Тема вежбе: Показивачи и низови

ПОКАЗИВАЧИ И НИЗОВИ

1 Показивачи

Показивачи су промењљиве чије су вредности адресе неких података у меморији. Показивачи се могу односити (могу показивати) на:

- променљиве
- функције

Декларација показивача: `data_type *ptr_name;`

- *data_type* – тип податка на који показивач показује, не и тип показивача (тип показивача је: показивач на *data_type*
- *ptr_name* – име показивача
- * – указује да је у питању показивач, а не обична променљива

Пример:

```
int *ptr;
```

Име: птр
Тип: показивач на целобројни тип

Показивач на неки тип може узети само адресу тог истог типа, у супротном компајлер пријављује упозорење или грешку. Једини изузетак је показивач на *void* тип.

```
int var;
float* fptr;
void* vptr;

fptr = &var;              /* compiler warning or error */
vptr = &var;              /* OK */
```

Додељивање вредности показивачу:

```
int *ptr;
int value = 5;
ptr = &value;
```

Дереференцирање показивача:

1) Читање вредности меморијске локације на коју показивач показује

```
int a = *ptr;              /* a = value */
```

- тип и величина прочитане меморије одређени су типом показивача

2) Упис у меморијску локацију на коју показивач показује

```
*ptr = 7;                  /* value = 7 */
```

- тип и величина података који ће бити уписан у меморију одређени су типом показивача

1.1 Const квалификатор

1) Показивач се може мењати али не и оно на шта он показује:
   - Кључна реч *const* мора бити лево од '*'

```
const int* ptr_variable;
int const* ptr_variable;

int var1 = 3;
int var2 = 5;
const int* ptr = &var1;
ptr = (const int*)var2;   /* OK */
*ptr = 7;                 /* ERROR */
```

   - Експлицитна конверзија мора бити коришћена када вредност оваквог показивача желимо да доделимо обичном, неконстантном, показивачу

```
int* ptr;
int const* cptr;
ptr = cptr;               /* compiler warning or error */
ptr = (int*) cptr;        /* OK */
```

2) Може се мењати оно на шта показивач показује, али не и сам показивач:
   - Кључна реч *const* мора бити десно од '*'

```
int* const ptr_variable;

int var1 = 3;
int var2 = 5;
int* const ptr = &var1;
ptr = var2;                 /* ERROR */
*ptr = 7;                   /* OK */
```

3) Дупло константни показивач – није могућа његова измена као ни измена онога на шта он показује

```
const int* const ptr_variable;
int const* const ptr_variable;

int var1 = 3;
int var2 = 5;
int const* const ptr = &var1;
ptr = var2;                   /* ERROR */
*ptr = 7;                     /* ERROR */
```

1.2 Sizeof оператор

Унарни оператор који срачунава величину типа и изражава је у бајтима. Може се користити над променљивом или над типом. У случају да се користи над променљивом, повратна вредност је величина те променљиве тј величина типа ког је та променљива. Када ради над типом враћа његову величину.

По дефиницији: `sizeof(char) = 1`

```
int* ptr;
int array[3] = {1, 2, 3};
int s;

s = sizeof(int);
printf("%d\n", s);       /* prints 4 */
s = sizeof(ptr);
printf("%d\n", s);       /* prints 4 */
s = sizeof(array);
printf("%d\n", s);       /* prints 12 (3*sizeof(int)) */
```

