Procedurálne programovanie

slido.com
2537503
PrPr – P5

Ján Zelenka

Ústav Informatiky

Slovenská akadémia vied

Obsah prednášky

slido slido.com
2537503
PrPr – P5

1. Opakovanie
2. Alokácia pamäte – statická alokácia
3. Alokácia pamäte – dynamická alokácia
4. Ukazovateľová aritmetika

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

Upozornenie

slido slido.com
2537503
PrPr – P5

1. počítačový test 1
   - príklady na zapracovanie cyklov, funkcií, práca s txt súborom

2. časť projektu (14 bodov)
   - zadaná v 5. týždni (16.10.)
   - odovzdanie v 7. týždni (5.11. do 23:59)
     - neskoré odovzdanie 8. týždeň (12.11. do 23:59),
       penalizácia => ***uznáva sa iba 80% zo získaných bodov***
   - za projekt musí získať študent **min. 4 body** (akceptovateľný)

IDstudenta_Rok_projekt_1.c

Opakovanie

Opakovanie - Definícia ukazovateľa

- špecifikujeme typ ukazovateľa
- pred menom premennej je *

```
int i;
int *p_i;
```

je ekvivalentné

```
int i, *p_i;
```

iba premenná **a** je ukazovateľ

```
int *a, b;
```

- Pozor na neinicializovaný ukazovateľ
  - Ukazuje na náhodné miesto v pamäti
  - Pred prvým použitím je potrebná jeho inicializácia

Opakovanie - Ukazovateľ, ktorý nikam neukazuje

- nulový ukazovateľ: **`NULL`** neobsahuje žiadnu platnú adresu – neukazuje na žiadne konkrétne miesto v pamäti, s ktorým chceme pracovať.
- **`NULL`** - symbolická konštanta definovaná v **`stdio.h`**:
  - `#define NULL 0`
  - `#define NULL ((void *) 0)`
- je možné priradiť ho ukazovateľom na ľubovoľný typ

```
int *p;
p = NULL;
```

```
if (p == NULL)
...
```

Opakovanie - Ukazovateľ typu `void`

- v čase definície ukazovateľa nie je zrejmé na aký typ premennej bude ukazovať - použijeme generický ukazovateľ
- môže ukazovať na ľubovoľný typ
- pred konkrétnym použitím generického ukazovateľa je potrebné pretypovanie na typ konkrétnej premennej, na ktorú bude ukazovať.

```
void *p;
```

```
int i;
float f;
void *p_void = &i;
*(int *)p_void = 2;
p_void = &f;
*(float *)p_void = 3.5;
```

Opakovanie - Ukazovateľ

slido slido.com
2537503
PrPr – P5

Ukazovateľ (**na dáta**):

```
int i = -10;
int *p_i = &i;

*p_i = 10;
```

Ukazovateľ (**na funkciu**):

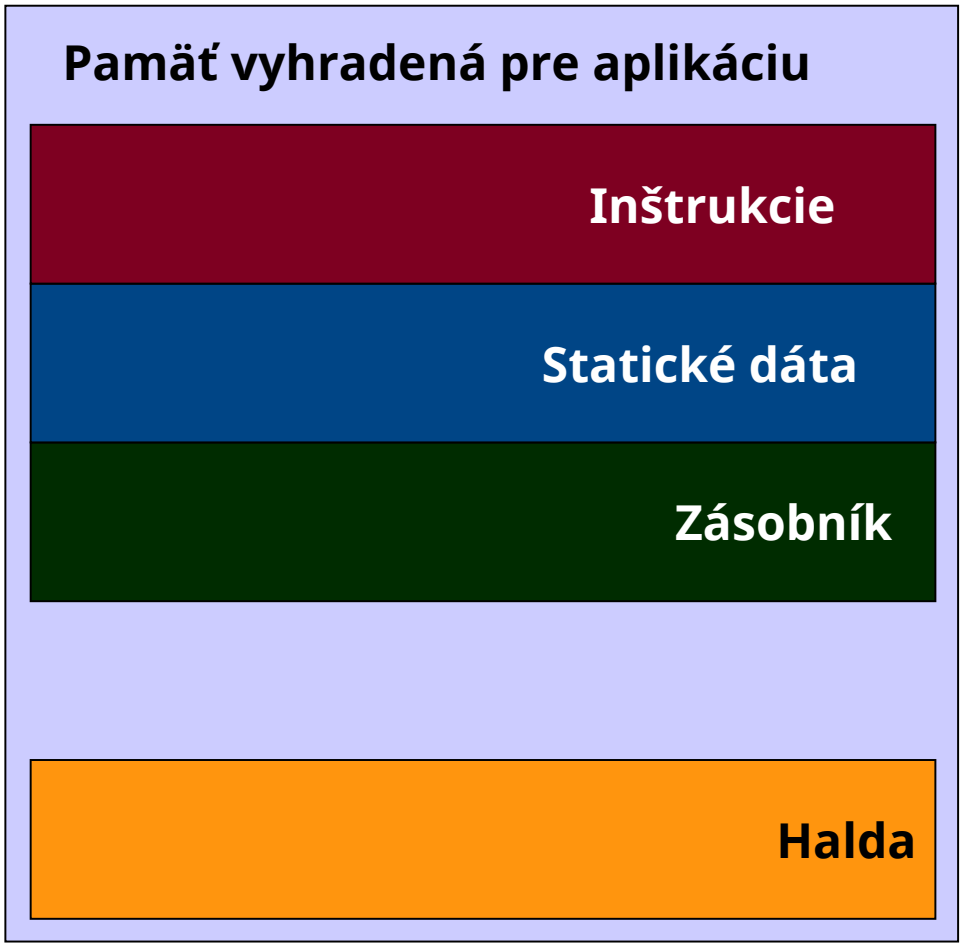
```
void (*p_funkciu)(int);
p_funkciu = &foo;

(*p_vypis)(10);
```

Alokácia pamäte

slido.com
2537503
PrPr – P5

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



Halda

slido slido.com
2537503
PrPr – P5

https://stack

Alokácia pamäte

slido slido.com
2537503
PrPr – P5

- 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

slido slido.com
2537503
PrPr – P5

Statická alokácia

Statická alokácia pamäte

- prekladač pozná vopred pamäťové nároky
  - napr. dve premenné typu **`double`** a jednu premennú typu **`char`**
- prekladač sám určí požiadavky pre všetky definované premenné a pri spustení programu sa pre ne alokuje miesto
- počas vykonávania programu sa nemanipuluje s touto pamäťou

Statická alokácia pamäte

- vymedzuje miesto v dátovej oblasti
- globálne premenné - statické
- nie vždy to stačí
  - napr. rekurzia alebo do pamäte potrebujeme načítať obsah súboru
  - použiť dynamickú alokáciu, alebo vymedzenie pamäte v zásobníku

Vymedzenie pamäte v zásobníku

slido slido.com
2537503
PrPr – P5

- zaisťuje kompilátor pri volaní funkcie
- väčšina lokálnych premenných definovaných vo funkciách
- existencia týchto premenných začína pri vstupe do funkcie a končí pri výstupe z funkcie
- ak chceme prenášať hodnotu premennej medzi jednotlivými volaniami funkcie - nemôže byť premenná alokovaná v zásobníku

slido.com
2537503
PrPr – P5

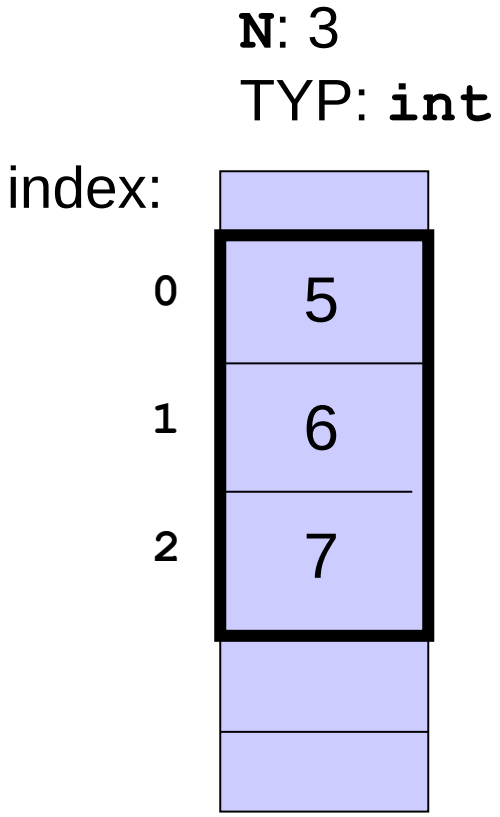
Statické jednorozmerné pole

Základy práce s poliami

- pole je štruktúra zložená z niekoľkých prvkov rovnakého typu (blok prvkov)

```
TYP x[N];
```

statická definícia poľa



- pole obsahuje **N** prvkov
- dolná hranica je vždy **0**
  - $\Rightarrow$ horná hranica je **N-1**
- číslo **N** musí byť známe v čase prekladu
- hodnoty nie sú inicializované na 0
- hranica pola nie je kontrolovaná

