# Pole, ukazatelé, dynamická alokace paměti

# Pole

- homogenní datová struktura
  - všechny prvky jsou stejného datového typu
  - jednorozměrné
    - analogické aritmetickému vektoru
  - dvourozměrné
    - analogické matici
  - třírozměrné
    - krychle

# Pole

- statické pole –
  - velikost (počet prvků) je dána konstantou při překladu
- dynamické pole
  - velikost je dána až při běhu programu a paměť se alokuje za běhu voláním speciální funkce, lze měnit velikost (realokovat)

# Statické pole

- deklarace

```
Typ identifikator[velikost];
```

```
int a[10];
float pole[30];
```

*Příklad pole* `a`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 8 | 12 | 1 | 8 | 5 | 7 | 9 | -50 |

- přístup k prvkům pomocí indexů
  `a[i], pole[10]`
- rozsah indexů: `0 .. n-1`
- meze polí se nekontrolují

Načítání do pole

- pomocí `scanf:`

```
scanf("%d",&a[i]);
```

- pomocí `cin:`

```
cin >> pole[i];
```

- prvky polí jsou neinicializované, mají náhodnou hodnotu
  - pouze globální podle jsou nulované

- v jazyce C je možnost definovat prvky pole při deklaraci (konstruktor pole):

```
int a[3] = {5,-6,10};
int b[] = {5,-6,10};
int c[10] = {5,-6,10};
```

Napíšeme program, který načte z klávesnice počet zpracovávaných čísel, uloží je do statického pole, čísla vypíše na obrazovku pozpátku a nalezne největší číslo. Pole deklarujeme statické.

Příklad pole_1D_1

```
#define MAX 50
int main(void)
{
  int i,n,max;
  int A[MAX];
  printf("Zadej pocet zadavanych cisel: ");
  scanf("%d",&n);
  if (n>MAX)
  {
   printf("Staticke pole ma pouze %d prvku",MAX);
   return -1;
  }
```

```
  printf("Zadej %d cisel: ",n);
  for(i=0;i<n;i++) scanf("%d",&A[i]);

  printf("Vypis cisel pozpatku:\n");
  for(i=n-1;i>=0;i--) printf("%d ",A[i]);

  /* hledani maxima */
  max = A[0];
  for(i=0;i<n;i++) // lépe for(i=1;i<n;i++)
   if (A[i]>max) max = A[i];

  printf("Nejvetsi cislo je %d.\n",max);
  return 0;
}
```

# Statické pole

- uživatelská definice typu pole

```
typedef int TPole20INT[20];
```

- pak mohu nadeklarovat pole:

```
TPole20INT pole;
```

# Statické pole

- výhoda:

```
typedef int TPole20INT[20];
TPole20INT pole_1;
TPole20INT pole_2;
TPole20INT pole_3;
```

- místo

```
int pole_1[20];
int pole_2[20];
int pole_3[20];
```

# Statické pole

*Poznámka:*

```
int moje_funkce(int n)
{
   float B[n];
}
```

ačkoliv není velikost pole specifikována konstantou
a pole je vytvořeno v okamžiku vstupu do funkce na zásobníku
(tzv. automatické proměnné ve třídě **auto**), pole není
považováno za dynamické, protože není vytvořeno voláním
speciální funkce (a není alokováno na tzv. hromadě)