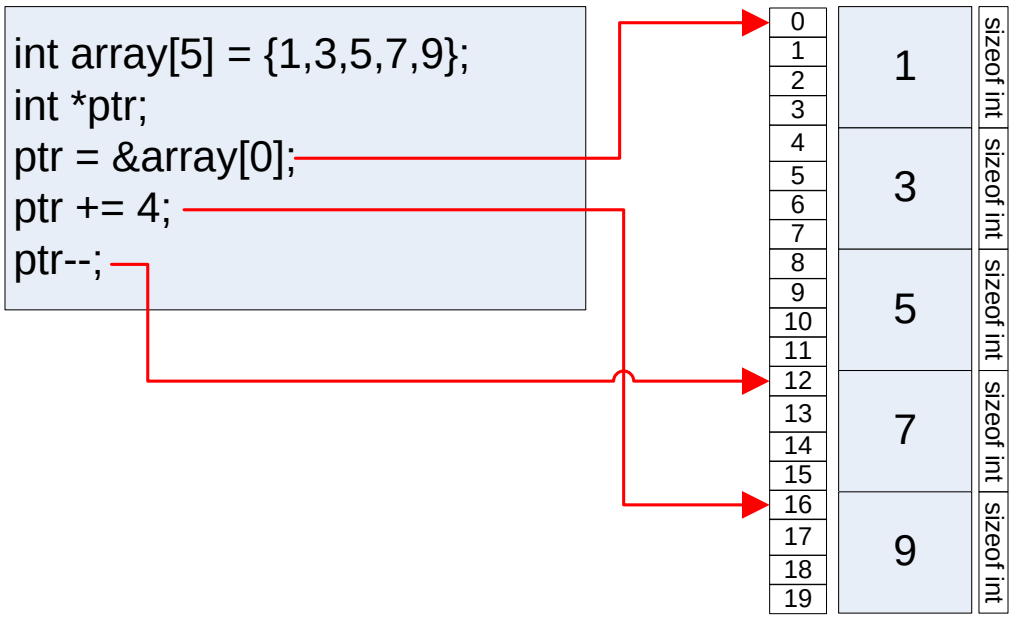
1.3 Показивачи и операције над њима

Операције сабирања и одузимања целог броја и показивача извршавају се на следећи начин:

```
data_type* ptr;
ptr ± n <=> ptr ± n * sizeof (data_type)
```

Напомена: исто важи и за унарне операторе ++ и --

Пример:



Одузимање два показивача је могуће само ако су показивачи истог типа. Сама операција одузимања једино има смисла ако показивачи показују на адресе унутар истог парчета меморије, у супротном резултат ће бити недефинисан. Са друге стране сабирање два показивача није дозвољено по стандарду те ће компајлер у том случају пријавити грешку. Слично је и за поређење два показивача односно поређење има смисла само ако показивачи указују на исто парче меморије.

Примери:

```
int array[5] = {1,3,5,7,9};
int* ptr1;
int* ptr2;
ptr1 = &array[1];
ptr2 = &array[4];
printf("%d\n", (ptr2-ptr1));        /* prints 3 */
```

```
int array1[5] = {1,3,5,7,9};
int array2[3] = {2,4,6};
int* ptr1;
int* ptr2;
ptr1 = &array1[0];
ptr2 = &array2[2];
```

```
printf("%d\n", (ptr2-ptr1));        /* undefined */
int sum = ptr1 + ptr2;              /* ERROR */
```

1.4 Оператор [ ]

По дефиницији: `A[B] <=> *(A + B)`

где *A+B* мора бити показивачког типа јер оператор * ради само са показивачем. Дакле или *A* мора бити показивач, а *B* целобројног типа, или обрнуто:

```
data_type* A;

int B;

A[B] <=> *(A + B) <=> *(A + B * sizeof(data_type))

B[A] <=> *(B + A) <=> *(A + B * sizeof(data_type))
```

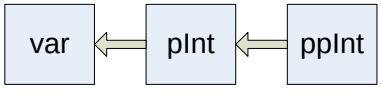
Пример:

```
float* p;
float x;

/* neka je p 1000, tj. p pokazuje na adresu 1000 */

x = *p;         // x je float vrednost sa adrese 1000
x = p[0];       // x je float vrednost sa adrese 1000
x = p[4];       // x je float vrednost sa adrese (1000 + 4 *
                // sizeof(float))
x = 4[p];       // x je float vrednost sa adrese (1000 + 4 *
                // sizeof(float))
```