Príklady definícií statického poľa

definícia konštanty (inak sa nejedná o statické pole!)

```
#define N 10

int x[N], y[N+1], z[N*2];
```

x má 10 prvkov poľa, od indexu 0 po index 9
y má 11 prvkov poľa, od indexu 0 po index 10
z má 20 prvkov poľa, od indexu 0 po index 19

Prístup k prvkom poľa

```
#define N 10

...
   int x[N], i;

   x[0] = 1;

   for (i = 0; i < N; i++)
      x[i] = i+1;

   for (i = 0; i < N; i++)
      printf("x[%d]: %d\n", i, x[i]);
```

priradenie hodnoty do prvého prvku poľa

v cykle priradenie hodnoty postupne všetkým prvkom poľa

výpis prvkov poľa

Prístup k prvkom poľa

Kompilátor nekontroluje rozsah hodnôt (range-checking) t.j. či index je mimo rozsahu poľa

```
x[10] = 22;
```

- program sa skompiluje, ale hodnota 22 sa zapíše na zlé miesto v pamäti
- prepísanie obsahu iných premenných
- prepísanie časti kódu

Inicializácia poľa

```
int A[3] = { 1, 2, 3 };

int B[4]; //spravne
B[4]={ 1, 2, 3, 4 }; //nespravne
B[0]=1; B[1]=2; B[2]=3; B[3]=4; //spravne

double C[5]={5.1, 6.9};
double C[5]={0};
```

pole tu nedefinujeme

meníme prvky existujúceho pola

jednoduchá inicializácia všetkých prvkov na 0

inicializuje C[0]=5.1, C[1]=6.9 a C[2]..C[4]=0

Porovnávanie poľa

- nie je možné vykonať pomocou operátora ==
- neporovná sa obsah poľa, ale adresa
- meno poľa bez [] vracia adresu na začiatok poľa (väčšinou prvý prvok – závisí od kompilátora)
- treba vykonať prvok po prvku

```
for (int i = 0; i < N; i++) {
      if(A[i] != B[i]) {
          return 0; // false
      }
  }
return 1; // true
```

Kopírovanie poľa

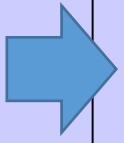
- nie je možné vykonať pomocou operátora =
- neskompiluje sa
- treba vykonať prvok po prvku

```
for (int i = 0; i < N; i++) {
      B[i] = A[i];
  }
```

Statické pole – histogram písmen

slido slido.com # 2537503 PrPr – P5

```
#include <stdio.h>
#include <stdlib.h>
 #define N ('Z' - 'A' + 1)  /* 90 – 65 + 1*/

int main()
{
    int I, hist[N]; char  slovo[100];

   scanf("%s", slovo);      /* nacitanie slova */
   for (i = 0; i < N; i++)      /* inicializacia hist */
         hist[i] = 0;             /* naplnenie hist */

  i = 0;
 while ( (i < 100)  &&  (slovo[i] != '\0' ) )
 {
    hist[toupper(slovo[i]) - 'A']++;
    i++;
 }
```



```
   for (i = 0; i < N; i++)      /* vypis hist */
       if ( hist[i] != 0 )
           printf("%c: %d\n", i+'A', hist[i] );

   return(0);
}
```

Statické pole – pole ako parameter funkcie


```
int maximum( int pole[],  int n )
{
   int i,  max = pole[0];
   for (i = 1; i < n; i++)
       if (pole[i] > max)
              max = pole[i];
    return max;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define N 15

int main()
{
    int i;  int cisla[N];

    for (i = 0; i < N; i++)
          scanf(" %d", & csila[i])     /* naplnenie pola*/

   printf("Maximum je: %d", maximum(cisla, N) ) ;
   return(0);
}
```

Statické pole – pole ako parameter funkcie


```
int maximum( int pole[],  int n )
{
   int i,  max = pole[0];
   for (i = 1; i < n; i++)
       if (pole[i] > max)
              max = pole[i];
    return max;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define N 15

int main()
{
    int i;  int cisla[N];

    for (i = 0; i < N; i++)
          scanf(" %d", & csila[i])     /* naplnenie pol
   printf("Maximum je: %d", maximum(cisla, N) ) ;
   return(0);
}
```

parameter sa dá vo funkcii meniť (lebo sa vytvorila jeho lokálna kópia (nezáleží na tom, či ide o statické alebo dynamické pole)

int maximum(int pole[15])

slido slido.com
2537503
PrPr – P5

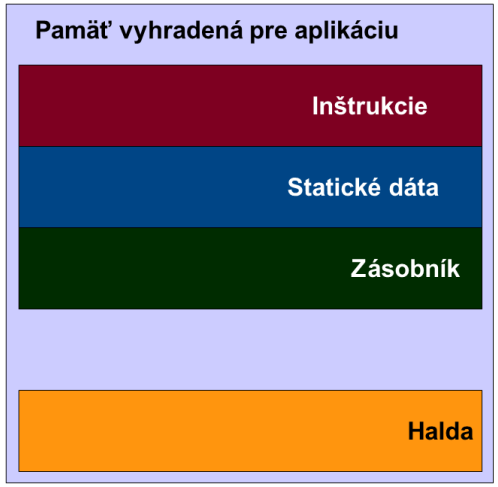
Dynamická alokácia

Dynamická alokácia

- vymedzenie pamäte v halde (heap)
- za behu programu dynamicky prideliť (alokovať) oblasť pamäte určitej veľkosti
- pristupuje sa do nej prostredníctvom ukazovateľov

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



Statická a dynamická aplokácia

slido slido.com
2537503
PrPr – P5

Statická alokácia vymedzuje pamäť na zásobníku

- Automaticky sa uvoľní po dokončení bloku kódu
  - Koniec funkcie, cyklu…
- Výrazne rýchlejšia
  - Na zásobníku nevzniká fragmentácia, ľahké uvoľnenie
  - Vrchol zásobníka je v cache
- Krátkodobé premenné

Dynamická alokácia vymedzuje pamäť na halde

- Existuje do explicitného uvolnenia (alebo do skončenia programu)
- Dlhodobé premenné

Funkcie pre dynamickú alokáciu pamäte

```
void *malloc(size_t n)
```

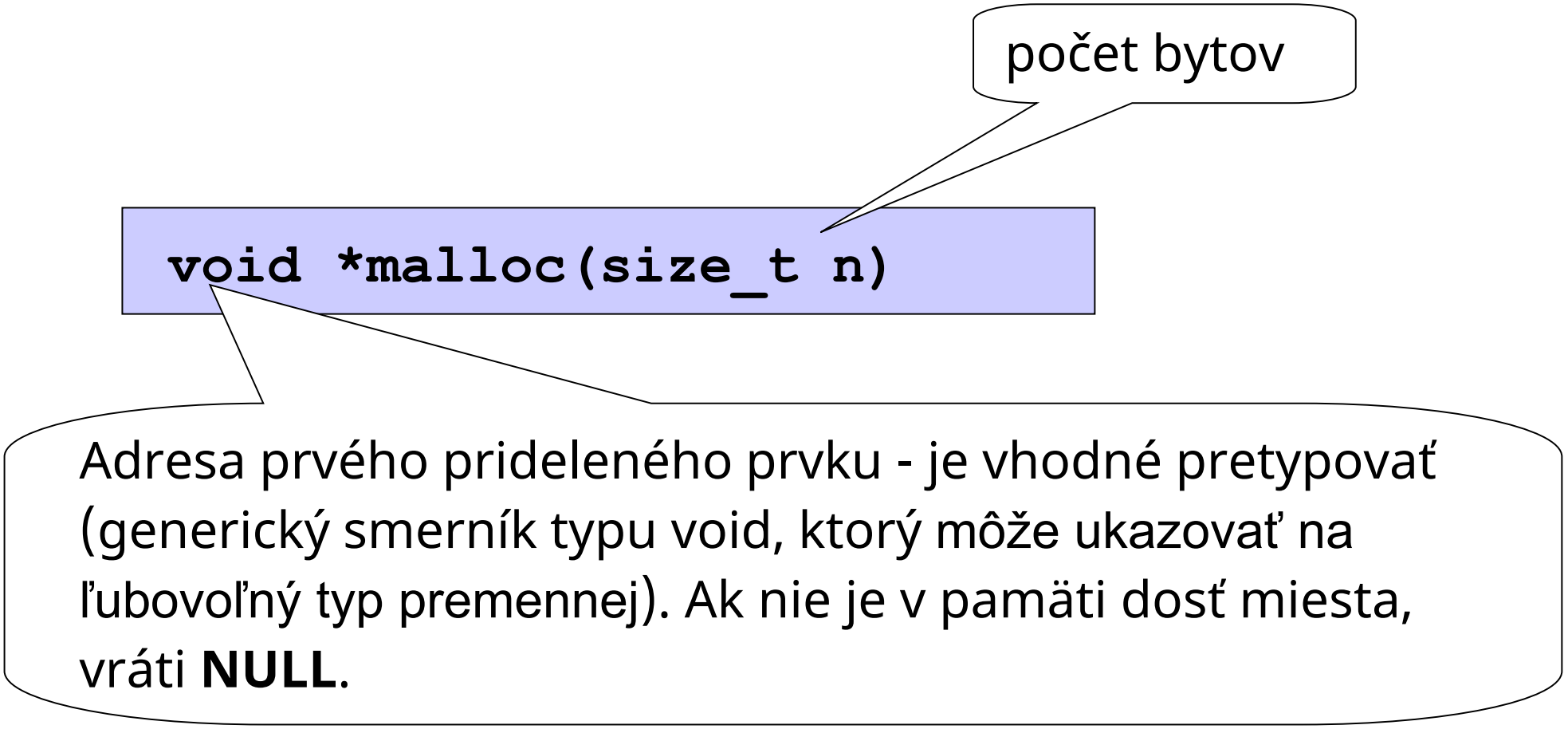
- Alokuje pamäť veľkosti n bytov
- Jeden súvislý blok (ako pole)
- Na halde
- Pamäť **nie je** inicializovaná (pozor na „neporiadok“)
- Ak je alokácia neúspešná, vráti NULL

