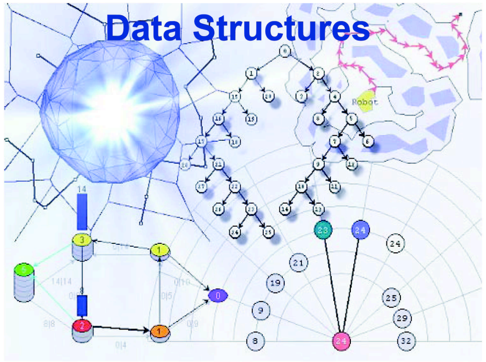
BBM 201
DATA STRUCTURES

Lecture 3:
Representation of Multidimensional Arrays



What is an Array?

- An array is a fixed size sequential collection of elements of identical types.
- A multidimensional array is treated as an array of arrays.
  - Let a be a k-dimensional array; the elements of A can be accessed using the following syntax:

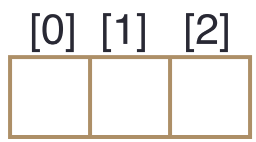
    $A[\,i_1\,][\,i_2\,]\ldots[\,i_k\,]$

The following loop stores 0 into each location in a two dimensional array A :

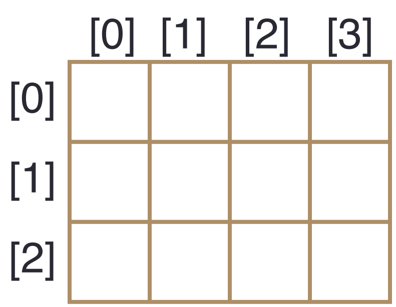
```
int row, column;
int A[3][4];
    for (row = 0; row < 3; row++)
    {
     for (column = 0; column < 4; column++)
         {
            A[row][column] = 0;
        }
    }
```

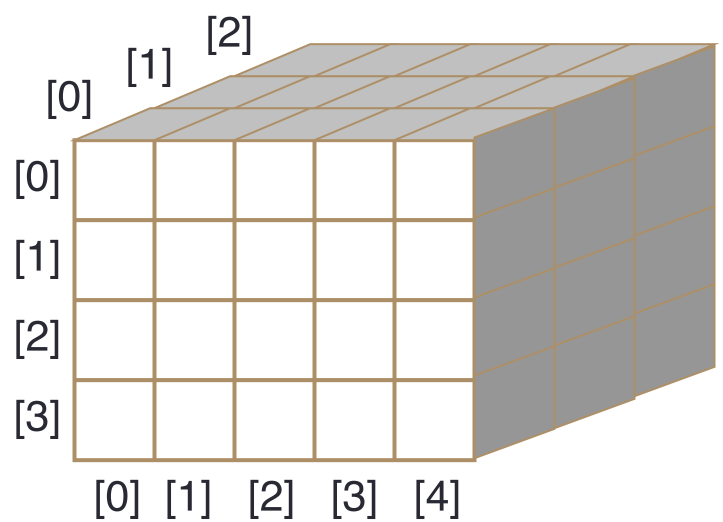
Definition of a Multidimensional Array

- **One-dimensional** arrays are linear containers.



