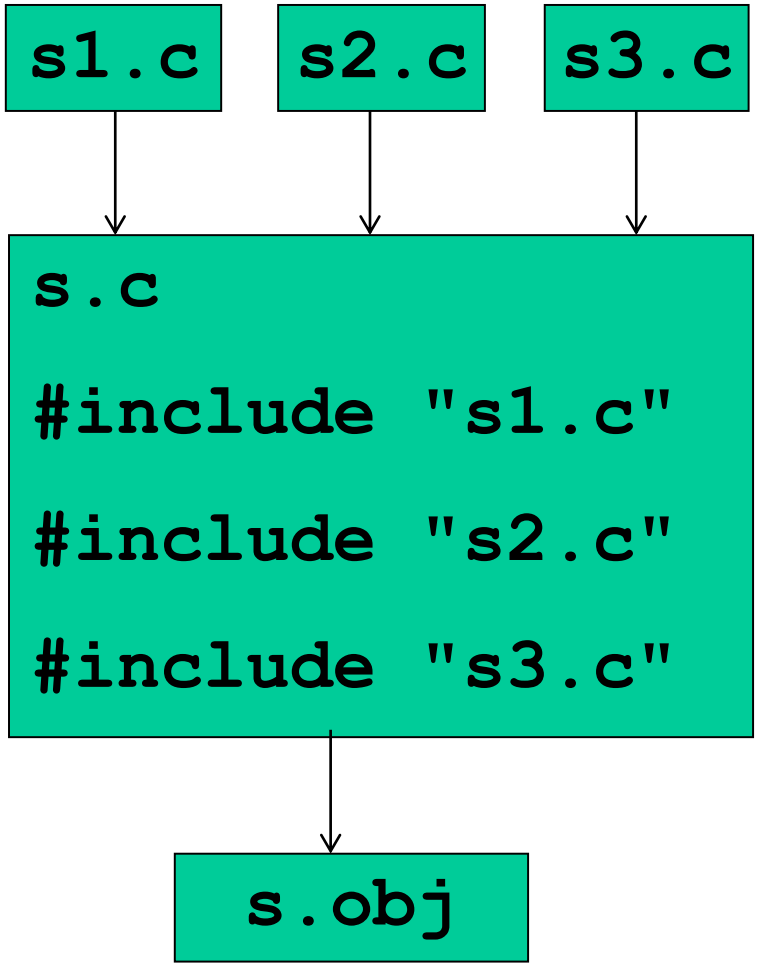


- prístup k i-temu prvku je drahý
  - potrebujeme prejsť sekvenčne cez všetky prvky zoznamu od jeho začiatku (pokročilé vyhľadávacie techniky napr. binárne vyhľadávanie sú časovo náročné, to isté triedenie)
- potrebujeme si navyše pamätať ukazovateľ pri každom prvku zoznamu

