Arrays, pointers, dynamic memory allocation

Arrays

- homogenous data structure
  - all elements are of the same data type
  - one dimensional
    - analogous to arithmetical vector in mathematic
  - two dimensional
    - analogous to matrix in mathematic
  - three dimensional
    - cube

Static Array

- static array –
  - the size (count of elements) is given in the source code by the constant
- dynamic array
  - the size is determined and memory is allocated during program run by calling special function

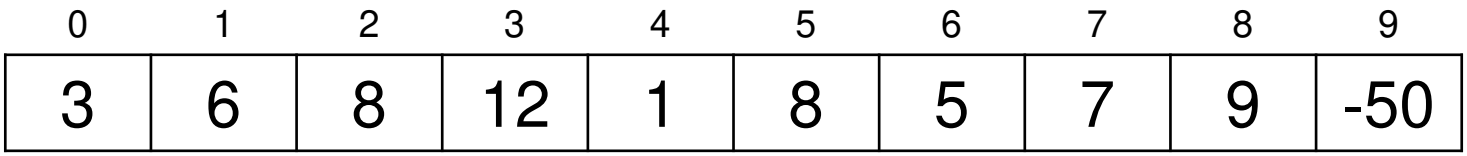
Static Array

- declaration

```
Type identifier[size];
```

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

Example of the array `a`



- access to elements: using indexes

```
a[i], my_vector[10]
```

- index range: `0 .. n-1`
- index range isn't automatically checked

Reading data into array

- with `scanf:`

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

- with stream `cin:`

```
cin >> my_vector[i];
```

- elements of arrays are random values
  - only global arrays are set to zero

- it is possible to initialize elements by the declaration (array constructor):

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

We create a program which reads count of processed numbers, then it reads these values and stores them into static array. It prints values in opposite order and it finds maximum.

Example array_1D_1

```
#define MAX 50

int main(void)
{
  int i,n,max;
  int A[MAX];
  printf("Enter count of input values: ");
  scanf("%d",&n);
  if (n>MAX)
  {
   printf("Static array has only %d elements.\n",MAX);
   return -1;
  }
```

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

  printf("Values in opposite order:\n");
  for(i=n-1;i>=0;i--) printf("%d ",A[i]);

  /* maximum search */
  max = A[0];
  for(i=0;i<n;i++) // better: for(i=1;i<n;i++)
   if (A[i]>max) max = A[i];

  printf("The maximum is %d.\n",max);
  return 0;
}
```

Static Array

- user definition of array type

```
typedef int TArray20INT[20];
```

- declaration of variable :

```
TArray20INT my_array;
```

Static Array

- advantage:

```
typedef int TArray20INT[20];
TArray20INT my_array_1;
TArray20INT my_array_2;
TArray20INT my_array_3;
```

- instead of

