Procedurálne programovanie

slido.com
5981245
PrPr – P10

Ján Zelenka

Ústav Informatiky

Slovenská akadémia vied

Oznamy

slido slido.com
5981245
PrPr – P10

Termín odovzdania druhého projektu (Spájaný zoznam štruktúr): odovzdanie v 11. týždni (3.12. do 23:59)

- neskoré odovzdanie 12. týždeň (10.12.do 23:59), penalizácia - uznáva sa 80% zo získaného počtu bodov
- za projekt musí získať študent **min. 6 bodov** (akceptovateľný),bez bodov za prezentáciu (2 bodov)

IDstudenta_Rok_projekt_2.c

Obsah prednášky

slido slido.com
5981245
PrPr – P10

1. **Opakovanie**
2. **Bitové operácie**
3. **Preprocesor jazyka C**

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

Štruktúry ukazujúce samy na seba

slido.com
5981245
PrPr – P10

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

odkaz na samého seba (na takú istú štruktúru)

aj štruktúra, aj typ musia byť pomenované

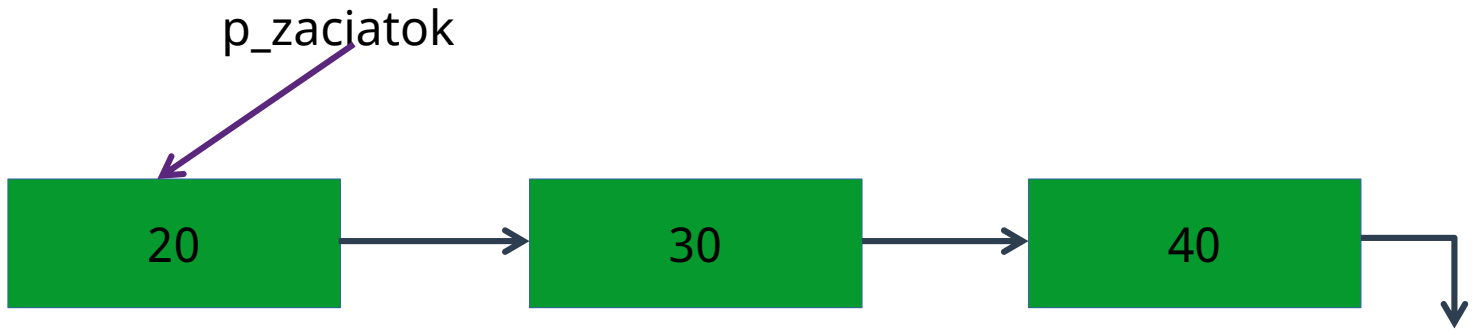
```
typedef struct {
    int hodnota;
    struct PRVOK *p_dalsi;
} PRVOK;
```

chyba: v čase, keď sa definuje **p_dalsi**, položka **PRVOK** ešte nie je známa

Spájaný zoznam

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

- postupnosť hodnôt
- reprezentovaný ukazovateľom na prvý prvok
- posledný prvok nemá nasledovníka
- ukazovateľ **`p_dalsi`** je nastavený na špeciálnu hodnotu, zvyčajne **`NULL`**



Pridanie prvku na začiatok zoznamu

1. vytvoríme prvok so vstupnými dátami
2. nasledovníka nového prvku nastavíme na začiatok zoznamu
3. začiatok zoznamu nastavíme na nový prvok

```
void priadaj_na_zaciatok(PRVOK** p_p_zaciatok, int hodnota)
{
    PRVOK* novy_prvok = (PRVOK*) malloc(sizeof(PRVOK));
    novy_prvok->data  = hodnota;
    novy_prvok->dalsi = (*p_p_zaciatok);
    (*p_p_zaciatok)    = novy_prvok;
}
```

Pridanie prvku na koniec zoznamu

1. vytvoríme prvok so vstupnými dátami
2. nasledovníka nového prvku nastavíme na NULL
3. nájdeme posledný prvok
4. nasledovník posledného prvku bude nový prvok

```
void priadaj_na_koniec(PRVOK** p_p_zaciatok, int hodnota) {
    PRVOK *novy_prvok = (PRVOK*) malloc(sizeof(PRVOK));
    PRVOK *posledny = *p_p_zaciatok;

    novy_prvok->data  = hodnota;
    novy_prvok->dalsi = NULL;

    if (*p_p_zaciatok == NULL) {
       *p_p_zaciatok = novy_prvok;
       return;}

    while (posledny->dalsi != NULL)
        posledny = posledny->dalsi;

    posledny->dalsi = novy_prvok;
    return;}
```

čo ak je spájaný zoznam prázdny?