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

```
void *calloc(size_t n,size_t size_item)
```

- Ako malloc
- Alokuje pamäť veľkosti n * size_item
- Inicializuje pamäť na 0

Prideľovanie pamäte



Testovanie pridelenia pamäte

slido slido.com
2537503
PrPr – P5

Kontrola, či **`malloc()`** pridelil pamäť:

Dynamický priestor pre 5 celých čísiel

```
int * p_i;

if((p_i = (int *) malloc(5 *sizeof(int))) == NULL)
{
    printf("Nepodarilo sa pridelit pamat\n");
    exit;
}
```

Uvoľnenie alokovanej pamäte

- Uvoľní alokovanú pamäť na halde
- Funguje na ukazovateľ vrátený z malloc(), calloc() alebo realloc()
  - *nie na hocijaký ukazovateľ (napr. z ukazovateľovej aritmetiky)*
- Prístup k pamäti po zavolaní free()
  - Pamäť môže byť pridelená inej premennej aj keď na ňu ukazovateľ stále ukazuje
  - Use-after-free (exploits)

```
void free(void *)
```

- Free() nemaže obsah pamäti
  - Citlivé údaje je potrebné pre uvoľnením zmazať
- Je vhodné nastaviť uvoľnený ukazovateľ na NULL
  - Opakované uvoľnenie je v poriadku, ak je argument NULL
  - Prístup na adresu je síce nevalidný, ale neprepisujú sa žiadne dáta

Uvoľnenie alokovanej pamäte

- nepotrebnú pamäť je vhodné ihneď vrátiť operačnému systému

```
char *p_c;

p_c = (char *) malloc(1000 * sizeof(char));
/* p_c=(char *)calloc(1000, sizeof(char));*/
...
free(p_c);
p_c = NULL;
```

Zmena veľkosti alokovanej pamäte

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

- prt == NULL, ako malloc()
- prt != NULL, zmení veľkosť akolovaného pamäťového bloku na hodnotu size
- size == 0, závisí od kompilátora (očakávame free(), ale nemusí to tak vždy byť)
- Obsah pôvodného pamäťového bloku je zachovaný
  - Ak je nová veľkosť väčšia ako pôvodná
  - Inak sa pamäťový blok skráti
- Pri zväčšení je dodatočná pamäť neinicializovaná
  - Tak isto ako pri malloc()

Rýchle nastavenie pamäte


```
void *memset(void *ptr,int value,size_t num);
```

- Nastaví pamäť na zadanú hodnotu (0/-1)
- Výrazne rýchlejšie ako inicializácia cez cyklus
- Pracuje na úrovni bytov
  - Často sa používa spolu so sizeof()

```
#include <string.h>
```

```
int array[10];
memset(array, 0, 10*sizeof(int));
```

Dynamická alokácia pamäte

1. Alokovať potrebný počet bytov
   - Zvyčajne ako `počet_prvkov*sizeof(typ_prvku)`
2. Uložiť adresu pamäte do ukazovateľa daného typu
   - `int *array = (int *) malloc(5 *sizeof(int)));`
3. Uvolniť alokovanú pamäť
   - `free(array);`

Operátor sizeof

- zistí veľkosť dátového typu alebo objektu v bytoch
- vyhodnotí sa v čase prekladu (nezdržuje beh)

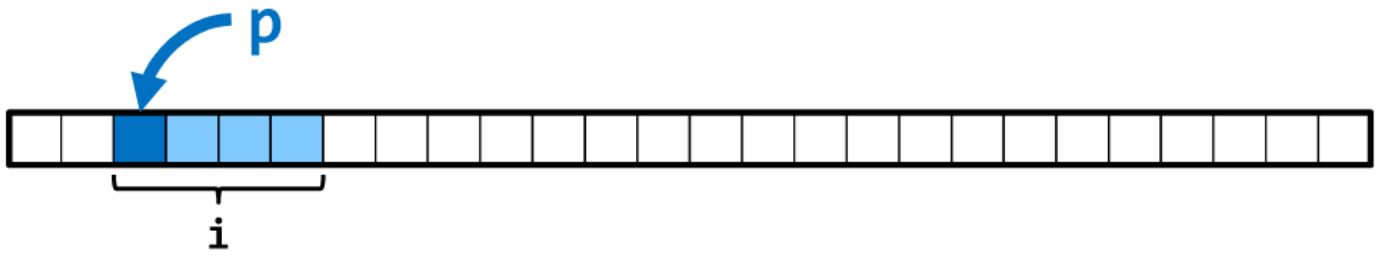
```
int i, *p_i;
i = sizeof(*p_i);
```

počet bytov potrebných na uloženie typu **`int`** – využíva sa často

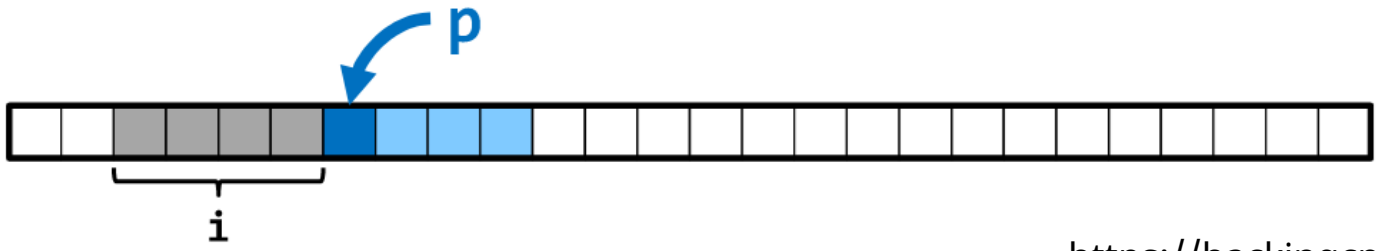
Ukazateľová aritmetika

- `sizeof(int) == 4 byte`

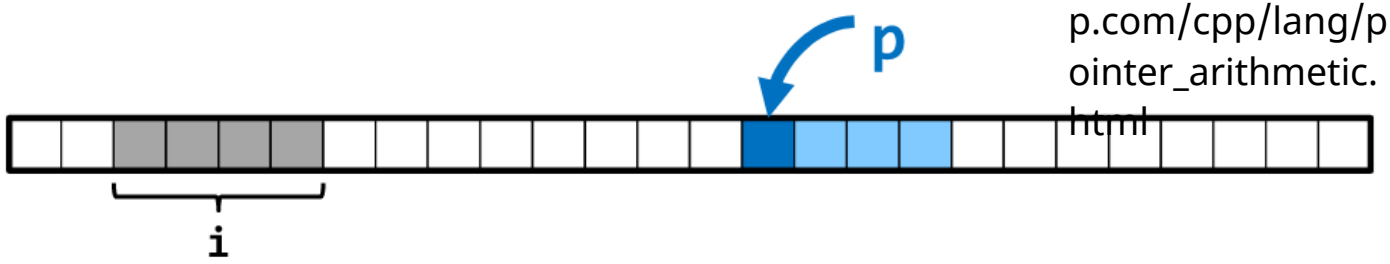
```
int i = 5;
int* p = &i;
```




```
p = &i + 1;
```




```
p += 2;
```



https://hackingcpp.com/cpp/lang/pointer_arithmetic.html

Ukazateľová aritmetika

Vieme:

```
sizeof(char)   == 1
sizeof(int)    == 4
sizeof(double) == 8
```

```
  char c, *p_c=&c; //&c 10
   int i, *p_i=&i; //&i 20
double f, *p_f =&f;//&f 30
```

