Algoritmizácia a programovanie: 9. prednáška

Ján Grman

Obsah

- Štruktúry a spájané zoznamy
- Bitové operácie

Štruktúry

Čo je to štruktúra?

- štruktúra (struct) je heterogénny dátový typ
  - heterogénny: je zložený z prvkov rozličných dátových typov (pole je homogénny dátový typ)

```
struct {
    položka_1;
    ...
    položka_n;
}
```

dá sa definovať piatimi rôznymi spôsobmi

Definícia štruktúry (1)

- základný spôsob
  - štruktúra nie je pomenovaná,
  - nedá sa inde v progame použiť
  - dajú sa použiť len definované premenné

```
struct {
    int vyska;
    float vaha;
} pavol, jan, karol;
```

Definícia štruktúry (2)

- modifikácia základného spôsobu
  - štruktúra je pomenovaná,
  - dá sa využiť aj inde v programe

```
struct miery {
    int vyska;
    float vaha;
} pavol, jan, karol;
```

Definícia štruktúry (3)

- podobne ako predchádzajúci spôsob
  - definícia štruktúry a premenných je oddelená od definície premenných (ktoré sa môžu robiť viackrát)

```
struct miery {
    int vyska;
    float vaha;
};
struct miery pavol;
struct miery jan, karol;
```

Definícia štruktúry (4)

- definícia nového typu (typedef)
  - štruktúra nie je pomenovaná, pomenovaný je typ
  - typ sa dá použiť na definíciu premenných, pretypovanie...

```
typedef struct {
    int vyska;
    float vaha;
} MIERY;
MIERY pavol, jan, karol;
```

nebolo použité "**struct**"

Definícia štruktúry (5)

- modifikácia predchádzajúceho spôsobu
  - štruktúry aj typ je pomenovaná (v tomto prípade to nie je potrebné, ale neskôr to budeme potrebovať)

```
typedef struct miery {
    int vyska;
    float vaha;
} MIERY;
MIERY pavol, jan, karol;
```

odporúča sa pomenovať typ aj štruktúru rovnako, odlíšiť ich len veľkosťou písma

Prístup k položkám štruktúry

- bodková notácia

```
typedef struct {
    int vyska;
    float vaha;
} MIERY;
MIERY pavol, jan, karol;

pavol.vyska = 182;
karol.vaha = 62.5;
jan.vyska = pavol.vyska;
```

často sa používa pole štruktúr:

```
MIERY ludia[100];

ludia[50].vyska = 156;

ludia[0] = ludia[50];
```

v ANSI C sa dá urobiť

Pole v štruktúre

```
typedef struct {
    int pole[10];
} STR_POLE;

void main()
{
    STR_POLE a, b;
    a.pole[0] = 5;
    b = a;
}
```

takto sa dá použiť štruktúra na to, aby sa dalo naraz skopírovať celé pole

Štruktúry a ukazovatele

Štruktúry a ukazovatele

- použitie:
  - štruktúra v dynamickej pamäti
  - štruktúra vo funkcii

```
typedef struct {
    char meno[30];
    int rocnik;
} STUDENT;

STUDENT s, *p_s;
```

***p_s** - ukazovateľ na štruktúru, nemá pridelené miesto v pamäti, preto je potrebné alokovať mu pamäť.

```
p_s = (STUDENT *) malloc(sizeof(STUDENT));
```

alebo nastaviť ukazovateľ na inú pamäť

```
p_s = &s;
```

Štruktúry a ukazovatele

```
typedef struct {
    char meno[30];
    int rocnik;
} STUDENT, *P_STUDENT;
```

definícia typu ukzovateľa na štruktúru

```
STUDENT s;
P_STUDENT  p_s;
p_s = (P_STUDENT *) malloc(sizeof(STUDENT));
```

```
s.rocnik = 2;
(*p_s).rocnik = 3;
p_s->rocnik = 4;
```

prístup k štruktúre pomocou ukazovateľa

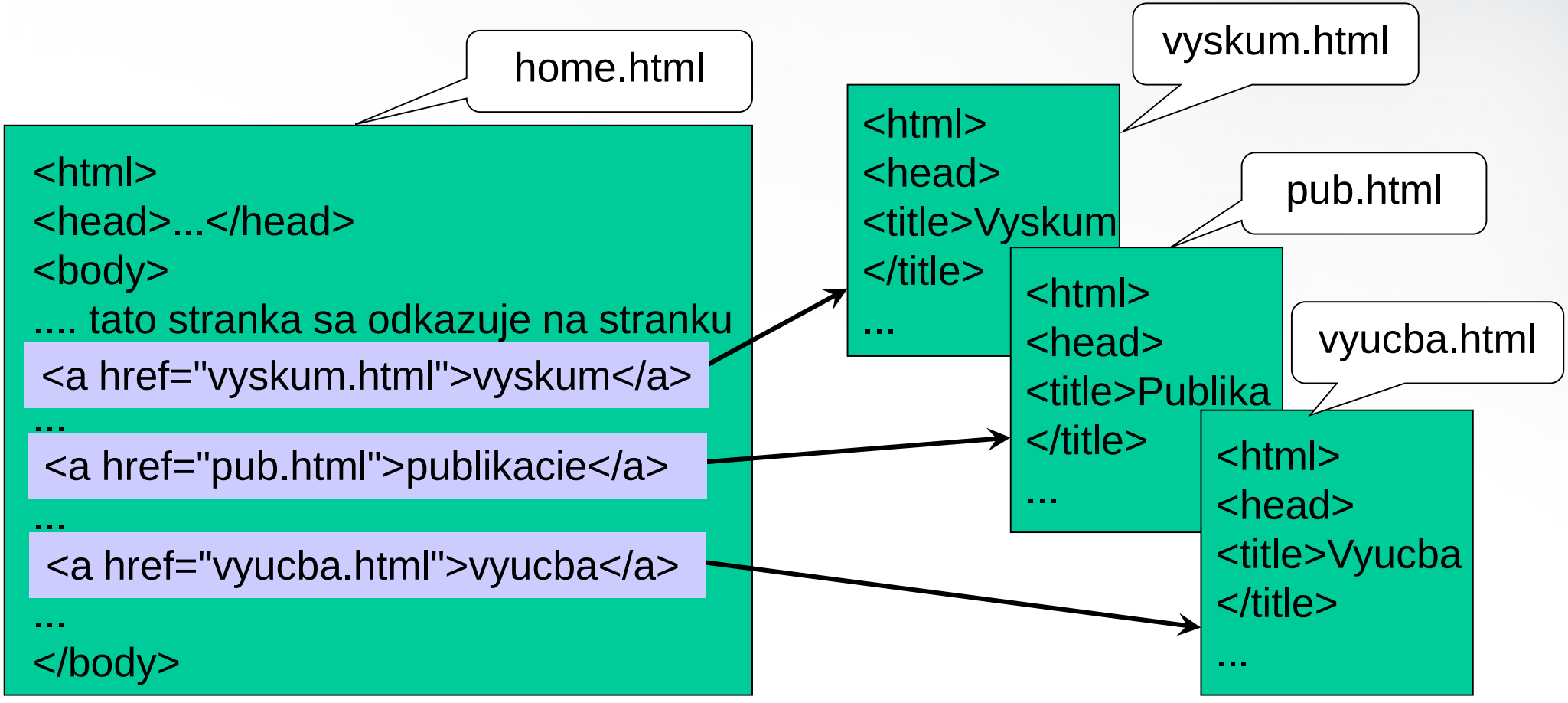
Prístup do štruktúry pomocou ukazovateľa

```
STUDENT s, *p_s;

s.rocnik = 2;
(*p_s).rocnik = 3;
 p_s->rocnik = 5;
```

