

# Základy procedurálneho programovania 1

## Práca s pamäťou, viacrozmerné polia

7. 11. 2016

zimný semester  
2016/2017

# Vyhradenie pamäte pre premenné

---

- Aké poznáme spôsoby vyhradenia pamäte?
- Pre premenné program automaticky vyhradí pamäť podľa rozsahu platnosti
  - Premenná (vo funkcií) – tzv. **lokálna premenná**  
rozsah platnosti je od deklarácie až po koniec bloku, v ktorom bola deklarovaná
  - **Globálna premenná**  
rozsah platnosti je od deklarácie až po koniec súboru, v ktorom bola deklarovaná

# Alokácia

---

- Programovacie jazyky umožňujú vyhradiť pamäť pre dáta aj manuálnym spôsobom
- Používame nato dve funkcie:
  - **ALOKUJ** – vyhradí pamäť
  - **DEALOKUJ** – uvoľní pamäť
- Blok pamäte, ktorý manuálne alokujeme zostane vyhradený až kým ho manuálne neuvoľníme

# Alokácia – vstupné a výstupné argumenty

## ■ ALOKUJ – vyhradí pamäť


- Vstupný argument?  
Koľko byte-ov chceme alokovať
- Výstupný argument – návratová hodnota  
Kde v pamäti sa miesto vyhradilo – t.j. adresa pamäte
- Príklad:

**x ← ALOKUJ(4)**

**y ← ALOKUJ(4)**

PRINT x            ... 41234

PRINT y            ... 41238



Aj zdanlivo rovnaké  
volania vrátia iné  
hodnoty, pretože sa  
vždy alokuje nový  
blok pamäte

# Alokácia – otázky a odpovede

---

- Keď si chcem v programe vyhradiť miesto pre int, koľko byte-ov musím alokovať?  
ALOKUJ(4)
- Keď si chcem vyhradiť miesto pre reťazec s desiatimi znakmi, koľko byte-ov musím alokovať?  
ALOKUJ(11)
- Keď si chcem vyhradiť miesto pre pole desiatich intov, koľko byte-ov musím alokovať?  
ALOKUJ (10\*4)  
ALOKUJ (40)

# Dealokácia – vstupné a výstupné argumenty

## ▪ DEALOKUJ – uvoľní pamäť

- Vstupný argument?  
Čo chceme uvoľniť – t.j. adresa pamäte, ktorú sme predtým alokovali
- Výstupný argument?  
Nič.
- Príklad:

**x ← ALOKUJ(4)**

**y ← ALOKUJ(4)**

PRINT x            ... 41234

PRINT y            ... 41238

**DEALOKUJ(41234)**

Vyhradené 4 byte na adrese 41234 už nie sú pre nás vyhradené (pamäť zostane existovať, ale už nie je pre nás a teda by sme do nej nemali ani zapisovať ani pristupovať (čítať jej obsah))

# Dealokácia – otázky a odpovede

---

- Je potrebné, aby nás funkcia DEALOKUJ informovala, či sa pamäť podarilo úspešne uvoľniť?

**Nie.** Keď predpokladáme fungujúcu a správnu implementáciu funkcie DEALOKUJ v operačnom systéme, tak volaním funkcie pre uvoľnenie pamäte nič nepožadujeme, ale naopak darujeme počítaču zdroje späť, a počítač ich s radosťou prijme.

- Je potrebná do funkcie DEALOKUJ poslať aj veľkosť vyhradenej pamäte, ktorú chceme uvoľniť?

**Nie.** Pamäťový systém si túto informáciu pamätá, aby vedelo všetko pekne fungovať.

# Alokovanie v programovacom jazyku C

---

- Premenná (vo funkcií) – tzv. **lokálna premenná**  
rozsah platnosti je od deklarácie až po koniec bloku, v ktorom bola deklarovaná
- **Globálna premenná**  
rozsah platnosti je od deklarácie až po koniec súboru, v ktorom bola deklarovaná
- Manuálnou alokáciou a dealokáciou  
**ALOKUJ – malloc**  
**DEALOKUJ – free**



# Alokovanie v programovacom jazyku C

---

- Aké poznáme spôsoby vyhradenia pamäte?
  - **Lokálna premenná**
  - **Gobálna premenná**
  - **malloc/free**
  
- Dôležitý rozdiel:
  - Globálna premenná je po vyhradení v pamäte **inicializovaná na nuly** – všetky byte sú vynulovaná
  - Lokálna premenná **nie je inicializovaná** (programátor musí naplniť počiatočnú hodnotu)
  - Manuálne vyhradená pamäť (malloc) **nie je inicializovaná** (calloc) je inicializovaná