Multi-dimensional Arrays



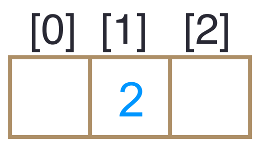


abstract view

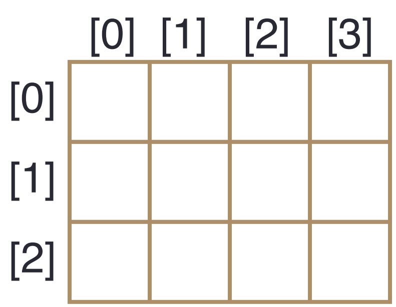
Definition of a Multidimensional Array

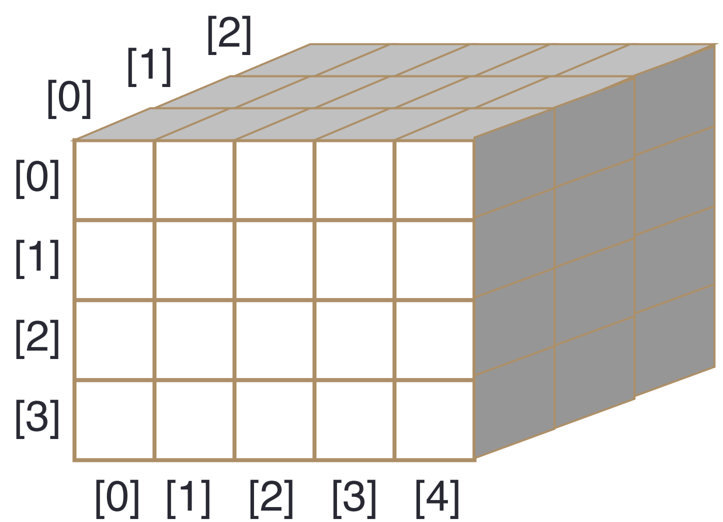
- **One-dimensional** arrays are linear containers.

```
int A[3];
A[1]=2;
```



Multi-dimensional Arrays



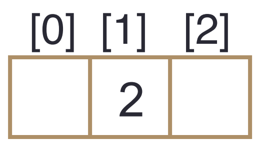


abstract view

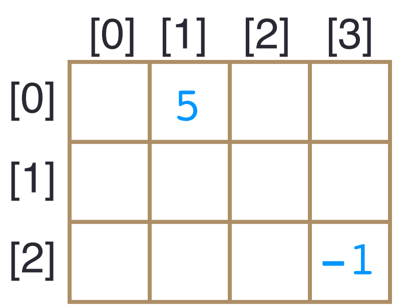
Definition of a Multidimensional Array

- **One-dimensional** arrays are linear containers.

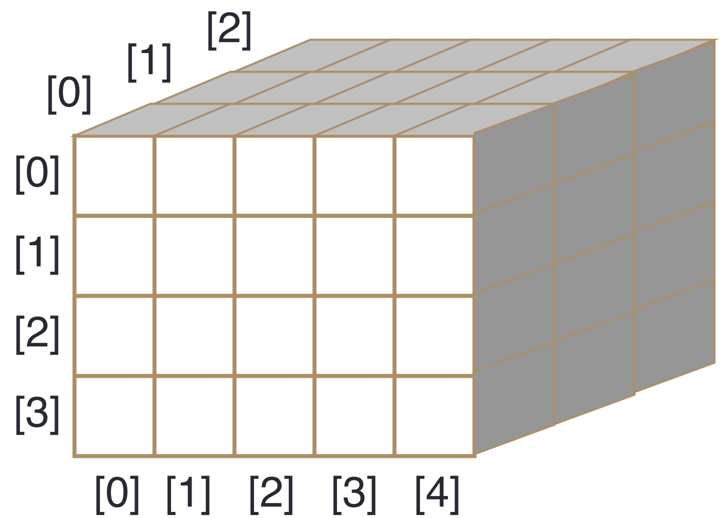
```
int A[3];
A[1]=2;
```



Multi-dimensional Arrays




```
int A[3][4];
A[0][1]=5;
A[2][3]=-1;
```

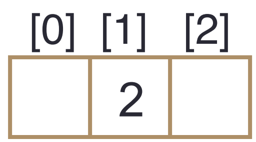


abstract view

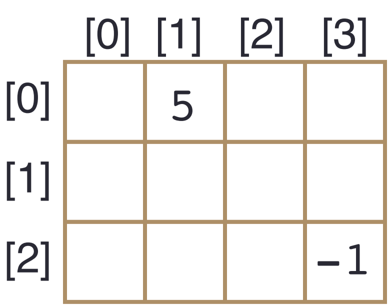
Definition of a Multidimensional Array

- **One-dimensional** arrays are linear containers.

