



# Programovanie v jazyku C

Parametre funkcií

Polia, reťazce

Smerníky, smerníková aritmetika

Dynamická alokácia pamäte

Pamäťové triedy, typové modifikátory



# [ Pamät' programu ]

```
// globálna premenná  
int y = 2;
```

```
int main(void)  
{
```

```
    // lokálna premenná  
    int x = a;
```

```
    printf("x je na adrese %p\n", &x);  
    printf("y je na adrese %p\n", &y);  
    return 0;
```

```
}
```

Read-only



# Smerník (pointer, ukazovateľ)

```
int *ptr;  
int i;
```

ptr



i



```
ptr = &i;  
*ptr = 112;
```

ptr



i



```
printf("Premenna i je na adrese %p a ma hodnotou %d\n", ptr, i);
```

&      *adresový operátor*

\*      *operátor dereferencie*

NULL      *konštanta reprezentujúca neplatnú adresu*

# [ Smerníky rôzneho typu (príklady deklarácie) ]

```
float *p, *q;  
unsigned long *r;  
char *s;  
FILE *f;  
int *t[5];  
char *fun(void);
```

p, q sú smerníky na typ *float*  
r je smerník na typ *unsigned long*  
s je smerník na typ *char*  
f je smerník na typ (štruktúru) *FILE*  
t je pole 5 smerníkov na typ *int*  
fun je funkcia vracajúca *smerník na char*

```
int **ptr;
```

ptr je smerník na typ smerník na typ int

premenná ptr obsahuje *adresu smerníka* ukazujúceho na premennú typu int

```
int x = 999;  
int *p = &x;  
int **pp = &p;
```



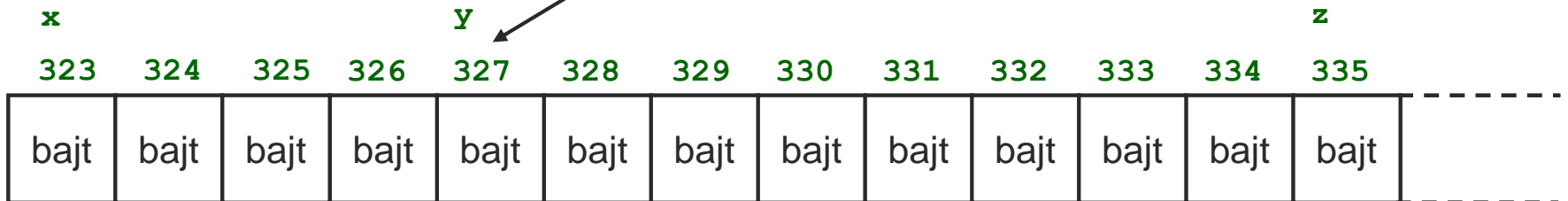
# Prečo v deklarácii uvádzame typ smerníka?

```
int x;  
double y;  
char z;
```

```
double *p;
```

p

327



```
y = *p + 15.7;
```

Smerník obsahuje adresu bajtu, kde premenná začína.

Vďaka deklarácii počítač vie, o aký typ smerníka sa jedná, a vie teda aj to, kde táto premenná končí (akú má veľkosť, koľko bajtov zaberá).

# [ Smerník typu *void* ]

`void *p;`      všeobecný smerník, pamäťové miesto pre „nejakú“ adresu  
môže ukazovať na premennú ľubovoľného typu

---

Často ako parameter alebo výsledok „všeobecne napísaných“ funkcií, napr.:

```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

```
void* malloc(size_t size);
```

---

```
void *p;  
int i;  
double x;
```

```
p = &i;  
p = &x;
```

OK!

```
p = &i;  
*p = 999;
```

CHYBA!