Pridanie prvku do zoznamu

Pridanie nového prvku za prvok zo zoznamu:

1. vytvoríme prvok so vstupnými dátami
2. nájdeme prvok v zozname
3. nasledovníka nového prvku nastavíme na nasledovníka nájdeného prvku zo zoznamu
4. nasledovníka nájdeného prvku zoznamu nastavíme na nový prvok

Odstránenie prvého prvku zo zoznamu

1. zapamätáme si prvý prvok zoznamu v pomocnej premennej
2. posunieme začiatok zoznamu na jeho nasledovníka
3. uvoľníme pamäť vyhradenú pre pôvodný prvý prvok zoznamu

```
void zmaz_zaciatok(PRVOK **p_p_zaciatok) {
    PRVOK *p_akt;

    if(p_p_zaciatok == NULL || *p_p_zaciatok == NULL)
        return;
    p_akt = *p_p_zaciatok;
    *p_p_zaciatok = (*p_p_zaciatok)->dalsi;
    free(p_akt);}
```

Spájaný zoznam a dynamické pole

slido.com
5981245
PrPr – P10

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

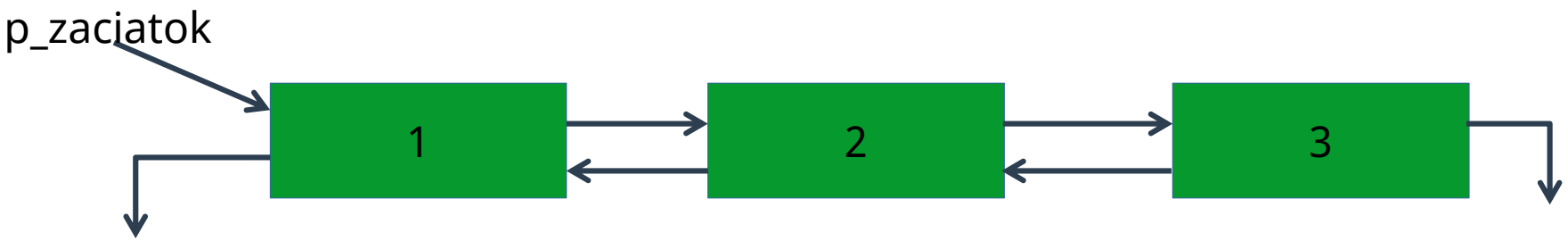
Zhrnutie

slido slido.com
5981245
PrPr – P10

- spájaný zoznam sa skladá z prvkov, ktoré sú prepojené ukazovateľmi
- celý zoznam je dostupný cez ukazovateľ na prvý prvok
- každý prvok zoznamu, okrem posledného má jediného nasledovníka (ukazovateľ `dalsi`)

Obojsmerný spájaný zoznam

- obsahuje ukazovateľ na nasledovníka a aj na predchodcu
- môžeme ho prechádzať oboma smermi