# Alokácia v jazyku C – otázky a odpovede

---

- Ked' si chcem v programe vyhradiť miesto pre int, ako to alokujem?  
**malloc(4)**
- Ked' si chcem vyhradiť miesto pre reťazec s desiatimi znakmi, ako to alokujem?  
**malloc(11)**
- Ked' si chcem vyhradiť miesto pre pole desiatich intov, ako to alokujem?  
**malloc(10\*4)**  
**malloc(40)**
- Na rôznych platformách môžu mať dátové typy inú veľkosť (32 vs. 64 bit) – operátor **sizeof**
  - Ako vyhradiť miesto pre pole desiatich intov?  
**malloc(10\*sizeof(int))**



# Pozrime sa do manuálu

---

- <http://man7.org/linux/man-pages/man3/malloc.3.html>

## NAME [top](#)

`malloc, free, calloc, realloc` - allocate and free dynamic memory

## SYNOPSIS [top](#)

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- ALOKUJ – vyhradí pamäť
  - Vstupný argument?  
**Kolko byte-ov chceme alokovať**
  - Výstupný argument – návratová hodnota  
**Kde v pamäti sa miesto vyhradilo – t.j. adresa pamäte**

# Pozrime sa do manuálu (2)

---

## DESCRIPTION

[top](#)

The **malloc()** function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized.* If *size* is 0, then **malloc()** returns either NULL, or a unique pointer value that can later be successfully passed to **free()**.

The **free()** function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc()**, **calloc()**, or **realloc()**. Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **calloc()** function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc()** returns either NULL, or a unique pointer value that can later be successfully passed to **free()**.

The **realloc()** function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**. If the area pointed to was moved, a *free(ptr)* is done.

# Pozrime sa do manuálu (2)

---

## RETURN VALUE [top](#)

The **malloc()** and **calloc()** functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc()** with a *size* of zero, or by a successful call to **calloc()** with *nmemb* or *size* equal to zero.

The **free()** function returns no value.

The **realloc()** function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from *ptr*, or NULL if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free()** is returned. If **realloc()** fails, the original block is left untouched; it is not freed or moved.

## ERRORS [top](#)

**calloc()**, **malloc()**, and **realloc()** can fail with the following error:

**ENOMEM** Out of memory. Possibly, the application hit the **RLIMIT\_AS** or **RLIMIT\_DATA** limit described in [getrlimit\(2\)](#).

# Pozrime sa do manuálu (3)

---

## ATTRIBUTES [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

Interface	Attribute	Value
<code>malloc()</code> , <code>free()</code> , <code>calloc()</code> , <code>realloc()</code>	Thread safety	MT-Safe

## CONFORMING TO [top](#)

POSIX.1-2001, POSIX.1-2008, C89, C99.

## NOTES [top](#)

By default, Linux follows an optimistic memory allocation strategy. This means that when `malloc()` returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of `/proc/sys/vm/overcommit_memory` and `/proc/sys/vm/oom_adj` in [proc\(5\)](#), and the Linux kernel source file *Documentation/vm/overcommit-accounting*.

# Pozrime sa do manuálu (4)

---

Normally, `malloc()` allocates memory from the heap, and adjusts the size of the heap as required, using `sbrk(2)`. When allocating blocks of memory larger than `MMAP_THRESHOLD` bytes, the glibc `malloc()` implementation allocates the memory as a private anonymous mapping using `mmap(2)`. `MMAP_THRESHOLD` is 128 kB by default, but is adjustable using `mallopt(3)`. Allocations performed using `mmap(2)` are unaffected by the `RLIMIT_DATA` resource limit (see `getrlimit(2)`).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using `brk(2)` or `mmap(2)`), and managed with its own mutexes.

SUSv2 requires `malloc()`, `calloc()`, and `realloc()` to set `errno` to `ENOMEM` upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private `malloc` implementation that does not set `errno`, then certain library routines may fail without having a reason in `errno`.

Crashes in `malloc()`, `calloc()`, `realloc()`, or `free()` are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The `malloc()` implementation is tunable via environment variables; see `mallopt(3)` for details.