Potom platí:

```
p_c + 1 == 11
p_i + 1 == 24
p_f + 1 == 38
```

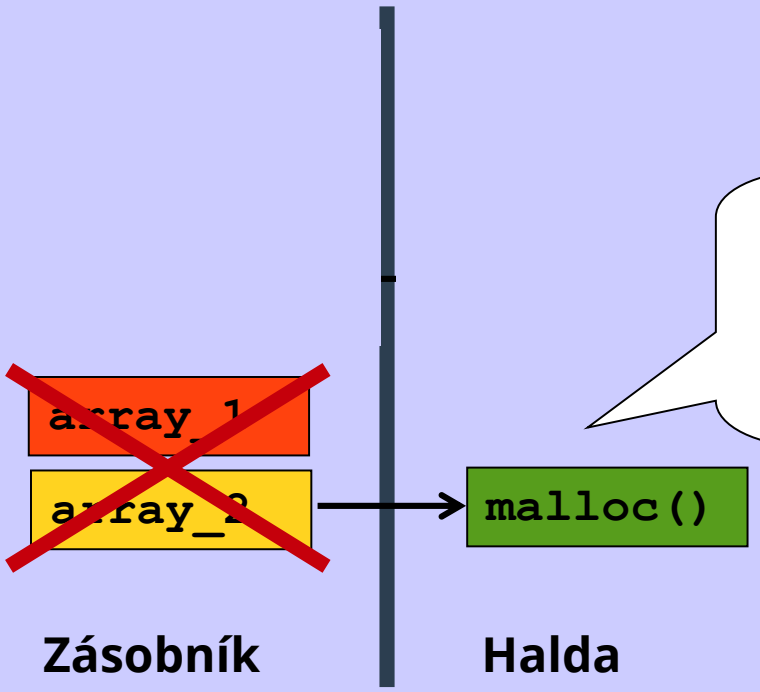
Porovnávanie ukazovateľov s konštantou NULL

- bez explicitného pretypovávania
- **`p = NULL`**
  - neukazuje na žiadne zmysluplné miesto v pamäti

```
int n, *p;
...
if (n >= 0)
    p = alokuj(n);
else
   p = NULL;
...
if (p != NULL)
  ...
```

Statická a dynamická alokácia pamäte

```
void procedura()
{
    int array_1[10];
    int *array_2 = (int *) malloc (10*sizeof(int));
    ...
}

int main(void)
{
    ...
    procedura();
    ...
    procedura();
    ...
    return 0;
}
```

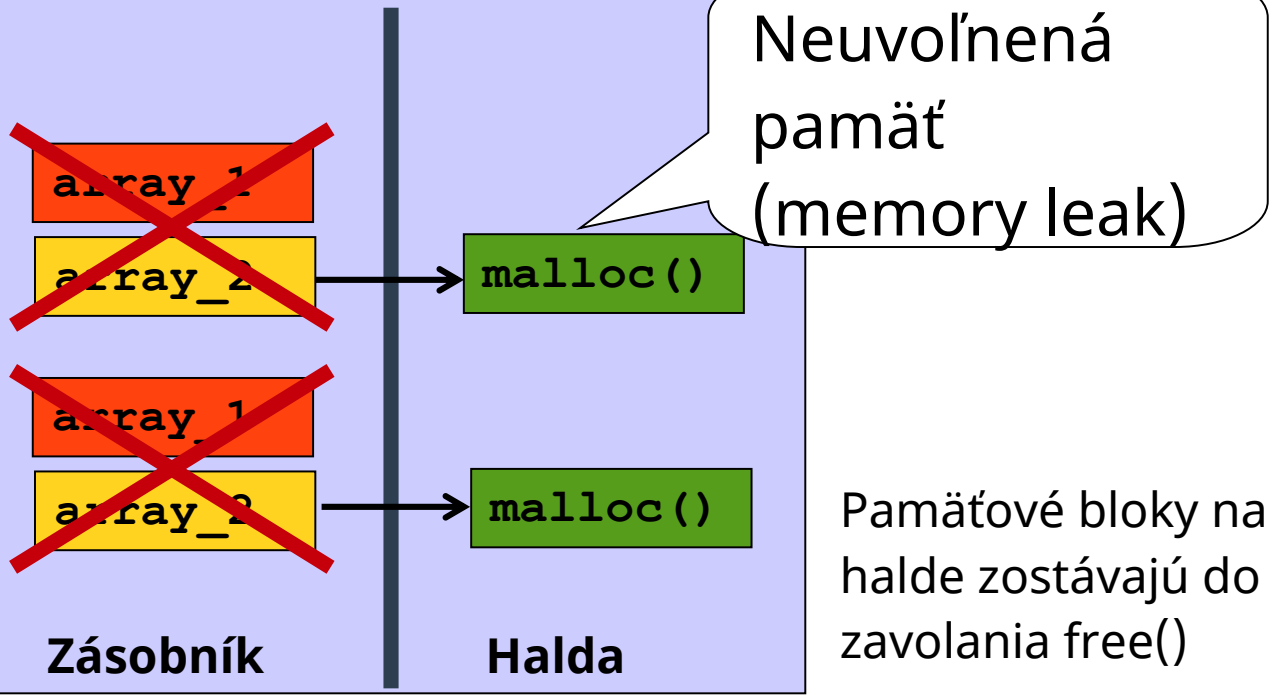


Neuvoľnená pamäť (memory leak)

Pamäťové bloky na halde zostávajú do zavolania free()

Statická a dynamická alokácia pamäte

```
void procedura()
{
    int array_1[10];
    int *array_2 = (int *) malloc (10*sizeof(int));
    ...
}

int main(void)
{
    ...
    procedura();
    ...
    procedura();
    ...
    return 0;
}
```



Neuvoľnená pamäť (memory leak)

Pamäťové bloky na halde zostávajú do zavolania free()

Neuvoľnená pamäť

- Dynamicky alokovaná pamäť sa musí uvoľniť
  - Explicitne programátorom
  - C nemá garbage collector
- Valgrind – nástroj na detekciu neuvoľnenej pamäti
  - **`valgrind -v –leak-check=full`** `./testovaný_program`
  - Eclipse Valgrind plugin

http://www.valgrind.org/

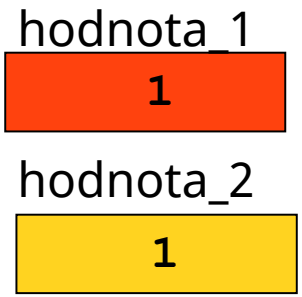
- Microsoft Visual Studio
  - Automaticky zobrazuje neuvoľnenú pamäť v debug režime
  - `_CrtDumpMemoryLeaks();` #include <crtdbg.h>

Parameter funkcie: Hodnota vs ukazovateľ

slido slido.com # 2537503 PrPr – P5

Lokálne premenné funkcie zaniknú (hodnota, ukazovateľ), ale zápis do hodnota_2 zostane

```
void predanieHodnoty(int hodnota, int *ukazovatel)
{
    hodnota = 5;
    *ukazovatel = 7;
}

int main()
{
    int hodnota_1 = 1;
    int hodnota_2  = 1;
    predanieHodnoty(hodnota_1, &hodnota_2);
    return 0;
}
```

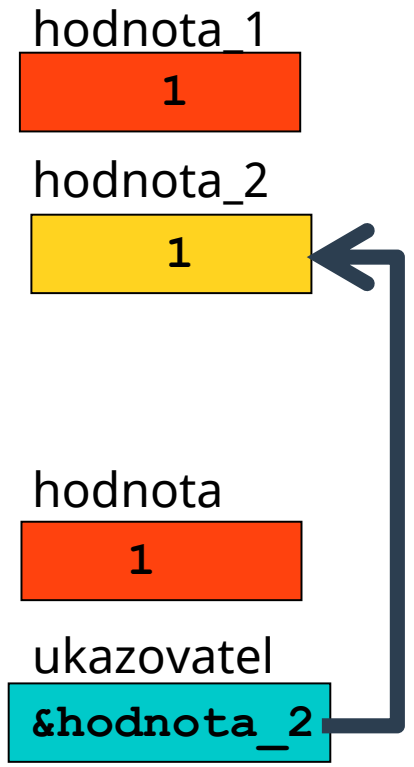


Zásobník

Parameter funkcie: Hodnota vs ukazovateľ

Lokálne premenné funkcie zaniknú (hodnota, ukazovateľ), ale zápis do hodnota_2 zostane

```
void predanieHodnoty(int hodnota, int *ukazovatel)
{
    hodnota = 5;
    *ukazovatel = 7;
}

int main()
{
    int hodnota_1 = 1;
    int hodnota_2  = 1;
    predanieHodnoty(hodnota_1, &hodnota_2);
    return 0;
}
```