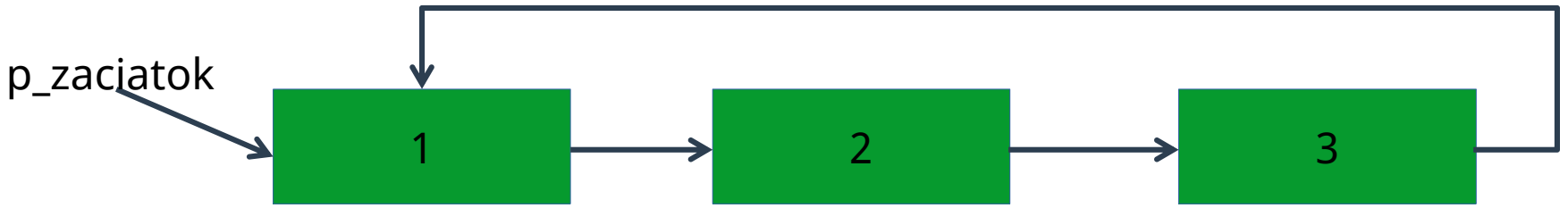


Kruhový spájaný zoznam

- ukazovateľ na nasledovníka posledného prvku ukazuje na prvý prvok zoznamu
- prechádzanie zoznamu môžeme začať v ľubovoľnom prvku (celý zoznam sme spracovali vtedy, ak sme druhýkrát navštívili prvý/vstupný prvok zoznamu)

```
typedef struct prvok {
    int hodnota;

    struct polozka *p_dalsi;
} PRVOK;
```



Bitové operácie

Práca s bitmi

práca s reprezentáciou čísla v dvojkovej sústave

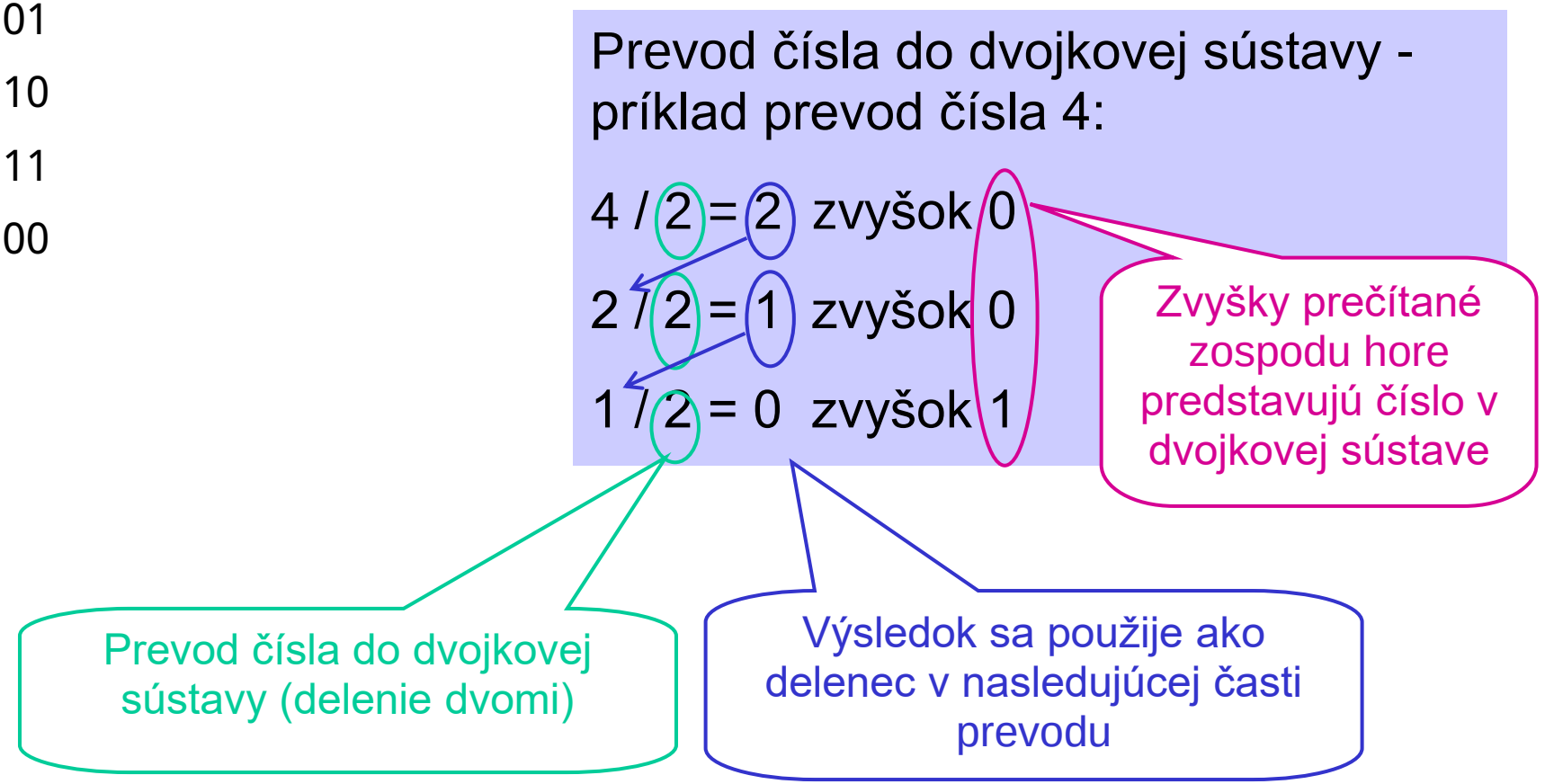
Príklady:

1: 001

2: 010

3: 011

4: 100



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

$$\mathbf{1}*2^2 + \mathbf{0}*2^1 + \mathbf{0}*2^0 = 4$$

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, ostatné vynulujeme

`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),* **`&&`** je logický súčin
- **`l`** = 0, pretože

  `&` je bitový súčin

```
1 = 0000 0001
2 = 0000 0010
1 = 0000 0000
```

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 `i`-ty bit `x` nerovná `i`-temu bitu `y` sú 1, 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 sprava 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;
```

vynásobenie $2^3 = 8$

Bitový posun doprava

- `x >> n` posunie bity v `x` o `n` pozícií doprava
- bity sprava 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

Priorita operátorov **<<** a **>>** je veľmi nízka, je nutné výrazy zátvorkovať.

Príklad: zistenie hodnoty konkrétneho bitu

Operátor >> sa často používa na získanie hodnoty konkrétneho bitu. Bity posúva až dokiaľ nie je požadovaný bit na najnižšej pozícii

```
#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);
}
```

```
   000 0000 1100 1001
& 0000 0000 0000 0001
-------------------
   0000 0000 0000 0001
```

vráti hodnotu **i**-teho bitu **x**

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

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 0100
→ **WRITE:** 2^4 = 0000 1000
→ **DELETE:** 2^5 = 0001 0000

nastaví 2., 3. a 4. bit na 1

nastaví 2., 3. bit na 1

nastaví 2., 3. a 4. bit na 0

nastaví 2. bit na 0

ak 2. a 3. bit sú nulové

Bitové pole

Bitové pole

slido slido.com
5981245
PrPr – P10

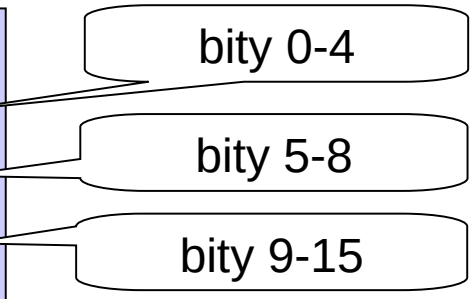
- štruktúra, ktorej veľkosť je obmedzená veľkosťou typu **`int`**
- najmenšia dĺžka položky je 1 bit
- definuje podobne ako štruktúra, ale každá položka bitového poľa je určená menom a dĺžkou v bitoch
- môže byť **`signed`** aj **`unsigned`** (preto vždy uviesť)
- oblasti použitia:
  - uloženie viac 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 = 4;
dnes. mesiac = 5;
dnes.rok = 2005 - 1980;
zajtra.den = dnes.den + 1;
```