# Alokácia v jazyku C – príklad

---

- Napíš program, ktorý malloc-om alokuje miesto pre int, naplní ho konkrétnym číslom (napr. 42) a vypíše ho na obrazovku

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

int main(void)
{
    int *x = (int*) malloc (sizeof(int));
    *x = 42;
    printf("%d\n", *x);
    return 0;
}
```



# Alokácia v jazyku C – príklad

---

- Napíš program, ktorý malloc-om alokuje miesto pre 10 znakov, načíta do vyhradeného miesta reťazec zo vstupu, a vypíše ho na obrazovku

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

int main(void)
{
    char *x = (char*) malloc (11);
    scanf("%s", x);
    printf("%s\n", x);
    return 0;
}
```

# Jednorozmerné pole

---

- Viac rovnakých dátových údajov za sebou
- Rôzne možnosti v programe:

- Jednorozmerné pole The diagram shows a horizontal row of four blue squares representing array elements. To the left of the first square is the letter 'a'. Below each square is its index: 0, 1, 2, and 3.

- Statické:

```
int a[4];
```

- Dynamické:

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

# Viacrozmerne polia

---

## ■ Dvojrozmerné polia:

- Obdĺžnikové:

Statické:

```
int b[2][3];
```

b	0	1	2
0			
1			

Dynamické:

```
int **b = (int**)malloc(2 * sizeof(int*));  
b[0] = (int*)malloc(3 * sizeof(int));  
b[1] = (int*)malloc(3 * sizeof(int));
```

# Viacrozmerne polia

## ■ Dvojrozmerné polia:

- Zubaté:  
(angl. jagged)

Statické: nie je možné

Dynamické:

c	0	1	2	3
0				
1				
2				
3				
4				

```
int **c = (int**)malloc(5 * sizeof(int*));  
c[0] = (int*)malloc(3 * sizeof(int));  
c[1] = (int*)malloc(4 * sizeof(int));  
c[2] = (int*)malloc(2 * sizeof(int));  
c[3] = (int*)malloc(0 * sizeof(int));  
c[4] = (int*)malloc(3 * sizeof(int));
```

# Viacrozmerne polia

## ▪ Trojrozmerné polia:

- Statické:

```
int d[2][2][3];
```

d[0]	0	1	2	d[1]	0	1	2
0				0			
1				1			

- Dynamické:

```
int ***d = (int***)malloc(2 * sizeof(int**));  
d[0] = (int**)malloc(2 * sizeof(int *));  
d[0][0] = (int*)malloc(3 * sizeof(int));  
d[0][1] = (int*)malloc(3 * sizeof(int));  
d[1] = (int**)malloc(2 * sizeof(int *));  
d[1][0] = (int*)malloc(3 * sizeof(int));  
d[1][1] = (int*)malloc(3 * sizeof(int));
```

- Zubaté – podľa potreby 😊



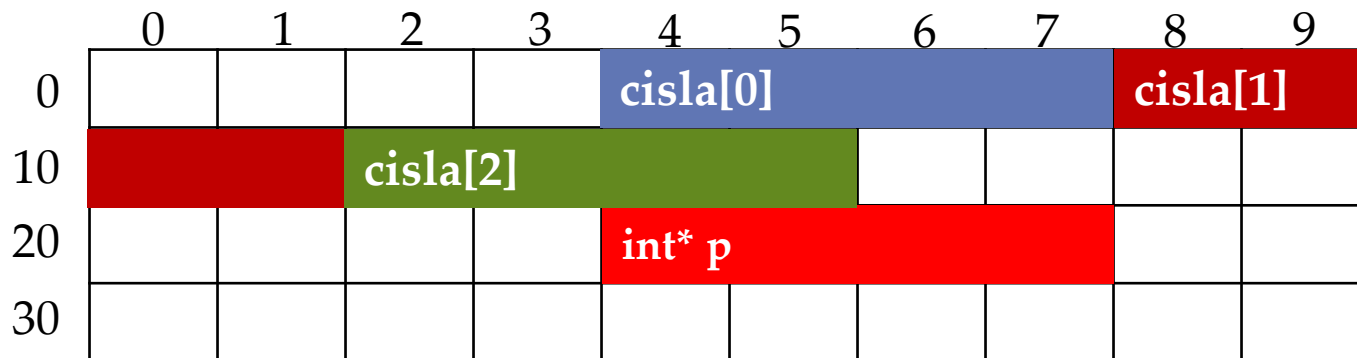
# Zložené dátové typy

---

- Čo ak chceme, aby premenná mohla uschovať **viac** ako poskytuje základný dátový typ?
- Viac rovnakých dátových typov za sebou – **pole**
  - množstvo píšeme v hranatých zátvorkách: **int ciska[100]**
  - k prvkom pristupujeme cez hranaté zátvorky **ciska[7]**, ...
  - zvyčajne prvky číslujeme od 0, prvý prvok je **ciska[0]**
- Viac rôznych dátových typov spolu – **štruktúra**
  - Kľúčové slovo **struct** – definícia štruktúry
  - Pred názvom píšeme struct: **struct Osoba**
  - K prvkom pristupujeme cez bodku: napr. **osoba.vek=30;**