chybné, lebo **.** má väčšiu prioritu ako *, teda tam, kam ukazuje **p_s.rocnik** (adresa 3) priraď hodnotu 4

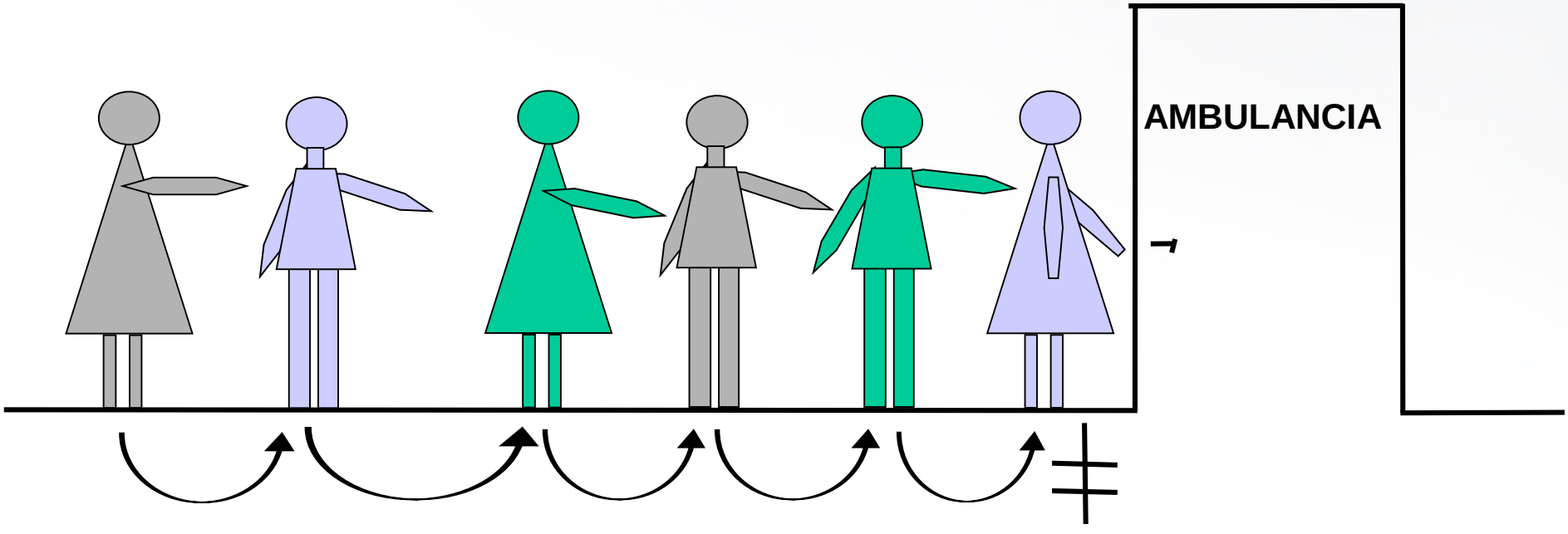
Štruktúry ukazujúce samy na seba: príklad 1

- hypertext
  - web stránka má odkaz na web stránku (ukazovateľ na rovnaký typ)



Štruktúry ukazujúce samy na seba: príklad 2

- pacienti v čakárni u lekára
  - "Kto je posledný?"
  - každý človek si pamätá človeka, ktorý je pred ním (ukazovateľ na ten istý typ)



Štruktúry ukazujúce samy na seba

```
typedef struct polozka {
    int hodnota;
    struct polozka *p_dalsi;
} POLOZKA;
```

odkaz na samého seba (na takú istú štruktúru)

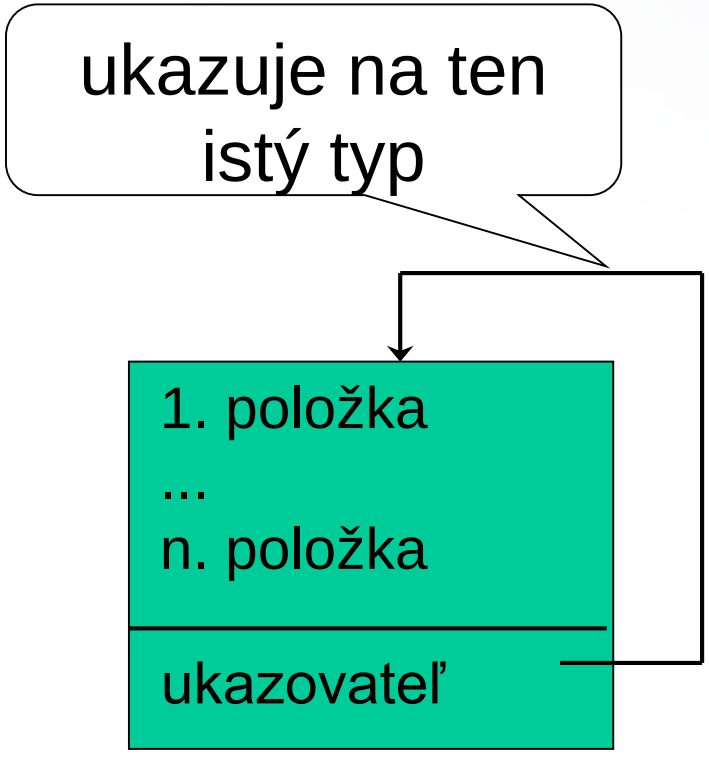
aj štruktúra, aj typ musia byť pomenované

```
typedef struct {
    int hodnota;
    struct POLOZKA *p_dalsi;
} POLOZKA;
```

chyba: v čase, keď sa definujem **p_dalsi**, **POLOZKA** ešte nie je známa

Spájaný zoznam

- dynanamický zoznam prvkov :
  - v pamäti je práve toľko prvkov, koľko je potreba
  - dá sa pridávať na ktorékoľvek miesto v zozname