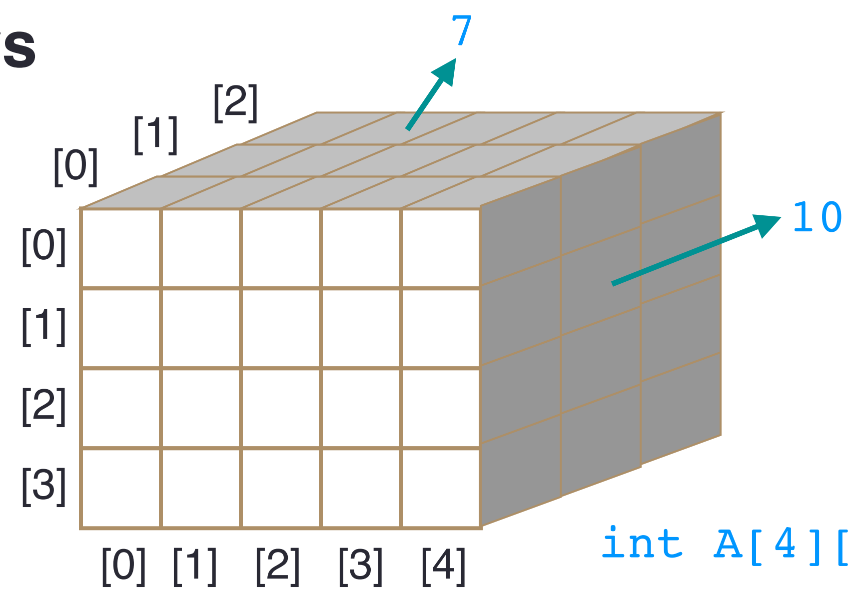
```
int A[3];
A[1]=2;
```



Multi-dimensional Arrays




```
int A[3][4];
A[0][1]=5;
A[2][3]=-1;
```



abstract view

```
int A[4][5][3];
A[1][4][1]=10;
A[0][1][2]=7;
```

Dynamic Allocation of 2d Arrays

A dynamically allocated 2d array
of dims: [3][5] could be considered
as a matrix with 3 rows and 5 columns

```
int** A;
A = new int*[3];
for(int i=0;i<3;i++)
    A[i] = new int[5];
```

A:

1	0	12	-1	4
7	-3	2	5	6
-5	-2	2	9	7

Dynamic Allocation of 2d Arrays

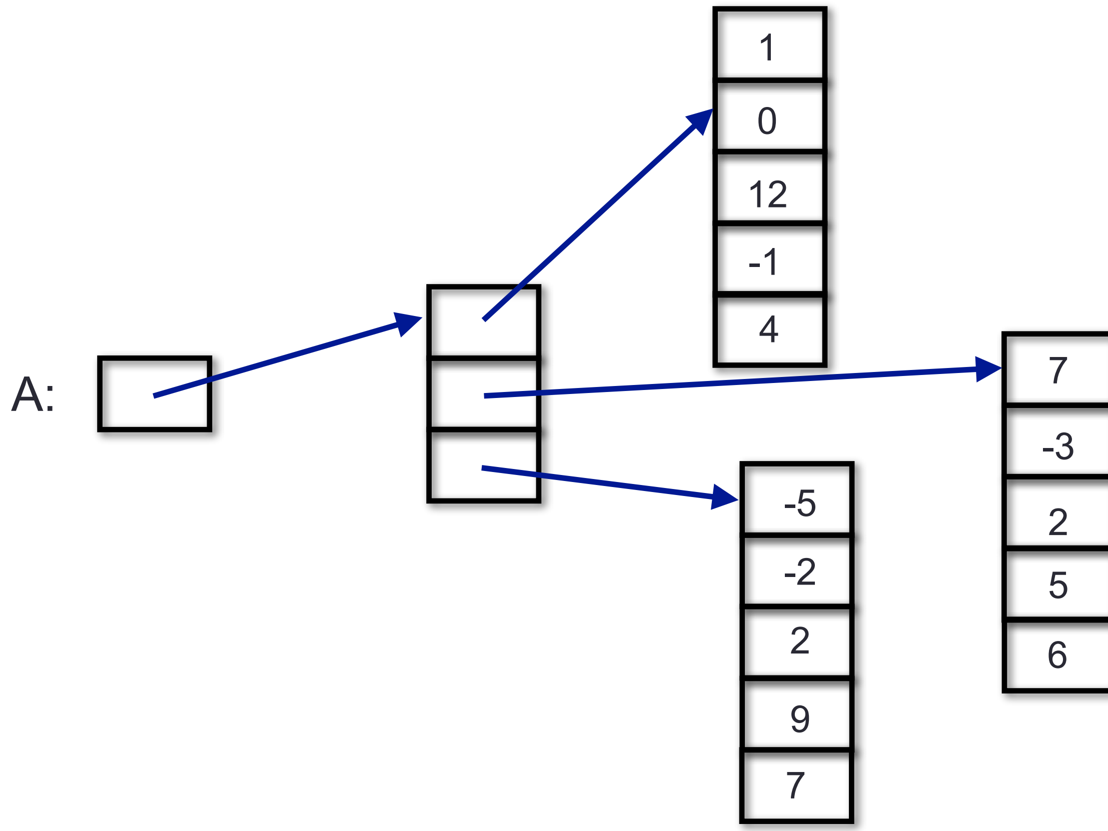
A dynamically allocated 2d array of dims: [3][5] could be considered as a matrix with 3 rows and 5 columns