```
int my_array_1[20];
int my_array_2[20];
int my_array_3[20];
```

Static Array

Notes

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

although size of array is not specified by the constant and the array is created at the moment of entrance into function at the stack (so-called **auto** variables), the array is not considered as dynamic one because it is not created by calling special function

Static Array

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

the same situation

- in K&R, the size of the array had to be defined only by constant

Two-dimensional static array

- declaration

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

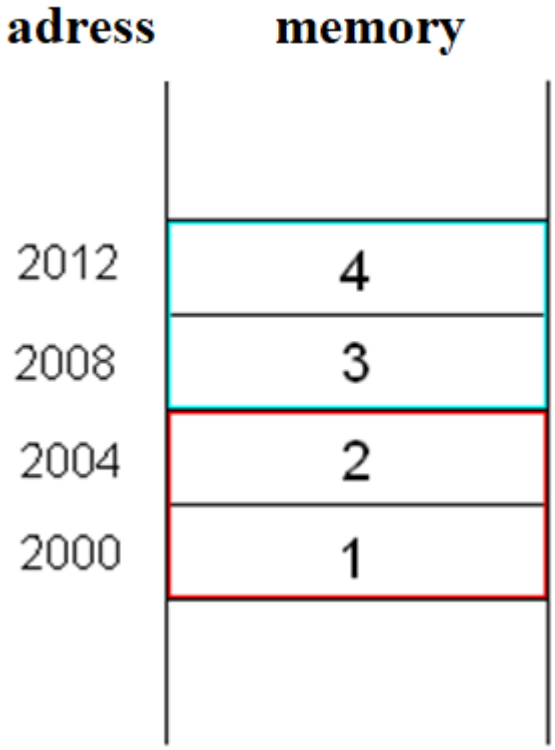
- constructor

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

	0	1
0	1	2
1	3	4

Two-dimensional static array

- Two-dimensional static array is stored by rows in C



Pointers, dynamic memory allocation

- pointer type – "special" data type

The value stored in the variable of pointer type is address into memory; it means that pointers doesn't carry values (data) but address where data is placed in memory.

  – the content of this variable points to data

- declaration

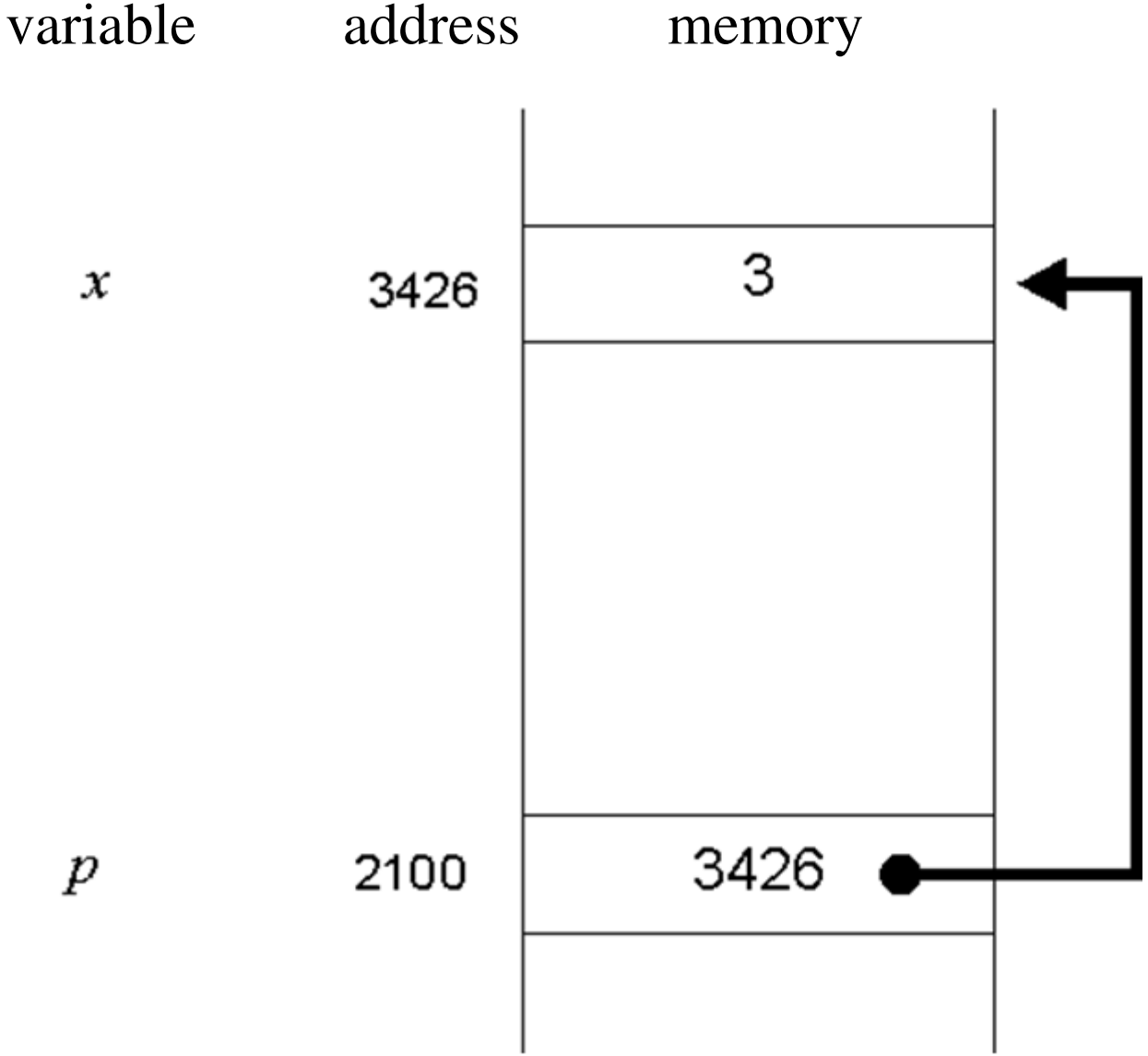
```
int *p;
float *pf;
```

- one possibility of initialization

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

- access to the value