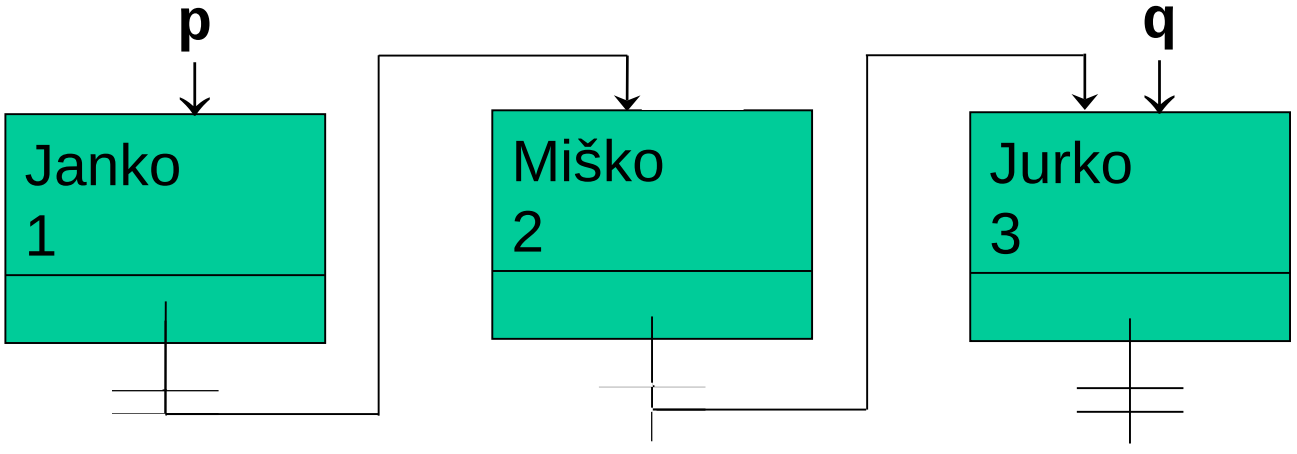



```
typedef struct clovek {
    char meno[30];
    int rocnik;
    struct clovek *dalsi;
} CLOVEK;
```

Spájaný zoznam

```
typedef struct clovek {
     char meno[30];
     int rocnik;
     struct clovek *dalsi;
} CLOVEK;

CLOVEK *p, *q;
```

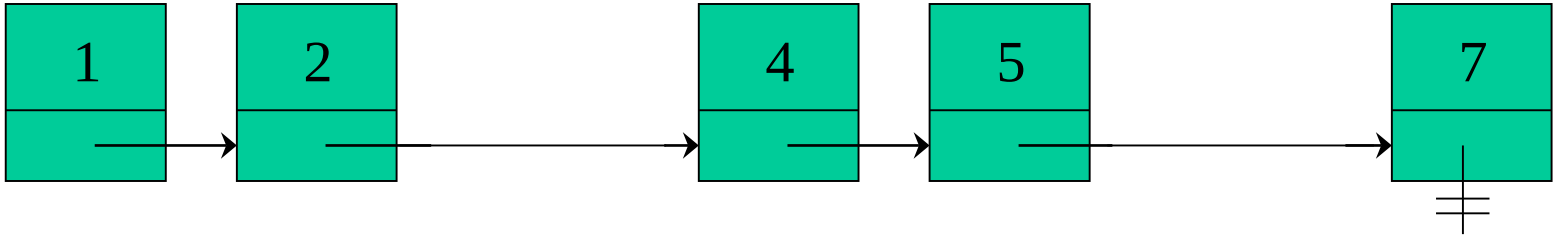
```
q = (CLOVEK *) malloc(sizeof(CLOVEK));
q->meno = Jurko;
q->rocnik =  3;
q->dalsi = NULL;

p->dalsi->dalsi = q;
```



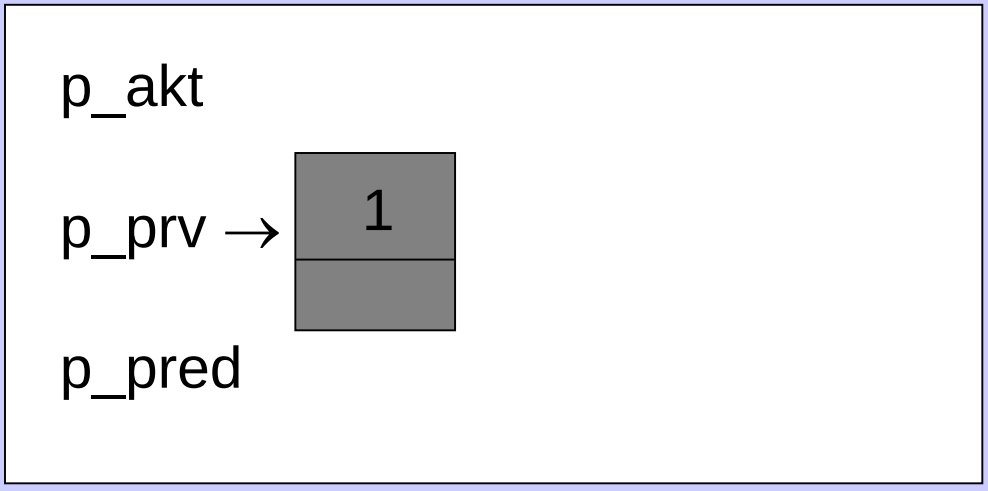
Spájaný zoznam: príklad

- Vytvorí sa spájaný zoznam s hodnotami 1...n, potom sa vymažú všetky prvky, ktoré sú deliteľné 3.