```
p = &i;  
*(int*)p = 999;
```

OK!

Ak chceme na smerník typu *void* použiť operátor \*, musíme ho najskôr správne pretypovať!



# [ Úloha (parametre funkcie)

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

```
void vymena(int a, int b)
{
    // TO DO:
}
```

```
int main(void)
{
    int x = 10;
    int y = 20;

    vymena(x, y);

    printf("Po vymene: x = %d, y = %d\n", x, y);
    return 0;
}
```

# Správne riešenie

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

```
void vymena(int *a, int *b)
```

```
{
```

```
    int pom = *a; *a = *b; *b = pom;
```

```
}
```

```
int main(void)
```

```
{
```

```
    int x = 10;
```

```
    int y = 20;
```

```
    vymena(&x, &y);
```

```
    printf("Po vymene: x = %d, y = %d\n", x, y);
```

```
    return 0;
```

```
}
```

*Ak chceme funkcii sprístupniť skutočné parametre (aby sa premenná zbytočne nekopírovala na zásobník, resp. aby do nej mohla funkcia zapísať hodnotu, ktorú z nej potrebujeme vyniesť), musíme do funkcie posilať adresu a parameter deklarovať ako smerník.*



# Úloha (pole ako parameter)

```
#include <stdio.h>


void show_msg(char msg[])
{
    printf("%s\n\n", msg);
    printf("sizeof(msg) == %d\n", sizeof(msg));
}

int main(void)
{
    char quote[] = "Only two things ...";

    show_msg(quote);
    printf("sizeof(quote) == %d\n\n", sizeof(quote));

    return 0;
}
```

*// mozeme pouzit aj tento zapis*  
*void show\_msg(char \*msg) { }*



# [ Polia a smerníky (diskusia) ]

```
char s[] = "Hallo there!";  
char *t = s;
```

```
sizeof(s) == 13
```

```
sizeof(t) == 4
```

*veľkosť údajovej štruktúry s*

*veľkosť smerníka t (všetky smerníky majú rovnakú veľkosť)*

```
&s == s
```

```
&t != t
```

*s je meno poľa a zároveň aj adresa, na ktorej toto pole v pamäti začína  
t je smerník, teda premenná, ktorá je v pamäti uložená na inej adrese ako  
pole, na ktoré ukazuje*

```
s = t;
```

*premenná s je síce adresou začiatku poľa, ale ide o konštantný smerník, nie  
je možné priradiť mu inú hodnotu*

# Úloha – čo sa udeje?

```
#include <stdio.h>

int main(void)
{
    char *cards = "JQKA";
    char c = cards[2];

    cards[2] = cards[1];
    cards[1] = cards[0];
    cards[0] = cards[2];
    cards[2] = cards[1];
    cards[1] = c;

    puts(cards);
    return 0;
}
```



# [ Dynamická alokácia pamäte ]

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

```
void* malloc(size_t size);
```

```
void* calloc(size_t num, size_t size);
```

```
void* realloc(void *memblock, size_t size);
```

používame, keď  
chceme dynamicky  
alokovať pamäť  
určitej veľkosti

alokovanú pamäť uvoľňujeme pomocou funkcie

```
void free(void *memblock);
```

# Vytvorenie / zrušenie dynamickej premennej

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

int main(void)
{
    int *p;

    p = (int*) malloc(sizeof(int));

    *p = 112;
    printf("%d\n", *p);

    free(p);           // free((void*)p);

    return 0;
}
```

# Pole alokované dynamicky (dynamické pole)

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

```
int main(void)
```

```
{
    int a[10];    statické pole
    int *b;       dynamické pole (smerník na jeho začiatok)
```

```
b = (int*)malloc( sizeof(int)*10 );
```

```
// práca s poľom
```