```
*p = 3;
```


variable
address
memory
x
3426
3
p
2100
3426

- what does it mean?

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

- such program will by probably stopped under normal operating system due to run-time error (memory access violation error or like this)
- the writing to the specific address is used for example in microchips

- dynamic memory allocation

`void` `*malloc(size_t  size)`

  - allocates memory block of the size `size` (in bytes), returns address to the beginning of block (pointer to the beginning)

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

  - allocates memory block of the size `size*num` bytes, returns address to the beginning of block (pointer to the beginning), the memory is set to zero

`void` `*`

  - general pointer

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

- reallocates the memory previously allocated by any function `malloc, calloc` or `realloc` (allocates memory of new size, the block can be elsewhere in memory);
- `block` is the pointer to the previously allocated memory, `size` is **new size** in bytes; if lock is NULL, the function behaves as malloc
- returns pointer to the beginning of newly allocated memory; it copies the contents of previously allocated memory to the beginning of newly allocated one and frees original block

- if operating system has insufficiency of memory, functions return value `NULL`
  - constant defined in stdio.h, generally 0

- memory deallocation ("return the memory to the operating system")

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

- how to allocate memory if we want to use it as an array (dynamically allocated array)

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

- the operating system allocates dynamic memory from block called **heap**

We rewrite the program from the slide 8 to allocate array dynamically; the array is exactly of such size to store the input data.

Example array_1D_2

```
int main(void)
{
  int i,n,max;
  int *A;
  printf("Enter count of input values: ");
  scanf("%d",&n);
  A = (int*)malloc(sizeof(int)*n);
  if (A == NULL)
  {
   printf("There is no free memory in OS.");
   return -1;
  }
```

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

  printf("Values in opposite order:\n");
  for(i=n-1;i>=0;i--) printf("%d ",A[i]);

  /* maximum search */
  max = A[0];
  for(i=0;i<n;i++) //better: for(i=1;i<n;i++)
   if (A[i]>max) max = A[i];

  printf(" The maximum is %d.\n",max);
  free(A);
  return 0;
}
```

Pointer arithmetic

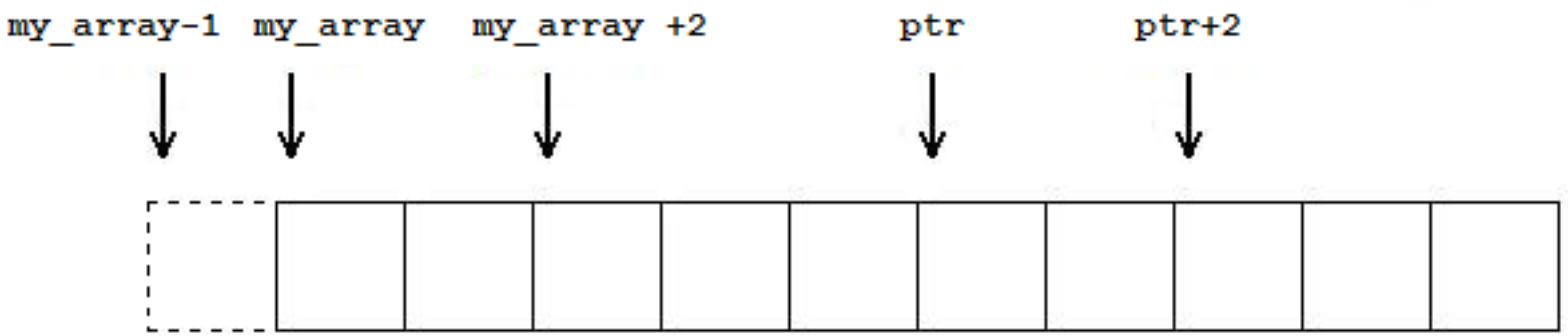
- pointers can be
  - compared
    - result: if two pointers point to the same address
  - subtracted
    - how far between data
  - added by constant
    - shift within memory