# Statické pole

```
int main()
{
   int n;
   scanf("%d",&n);
   int A[n];
}
```

stejná situace

- ve verzi K&R, velikost pole mohla být dána pouze konstantou

# Dvourozměrné statické pole

- deklarace

počet řádek

```
int x[10][10];
float matice[10][20];
```

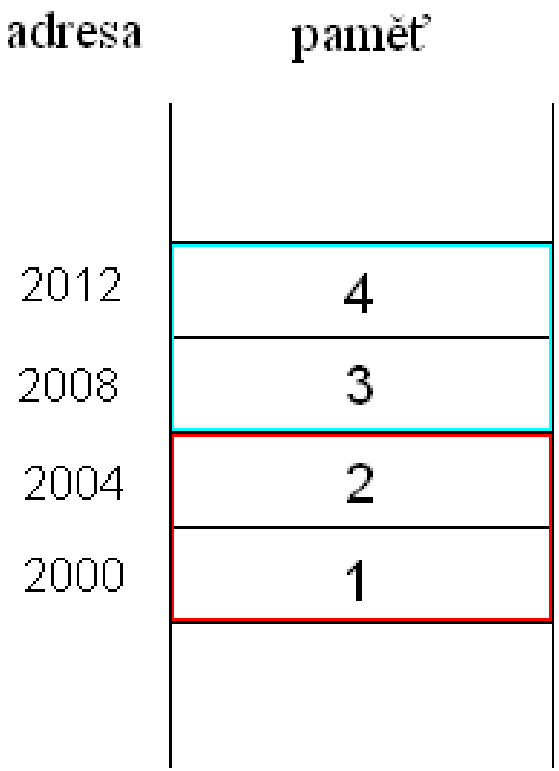
- konstruktor pole

```
int m[2][2] = {{1,2},{3,4}};
```

| | 0 | 1 |
|---|---|---|
| 0 | **1** | **2** |
| 1 | **3** | **4** |

# Dvourozměrné statické pole

- dvourozměrné statické pole je v jazyce C uloženo po řádcích



# Ukazatelé, dynamická alokace paměti

- typ ukazatel – speciální datový typ

Obsahem proměnných typu ukazatel je adresa do paměti, tedy obsahem není přímo hodnota, ale ukazatel (odkaz) na místo, kde se hodnota nachází