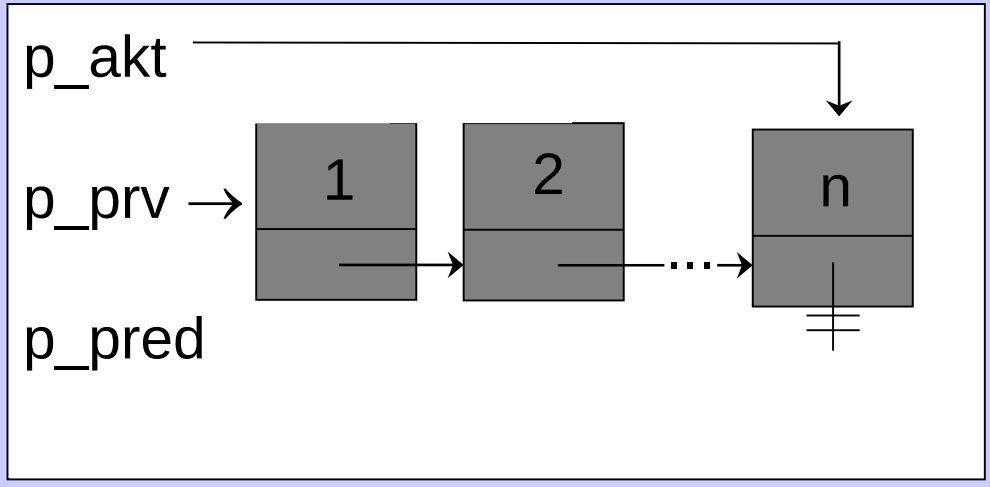
```
typedef struct prvok {
    int hodnota;
    struct prvok *dalsi;
} PRVOK;
```



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

typedef struct prvok {
    int hodnota;
    struct prvok *dalsi;
} PRVOK;

void main()
{
    int i, n;
    PRVOK *p_prv, *p_akt, *p_pred;

    printf("Zadaj pocet prvkov zoznamu: ");
    scanf("%d", &n);

    if((p_prv = (PRVOK *) malloc(sizeof(PRVOK))) == NULL){
        printf("Malo pamate.\n")
        return;
    }
    p_prv->hodnota = 1;
```



```
    p_akt = p_prv;
    for(i = 2; i < n; i++) {
        if(p_akt->dalsi =
                (PRVOK *) malloc(sizeof(PRVOK))) == NULL) {
            printf("Malo pamate.\n")
            break;
        }
        p_akt = p_akt->dalsi;
        p_akt->hodnota = i;
    }
    p_akt->dalsi = NULL;

    for(p_pred = p_akt = p_prv;
              p_akt != NULL;
              p_pred = p_akt, p_akt = p_akt->dalsi) {
        if(p_akt->hodnota % 3 == 0) {
            p_pred->dalsi = p_akt->dalsi;
            free((void *) p_akt);
            p_akt = p_pred;
        }
    }
}
```



Štruktúra v inej štruktúre

- štruktúra, ktorá je v inej štruktúre musí byť definovaná skôr ako je do inej štruktúry uložená (vhniezdená štruktúra)
  - predtým sme mali len odkaz na štruktúru

```
typedef struct {
    char ulica[30];
    int cislo;
} ADRESA;
```

```
typedef struct  {
    char meno[30];
    ADRESA adresa;
    float plat;
} OSOBA;
```

príklad: vypísať adresu zamestnanca s najvyšším platom (**`OSOBA ludia[100]`**)

Príklad: štruktúra v inej štruktúre

```
typedef struct {
    char ulica[30];
    int cislo;
} ADRESA;

typedef struct  {
    char meno[30];
    ADRESA adresa;
    float plat;
} OSOBA;
```

```
int i, kto = 0;
float max = 0.0, pom;
OSOBA ludia[100];

... /* inicializacia */

for (i = 0; i < 100; i++) {
    if((pom = ludia[i].plat) > max) {
        max = pom;
        kto = i;
    }
}
```

```
printf("Zamestnanec s najvyssim platom byva: %s %d",
ludia[kto].adresa.ulica, ludia[kto].adresa.cislo);
```

Príklad: štruktúra v inej štruktúre - cez ukazovateľe

```
typedef struct {
    char ulica[30];
    int cislo;
} ADRESA;

typedef struct  {
    char meno[30];
    ADRESA adresa;
    float plat;
} OSOBA;
```

```
float max = ludia[0].plat, pom;
OSOBA ludia[100], *p_kto, *p_pom;

... /* inicializacia */

for (p_pom = p_kto = ludia;
       p_pom < ludia + 100;
       p_pom++) {
    if ((p_pom->plat) > max) {
        p_kto = p_pom;
        max = p_pom->plat;
    }
}
```

```
printf("Zamestnanec s najvyssim platom byva: %s %d",
p_kto->adresa.ulica, p_kto->adresa.cislo);
```

Alokácia pamäte pre jednotlivé položky štruktúry

```
typedef struct {
    char c;
    int i, j, k;
    char d;
} POKUS;

POKUS p;
```

- položky obsadzujú pamäť zhora dole a zľava doprava
  - pre **p** v poradí: **c i j k d**
- položky sú väčšinou zarovnávané na párne adresy
  - za položkou **c** je väčšinou 1 prázdny Byte
- štruktúra väčšinou končí na párnej adrese
  - za položkou **d** - 1 volný Byte

na zistenie veľkosti: **sizeof(p);**

Štruktúry a funkcie

- K&R verzia jazyka C:
  - parameter: ukazovateľ na štruktúru
  - návratový typ: ukazovateľ na štruktúru
- ANSI C:
  - parameter: ukazovateľ na štruktúru aj samotná štruktúra
  - návratový typ: ukazovateľ na štruktúru aj samotná štruktúra

Príklad: štruktúry a funkcie

- sčítanie komplexných čísel

```
typedef struct {
    double re, im;
} KOMP;

KOMP sucet(KOMP a, KOMP b)
{
    KOMP c;

    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}
```

```
void main()
{
    KOMP x, y, z;
    x.re = 1.4; x.im = 3.2;
    y = x;
    z = sucet(x, y);
}
```

ak sú štruktúry veľké, nevýhodné, lebo sa vytvárajú lokálne kópie (časovo aj pamäťovo náročné) ⇒ vhodné používať ukazovatele

Príklad: štruktúry a funkcie - pomocou ukazovateľov

- sčítanie komplexných čísel

```
typedef struct {
    double re, im;
} KOMP;

void sucet(KOMP *a, KOMP *b, KOMP *c)
{
    c->re = a->re + b->re;
    c->im = a->im + b->im;
}
```

```
void main()
{
    KOMP x, y, z;
    x.re = 1.4; x.im = 3.2;
    y = x;
    sucet(&x, &y, &z);
}
```

Príklad: dynamické vytváranie štruktúry vo funkciách

vracia štruktúru ako ukazovateľ

```
typedef struct {
    char meno[30];
    int rocnik;
} STUDENT;
```

```
STUDENT *vytvor1(void)
{
    STUDENT p;
    p = (STUDENT *) malloc(sizeof(STUDENT));
    if (p = NULL)
        printf("Malo pamate.\n");
    return p;
}
```

vracia štruktúru cez parameter