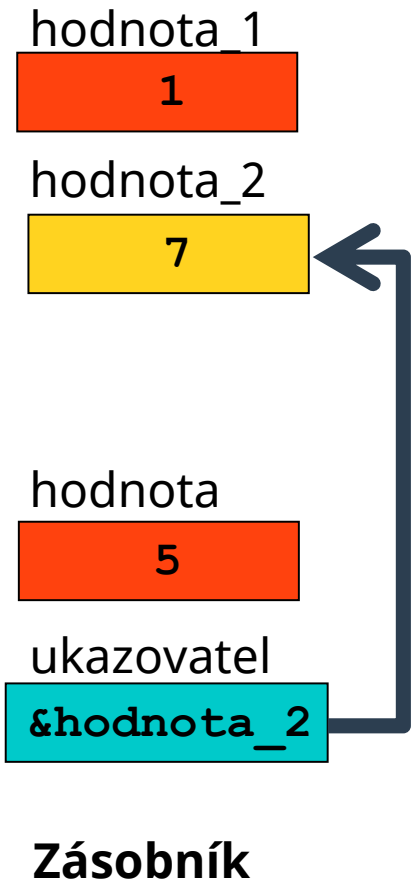


Parameter funkcie: Hodnota vs ukazovateľ

slido slido.com # 2537503 PrPr – P5

Lokálne premenné funkcie zaniknú (hodnota, ukazovateľ), ale zápis do hodnota_2 zostane

```
void predanieHodnoty(int hodnota, int *ukazovatel)
{
    hodnota = 5;
    *ukazovatel = 7;
}

int main()
{
    int hodnota_1 = 1;
    int hodnota_2  = 1;
    predanieHodnoty(hodnota_1, &hodnota_2);
    return 0;
}
```

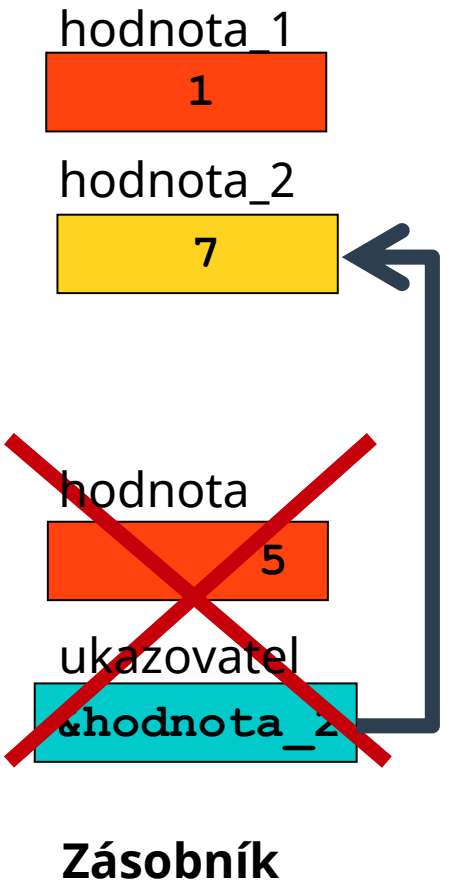


Parameter funkcie: Hodnota vs ukazovateľ

slido slido.com # 2537503 PrPr – P5

Lokálne premenné funkcie zaniknú (hodnota, ukazovateľ), ale zápis do hodnota_2 zostane

```
void predanieHodnoty(int hodnota, int *ukazovatel)
{
    hodnota = 5;
    *ukazovatel = 7;
}

int main()
{
    int hodnota_1 = 1;
    int hodnota_2  = 1;
    predanieHodnoty(hodnota_1, &hodnota_2);
    return 0;
}
```



Zásobník

Parameter funkcie: Hodnota vs ukazovateľ

slido slido.com # 2537503 PrPr – P5

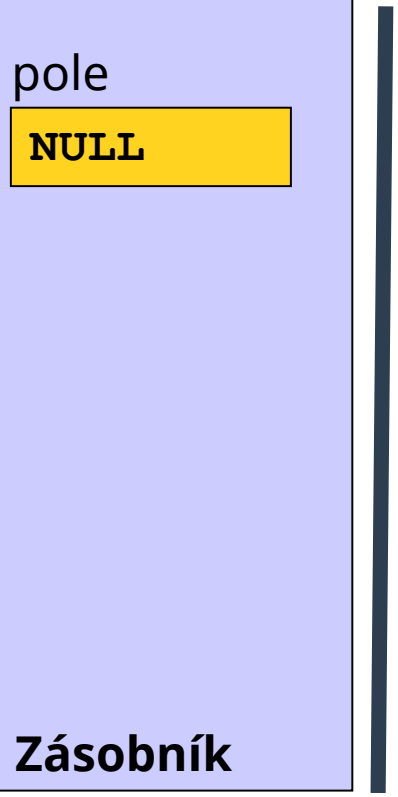
Lokálne premenné funkcie zaniknú (hodnota, ukazovateľ), ale zápis do hodnota_2 zostane

```
void predanieHodnoty(int hodnota, int *ukazovatel)
{
    hodnota = 5;
    *ukazovatel = 7;
}

int main()
{
    int hodnota_1 = 1;
    int hodnota_2  = 1;
        int *p_h2 = &hodnota_2;
    predanieHodnoty(hodnota_1, p_h2);
    return 0;
}
```

Predanie dynamického poľa funkciou

Ukazovateľ na ukazovateľ

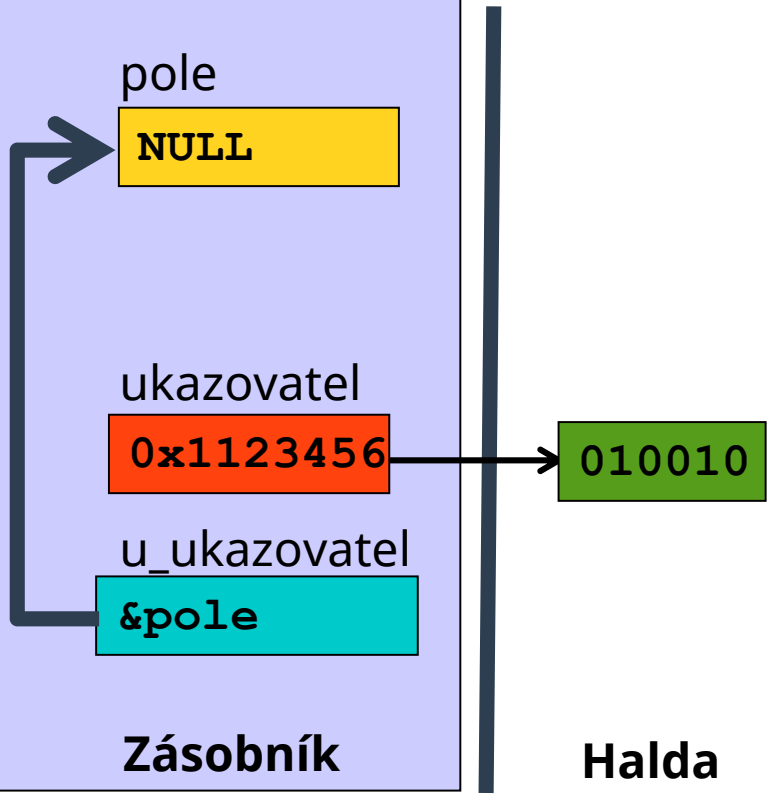
```
void alokujPole(int *ukazovatel, int **u_ukazovatel)
{
    // adresa sa stratí, neuvolnena pamat
    ukazovatel = malloc(10*sizeof(int));
    // warning: integer from pointer
    *ukazovatel = malloc(10 *sizeof(int));
    // OK
    *u_ukazovatel = malloc(10*sizeof(int));
}

int main()
{
    int *pole = NULL;
    alokujPole(pole, &pole);
    free(pole);
    return 0;
}
```