# Opakovanie: Zložené dátové typy – pole

- Viac rovnakých dátových prvkov za sebou v pamäti
- Môžeme k prvkom pristupovať výpočtom presného miesta od začiatku.
- Napr. **int ciska[3]** ...



- Adresa premennej **ciska**? ... Prvý byte v pamäti – 4
- Adresa tretieho čísla? ... **&ciska[2]** ... 12
- Nech: **p=ciska**; **\*(p+1) = 40**; ... kam sa naplní 40?

# Zložené dátové typy – štruktúra

---

- Štruktúra združuje rôzne typy do jedného, a umožňuje s nimi pracovať využitím mien (program mená položiek v štruktúre automaticky prekladá na adresy pamäte podľa toho ako sú položky v pamäti)
- **Definícia** – musíme najskôr zadať definovať ako bude vyzerat', podľa definície sa rozloží v pamäti – zarovná sa podľa veľkosti dátových typov (alignment)

- Napr. komplexné čísla:

```
struct KomplexneCislo {  
    int r;          // celá časť  
    int i;          // imaginárna časť  
}
```

- **Použitie** – `struct KomplexneCislo x;`  
                  `x.r=20; x.i = 5;`



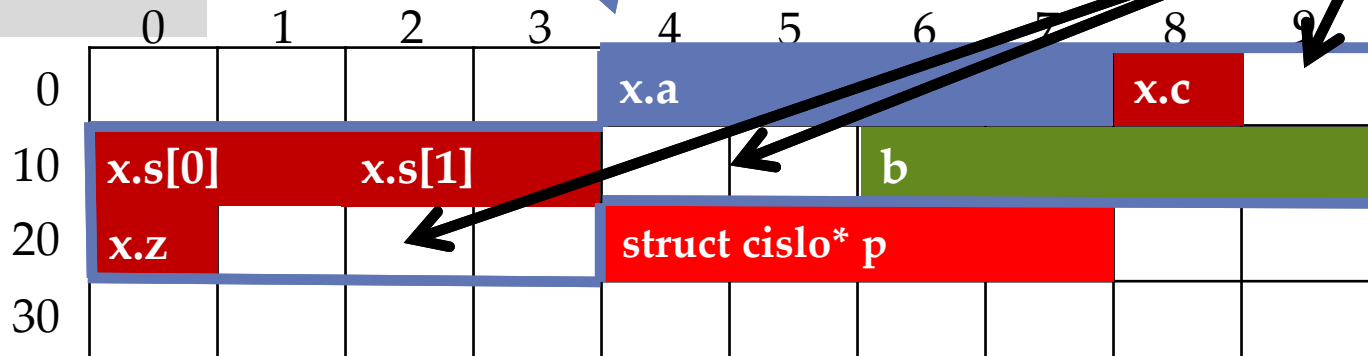
# Zložené dátové typy – štruktúra

```
struct cislo {  
    int a;  
    char c;  
    short s[2];  
    int b;  
    char z;  
}
```

**struct cislo x;**

Modrý okraj: Celková veľkosť  
vyhradenej pamäte pre jednu  
štruktúru typu **struct cislo** je 20 bytes  
(celá musí byť zarovnaná na 4 bytes)

Nevyužívané miesta,  
kvôli zarovnaniu



- Adresa premennej x? ... Prvý byte v pamäti – 4
- Adresa druhého čísla v poli x.s[1]? ... &x.s[1] ... 12
- Nech: p=&x; (\*p).c = 'w'; Pre prístup cez smerník na štruktúru môžeme priamo použiť **šípku** (p->c = 'w')



# typedef – nové pomenovanie pre typ

---

- Do priestoru mien (identifikátorov) zavedie nový názov pre nejaký typ
- Najčastejšie sa používa pre štruktúry, aby sa nemuselo písať „**struct**“

```
typedef struct Osoba
{
    char *meno;
    int vek;
} OSOBA;
```

```
struct Osoba *o;
OSOBA *p;
```

# typedef – nové pomenovanie pre typ

---

- Je možné použiť aj na existujúce jednoduché typy

```
typedef int Cislo;
```

```
Cislo i;
```

# Základy procedurálneho programovania 1

## Programujeme

7. 11. 2016

zimný semester  
2016/2017

# Úloha: Špirála

---

- Nakresli špirálu  $N \times N$ .