  – obsah proměnné typu ukazatel ukazuje na data

- deklarace

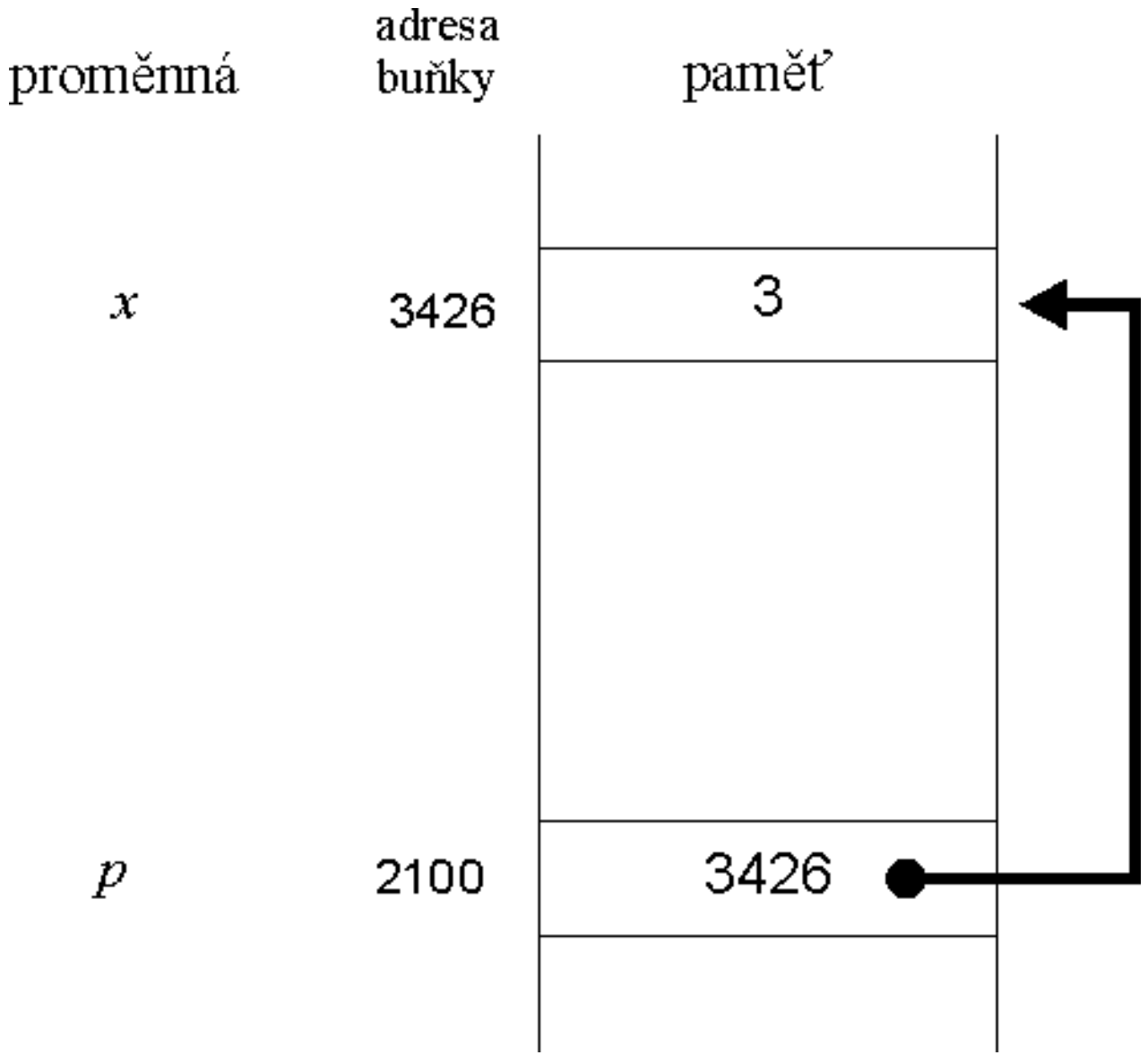
```
int *p;
float* pf;
```

- možná inicializace

```
int x;
p = &x;
```

- přístup k hodnotě

```
*p = 3;
```


proměnná
adresa buňky
paměť
x
3426
3
p
2100
3426

- co znamená ?

```
int *p;
p = 3;
*p = 8;
```

- program by byl zastaven operačním díky chybě při běhy typu „memory access violation“ atp.
- zápis na specifickou adresu se hodí např. u jednočipových mikropočítačů

- dynamická alokace paměti

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

  - alokuje paměť o velikosti `size` bytů, vrací ukazatel na počátek alokované paměti (adresu počátku bloku)

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

  - alokuje paměť o velikosti `size*num` bytů, vrací ukazatel na počátek alokované paměti, alokovanou paměť vynuluje

**`void *`**

  - „obecný“ ukazatel

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

- mění velikost již dříve alokované paměti (funkcemi `malloc, calloc` nebo `realloc);`
- `block` je ukazatel na původně alokovanou paměť a `size` je nová velikost v bytech; je-li block NULL, funkce se chová jako malloc
- vrací ukazatel na počátek nově alokované paměti; překopíruje obsah původně alokované paměti do nové a původní paměť uvolní

- při nedostatku paměti vracejí funkce hodnotu `NULL`
  - konstanta definovaná v stdio.h, zpravidla 0

- dealokace paměti ("návrat paměti operačnímu systému")

```
free(void *p);
```

- jak alokovat pamět, chceme-li ji využít jako pole o počtu *n* položek (dynamicky alokované pole)

```
int *pi;
pi = (int*)malloc(sizeof(int)*n);
pi = (int*)calloc(n,sizeof(int));
```

- operační systém alokuje dynamickou paměť z bloku paměti zvané **hromada (heap)**

Program ze snímku 8 přepíšeme tak, aby se pole alokovalo dynamicky podle počtu zadaných prvků.