Predanie dynamického poľa funkciou

Ukazovateľ na ukazovateľ

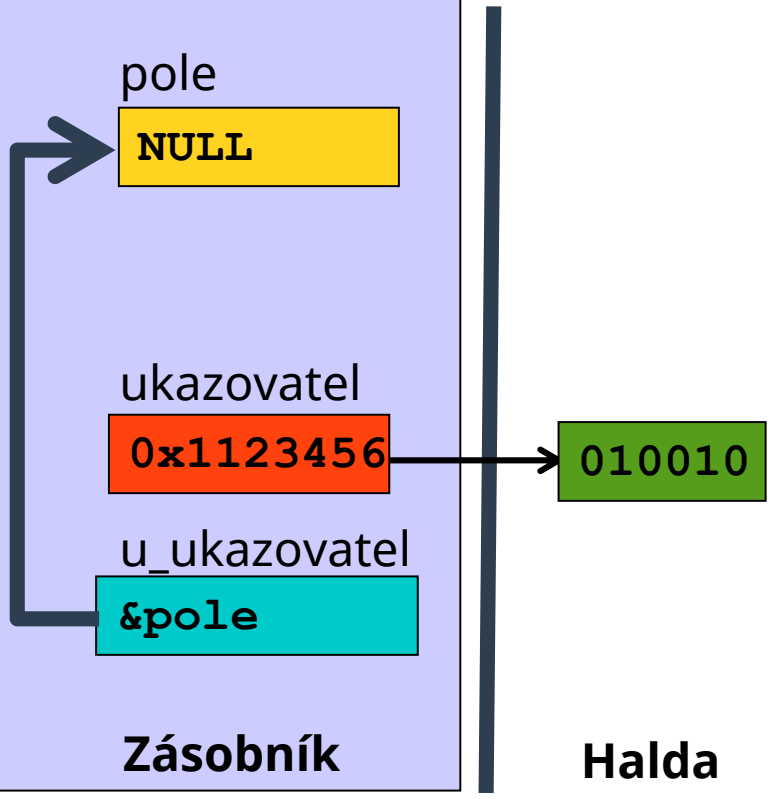
```
void alokujPole(int *ukazovatel, int **u_ukazovatel)
{
    // adresa sa stratí, neuvolnena pamat
    ukazovatel = malloc(10*sizeof(int));
    // warning: integer from pointer
    *ukazovatel = malloc(10 *sizeof(int));
    // OK
    *u_ukazovatel = malloc(10*sizeof(int));
}

int main()
{
    int *pole = NULL;
    alokujPole(pole, &pole);
    free(pole);
    return 0;
}
```



Predanie dynamického poľa funkciou

Ukazovateľ na ukazovateľ

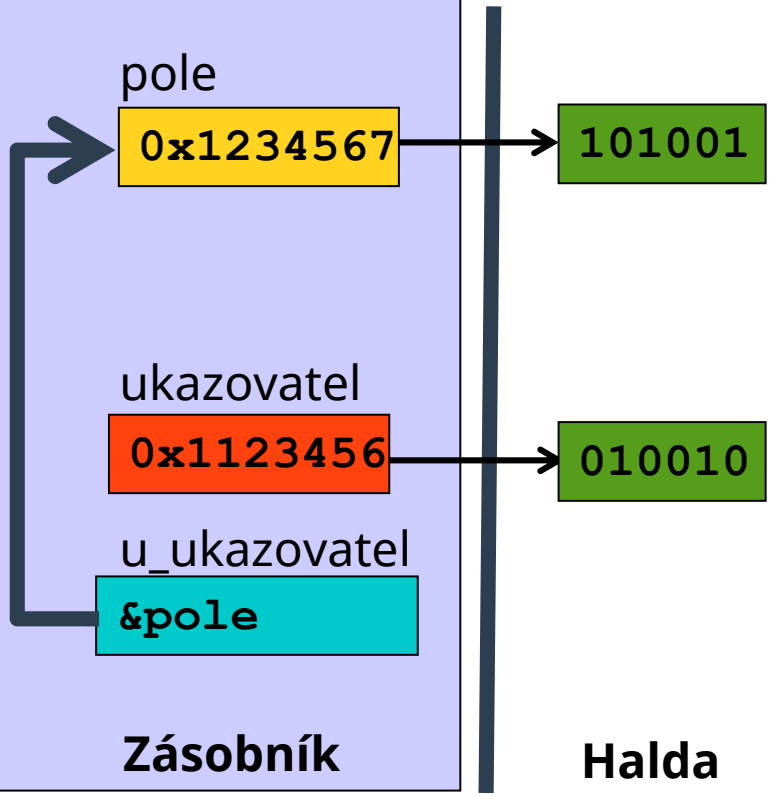
```
void alokujPole(int *ukazovatel, int **u_ukazovatel)
{
    // adresa sa stratí, neuvolnena pamat
    ukazovatel = malloc(10*sizeof(int));
    // warning: integer from pointer
    *ukazovatel = malloc(10 *sizeof(int));
    // OK
    *u_ukazovatel = malloc(10*sizeof(int));
}

int main()
{
    int *pole = NULL;
    alokujPole(pole, &pole);
    free(pole);
    return 0;
}
```



Predanie dynamického poľa funkciou

Ukazovateľ na ukazovateľ

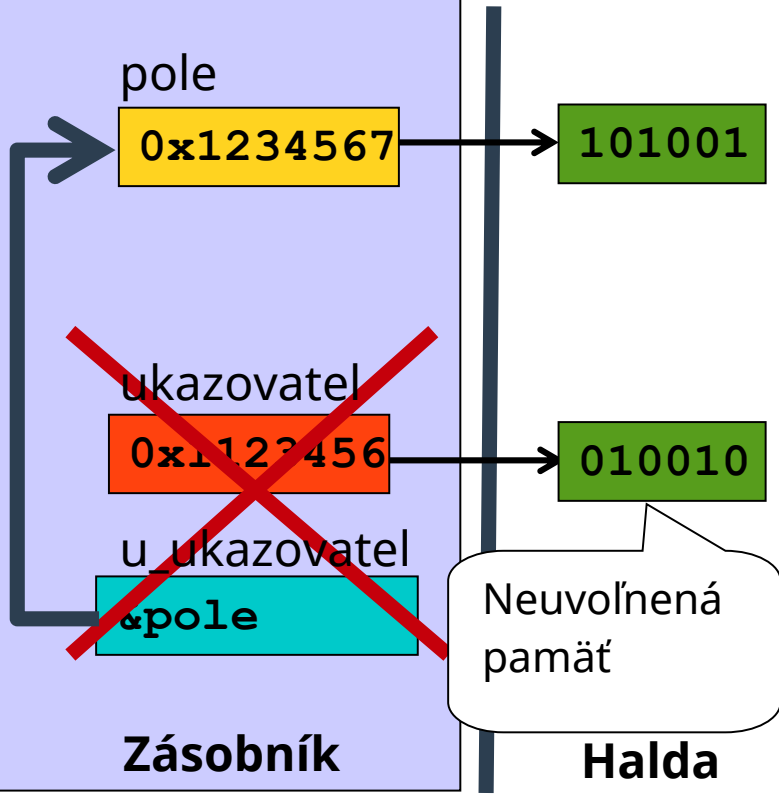
```
void alokujPole(int *ukazovatel, int **u_ukazovatel)
{
    // adresa sa stratí, neuvolnena pamat
    ukazovatel = malloc(10*sizeof(int));
    // warning: integer from pointer
    *ukazovatel = malloc(10 *sizeof(int));
    // OK
    *u_ukazovatel = malloc(10*sizeof(int));
}

int main()
{
    int *pole = NULL;
    alokujPole(pole, &pole);
    free(pole);
    return 0;
}
```



Predanie dynamického poľa funkciou

Ukazovateľ na ukazovateľ

```
void alokujPole(int *ukazovatel, int **u_ukazovatel)
{
    // adresa sa stratí, neuvolnena pamat
    ukazovatel = malloc(10*sizeof(int));
    // warning: integer from pointer
    *ukazovatel = malloc(10 *sizeof(int));
    // OK
    *u_ukazovatel = malloc(10*sizeof(int));
}

int main()
{
    int *pole = NULL;
    alokujPole(pole, &pole);
    free(pole);
    return 0;
}
```



Predanie dynamického poľa funkciou

Ukazovateľ na ukazovateľ

```
void alokujPole(int *ukazovatel, int **u_ukazovatel)
{
    // adresa sa stratí, neuvolnena pamat
    ukazovatel = malloc(10*sizeof(int));
    // warning: integer from pointer
    *ukazovatel = malloc(10 *sizeof(int));
    // OK
    *u_ukazovatel = malloc(10*sizeof(int));
}

int main()
{
    int *pole = NULL;
    int **p_pole = &pole;
    alokujPole(pole, p_pole);
    free(pole);
    return 0;
}
```

slido slido.com
2537503
PrPr – P5

Ukazovateľová aritmetika

Ukazovateľová aritmetika

- aritmetické operácie nad ukazovateľmi
- založené na:
  - `pole[X]` je definované ako `*(pole + X)`
- realizované **na úrovni prvkov**
  - bie bytov
  - majme `pole` typu `int*`
    - pripočíta sa `X * sizeof(int) bytov`
- adresa začiatku pola je priradená do ukazovateľa

Ukazateľová aritmetika

- s ukazovateľmi je možné vykonávať nasledovné aritmetické operácie:
  - súčet ukazovateľa a celého čísla
  - rozdiel ukazovateľa a celého čísla
  - porovnávanie ukazovateľov rovnakého typu
  - rozdiel dvoch ukazovateľov rovnakého typu
- majú zmysel len v rámci bloku dynamicky vytvorenej pamäte (POLIA)
- ostatné operácie nedávajú zmysel
  - OS nezaručí, že neskôr alokovaný blok bude na vyššej adrese

Operátor sizeof

- zistí veľkosť dátového typu alebo objektu v bytoch
- vyhodnotí sa v čase prekladu (nezdržuje beh)