1.5 Показивач на показивач

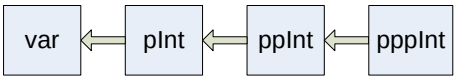
Показивач може показивати на било који тип, па тако и на тип показивача:

```
int var;
int* pInt = &var;
int** ppInt = &pInt;
```



и то тако у недоглед:

```
int var;
int* pInt = &var;
int** ppInt = &pInt;
int*** pppInt = &ppInt;
```



Вишеструки показивачи су потребни у два сценарија. Први је, када је у програму неопходно проследити показивач по референци, а други, у случају рада са вишедимензионалним низовима:

1) Прослеђивање показивача по референци:

```
int g_var;

void bar(int** p)
{
     *p = &g_var;    // change pointer value
     **p = 39;       // change value of variable to which
                     // pointer points to
}

void foo()
{
     int var;
     int* ptr = &var;
     bar (&ptr);
     . . .
}
```

2) Вишедимензионални низови

```
void bar(int** mat)
{
    mat[0][0] = 1;
}

void foo()
{
    int** matrix;
    matrix = new int* [2];
    matrix[0] = new int[2];
    matrix[1] = new int[2];
    bar(matrix);
}
```

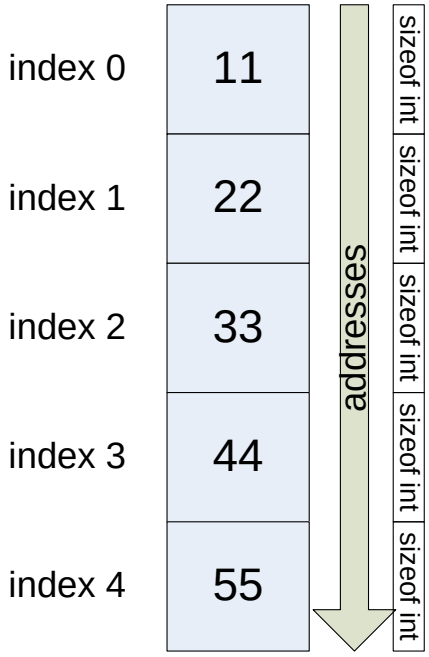
2 Низови

Низ у програмском језику Це представља блок меморије који се тумачи као да се у њему, један за другим, налазе елементи исте величине. Елементи низа су континуалне меморијске локације. Кључни аспект низа је његова декларација:

```
element_type array_name[dimension] = {declaration_list};
```

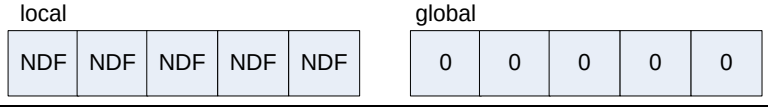
По осталим аспектима је јако сличан показивачу који се не може мењати (нпр. смисао оператора [ ] је исти).

```
int array[5] = {11, 22, 33, 44, 55};
printf(element with index 3: %d\n, array[3]);
// prints 44
```



Као и код регуларних променљивих, ако нема експлицитне инцијализације, низови статичке трајности постављају се на нулу (сви њихови елементи), а аутоматске неће (остаће недефинисане).

```
int array[5];
```



За експлицитну иницијализацију користи се листа елемената у витичастим заградама. Листа не сме имати више елемената од димензије низа. Ако има мање, преостали елементи се попуњавају нулама.

```
int array[5] = {1,3,5};
```

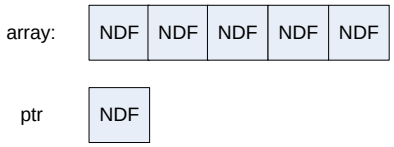
Ако постоји иницијализација низа, димензија може бити изостављена. Тада ће димензија бити одређена бројем елемената у листи за иницијализацију. У случају да је потребно иницијализовати само неке елементе низа, C99 стандард нуди решење:

```
int array[100] = {[13] = 5, [77] = 6};
```