Příklad pole_1D_2

```
int main(void)
{
  int i,n,max;
  int *A;
  printf("Zadej pocet zadavanych cisel: ");
  scanf("%d",&n);
  A = (int*)malloc(sizeof(int)*n);
  if (A == NULL)
  {
   printf("Pocitac uz nema volnou pamet.");
   return -1;
  }
```

```
  printf("Zadej %d cisel: ",n);
  for(i=0;i<n;i++) scanf("%d",&A[i]);

  printf("Vypis cisel pozpatku:\n");
  for(i=n-1;i>=0;i--) printf("%d ",A[i]);

  /* hledani maxima */
  max = A[0];
  for(i=0;i<n;i++)
   if (A[i]>max) max = A[i];

  printf("Nejvetsi cislo je %d.\n",max);
  free(A);
  return 0;
}
```

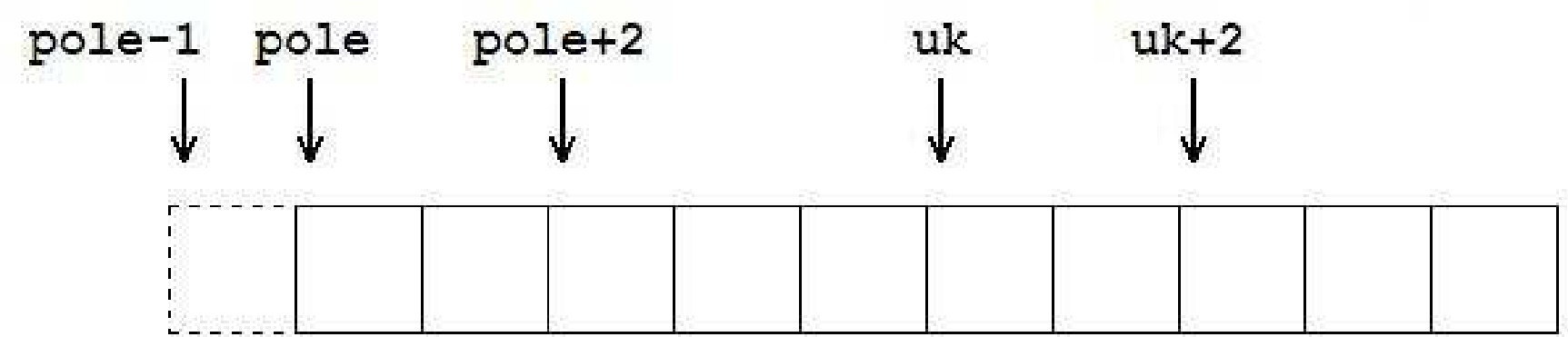
# Ukazatelová aritmetika

- ukazatelé mohou být
  - porovnávány
    - výsledek: zda dva ukazatelé ukazují na stejné místo v paměti
  - odečítány
    - jak daleko jsou v paměti data
  - přičtena konstanta
    - posuv v paměti
- sčítání dvou ukazatelů nemá smysl

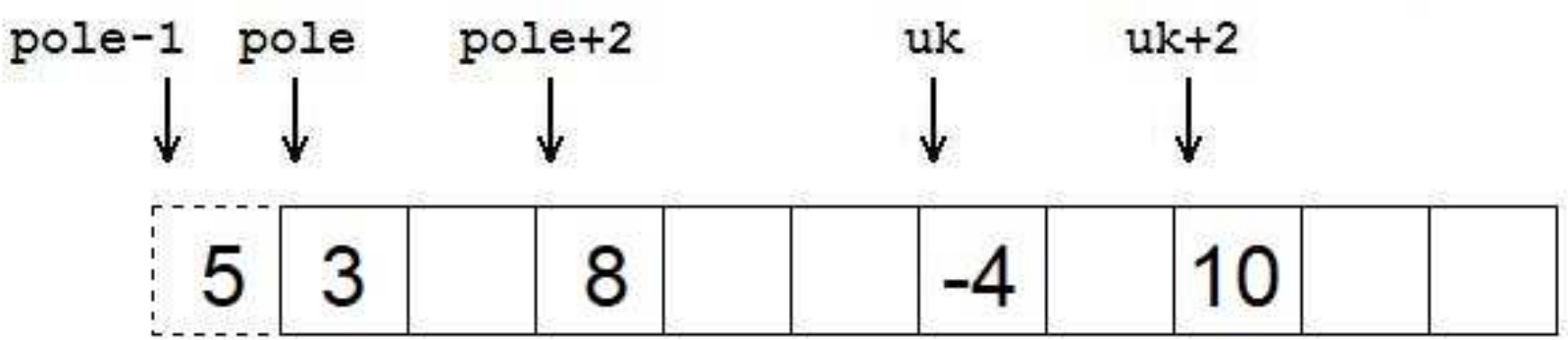
# Ukazatelová aritmetika

```
int *pole;
int *uk;  // pomocný ukzatel

pole = (int*)malloc(sizeof(int)*10);
uk = pole + 5;
```