- $N=7$ :

#####

. . . . . #

##### . #

# . . . # . #

# . #### . #

# . . . . . #

#####

- Dvojrozmerné pole reťazcov

# Úloha: Špirála – riešenie

---

- Kreslí sa nám ľahko ceruzkou na papier:
  - idem doprava, potom dole, potom doľava, ... striedam smery
- ale určiť konkrétny znak v  $i$ -tom riadku a  $j$ -tom stĺpci je podstatne ťažšie...

# Úloha: Špirála – riešenie (postup)

---

- Znaký komplikovaného obrázku si najskôr určím v dvojrozmernom poli **znak[N][N]** a až nakoniec vykreslím.
- Korytnačia grafika: Keď som aktuálne na nejakej pozícii  $i, j$  sú **štyri základné smery** vykresľovania:
  - **Doprava** ( $\Delta i=0, \Delta j=+1$ )
  - **Dole** ( $\Delta i=+1, \Delta j=0$ )
  - **Doľava** ( $\Delta i=0, \Delta j=-1$ )
  - **Hore** ( $\Delta i=-1, \Delta j=0$ )
- Smery si očísľujem: 0, 1, 2, 3 ...
- Ako vykreslím špirálu využitím týchto smerov?
  - Doprava 7, dole 6, doľava 6, hore 4, doprava 4, ...
  - Smer 0 7, Smer 1 6, Smer 2 6, Smer 3 4, Smer 0 4, ...

# Základy procedurálneho programovania 1

## **Projekt: Bludisko**

(termín odovzdania 29.11.2016)

7. 11. 2016

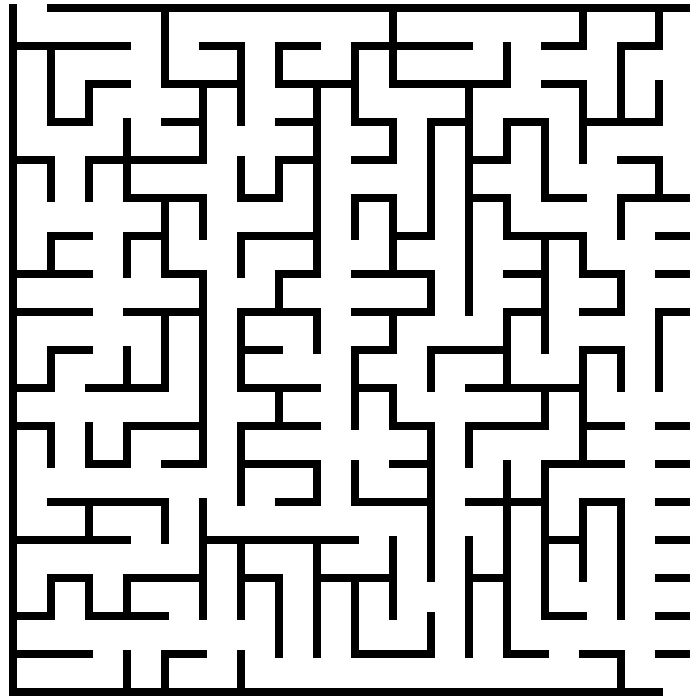
zimný semester  
2016/2017



# Čo je to bludisko

---

- Miestnosť – štvorček so stenami naokolo
- Bludisko – 2D pole miestností



# Čo je to bludisko

---

- Nejde až tak o umenie, ako o logické prepojenie miestností v bludisku tak, aby bolo blúdenie náročné / zaujímavé
- Kedy je bludisko **náročné**?
  - Dlho blúdim – skúšam nejakú vetvu, keď nakoniec zistím, že nie je správna, musím sa vrátiť a skúsiť inú možnosť
- Kedy je bludisko **ľahké**?
  - Skoro hociako idem a hneď prídem do cieľa, nemusím sa vracat' a skúšať inú možnosť