2.1 Однос низова и показивача

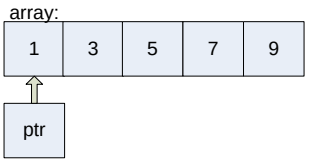
Иако је већ речено да су низови и показивачи врло слични, ипак постоје разлике. Када се дефинише низ, заузима се меморија за све његове елементе. Када су у питању показивачи то није случај, јер се меморија заузима само за тај показивач.

```
char array[5];

char* ptr;
```



Низ, односно име низа, представља његову адресу, адресу почетка низа (првог елемента). Показивач је са друге стране само променљива чија је вредност нека адреса. У том смислу се низ може посматрати као показивачка константа.

```
int array[] = {1, 3, 5, 7, 9};

int* ptr = array;
```



Поред претходно наведених разлика, разлика низова и показивача се огледа и у резултату *sizeof* операције:

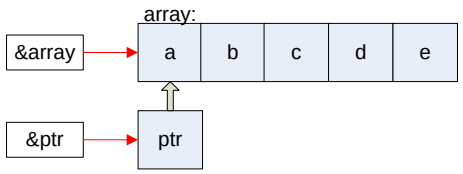
- за низ враћа број меморијских речи које су заузете за цео низ
- за показивач враћа колико меморијских речи је заузето за адресу

```
char array[15];
char* ptr;
printf("%d\n", sizeof(array));      /* prints 15 */
printf("%d\n", sizeof(ptr));        /* prints 4 */
```

Разлика у & оператору:

- За низ враћа адресу првог елемента (исто што и &array[0])
- За показивач враћа адресу показивача (на којој адреси се налази показивач као променљива)

```
char array[] = {'a', 'b', 'c', 'd', 'e'};
char* ptr = array;

if (array[0] == ptr[0])         /* prints equal */
    printf("equal");
else
    printf("not equal");

if (&array == &ptr)             /* prints not equal */
    printf("equal");
else
    printf("not equal");
```



2.2 Додела меморије

Це језик подржава два начина заузимања меморије кроз дефинисање самих променљивих:

- Статичко заузимање – које се примењује за све глобалне и статички дефинисане променљиве. Свака променљива одређује један блок меморије, непромељиве величине. Меморија се заузима једном (приликом покретања програма) и никад се не ослобађа.
- Аутоматско заузимање – које се примењује за аутоматске променљиве тј за аргументе функција или локално дефинисане променљиве. Меморијски простор за овакве промељиве се заузима сваки пут кад се у току извршавања програма уђе у програмски блок који садржи дефиницију промељиве, а ослобађа се на крају тог програмског блока.

Заједничко за статичко и аутоматско заузимање меморије је да величина коју је потребно заузети мора бити константна вредност:

```
static int var = 5;
int array1[10];
int array2[var];     /* compiler error */
```

Трећи начин заузимања меморије - динамичко заузимање није подржано преко Це променљивих, али је доступно преко стандардних библиотека (*stdlib*) и преко C++ оператора.

- Динамичко заузимање – техника која се примењује у случајевима када информације о величини и трајању променљивих нису доступне пре почетка извршавања програма. Представља акцију где програмер експлицитно затражи заузимање меморије. Са обзиром да заузимањем управља програмер, дужан је и да заузету меморију на исправан начин ослободи. У супротном може доћи до грешака које је тешко уочити и отклонити (цурење меморије).

У Це језику функције стандардне библиотеке (*stdlib*) за динамичко заузимање и ослобађање меморије су:

- calloc() – заузимање и постављање заузете меморије на вредност 0
- malloc() – заузимање меморије
- realloc() – реалокација претходно заузете меморијске зоне
- free() – ослобађање заузете меморије

У C++ језику на располагању су оператори:

- new – заузимање меморије
- new[] – заузимање меморије за низ елемената
  - повратна вредност *new* оператора је показивач на почетак заузете меморијске зоне
- delete – ослобађање меморије
- delete[] – ослобађање меморије за заузет низ елемената
  - *delete* оператори се користе над показивачем који показује на почетак меморијске зоне коју треба ослободити

Напомена: Да би се избегле грешке у пару увек треба користити:

- calloc/malloc – free
- new – delete
- new[] – delete[]

Од интереса за овај предмет ће бити C++ оператори, а њихова примена описана је у примерима који следе.