- přístup ke statickým a dynamickým polím je v jazyce C/C++ stejný:
- `*(pole+0)=3; pole[0] = 3;`
  `*(pole)=3;`
- `*(pole-1)=5; pole[-1] = 5;`
- `*(pole+2)=8; pole[2] = 8;`
- `*(uk+0)= -4; *uk = -4;`
  `uk[0] = -4;`
- `*(uk+2)= 10; uk[2] = 10;`



- příklad: posuv ukazatele

```
uk = uk + 2;
```

- **int** A[10];
  - A je konstantní ukazatel na začátek statického pole
  - lze použít ukazatelovou aritmetiku

```
A[3] = 7;
```
nebo
```
*(A+3) = 7;
```

  - ale nelze provést přiřazení:

```
A = (int*)malloc(…);
A = A + 2;
```

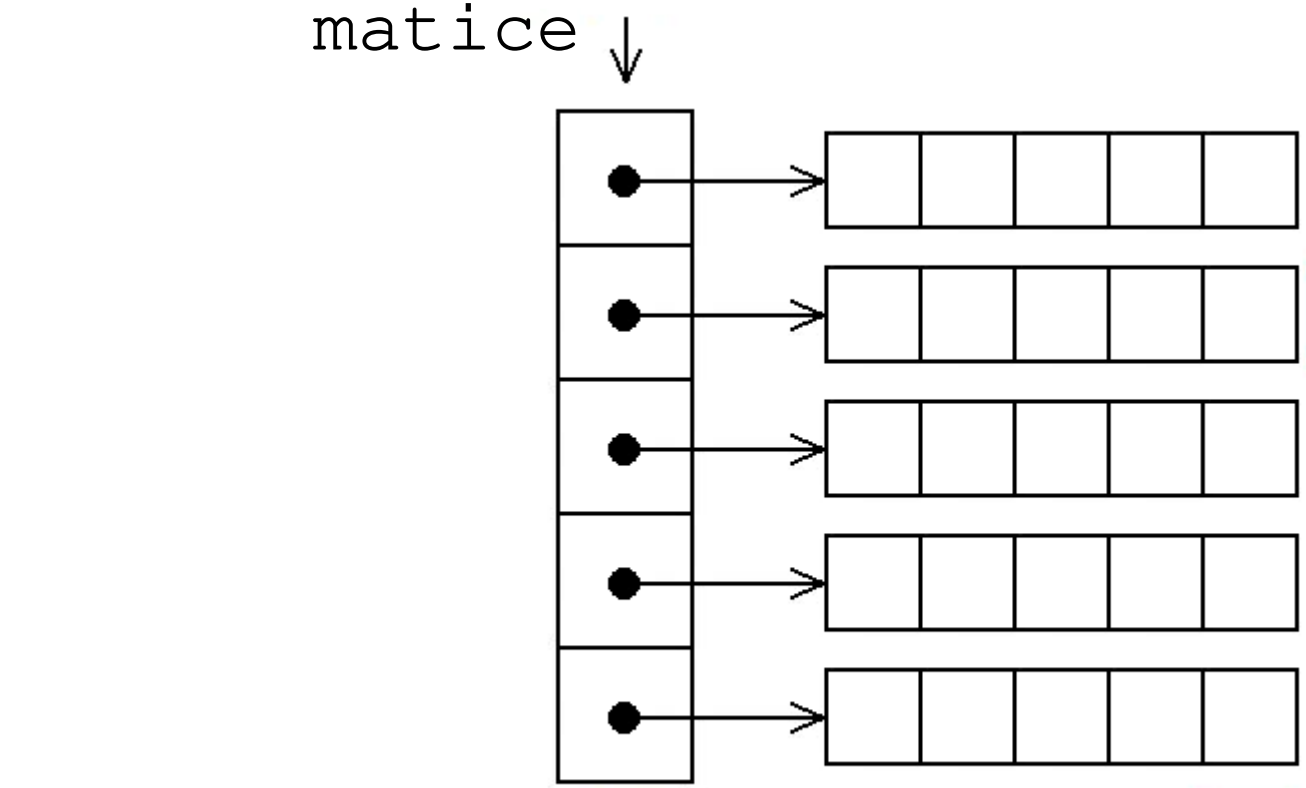
protože A je konstanta

# Příklad – ukazatelová aritmetika

Program `tisk_pole` na webových stránkách, který tiskne pole několika způsoby využívající ukazatelovou aritmetiku.

# Dynamická alokace dvourozměrného pole

- dynamicky alokované dvourozměrné pole je uloženo jako pole polí



# Dynamická alokace dvourozměrného pole

- alokace (n řádek, m sloupců)

```
int **matice;
matice = (int**)malloc(n*sizeof(int*));
for(int i=0;i<n;i++)
  matice[i] = (int*)malloc(m*sizeof(int));
```

- dealokace

```
for(int i=0;i<n;i++) free(matice[i]);
free(matice);
```

```
int main(void)
{
  int i,j,n,m,min; int **matice;
  printf("Zadej pocet radku matice: ");
  scanf("%d",&n);
  printf("Zadej pocet sloupcu matice: ");
  scanf("%d",&m);
  matice = (int**)malloc(n*sizeof(int*));
  for(i=0;i<n;i++)
    matice[i] = (int*)malloc(m*sizeof(int));

  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      scanf("%d",&matice[i][j]);
```

```
  /* hledam minimum */
  min = matice[0][0];
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      if (matice[i][j]<min) min=matice[i][j];

  printf("Minimum je: %d\n",min);
  /* dealokace */
  for(i=0;i<n;i++) free(matice[i]);
  free(matice);
}
```