# Čo je to bludisko

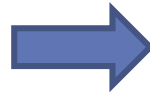
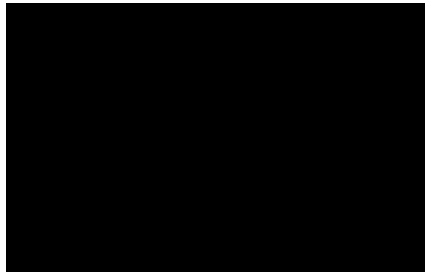
---

- Kedy je bludisko **náročné**?
  - Dlho blúdim – skúšam nejakú vetvu, keď nakoniec zistím, že nie je správna, musím sa vrátiť a skúsiť inú možnosť
  - **Medzi dvoma miestami v bludisku existuje len jedna cesta, a teda nie je ľahké (si na križovatke) zvoliť tú správnu**
- Kedy je bludisko **ľahké**?
  - Skoro hociako idem a hneď prídem do cieľa, nemusím sa vracat' a skúšať inú možnosť
  - **Medzi dvoma miestami v bludisku existuje viacero ciest, a teda sa mi ľahko stane, že si (na križovatke) zvolím nejakú správnu**

# Čo je to bludisko

---

- Reprezentácia bludiska

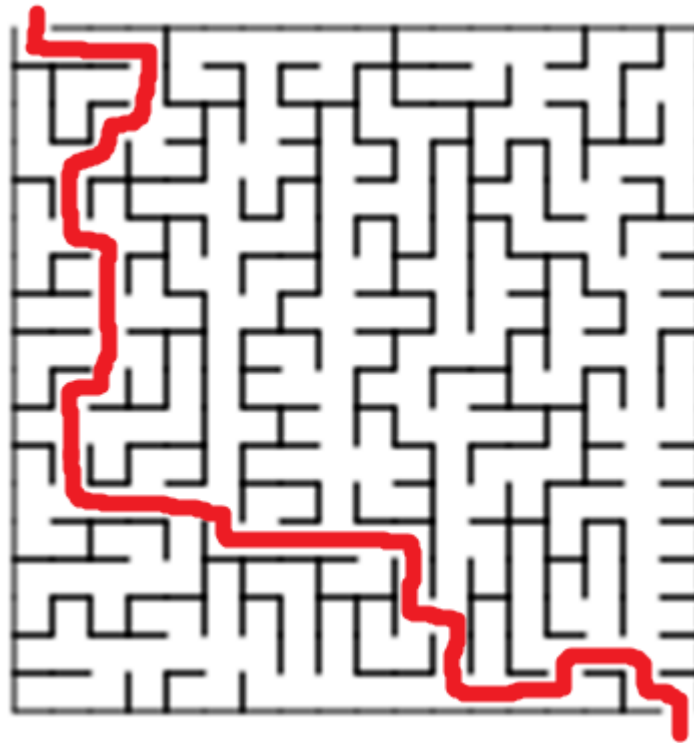


```
# . ###  
# . # . #  
### . #  
. . . . #
```

- 2x2 miestnosti vykreslí bludisko 5x5 (znakov)

# Zadanie projektu

- Dané je bludisko (chodby – znak bodka . a steny – znak mriežka #) **vyznačte prechod z ľavého horného rohu do pravého dolného rohu** znakmi hviezdička \*



# Ukážka riešenia (prechod bludiskom)

```
#.#####  
#...#.#...#.#...#  
#.####.####.#.#.####  
#.#.....#.....#  
#.####.#####.#.####  
#.....#.#...#...#  
#####.#.#.#.#.####  
#.#.....#.#.#.#.  
#.#.####.#####.#.#  
#...#.#...#...#...#  
#.####.#.#####.#  
#...#.....#.....#  
#####.####.#####.#  
#...#...#.#.#...#  
#.####.#####.#.####  
#.....#...#...#  
#####.#####.#
```



```
#*#####  
#*...#.#...#.#...#  
#*####.####.#.#.####  
#*#.....#...***.#  
#*####.#####*#*####  
#*****#.#***#*...#  
#####*#.#*#.#*####  
#.#...*****#.#*#.#  
#.#.####.#####*#.#  
#...#.#...#...#...***#  
#.####.#.#####*#  
#...#.....#...#*#  
#####.####.#####*#  
#...#...#.#.#***#  
#.####.#####.#*####  
#.....#...***#  
#####.#####*#
```

# Zadanie projektu

- Dané je bludisko (chodby – znak bodka . a steny – znak mriežka #) **vyznačte prechod z ľavého horného rohu do pravého dolného rohu** znakmi hviezdička \*
- Termín odovzdania:  
utorok 29.11.2016 13:00