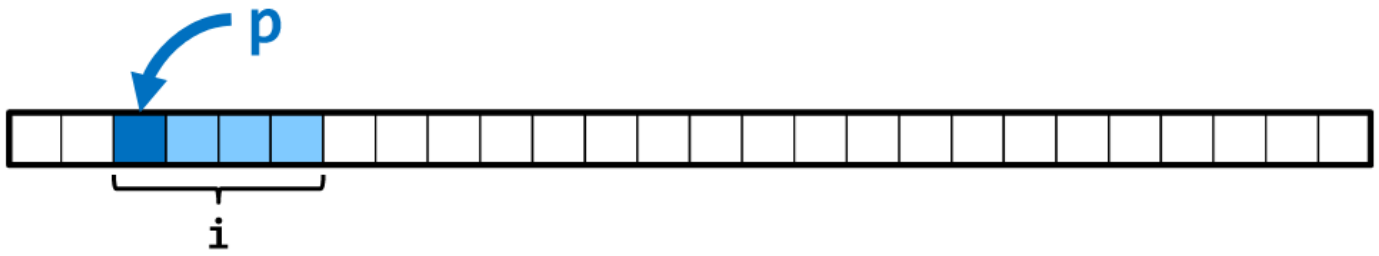
```
int i, *p_i;
i = sizeof(*p_i);
```

počet bytov potrebných na uloženie typu **`int`** – využíva sa často

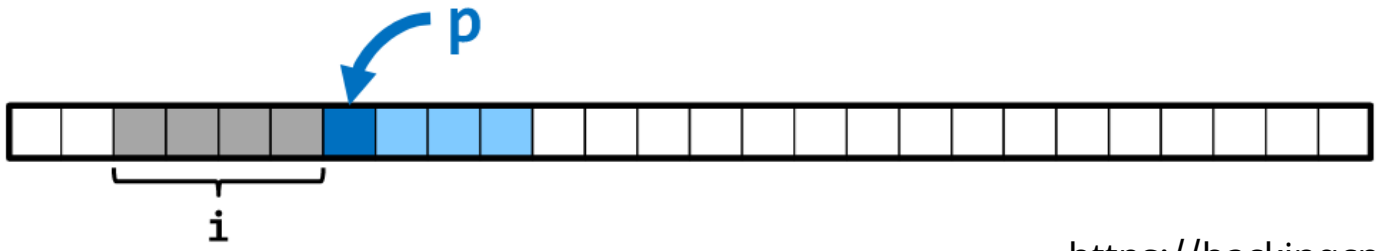
Ukazateľová aritmetika

- `sizeof(int) == 4 byte`

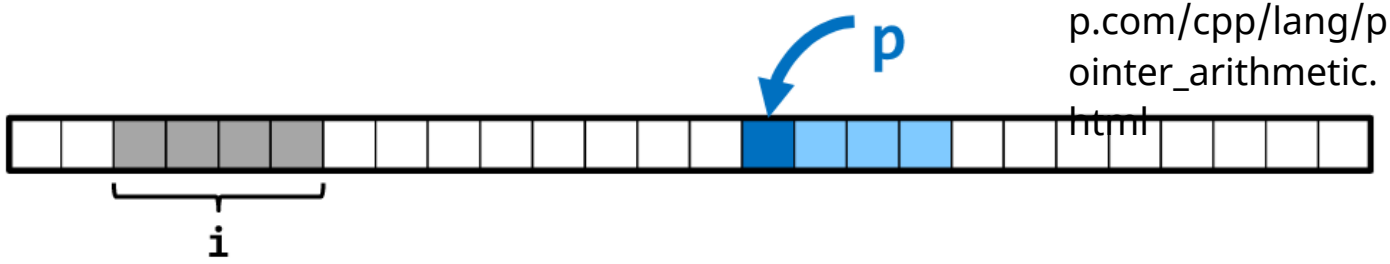
```
int i = 5;
int* p = &i;
```




```
p = &i + 1;
```




```
p += 2;
```



https://hackingcpp.com/cpp/lang/pointer_arithmetic.html

Ukazateľová aritmetika

Vieme:

```
sizeof(char)   == 1
sizeof(int)    == 4
sizeof(double) == 8
```

```
  char c, *p_c=&c; //&c 10
   int i, *p_i=&i; //&i 20
double f, *p_f =&f;//&f 30
```

Potom platí:

```
p_c + 1 == 11
p_i + 1 == 24
p_f + 1 == 38
```

Porovnávanie ukazovateľov

- operátory: `<` `<=` `>` `>=` `==` `!=`
- porovnávanie má zmysel len keď ukazovatele:
  - sú rovnakého typu
  - ukazujú na ten istý úsek pamäte
- výsledok porovnania:
  - ak je podmienka splnená: 1
  - inak: 0

Porovnávanie ukazovateľov s konštantou NULL

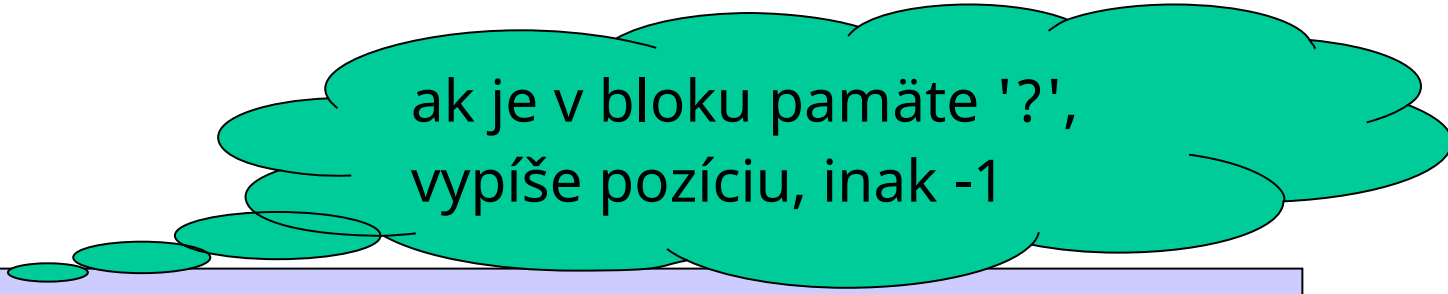
- bez explicitného pretypovávania
- **`p = NULL`**
  - neukazuje na žiadne zmysluplné miesto v pamäti

```
int n, *p;
...
if (n >= 0)
    p = alokuj(n);
else
   p = NULL;
...
if (p != NULL)
  ...
```

Rozdiel dvoch ukazovateľov rovnakého typu

```
int n, *p1, *p2;
...
n = p1 - p2;
```

```
 n = ((int *) p1 - (int *) p2) / sizeof(*p1);
```




```
char *p1, *p2 , str[5]={'A','B','c','?','o'};
p1 = str;
for (p2=p1; (p2-p1)<5 && *p2 != '?'; p2++)
   ;
printf("%d", (p2 < p1+5) ? (p2-p1+1) :-1);
```

slido slido.com
2537503
PrPr – P5

Polia a ukazovatele

Polia a ukazovatele

`TYP x[N];` definícia statického poľa

`TYP *p_x;` definícia smerníka

`x` a `p_x` sú smerníky na typ `TYP`

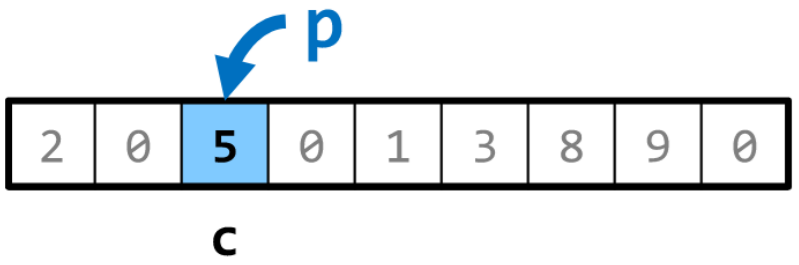
Porovnanie:

- **`x`** je konštantný smerník jeho hodnota sa nedá meniť
- **`x`** je adresa začiatku bloku pamäti alokovaného pre statické pole. (statické pole počas behu programu nemôže meniť svoju polohu v pamäti)
- **`p_x`** je smerník s neurčenou počiatočnou hodnotou (zatiaľ nie je inicializovaný, neukazuje na konkrétnu oblasť pamäte), alokuje pamäť iba pre seba

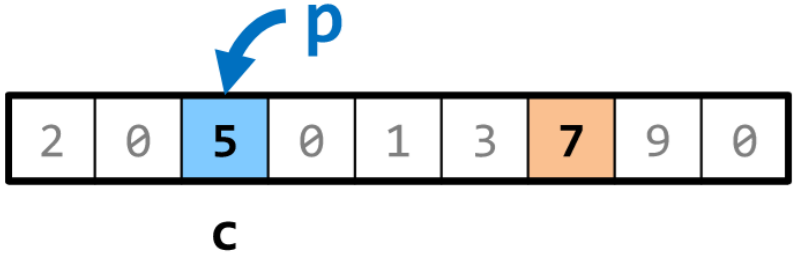
Operátor []