Príklad oddeleného prekladu

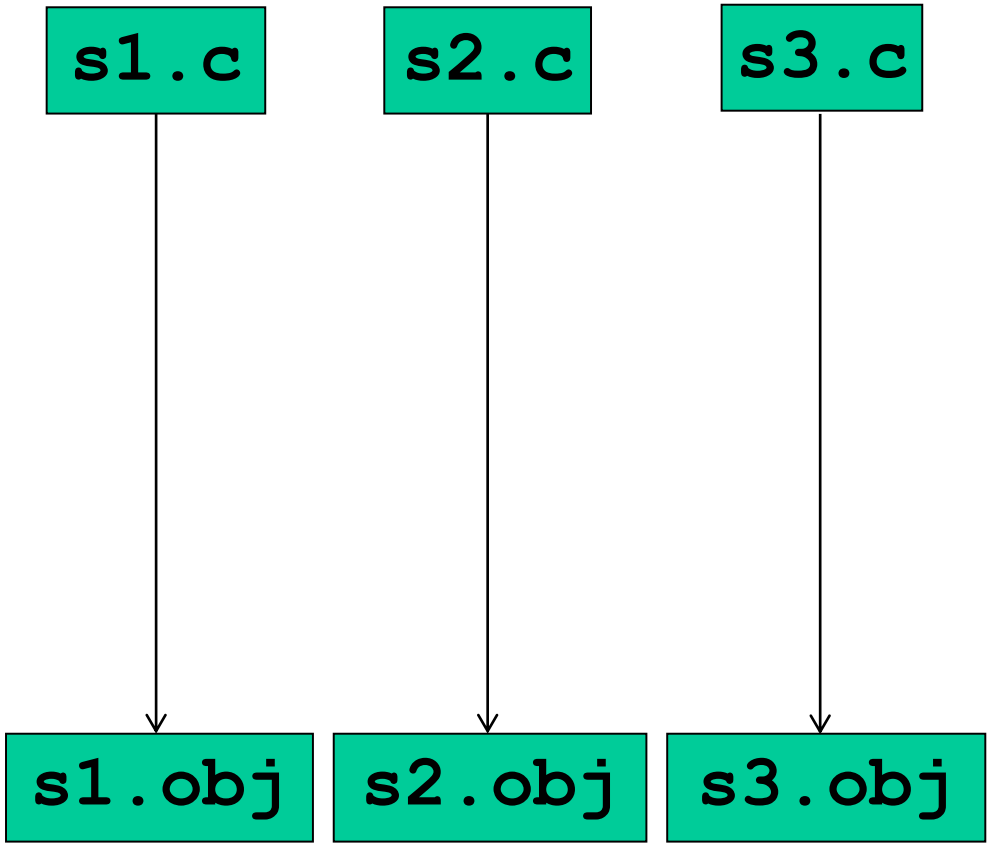


vkladanie súborov:



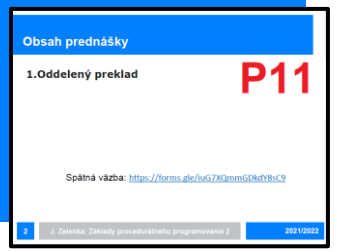
link s

oddelený preklad:



link s1, s2, s3

Oddelený preklad



- vkladanie súborov
  - **`#include`** → vznikne jeden .OBJ súbor
    - Vkladá sa jeden, alebo viac súborov
      - Nevýhody:
        - pri zmene v ľubovoľnom súbore musíme zbytočne prekladať aj všetky ostatné
        - medzi jednotlivými súbormi nie je možné skryť ich globálne identifikátory
- oddelený preklad
  - každý súbor sa preloží zvlášť (vznikne viac .OBJ súborov) a potom sa spoja pomocou linkeru
  - možnosť rozdelenia problému na viacero menších nezávislých častí, na ktorých pracuje súčasne viacero programátorov

Príklad: oddelený preklad

- program na výpočet obsahu a obvodu kruhu:

  `kruh_main.c, kruh_main.h, kruh_io.c, kruh_io.h`

	globálne premenné a funkcie	lokálne premenné a funkcie

`kruh_main`	`double obvod`
	`double vyp_priemer(double polomer)`
		`double polomer, obsah`
		`void vyp_obsah(double polomer)`
		`doble vyp_obvod()`
		`void vypocet(void)`

`kruh_io`	`double vstup_dat()` `double vystup_dat(double obsah)`	`double polomer`

premenné a funkcie definované v jednom z modulov - môžu sa používať aj v rámci druhého modulu

Oddelený preklad

kruh_main.h

```
extern double obvod;
extern double vyp_priemer(double polomer);
```

kruh_io.h

```
#define CHYBA_DAT -1

extern double vstup_dat();
extern double vystup_dat(double obsah);
```

kruh_main.c

```
#include <stdio.h>
#include "kruh_main.h"
#include "kruh_io.h"

double obvod;
#define pi 3.14;

static double polomer;
static double obsah;

static void vyp_obsah
            (double polomer);
static doble vyp_obvod();
static void vypocet(void);

void main() {
  polomer = vstup_dat();
  if (polomer == CHYBA_DAT)
    printf("Chybny polomer.");
  else {
    vypocet();
    vystup_dat(obsah);
  }
}

static void vyp_obsah(double
                      polomer)
{
  obsah = PI * polomer *polomer;
}

static doble vyp_obvod()
{
  return(PI*vyp_priemer(polomer));
}

static void vypocet(void)
{
  obvod = vyp_obvod();
  vyp_obsah(polomer);
}

double vyp_priemer(double polomer)
{
  return (2 * polomer);
}
```