- adding two pointers has no sense

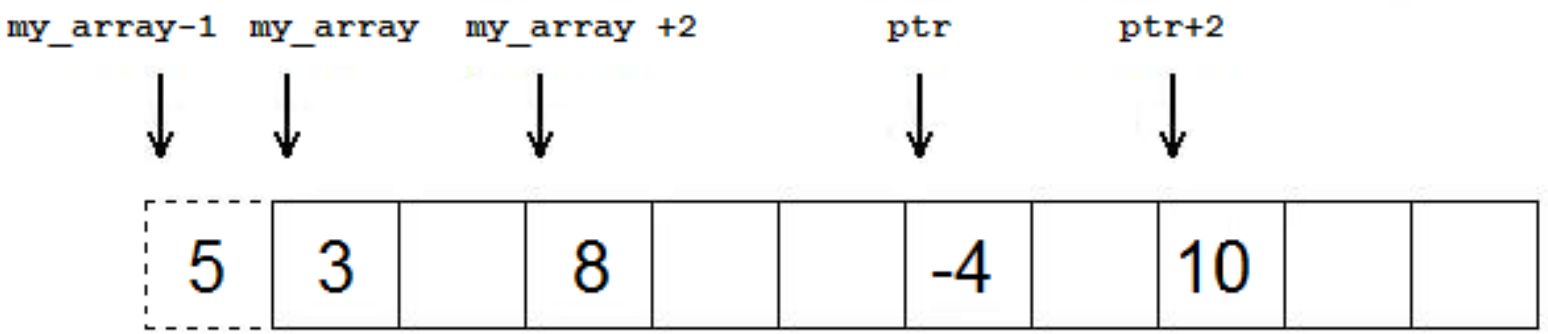
Pointer arithmetic

```
int *my_array;
int *ptr;  // auxiliary pointer

my_array = (int*)malloc(sizeof(int)*10);
ptr = my_array+5;
```



- the access to the static/dynamic array using pointers or indexes is the same in C/C++:
- `*(my_array+0)=3; my_array[0] = 3;`
  `*(my_array)=3;`
- `*(my_array-1)=5; my_array[-1] = 5;`
- `*(my_array+2)=8; my_array[2] = 8;`
- `*(ptr+0)= -4; ptr[0] = -4;`
  `*ptr = -4;`
- `*(ptr+2)= 10; ptr[2] = 10;`



- example of the shift of the pointer

```
ptr = ptr + 2;
```

- **int** A[10];
  - A is a constant pointer to the beginning of the static array
  - the pointer arithmetic can be used