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

```c
#include <stdio.h>

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

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

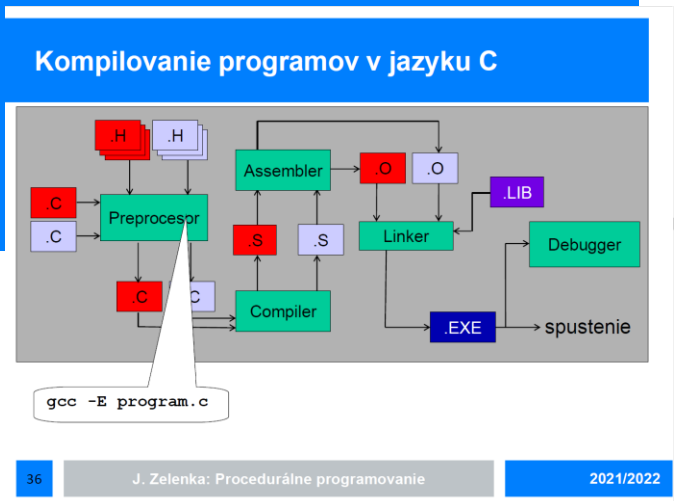
```c
int main(void)
{
  BITY dnes;
  int d, m, r;

  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;
}
```

Predprocesor jazyka C

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 **#**

Konštrukcie pre preprocesor

- definovanie makra

```
#define meno_makra text
```

- zrušenie definície makra

```
#undef meno_makra
```

- podmienený preklad v závislosti na konštante **`konst`**

```
#if konst
#elif #else #endif
```

Konštrukcie pre preprocesor

- vloženie textu zo špecifikovaného súbora zo systémového adresára

```
#include <filename>
```

- vloženie textu zo špecifikovaného súbora v adresári používateľa

```
#include "filename"
```

- výpis chybových správ vo fáze predspracovania

```
#error text
```

Konštrukcie pre preprocesor

- podmienený preklad v závislosti od toho, či je makro definované, alebo nedefinované

```
#ifdef meno_makra
#elif #else #endif
```

- podmienený preklad v závislosti od toho, či je makro nedefinované, alebo definované

```
#ifndef meno_makra
#elif #else #endif
```

Konštanty - makrá bez parametrov

slido slido.com
5981245
PrPr – P10

- symbolické konštanty
- používajú sa často (zbavujú program "magických čísel")
- väčšinou definované na začiatku modulu
- platnosť konštánt je do konca modulu
- náhrada konštanty hodnotou - rozvoj (expanzia) makra

Pravidlá pre písanie konštánt

slido slido.com
5981245
PrPr – P10

- mená konštánt - veľkými písmenami
- meno konštanty je od hodnoty oddelené aspoň jednou medzerou
- za hodnotou by mal byť vysvetľujúci komentár
- nové konštanty môžu využívať skôr definované konštanty
- ak je hodnota konštanty dlhšia ako riadok, musí byť na konci riadku znak \ (nie je súčasťou makra, nerozvinie sa, je to iba pomocný znak)

Príklady defninovania konštánt

```
#define MAX 1000
#define PI 3.14
#define DVE_PI (2 * PI)
#define MOD %
#define AND &&
#define MENO_SUBORU "list.txt"
#define DLHA_KONSTANTA Toto je dlha konstanta, \
            ktora sa nezmesti do jednoho riadku.
```

- za hodnotou nie je ;
- medzi menom konštanty a jej hodnotou nie je =
- platnosť konštanty – od definovania do konca súboru

Kedy sa nerozvinie makro

slido slido.com
5981245
PrPr – P10

- makro sa nerozvinie, ak je uzatvorené v úvodzovkách

```
#define MENO    "Jozef"

...
printf("Volam sa MENO");

printf("Volam sa %s", MENO);
```

vypíše sa:
Volam sa MENO

vypíše sa:
Volam sa Jozef

Prekrývanie definícií

- nová definícia prekrýva starú, pokiaľ je rovnaká (to ani nemá zmysel)
- ak nie je rovnaká:
  - zrušiť starú definíciu:

```
#undef meno_makra
```

  - definovať meno_makra

```
#define POCET 10
#undef POCET
#define POCET 20
```

Makro ako skrytá časť programu

```
#define ERROR { printf("Chyba v datach. \n"); }
```

- pri použití nie je makro ukončené bodkočiarkou:

```
if (x == 0)
    ERROR
else
    y = y / x;
```

Makrá s parametrami

slido slido.com
5981245
PrPr – P10

- krátka a často používaná funkcia vykonávajúca jednoduchý výpočet
  - problém s efektivitou (prenášanie parametrov a úschova návratovej hodnoty je časovo náročnejšia ako výpočet)
  - preto namiesto funkcie - makro (to sa pri preprocesingu rozvinie)
- je potrebné sa rozhodnúť medzi
  - funkcia: kratší ale pomalší program
  - makro: rýchlejší ale dlhší program

Makrá s parametrami

- nazývajú sa vkladané funkcie - rozvitie makra znamená, že sa meno makra nahradí jeho telom

definícia makra

```
#define je_velke(c) ((c) >= 'A' && (c) <= 'Z')
```

- zátvorka, v ktorej sú argumenty funkcie - hneď za názvom makra (bez medzery)

v zdrojovom súbore

```
ch = je_velke(ch) ? ch + ('a' - 'A') : ch;
```

rozvinie sa

```
ch = ((ch) >= 'A' && (ch) <= 'Z') ? ch + ('a'-'A') : ch;
```

Makrá s parametrami

- telo makra - uzavrieť do zátvoriek, inak môžu nastať chyby, napr.:

```
#define sqrt(x) x * x
...
sqrt(f + g);
```

po rozvinutí makra

```
f + g * f + g;
```

- správne

```
#define sqrt(x) (x * x)
...
sqrt((f + g));
```

po rozvinutí makra

```
(f + g) * (f + g);
```

Preddefinované makrá

`getchar()` a `putchar()` (v `stdio.h`)

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

makrá v `ctype.h`:

- na určenie typu znaku začínajú písmenami **is** (`isalnum`, `isalpha`...)
- na konverziu znaku začínajú písmenami **to** (`tolower`, `toupper`)

Vkladanie súborov

- vkladanie systémových súborov **< >**
- vkladanie súborov v aktuálnom adresári **" "**

```
#include <stdio.h>
#include <ctype.h>
#include "VLASTNY.H"
```

Podmienený preklad

slido slido.com
5981245
PrPr – P10

U väčších programov

- ladiace časti - napr. pomocné výpisy

Program

- trvalá časť zdrojového kódu – už pri vytváraní kódu ich označíme ako podmienene prekladateľné
- voliteľná časť zdrojového kódu (napr. pri ladení, alebo ak je argumentom programu nejaký prepínač)

Riadenie prekladu hodnotou konštantného výrazu

```
#if konstantny_vyraz
    cast_1
#else
    cast_2
#endif
```

ak je hodnota konštantného výrazu nenulová, vykoná sa časť 1, inak časť 2

```
#if 0
    cast programu, co
    ma byt vynechana
#endif
```

ak pri testovaní nechcete prekladať časť programu, namiesto `/* */` (problém by robili vhniezdené komentáre)

Riadenie prekladu hodnotou konštantného makra

```
#define PCAT 1

#if PCAT
    #include <conio.h>
#else
    #include <stdio.h>
#endif
```

- ak je program závislý na konkrétnom počítači
- ak na PC/AT - definujeme PCAT na 1, inak na 0

Riadenie prekladu definíciou makra


```
#define PCAT

#ifdef PCAT
    #include <conio.h>
#else
    #include <stdio.h>
#endif
```

- ak je program závislý na konkrétnom počítači
- ak na PC/AT - definujeme PCAT (bez hodnoty),
- stačí, že je konštanta definovaná

```
#ifndef PCAT
```

- ak nie je definovaná konštanta

```
#undef PCAT
```

- zrušenie definície makra

Operátory defined, #elif a #error

- **`#ifdef`**, alebo **`#ifndef`** zisťujú existenciu len jedného symbolu, čo neumožňuje kombinovať viaceré

- ak treba kombinovať viaceré podmienky:

```
#if defined TEST
```

```
#if !defined TEST
```

- **`#elif`** - má význam else-if

- **`#error`** - umožňuje výpis chybových správ (v priebehu preprocesingu - nespustí sa kompilácia)

Ďakujem vám za pozornosť!

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