Пример *new* и *delete* оператора:

```
int* ptr = NULL;
ptr = new int;      // allocates memory for one integer and
                    // returns a pointer to that object
*ptr = 10;          // sets the object value to 10 */
delete ptr;         // deletes the object (frees the memory)
```

```
int* ptr = new int(5);
// allocates memory for one integer with value 5

delete ptr;         // frees the memory
```

Пример: *new[]* и *delete[]* оператора:

```
int arraySize = 0;
int* array;
cout << "Please enter desired array size: ";
cin >> arraySize;
array = new int[arraySize];
. . . // do something with array
```

```
delete[] array;
```

```
int** matrix;
int rows = 2;
int cols = 2;
matrix = new int* [rows];
// allocates memory for an array of pointers and returns
// a pointer to the allocated array of pointers
// note: return value is pointer to a pointer

for (int i = 0; i < rows; i++)
     matrix[i] = new int[cols];
// allocating memory for each pointer from the array of
// pointers

for (int i = 0; i < rows; i++)
     delete[] matrix[i];
// freeing memory of each array

delete[] matrix;
// freeing memory for the array of pointers
```

2.3 Вишедимензионални низови

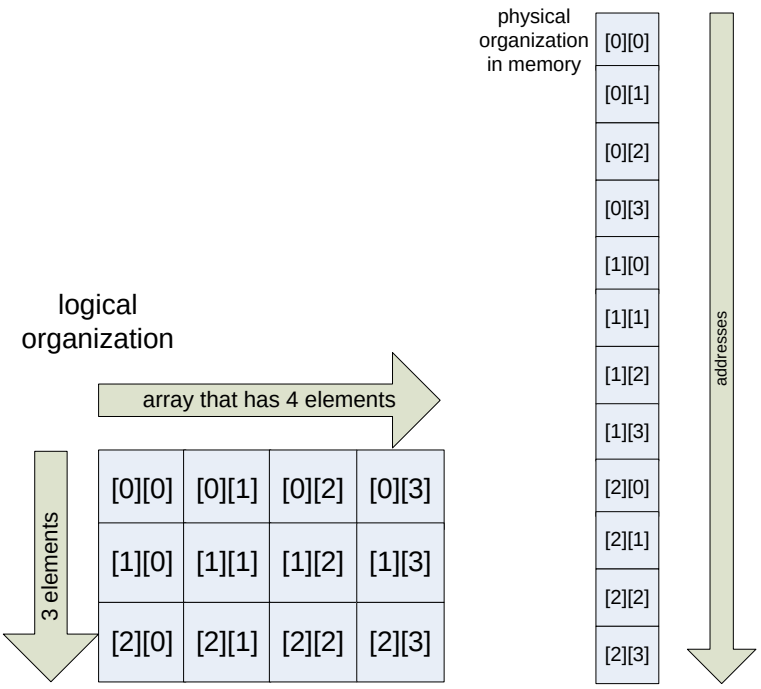
У Цеу вишедимензионални низови су у ствари низови низова. Ако говоримо о дводимензионалном низу то је у суштини низ елемената од којих је сваки елемент тог низа, опет низ. Постоји разлика у начину заузимања меморије ког дводимензионалних низова и то само у физичкој организацији, што је објашњено у примери у наставку:

Пример статички заузетог дводимензионалног низа:

```
int matrix[3][4];
```

- то је низ који има 3 елемента
- Питање: Ког су типа та 3 елемента?
- Одговор: Елементи тог низа су типа низ од 4 целобројне вредности.