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

  - but I cannot make assignments

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

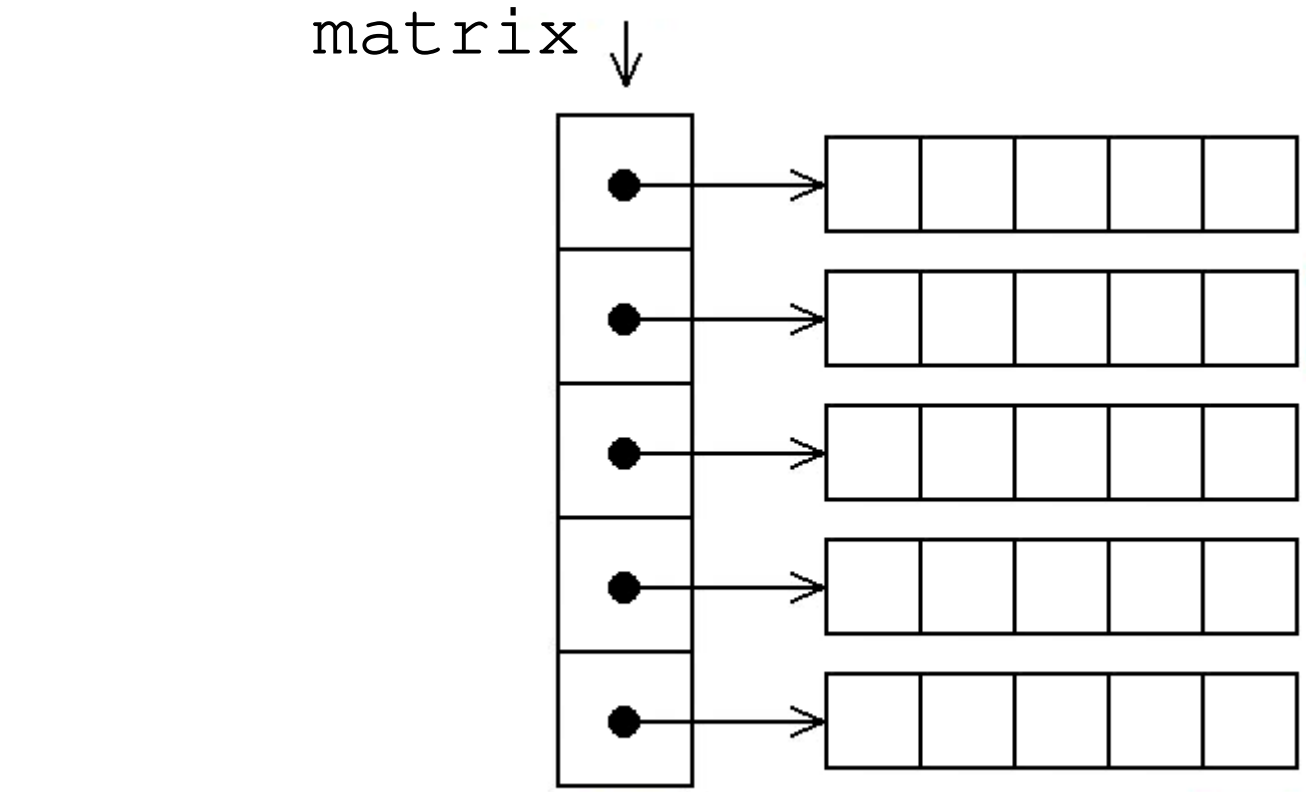
because A is the constant

Example – pointer arithmetic

Program `print_array` on web pages which prints elements of array using indexes and pointer arithmetic.

Dynamic allocation of two-dimensional array

- dynamically allocated 2D array is stored as array of 1D arrays



Dynamic allocation of two-dimensional array

- allocation (`n` rows, `m` columns)

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

- deallocation

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

```c
int main(void)
{
  int i,j,n,m,min; int **matrix;
  printf("Enter count of rows: ");
  scanf("%d",&n);
  printf("Enter count of columns: ");
  scanf("%d",&m);
  matrix = (int**)malloc(n*sizeof(int*));
  for(i=0;i<n;i++)
    matrix[i] = (int*)malloc(m*sizeof(int));

  printf("Enter elements of matrix: ");
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      scanf("%d",&matrix[i][j]);
```

```
/* print matrix*/
for(i=0;i<n;i++)
{
  for(j=0;j<m;j++)
    printf("%d ",matrix[i][j]);
  printf("\n");
}

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

```
printf("Minimum is: %d\n",min);