```
void vytvor2(STUDENT **p)
{
    p = (STUDENT *) malloc(sizeof(STUDENT));
    if (p = NULL)
        printf("Malo pamate.\n");
}
```

Príklad: dynamické vytváranie štruktúry vo funkciách

```
void nastav(STUDENT *p, char *meno, int rok)
{
    p->rocnik = rok;
    strcpy(p->meno, meno);
}
```

```
void main()
{
    STUDENT  s, *p_s1, *p_s2;
    p_s1 = vytvor1();
    vytvor2(&p_s2);
    s.rocnik = p_s1->rocnik = 1;
    nastav(&s, "Martin", 1);
    nastav(p_s1, "Peter", 2);
    nastav(p_s2, "Michal", 3);
}
```

Uniony

- dátový typ
  - vyhradí sa pamäť pre najväčšiu položku
  - všetky položky sa prekrývajú

```
typedef union {
    char c;
    int i;
    float f;
} ZIF;
ZIF a, *p_a = &a;
```

vyhradí sa pamäť o veľkosti najväčšieho prvku

```
a.c = '#';
p_a->i = 1;
a.f = 2.3;
```

premazávajú sa hodnoty

Union neposkytuje informáciu o typu prvku, ktorý bol naposledy do neho uložený!

Union vložený do štruktúry

```
typedef enum {
     ZNAK, CELE, REALNE
} TYP;

typedef union {
     char c;
     int i;
     float f;
} ZIF;