↑  
b je pole s 10 prvkami typu int

```
free(b) ;
```

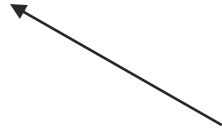
```
return 0;
```

```
}
```

# Pole alokované dynamicky (dynamické pole)

```
if ((b = (int*)malloc( sizeof(int)*10 )) == NULL)
{
    printf("Malo pamati!\n");
    exit(1);
}
```

funkcia *malloc* vracia v prípade neúspechu hodnotu NULL

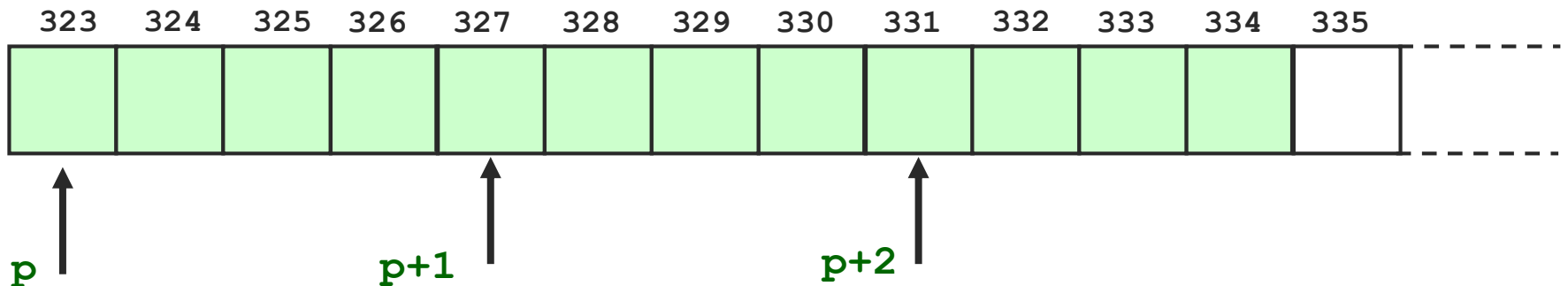


```
free(b);
b = NULL;
```

po uvoľnení pamäte ukazuje *b* na neexistujúcu premennú!  
„zo slušnosti“ resp. pre istotu nastavíme smerník na NULL

# Smerníková aritmetika – súčet smerníka a čísla

```
int *p = (int*)malloc( sizeof(int)*3 );
```



Keď ku smerníku  $p$  typu `int` pripočítame celé číslo  $n$ , posunieme sa v pamäti o  `$n * \text{sizeof}(\text{int})$`  bajtov ďalej (čiže o  $n$  prvkov ďalej)

<code>p[0]</code>	<code>*p</code>	<code>*(p + 0)</code>	0. prvok poľa
<code>p[1]</code>		<code>*(p + 1)</code>	1. prvok poľa
<code>p[i]</code>		<code>*(p + i)</code>	$i$ . prvok poľa



# Príklad (najväčší prvok v poli)

```
void main(void)
{
    int *p;
    int  n, i, max;

    printf("pocet prvkov n = "); scanf("%d\n", &n);
    p = (int*)malloc(n*sizeof(int));

    for (i = 0; i < n ; i++) {
        printf("%d. prvok:  ", i + 1);
        scanf("%d", p + i);
    }

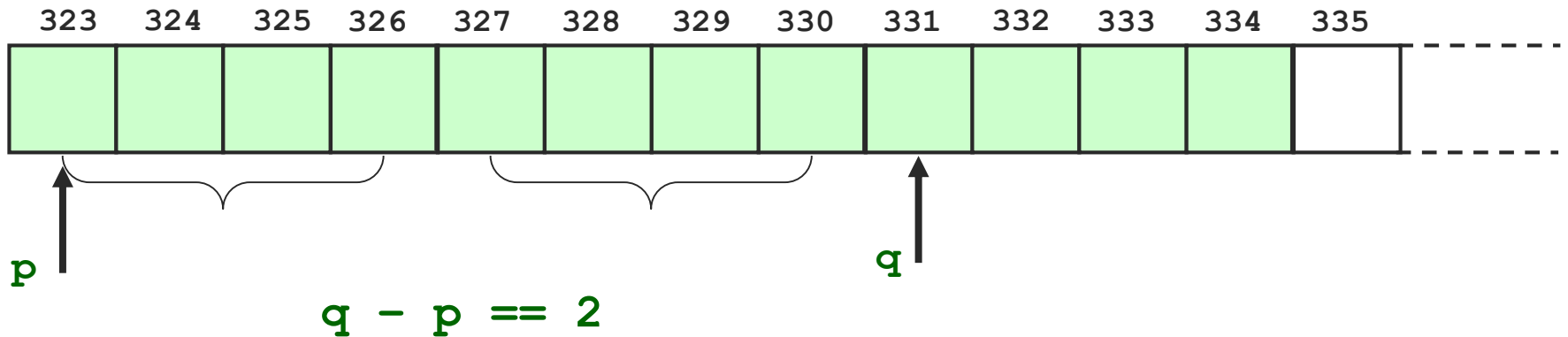
    max = *p;
    for (i = 1; i < n ; i++) {
        if (*(p + i) > max) max = *(p + i);
    }
    printf("Maximum je %d \n", max);
}
```