- p[n] == hodnota na adrese ukazovateľa + n

```
char c='5';
char *p =&c;
```

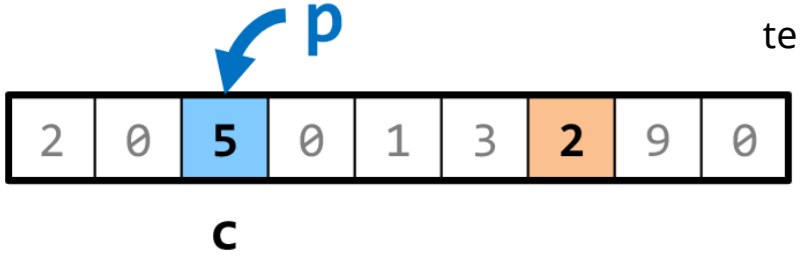



```
*(p+4) ='7';
```



https://hackingcpp.com/cpp/lang/pointer_arithmetic.html

```
p[4] = '2';
```



Polia a ukazovatele

```
int *p;
p = (int *) malloc(4 * sizeof(int));
```

- `*p` je smerník na blok pamäti alokovanej pomocou funkcie `malloc()`
- `p` je dynamické pole, ktoré vzniká v čase behu programu

Platí:

- `p[0] == *p`
- `p[1] == *(p + 1)`
- `p[2] == *(p + 2)`
- `p[3] == *(p + 3)`

Rozdiel medzi statickými a dynamickými poliami je v definícii a v spôsobe prideľovania pamäte

Polia a ukazovatele

Ak sú definované `int x[1],*p_x;`

- `x` je statický ukazovateľ, nemôže byť zmenený, ale
- `*x` je obsah staticky alokovanej pamäti.

  t.j. `*x = 2;` je to isté ako `x[0] = 2;`
  - smerník `p_x` nie je inicializovaný
  - priradenie `*p_x = 2;` je staticky správne, ale dynamicky chybné (chceme meniť obsah na neznámej adrese, ktorá nebola alokovaná)
  - `p_x = x;` – ukazujú na rovnakú adresu v pamäti (dynamický smerník nasmerujeme na začiatok statického poľa – ok.)
  - `x = p_x;` – chybné priradenie, x je konštantný smerník jeho hodnotu nemôžeme meniť (chceme zmeniť začiatok statického poľa na adresu kde ukazuje smerník – statické pole nemôžeme v pamäti premiestňovať)

Polia a ukazovatele - operátor []

- p[n] == hodnota na adrese ukazovateľa + n

```
#define N 10

...
   int x[N], i;

   for (i = 0; i < N; i++)
       x[i] = i+1;

   for (i = 0; i < N; i++)
      printf("x[%d]: %d\n", i, x[i]);

   for (i = 0; i < N; i++)
      printf("x[%d]: %d\n", i, *(x+i));
```

```
X[0] = 1
X[1] = 2
X[2] = 3
X[3] = 4
X[4] = 5
X[5] = 6
X[6] = 7
X[7] = 8
X[8] = 9
X[9] = 10
X[0] = 1
X[1] = 2
X[2] = 3
X[3] = 4
X[4] = 5
X[5] = 6
X[6] = 7
X[7] = 8
X[8] = 9
X[9] = 10
```

Veľkosť statického a dynamického ukazovateľa na pole

```
int x[10], *p_x;
p_x = (int *) malloc (10 * sizeof(int));
```

- po alokovaní pamäte pre **p_x** budú **x** aj **p_x** ukazovatele na pole 10 prvkov typu **int**, s rozdielom, že:
  - **x** je statický ukazovateľ
  - **p_x** je dynamický ukazovateľ
- preto dáva **sizeof()** iné výsledky:
  - **sizeof(x) == 10 * sizeof(int)** (napr. 40)
  - **sizeof(p_x) == sizeof(int *)** (napr. 8)

Pole meniace svoju veľkosť

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

- pomocou funkcie **`realloc()`** definovaná v **`stdlib.h`**
- zmenší **`pole`**, alebo vytvorí nové (väčšie) pole a prekopíruje tam hodnoty z pôvodného poľa

```
x = realloc(x, 10 * n * sizeof(int));
```

Pole ako parameter funkcie

slido slido.com
2537503
PrPr – P5

```
int maximum(int pole[], int n)
```

vo funkcii sa nedá zistiť veľkosť poľa

aj keď:

```
int maximum(int pole[10], int n)
```

parameter bude stále považovaný za `pole[]`

Pole ako parameter funkcie

volanie odkazom: odovzdá sa adresa začiatku poľa

`int pole[]` je ekvivalentné `int *pole`

Pri použití `int pole[]` je jasnejšie, že ide o pole a nie o ukazovateľ na `int`.

Volanie funkcie s poľom ako parametrom:

```
max = maximum(pole, 10);
```

Pole ako parameter funkcie

```
int maximum(int *pole, int n) {...}
```

dá sa použiť aj na zistenie maxima napr. na zistenie maxima 3. až 7. prvku

```
int x[10];
max = maximum(&x[2], 5);
```

Pole ako parameter funkcie

slido slido.com
2537503
PrPr – P5

- pole môže byť parametrom funkcie
- skutočný parameter sa do funkcie odovzdáva odkazom
- pomocou mena poľa sa odovzdá adresa začiatku poľa
- prvky poľa môžeme vo funkcii meniť a táto zmena je trvalá - pracujeme s originálom poľa, nie s jeho lokálnou kópiou

Typické problémy pri práci s poľom

slido slido.com
2537503
PrPr – P5

- Zápis do poľa bez špecifikácie miesta
  - `int pole[10]; pole=1;`
  - Do premennej typu pole sa nedá priradiť hodnota (narozdiel od ukazovateľa)
- Zápis mimo poľa
  - Prvok N+1 (najčastejší problém)
    - `int pole[N]; pole[N]=1;`
    - V C sa pole indexuje od 0
  - Zápis za koniec pola, alebo pred začiatok pola
    - Chyba v ukazovateľovej aritmetike, iteračnej premennej cyklu…
  - Práca s nepridelenou pamäťou môže spôsobiť pád programu, alebo nežiadúcu zmenu dát, ktoré sa na danom mieste nachádzajú

Pole ukazovateľov

prvkami poľa môžu byť aj ukazovatele

- na prvky → viacrozmerné polia
- na funkcie (všetky funkcie musia byť toho istého typu)

```
void (*funkcia[5])() = {file, edit,
search, compile, run} ;
...
funkcia[1]();
```

Dynamická alokácia

- Pri uvoľnení nastavte premennú späť na NULL
  - Opakované uvoľnenie nie je problém
  - Správnosť ukazovateľa sa nedá testovať, NULL áno
- Dynamicky alokovanú pamäť priraďujeme do ukazovateľa
  - `sizeof(ukazovatel)` vracia veľkosť ukazovateľa, nie poľa
- Dynamická alokácia (a jej uvoľnenie) nerobí nič s pamäťou
  - Neinicializovaná pamäť
  - Zabudnuté dáta (heslá...)

Dynamická a statická alokácia

slido slido.com
2537503
PrPr – P5

Pamäť alokovaná v dobe prekladu s pevnou dĺžkou v statickej časti

- Konštanty, reťazce, konštantné pole
- `const int n = 10;` (neskôr)
- Dĺžka známa v dobe prekladu
- Alokované v statickej sekcii programu (nachádzajú sa v nespustenom programe)

Pamäť alokovaná za behu na zásobníku, dĺžka známa v dobe prekladu

- Lokálne premenná, lokálne pole
- `int pole[10];`
- Pamäť je alokovaná a uvoľnená automaticky

Pamäť alokovaná za behu na halde, dĺžka nie je známa v dobe prekladu

- Alokácia a uvoľnenie explicitne pomocou funkcií malloc a free
- Neuvoľnená pamäť
- `int *pole=malloc(velkost*sizeof(int)); free(pole);`

Veľkosť poľa

- uchovávať v (celočíselnej) premennej
  - pri odovzdaní poľa ako argument funkcie treba dve premenné: smerník na adresu začiatku poľa a veľkosť poľa
- Dynamické pole
  - **`sizeof(*pole)`** vráti koľko bytov treba na uloženie jedného prvku poľa (napr. 4 byty v prípade poľa celých čísiel).
  - **`sizeof(pole)`** vráti koľko bytov treba na uloženie samotného ukazovateľa `pole` -> to je 8 bytov (respektíve 4 byty na 32-bitovom OS).
- Statické pole
  - **`sizeof(pole)`** vráti veľkosť poľa v bytoch
  - po predaní poľa cez argument funkcie informácia o jeho veľkosti sa stráca

Ďakujem vám za pozornosť!

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