kruh_io.c

```
#include <stdio.h>
#include "kruh_main.h"
#include "kruh_io.h"

#define kontrola(x) ((x >= 0.0) ? x : CHYBA_DAT)

static double polomer;

double vstup_dat(void) {
    printf("Zadaj polomer kruznice: ");
    scanf("%lf", &polomer);
    return kontrola(polomer);
}


void vystup_dat(double obsah) {
    double priemer;

    printf("Obsah kruhu s polomerom %6.2f je %.2f\n",
        polomer, obsah);
    priemer = vyp_priemer(polomer);
    printf("Obvod kruhu s polomerom %6.2f je %.2f\n",
        polomer, obvod);
}
```

Ako udržať poriadok vo veľkom programe

1. definície funkcií a premenných v súbore *program_1.c*
2. v *program_1.c* - striktne rozlíšené, čo sa bude používať v rámci súboru a čo mimo neho (čo najmenej)
3. všetky ostatné globálne premenné a funkcie označiť **`static`**
4. funkčné prototypy zdielaných funkcií a premenných - do *program_1.h*, označíme ich **`extern`**
5. ak poskytujeme aj symbolické konštanty - dáme ich len do *program_1.h*
6. *program_1.c* začína:

```
#include <stdio.h>
#include "program_1.h"
```

Ako udržať poriadok vo veľkom programe

7. súbor *program_2.c* obsahujúci ďalší modul programu a využíva premenné, funkcie alebo konštanty z modulu *program_1*, začína:

zdielané funkcie a premenné
deklarácia vlastného súboru

```
#include <stdio.h>
#include "program_1.h"
#include "program_2.h"
```

Odporučený obsah .c súborov

slido slido.com
#1021046
PrPr – P12

1. dokumentačná časť

   (meno súboru, verzie, stručný popis modulu, meno autora a dátum)

2. všetky potrebné **`#include`**

   len súbory **.h**, nie **.c**!

   najprv systémové **< >** a potom vlastné **" "**

3. deklarácie importovaných objektov

   len ak nie sú v **.h** súbore spolupracujúceho modulu (čomu dávame prednosť) - v iných moduloch sú tieto objetky bez špecifikovanej triedy

4. definície globálnych premenných

   premenné definované mimo funkcií zdielané inými modulmi (čo najmenej!)

Odporučený obsah .c súborov

5. lokálne **`#define`**

   definícia konštánt a makier len pre aktuálny modul (ak veľmi rozsiahle - do zvlášť **.h**)

6. definície lokálnych typov

   `typedef` len pre aktuálny modul (ak veľmi rozsiahle - do zvlášť **.h**)

7. definície premenných len pre aktuálny modul

   globálne len pre aktuálny modul - **`static`**

8. úplné funkčné prototypy funkcií len pre aktuálny modul
9. funkcia **`main()`**
10. definície funkcií aj pre iné moduly
11. definície funkcií len pre aktuálny modul

Odporučený obsah .h súborov

1. dokumentačná časť
   (meno súboru, verzie, stručný popis modulu, meno autora a dátum)
2. definície symbolických konštánt ktoré sa budú používať aj v iných moduloch
3. definície makier s parametrami, ktoré sa budú používať aj v iných moduloch
4. definície typov, ktoré sa budú používať aj v iných moduloch
5. deklarácie premenných aktuálneho modulu, ktoré sa budú používať v iných moduloch (tu označené **`extern`**)
6. funkčné prototypy funkcií aktuálneho modulu, ktoré sa budú používať v iných moduloch (tu označené **`extern`**)

Ďakujem vám za pozornosť!

Spätná väzba: *https://forms.gle/6q5D2G6UwrtimXEx9*