Příklad pole_2D_1

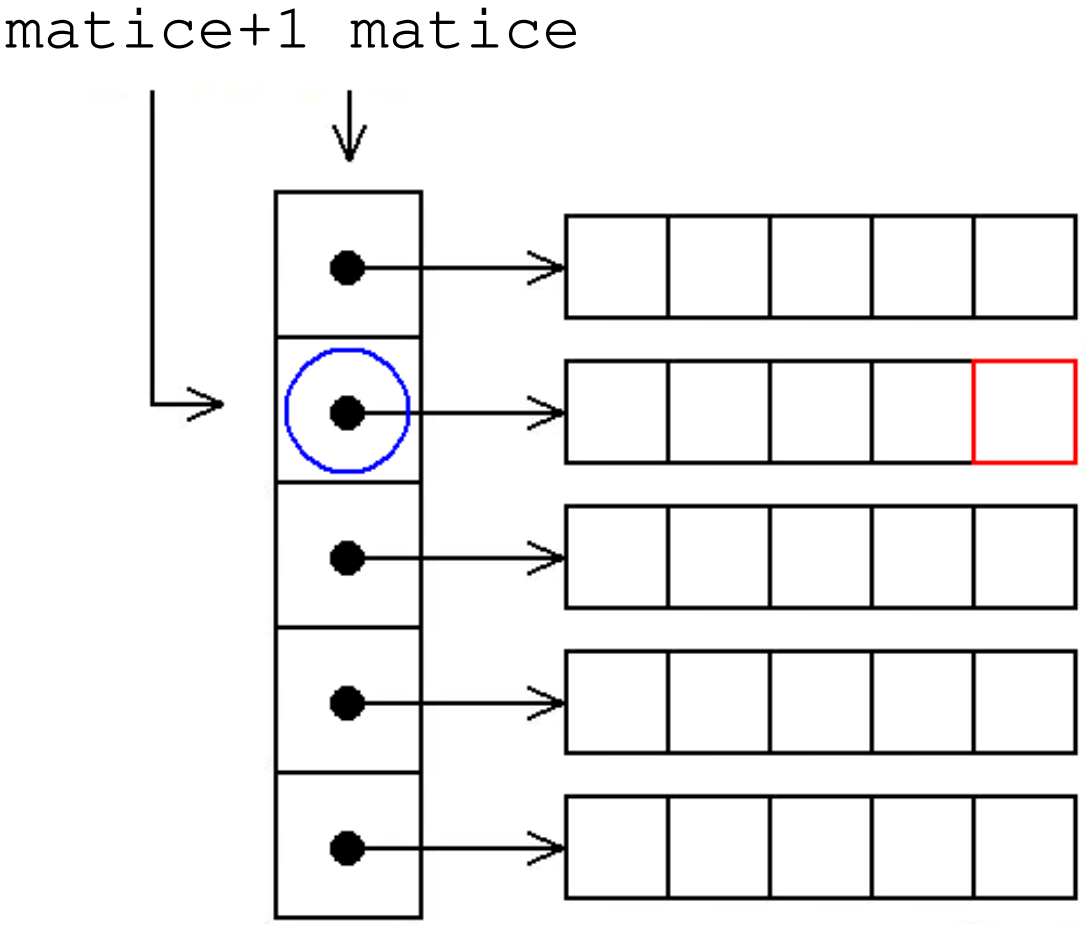
# Dvoudimenzionální dynamické pole a ukazatelová aritmetika

```
matice[i][j]      *((*(matice+i))+j)
```

- ukazatel na i-tý řádek

```
matice[i]         *(matice+i)
```

# Dvoudimenzionální dynamické pole a ukazatelová aritmetika




`matice+1` ukazuje na modrý prvek

`*(matice+1)+4` ukazuje na červený prvek

přístup k červenému prvku:

`*(*(matice+1)+4)`

nebo

`matice[1][4]`

# Alokace paměti v C++

- kromě `malloc` existuje v C++ operátor new
  - oproti funkci `malloc` je operátor new přetížen, tj. je předefinovaná jeho funkce pro určitý datový typ
- dynamická alokace

```
int *p;
p = new int;
```

- dynamická alokace pole

```
int *pi;
pi = new int[velikost];
```

- **velikost je v položkách, nikoliv ve slabikách!**
  - důvod: new je přetížen, tj. uvažování velikosti typu je „schováno“ v přetěžování
- pomocí new dynamicky alokujeme i objekty
  - dynamicky se alokovat objekty pomocí malloc nedají, nespustí se konstruktor
- paměť alokovaná pomocí new se dealokuje pomocí delete

```
delete p;
delete [] pi;
```

příznak rušení pole

- při nemožnosti alokovat paměť:
  - vyhodí se výjimka (rys C++, který poznáme později)
    - původní verze operátoru `new`
  - od verze jazyka 98 existuje i verze operátoru vracející NULL (nevyhazující výjimku)
  - `p = new (std::nothrow) int[vel]`

```
int main(void)
{
  int i,j,n,m,min; int **matice;
  cout << "Zadej pocet radku matice: ";
  cin >> n;
  cout << "Zadej pocet sloupcu matice: ";
  cin >> m;
  matice = new int*[n];
  for(i=0;i<n;i++)
    matice[i] = new int[m];

  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      cin >> matice[i][j];
```

```
  /* hledam minimum */
  min = matice[0][0];
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      if (matice[i][j]<min) min=matice[i][j];

  cout << "Minimum je " << min << endl;
  /* dealokace */
  for(i=0;i<n;i++) delete [] matice[i];
  delete [] matice;
}
```

Příklad pole_2D_2