Логичка и физичка организацији дата је на слици:



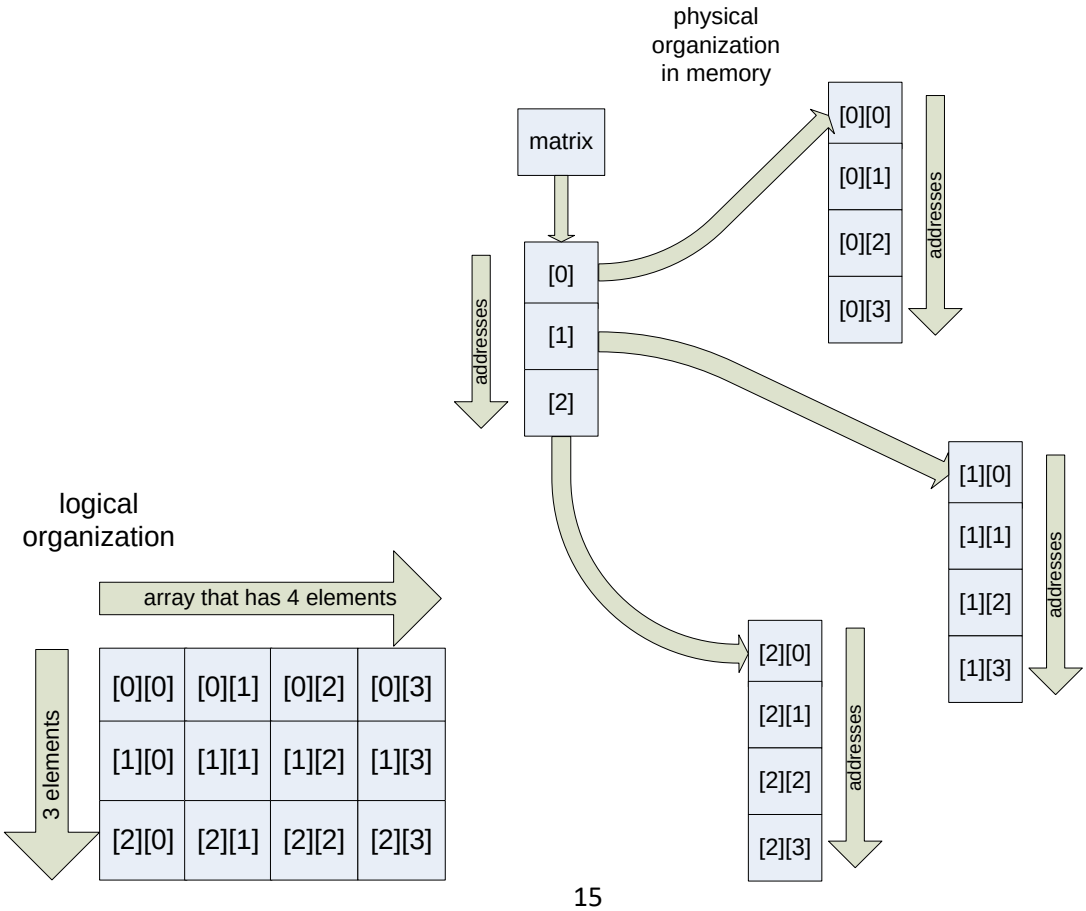
Еквивалентна дефиниција горенаведеног низа је:

```
typedef int array4[4];
array4 matrix[3];         // equals to int matrix[3][4];
array4 y;                 // equal to int y[4];
int i;
int j;

y[i];                     // equal to *(y + i*sizeof(int))

matrix[i][j];
// equals to
// (matrix[i])[j]
// (*(matrix + i*sizeof(array4)))[j]    // array4 T
// T[j]
// *(T + j*sizeof(int))
// *((*(matrix + i*sizeof(array4))) + j*sizeof(int))
// sizeof(array4) = 4*sizeof(int)
// *(*(matrix + i*4*sizeof(int)) + j*sizeof(int))
```