typedef struct {
     TYP typ;
     ZIF polozka;
} ZN_INT_FL;
```

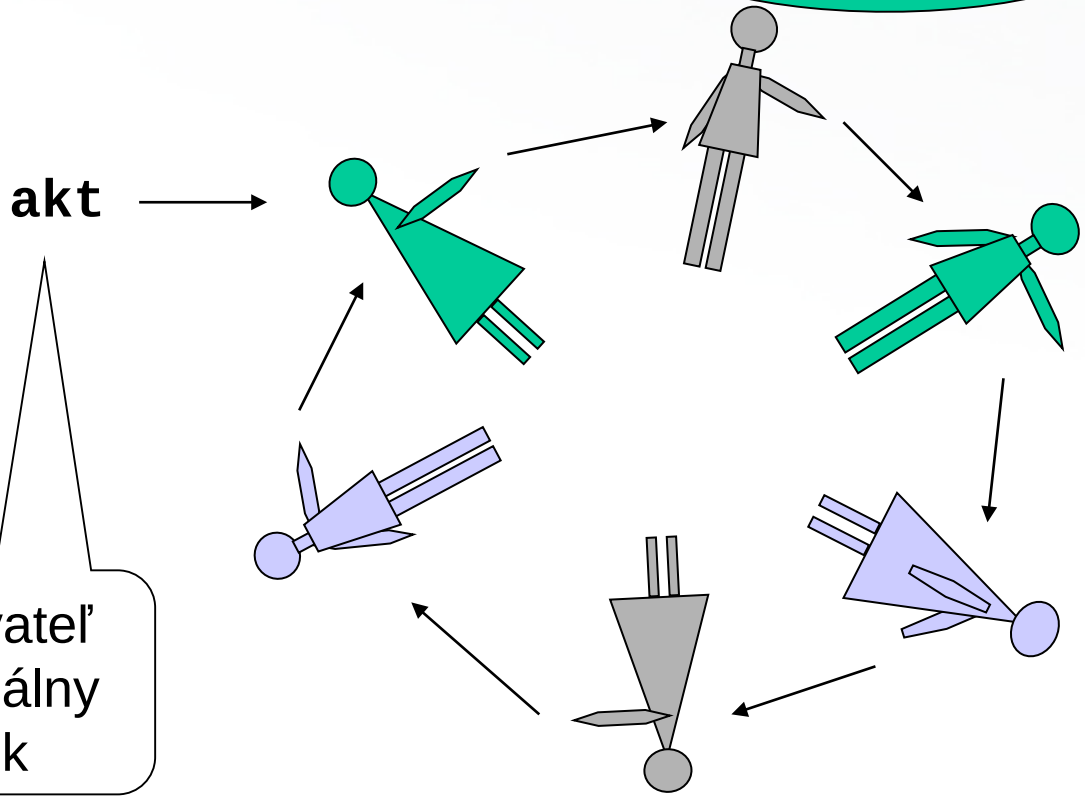
vymenovaný typ: slúži na rozlíšenie typov

union: umožňuje uchovávať znak, celé a reálne číslo

štruktúra: obsahuje informáciu o type položky a samotnú položku

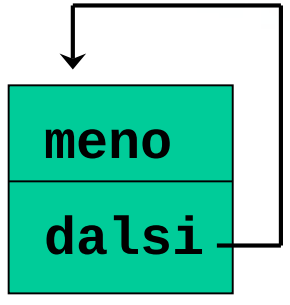
Príklad 1: Kruhový zoznam

"Simulácia hry": záznamy o deťoch, ktoré hrajú "*Kolo, kolo mlynské*" a ešte nevypadli...



ukazovateľ na aktuálny prvok

štruktúra prvku



Kruhový zoznam: prázdny zoznam

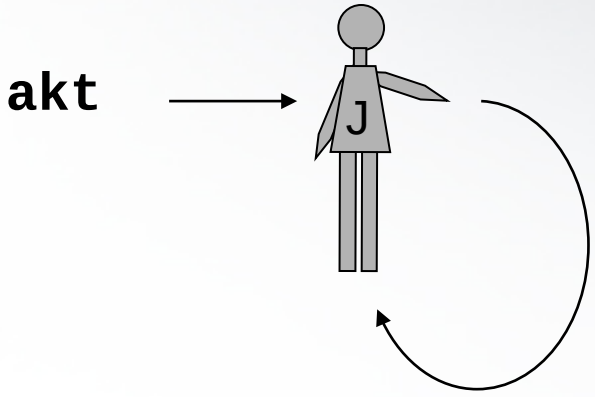
akt

na začiatku je zoznam prázdny (**`akt = NULL`**), musíme ho naplniť záznamami

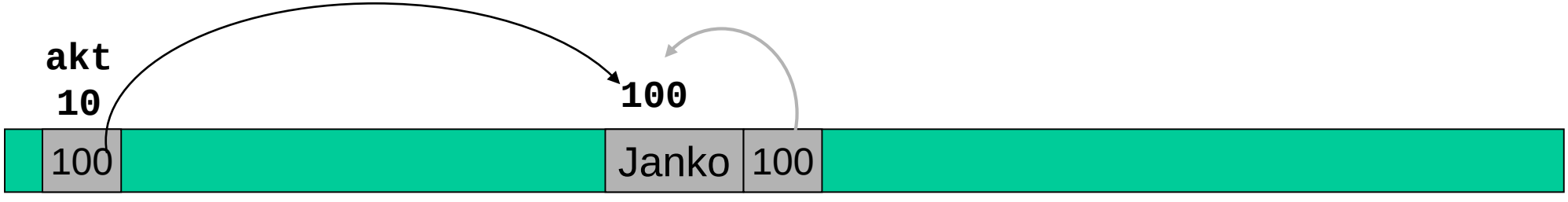
pridáme prvý záznam, aktuálny prvok bude ukazovať naňho

akt
10

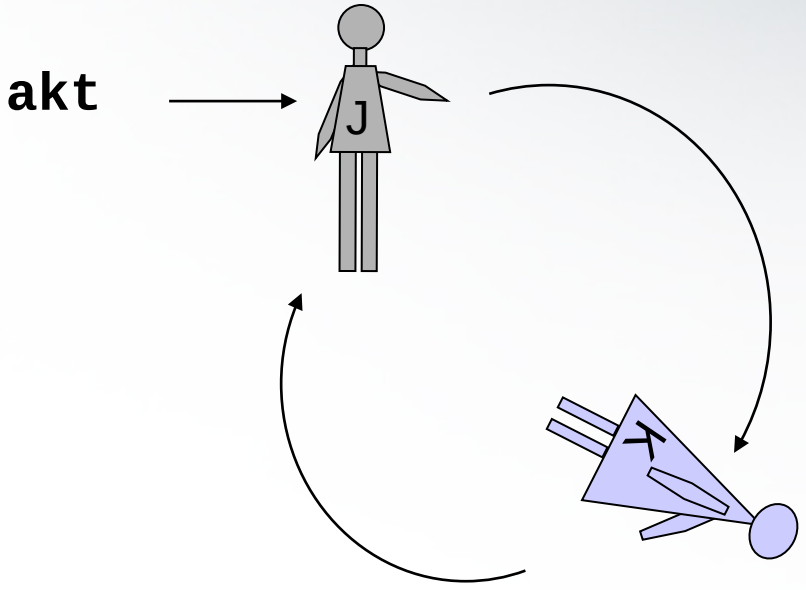
Kruhový zoznam: jeden prvok



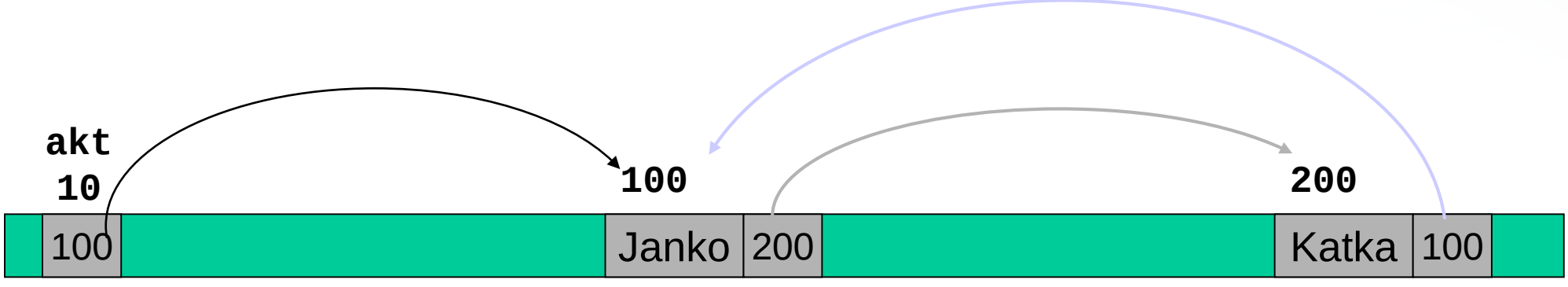
pridáme druhý záznam, Janko bude ukazovať na nový prvok a nový prvok bude ukazovať na Janka (za aktuálny prvok, ten sa nemení)



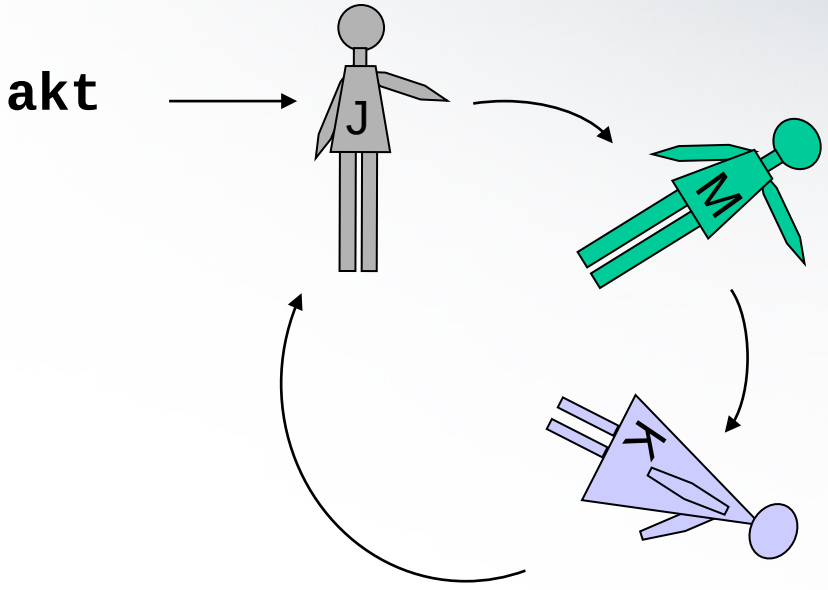
Kruhový zoznam: jeden prvok

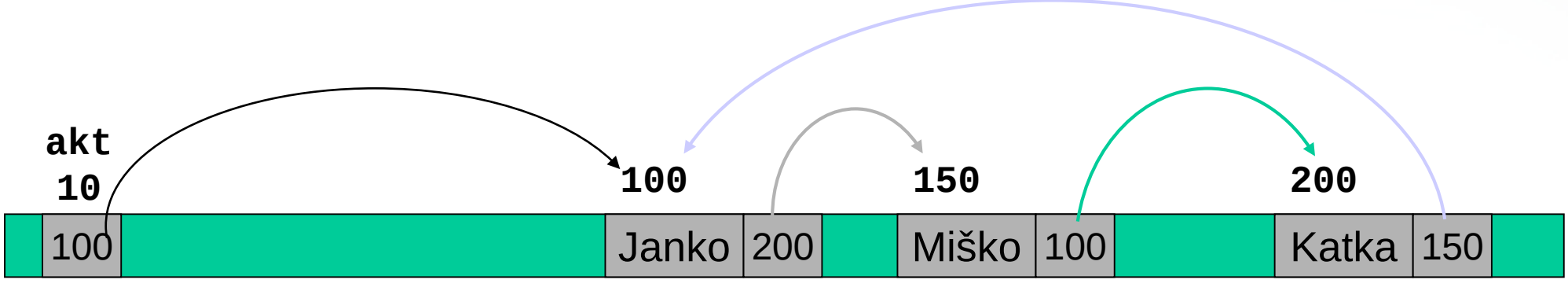


pridáme tretí záznam (za aktuálny (medzi Janka a Katku), zmení sa ukazovateľ Janka, nový záznam bude ukazovať na Katku)



Kruhový zoznam: dva prvky





Kruhový zoznam: funkcie

- funkcie:
  - `DIETA *pridaj(DIETA *akt)`
    pridanie do zoznamu (za aktuálny prvok)
  - `DIETA *zmaz(DIETA *akt)`
    zmazanie zo zoznamu (za aktuálnym prvkom)
  - `DIETA *posun(DIETA *akt)`
    posunie ukazovateľa na aktuálny prvok na `akt->dalsi`

  menia ukazovateľ na aktuálny prvok

  - `void vypis(DIETA *akt)`
    výpis zoznamu

Kruhový zoznam: implementácia

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define N 50

typedef struct dieta {
      char meno[N];
      struct dieta *dalsi;
} DIETA;

DIETA *pridaj(DIETA *akt);
DIETA *zmaz(DIETA *akt);
DIETA *posun(DIETA *akt);
void vypis(DIETA *akt);
```

```
int main()
{
   DIETA *akt = NULL;
   char c;

   do {
      printf("Kruhovy zoznam.\n");
      printf("p: pridaj\n");
      printf("z: zmaz\n");
      printf("s: posun\n");
      printf("k: koniec\n\n");

      c = getch();

      switch(c) {
      case 'p': akt = pridaj(akt); break;
      case 'z': akt = zmaz(akt); break;
      case 's': akt = posun(akt); break;
      }

      vypis(akt);
   } while (c != 'k');

   return 0;
}
```

```
/* pridava za aktualny prvok */
DIETA *pridaj(DIETA *akt)
{
   DIETA *p = (DIETA *) malloc (sizeof(DIETA));

   printf("Zadajte meno: ");
   scanf("%s", p->meno);

   if (akt == NULL) { /* ak sa vklada prvy zaznam */
      akt = p;
      akt->dalsi = p; /* ukazuje sam na seba */
      return akt;
   }
   else {              /* akt uz ukazuje na nejaky prvok */
      p->dalsi = akt->dalsi; /* vlozenie za aktualny */
      akt->dalsi = p;
      return akt->dalsi;
   }
}
```

```
/* maze za aktualnym prvkom */
DIETA *zmaz(DIETA *akt)
{
   DIETA *p;

   if (akt == NULL)          /* zoznam je prazdny => nic */
      return NULL;
   else if (akt->dalsi == akt) {  /* jednoprvkovy zoznam */
       free(akt);
       return NULL;
   }
   else {                     /* aspon 2 prvky v zozname */
      p = akt->dalsi->dalsi;
      free(akt->dalsi);
      akt->dalsi = p;
   }
   return akt;
}
```

```
/* vrati ukazovatel na dalsi prvok */
DIETA *posun(DIETA *akt) {
      return (akt == NULL ? NULL : akt->dalsi);
}


/* vypise zoznam */
void vypis(DIETA *akt) {
      DIETA * p = akt;

      if (p == NULL)
            printf("Zoznam je prazdny.\n\n");
      else {
            do {
                  printf("%s -> ", p->meno);
                  p = p->dalsi;
            } while (p != akt);
            printf("\n\n");
      }
}
```