/* deallocation */
for(i=0;i<n;i++) free(matrix[i]);
free(matrix);
return 0;
```

Example array_2D_1

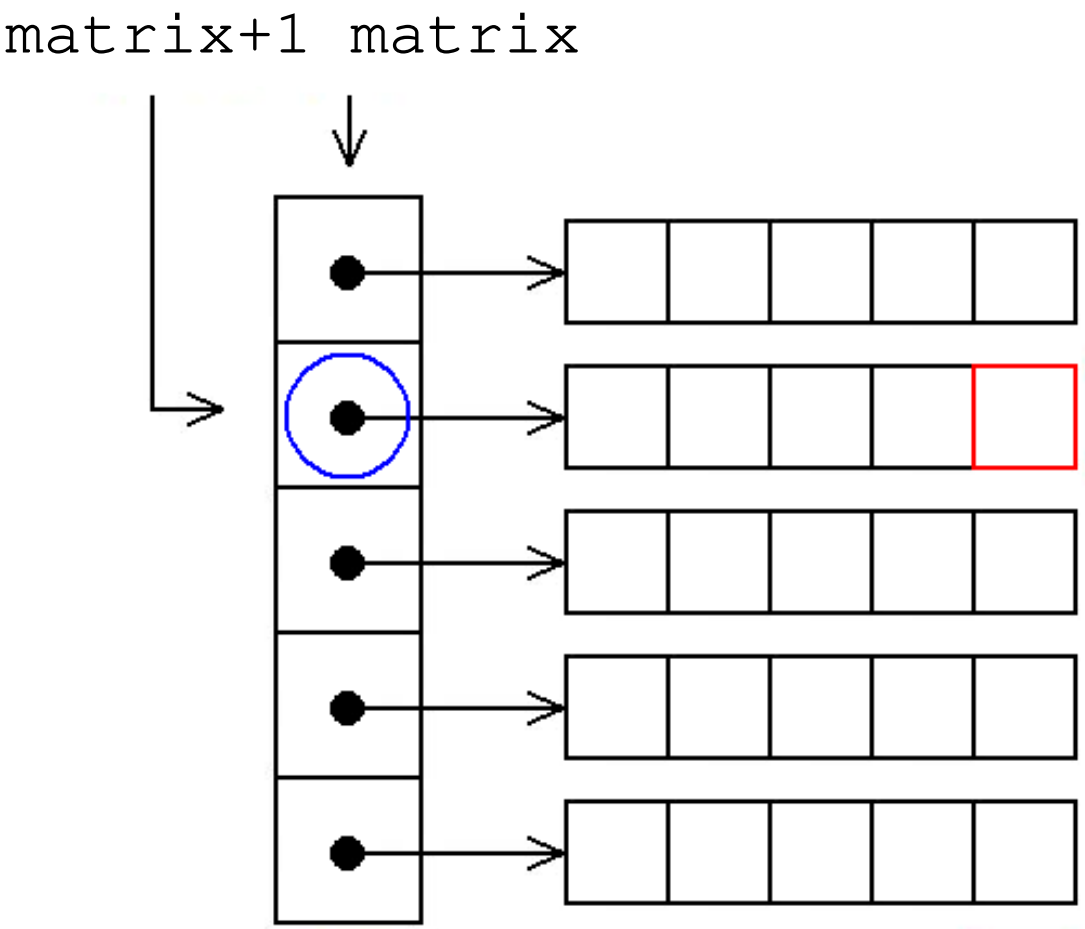
Two-dimensional dynamic array and pointer arithmetic

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

- pointer to the i-th row

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

Two-dimensional dynamic array and pointer arithmetic



`matrix+1` points to the blue element

`*(matrix+1)+4` points to the red element

access to the red element:

`*(*(matrix+1)+4)`

or

`matrix[1][4]`

Memory Allocation in C++

- new possibility to allocate memory in C++ - operator new
  - this operator is overloaded, i.e. there are several code for each data type
- dynamical allocation of one int

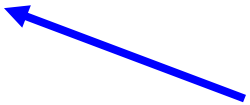
```
int *p;
p = new int;
```

- dynamical allocation of the array

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

- **the size is in elements, not in bytes!**
  - reason: overloading of the operator new, sizeof(type) is in implementation
- new is used to allocate objects dynamically
  - if we used malloc in this case the constructor wouldn't executed
- the memory allocated by new should be deallocated by the operator delete

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



flag to deallocate array

- if operating system has not sufficient amount of memory:
  - the exception is raised (thrown) - feature of object oriented languages to process errors
    - original version of the operator `new`
  - the version returning NULL exists since C++98
  - `p = new (std::nothrow) int[vel]`

```
int main(void)
{
  int i,j,n,m,min; int **matrix;
  cout << "Enter count of rows: ";
  cin >> n;
  cout << " Enter count of columns: ";
  cin >> m;
  matrix = new int*[n];
  for(i=0;i<n;i++)
    matrix[i] = new int[m];

  cout << "Enter elements of matrix: ";
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      cin >> matrix[i][j];
```

```
/* print matrix */
for(i=0;i<n;i++)
{
  for(j=0;j<m;j++)
    cout << matrix[i][j] << ' ';
  cout << '\n';
}

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

```
  cout << "Minimum is " << min << endl;

  /* deallocation */
  for(i=0;i<n;i++) delete [] matrix[i];
  delete [] matrix;

  return 0;
}
```

Example array_2D_2