Пример динамички заузетог дводимензионалног низа:

```
int** matrix;
matrix = new int*[3];
matrix[0] = new int[4];
matrix[1] = new int[4];
matrix[2] = new int[4];

matrix[i][j];
// equals to
// (matrix[i])[j]
// (*(matrix + i*sizeof(int*)))[j]
// T[j]
// *(T + j*sizeof(int))
// *(*(matrix + i*sizeof(int*))) + j*sizeof(int))
// *(*(matrix + i*4) + j*4)
```

Логичка и физичка организацији дата је на слици:



2.4 Прослеђивање низова функцији

Низови се функцијама не могу проследити по вредности. Прослеђивање се увек обавља на тај начин што се проследи адреса првог елемента. Последица је да су наведене декларације практично исте:

```
int func(int arr[10]);
int func(int arr[]);
int func(int* arr);
```

Број елемената наведен у угластим заградама се игнорише. За вишедимензионалне низове, све димензије се морају навести осим прве.

Питање: Зашто?

```
int func(int mat[][7]);
int func(int (*)[7]);
int func(int (*mat)[7]);
```

Уколико смо користили динамичко заузимање меморије за дводимензионални низ, тада се ради о показивачу на показивач па декларација и позив функције изгледају овако:

```
void initMatrix(int** mat, int rowNumber, int colNumber)
{
	for (int i = 0; i < rowNumber; i++)
	{
		for (int j = 0; j < colNumber; j++)
		{
			mat[i][j] = 0;
		}
	}
}

void foo()
{
	int** matrix;
	int rows = 2;
	int cols = 2;
	matrix = new int* [rows];

	for (int i = 0; i < rows; i++)
		matrix[i] = new int[cols];

	initMatrix(matrix, rows, cols);

	for (int i = 0; i < rows; i++)
```

```
                delete[] matrix[i];

            delete[] matrix;
        }
```

ЗАДАТАК

I. Написати програм за рад са матрицом од **n x m** целобројних елемената. Програм се састоји из три класе:

- ***Matrix1D (Matrix1D.h* i *Matrix1D.cpp)*** – која садржи имплементацију матрице као једнодимензионог низа
- ***Matrix2D (Matrix2D.h* i *Matrix2D.cpp)*** – која садржи имплементацију матрице као низ показивача на низове (редове матрице)
- ***MatrixVec (MatrixVec.h i MatrixVec.cpp)*** – која садржи имплементацију матрице као вектор вектора (користити класу std::*vector* из стандардне библиотеке шаблона *STL*

За сваку од три класе потребно је:

a) У заглављу класе додати приватно поље које представља матрицу.

b) Имплементирати иницијализацију матрице у оквиру конструктора. Конструктор прима три параметра: број редова (***rows***), број колона (**cols**) и параметар за одређивање максималне вредности елемента матрице. (***range***). Попуњавање елемената матрице свести на доделу случајних вредности које ће бити мање или једнаке од датог параметра *range*. Случајне вредноасти генерисати употребом функције:

- *int* ***rand*** *();*

c) Имплементирати методу за испис матрице на екран:

- ***void print();***

Испис форматирати тако да сваки ред матрице буде исписан у новом реду, а елементи унутар једног реда одвојени са "\t" карактером.

d) Имплементирати методу која рачуна број непарних елемената у свакој од колона матрице и исписује на екран:

- ***void process();***

e) У функцији ***main*** направити објекат за сваку од три класе и позвати редом методе *print* па *process* над њима.

f) Задавање димензија матрице **n** и **m** реализовати уз помоћ аргумената командне линије

ДОДАТНИ ЗАДАТАК

I. У сваку класу додати функцију за ротирање матрице за 90 степени у смеру казаљке на сату. Ограничити се само на квадратне матрице.

II. Од три направљене класе за рад са матрицама направити статичку библиотеку и испробати њен рад у новом пројекту који ће је користити.