Práca s bitmi

- práca s reprezentáciou čísla v dvojkovej sústave
- Príklady:
  - 1: 001
  - 2: 010
  - 3: 011
  - 4: 100

Prevod čísla do dvojkovej sústavy

-

príklad prevod čísla 4:

4 / 2 = 2 zvyšok 0

2 / 2 = 1 zvyšok 0

1 / 2 = 0 zvyšok 1

Zvyšky prečítané zospodu hore predstavujú číslo v dvojkovej sústave

Prevod čísla do dvojkovej sústavy (delenie dvomi)

Výsledok sa použije ako delenec v nasledujúcej časti prevodu

Oprácie s jednotlivými bitmi

- operátory:
  - **&** - bitový súčin (AND)
  - **|** - bitový súčet (OR)
  - **^** - bitový exkluzívny súčet (XOR)
  - **<<** - posun doľava
  - **>>** - posun doprava
  - **~** - jednotkový doplnok (negácia bit po bite)
- argumenty nemôžu byť **`float`**, **`double`** ani **`long double`**

Bitový súčin

- **i**-ty bit výsledku **x & y** bude 1 vtedy, ak **i**-ty bit **x** aj **i**-ty bit **y** sú 1, inak 0 (*AND* po bitoch)

```
#define je_neparne(x) (1 & (unsigned) (x))
```

```
    0000 0000 0000 0001
&   xxxx xxxx xxxx xxx1
-------------------------
    0000 0000 0000 0001
```

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

- ak chceme premennú typu int použiť ako ASCII znak, teda potrebujeme najmenších 7 bitov

`0000 0000 0111 1111 = 0x7F`

```
c = c & 0x7F;
```

```
c &= 0x7F;
```

Rozdiel medzi bitovým a logickým súčin

```
unsigned int i = 1, j = 2, k, l;
k = i && j;
l = i & j;
```

- **k**: 1, pretože 1 a 2 sú kladné číla, teda majú logickú hodnotu *true (pravda)* a **&&** je logický súčin

- **l**: 0, pretože 1 = 0000 0001

  2 = 0000 0010

  a **&** je bitový súčin

Bitový súčet

- **`i`**-ty bit výsledku **`x | y`** bude 1 vtedy, ak **`i`**-ty bit **`x`** alebo **`i`**-ty bit **`y`** sú 1, inak 0 (*OR* po bitoch)
- používa sa na nastavenie niektorých bitov na jednotku, pričom nechá ostatné bity nezmenené

```
#define na_neparne(x) (1 | (unsigned) (x))
```

```
    0000 0000 0000 0001
|   xxxx xxxx xxxx xxx1
-----------------------
    xxxx xxxx xxxx xxx1
```

x	y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

makro vráti nepárne číslo nezmenené a párne zväčší o 1

Bitový exkluzívny súčet

- **`i`**-ty bit výsledku **`x ^ y`** bude 1 vtedy, ak sa **`i`**-ty bit **`x`** nerovná **`i`**-temu bitu **`y`**, inak 0 (*XOR* po bitoch)

```
if (x ^ y) /* cisla sú rozdielne */
```