adresa prvku, to isté ako  $\&p[i]$

hodnota prvku, to isté ako  $p[i]$

# Smerníková aritmetika - rozdiel smerníkov

```
int *p, *q;
```



Ak sú  $p$ ,  $q$  smerníky rovnakého typu a ukazujú do toho istého súvislého bloku pamäte (na prvky toho istého poľa), potom rozdiel  $q - p$  predstavuje počet prvkov medzi týmito smerníkmi



# [ Úloha – je toto možné?

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

```
int main(void)
```

```
{
```

```
    int a[] = {10,20,30};
```

```
    printf("%d\n", a[2]);
```

```
    printf("%d\n", 2[a]);
```

```
    return 0;
```

```
}
```



# [ Reťazec = pole znakov ]

```
char str[10];
```

premenná *str* je reťazec, ktorý môže obsahovať najviac 9 znakov  
posledný bajt je vyhradený pre ukončovací znak `'\0'`

str	'a'	'h'	'o'	'j'	'\0'					
-----	-----	-----	-----	-----	------	--	--	--	--	--

str	'\0'									
-----	------	--	--	--	--	--	--	--	--	--

# Príklad – práca s reťazcom

```
#include <stdio.h>
#include <string.h> // strlen, strcpy, strcat, strcmp
```


```
int main(void)
{
```

```
    char str1[] = "Bratislava";
    char str2[] = {'N','i','t','r','a','\0' };
    char meno[25], priezvisko[25];
```

```
    int i;
```

```
    puts("Tvoje meno:");
    fgets(meno, 25, stdin);
```

*& sa v prípade  
reťazca nepíše!*



```
// scanf("%24s", meno);
```

```
    puts("Tvoje priezvisko:");
    fgets(priezvisko, 25, stdin);
```

```
    puts(str1);
    printf(str2);
    printf("Volas sa %s %s.\n", meno, priezvisko);
```

# Príklad – práca s reťazcom pokračovanie

```
for (i = strlen(str1)-1; i>=0; i--)  
    putchar(str1[i]);  
  
strcpy(meno, "Jozef");  
strcpy(priezvisko, "Mrkvicka");  
  
printf("\n%s ma %d znakov\n", meno, strlen(meno));  
  
strcat(meno, priezvisko);  
  
printf("%s ma %d znakov\n", meno, strlen(meno));  
// puts(strcat(meno, priezvisko));  
  
strcpy(priezvisko, "");  
  
strcpy(str1, "Berlin");
```

# Príklad – práca s reťazcom pokračovanie

```
if (!strcmp(str1, str2)) printf("%s == %s", str1, str2);  
else if (strcmp(str1, str2) < 0) printf("%s < %s", str1, str2);  
    else printf("%s > %s", str1, str2);  
  
return 0;  
}
```

ak sú porovnávané reťazce rovnaké, funkcia *strcmp* vráti hodnotu 0

```
if (strcmp(str1, str2) == 0) {}
```

```
if (!strcmp(str1, str2)) {}
```

```
if (!strcmp(str1, "hura")) {}
```