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

Bitový posun doľava

- **x << n** posunie bity v **x** o **n** pozícií doľava
- bity zľava sa strácajú bity zprava sú dopĺňané nulami

```
x = x << 1;
```

na rýchle násobenie dvomi

x = 0001 1011 0010 0101 = 6949

x << 1 = 0011 0110 0100 1010 = 13898 = 2 * 6949

```
x = x << 3;
```

vynásobesnie $2^3 = 8$

Bitový posun doprava

- **x >> n** posunie bity v **x** o **n** pozícií doprava
- bity zprava sa strácajú bity zľava sú dopĺňané nulami

```
x = x >> 1;
```

na rýchle celočíselné delenie dvomi

x = 0011 0110 0100 1010 = 13898

x >> 1 = 0001 1011 0010 0101 = 6949 = 13898 / 2

```
x = x >> 3;
```

celočíselné delenie $2^3 = 8$

Príklad: delenie a násobenie

- bitové posuny sú rýchlejšie ako násobenie a delenie násobkami dvojky

```
i = j * 80;
i = (j << 6) + (j << 4);
```

80 = 64 + 16
rýchlejšie

Príklad: zistenie hodnoty konkrétneho bitu

vráti hodnotu **i**-teho bitu **x**

```
#define ERROR -1
#define CLEAR 1
#define BIT_V_CHAR 8

int bit(unsigned x, unsigned i)
{
  if (i >= sizeof(x) * BIT_V_CHAR)
    return (ERROR);
  else
    return ((x >> i) & CLEAR);
}
```

```
  0011 0010 0101 0010
  0001 1001 0010 1001
  0000 1100 1001 0100
  0000 0110 0100 1010
  0000 0011 0010 0101
  0000 0001 1001 0010
  0000 0000 1100 1001

  0000 0000 1100 1001
& 0000 0000 0000 0001
-------------------
  0000 0000 0000 0001
```

Negácia po bitoch

- jednotkový doplnok **~x**
- prevráti nulové bity na jednotkové a naopak
- Použitie napr. ak sa chceme vyhnúť na počítači závislej dĺžke celého čísla:

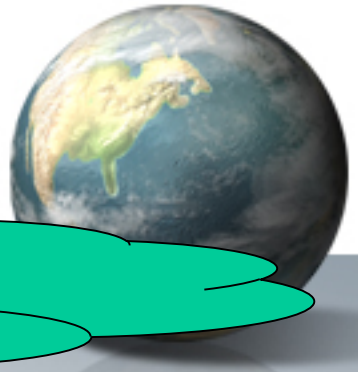
```
x &= 0xFFF0;
```

nastavenie posledných 4 bitov na nulu - len ak platí **sizeof(int) == 2**

```
x &= ~0xF;
```

nastavenie posledných 4 bitov na nulu - platí pre všade

Príklad



Zistenie dĺžky typu **int** v bitoch

```
#include <stdio.h>

int dlzka_int()
{
   unsigned int x, i = 0;

   x = ~0;      /* negácia 0 -> same 1 */

   while ((x >> 1) != 0)
      i++;
   return (++i);
}

int main()
{
   printf("Dlzka typu int je %d bitov\n", dlzka_int());
   return 0;
}
```

Práca so skupinou bitov

- stavová premenná **`stav`** - definuje práva na prístup k súboru

```
#define READ 0x8
#define WRITE 0x10
#define DELETE 0x20

unsigned int stav;

stav |= READ | WRITE | DELETE;
stav |= READ | WRITE;
stav  &= ~(READ | WRITE | DELETE);
stav & = ~READ;
if ( ! (stav & (WRITE | DELETE)))
   ...
```

→ **READ:** 2^3 = 0000 1000
→ **WRITE:** 2^4 = 0001 0000
→ **DELETE:** 2^5 = 0010 0000

nastaví 3., 4. a 5. bit na 1
(bity sú počítané od 0)

nastaví 3., 4. bit na 1

nastaví 3., 4. a 5. bit na 0

nastaví 3. bit na 0

ak 3. a 4. bit sú nulové

Bitové pole

- štruktúra, ktorej veľkosť je obmedzená veľkosťou typu **`int`**
- najmenšia dĺžka položky je 1 bit
- definuje podobne ako štruktúra, ale každá položka bitového poľa je určená menom a dĺžkou v bitoch
- môže byť **`signed`** aj **`unsigned`** (preto vždy uviesť)
- oblasti použitia:
  - uloženie viac celých čísel v jednom (šetrenie pamäte)
  - pre prístup k jednotlivým bitom (často)

Príklad bitového poľa

- uloženie dátumu do jednotho **`int`**-u:
  - deň - najmenších 5 bitov,
  - mesiac - ďalšie 4 bity,
  - rok - zvyšných 7 bitov (max. 127, preto rok - 1980)

```
typedef struct {
    unsigned den    : 5;
    unsigned mesiac : 4;
    unsigned rok    : 7;
} DATUM;

DATUM dnes, zajtra;
dnes.den = 29;
dnes.mesiac = 11;
dnes.rok = 2012 - 1980;
zajtra.den = dnes.den + 1;
```

bity 0-4

bity 5-8

bity 9-15

Príklad bitového poľa

Dátum ako bitové pole aj
hexadecimálne číslo (union)

```
#include <stdio.h>

typedef struct {
  unsigned den    : 5;   /* bity 0 -  4 */
  unsigned mesiac : 4;   /* bity 5 -  8 */
  unsigned rok    : 7;   /* bity 9 - 15 */
} DATUM;

typedef union {
  DATUM         datum;
  unsigned int  cislo;
} BITY;
```

Príklad bitového poľa - pokračovanie

```
#include <stdio.h>

typedef struct {
  unsigned den      : 5;
  unsigned mesiac   : 4;
  unsigned rok      : 7;
} DATUM;

typedef union {
  DATUM          datum;
  unsigned int   cislo;
} BITY;
```

```
int main(void)
{
  BITY dnes;
  int d, m, r;

  printf("Zadaj dnesny datum [dd mm rrrr]: ");
  scanf("%d %d %d", &d, &m, &r);
  dnes.datum.den = d;
  dnes.datum.mesiac = m;
  dnes.datum.rok = r - 1980;

  printf("datum: %2d.%2d.%4d - cislo: %X hexa\n",
          dnes.datum.den, dnes.datum.mesiac,
          dnes.datum.rok + 1980, dnes.cislo);
  return 0;
}
```