# Príklad (prevod reťazca na reťazec s veľkými písmenami)

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h> //toupper
```

```
void strToUpper(char *s)
{
    while (*s) {
        *s = toupper(*s);
        s++;
    }
}
```

```
int main(void)
{
    char *s = (char*)malloc(100);

    strcpy(s, "abcdefg");
    strToUpper(s);
    printf("%s\n", s);

    free(s);
    return 0;
}
```





# [ Úloha - Aký bude výstup programu? ]

```
int main(void)
{
    char *p;

    for (p = "VIP"; *p; p++) {
        printf("%c", *p - 1);
    }

    printf("\n");

    return 0;
}
```

Riešenie: UHO



# [ Pamäťová trieda *auto*

## automatické premenné

implicitne lokálne premenné funkcií  
vytvoria sa pri vstupe do funkcie v zásobníku  
nie sú inicializované, obsahujú náhodnú hodnotu

```
int main(void)
{
    auto int x, y;                // to isté ako int x, y;

    printf("%d %d\n", x, y);
    x = y = 0;
    printf("%d %d\n", x, y);

    return 0;
}
```

# [ Pamäťová trieda *extern* ]

## externé premenné

implicitne globálne premenné

sú uložené v dátovom segmente, inicializované nulovými hodnotami

slovo `extern` sa používa *hlavne* pri oddelenom preklade modulov na rozšírenie rozsahu platnosti na iný modul ako ten, v ktorom je premenná deklarovaná

```
extern int x;
```

```
int main(void)
{
    printf("%d\n", x);

    return 0;
}
```

```
int x;
```

```
int main(void)
{
    int x;
    printf("%d\n", x);

    return 0;
}
```

```
int x;
```

```
int main(void)
{
    extern int x;
    printf("%d\n", x);

    return 0;
}
```



# [ Pamäťová trieda *register*

## registrové premenné

môžeme požadovať, aby boli niektoré lokálne premenné umiestnené v registroch počítača (kvôli rýchlosti výpočtu)

istotu, či naozaj budú v registroch, ale nemáme

```
int main(void)
{
    register int i;

    for (i = 0; i < 1000; i++) {
        // príkazy
    }

    return 0;
}
```



# [ Pamäťová trieda *static*

## statická *lokálna* premenná

pamäť sa pre ňu vyhradí v dátovom segmente  
máme k nej prístup len v príslušnej funkcii  
inicializujeme ju pri prvom vstupe do funkcie  
hodnota sa uchováva aj po skončení funkcie

## statická *globálna* premenná

bude viditeľná len v príslušnom module  
používa sa hlavne pri oddelenom preklade modulov

# [ Funkcia, ktorá vie, koľký raz ju voláme ]

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

void fun(void)
{
    static int p = 0;

    printf("Volas ma %3d. krat!\n", ++p);
}

void main(void)
{
    int i;
    for (i = 0; i < 1000; i++)
        if (rand()%2) fun();
}
```

# [ Typový modifikátor *const* ]

```
const int n = 100;
```

hodnota *n* už nemôže byť po inicializácii menená  
(takúto konštantu ale v C nemožno použiť ako rozmer poľa)

```
int main(void)
{
    int x = 1, y = 2;
    int* const p = &x;

    p = &y;
    chyba

    return 0;
}
```

```
int main(void)
{
    int x = 1, y = 2;
    const int* p = &x;

    *p = 34;
    chyba

    return 0;
}
```

```
char* strcpy (char *str1, const char *str2);
```



reťazec *str2* nie je možné vo funkcii zmeniť



# [ Typový modifikátor *volatile*

```
volatile int pocet;
```

```
void cakaj (int maximum)
{
    pocet = 0;
    while (pocet < maximum)
        ;
}
```

Modifikátor *volatile* upozorňuje kompilátor, že takto definovaná premenná môže byť modifikovaná nejakou bližšie nešpecifikovanou asynchrónnou udalosťou mimo váš program, napr. pomocou prerušenia.

Kompilátor teda nemôže robiť pri optimalizácii „predčasné závery“.