But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 5 items

```
int** A;
A = new int*[3];
for(int i=0;i<3;i++)
    A[i] = new int[5];
```

A:

1	0	12	-1	4
7	-3	2	5	6
-5	-2	2	9	7



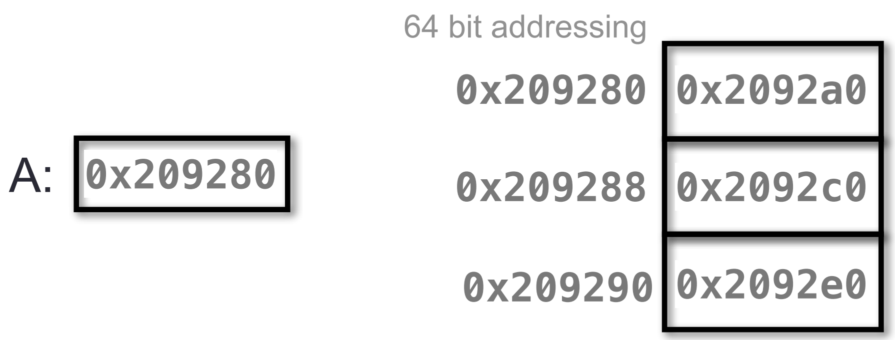
Dynamic Allocation

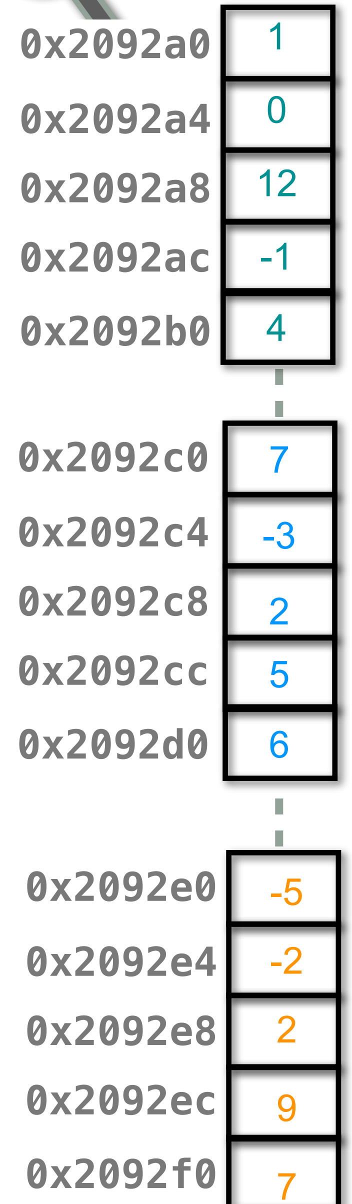
behind the scenes...

```
int** A;
A = new int*[3];
for(int i=0;i<3;i++)
     A[i] = new int[5];
```

A:

1	0	12	-1	4
7	-3	2	5	6
-5	-2	2	9	7





Array size

- In a *d-dimensional* array, which is declared as

```
<type> a[N1][N2]…[Nd];
```

the number of items is: $$\prod_{i=1}^{i=d} N_i$$

Example: What is the number of items in `a[20][20][1]`?

Storage Allocation

- The storage arrangement shown in this example uses the array subscript, i.e. array indices.

Array declaration: `int a[3][4];`

Array elements:

```
a[0][0]  a[0][1]  a[0][2]  a[0][3]
a[1][0]  a[1][1]  a[1][2]  a[1][3]
a[2][0]  a[2][1]  a[2][2]  a[2][3]
```

Two-Dimensional Storage Allocation

A 2d array declared in C++ as:

int A[3][5];

could be considered as a matrix

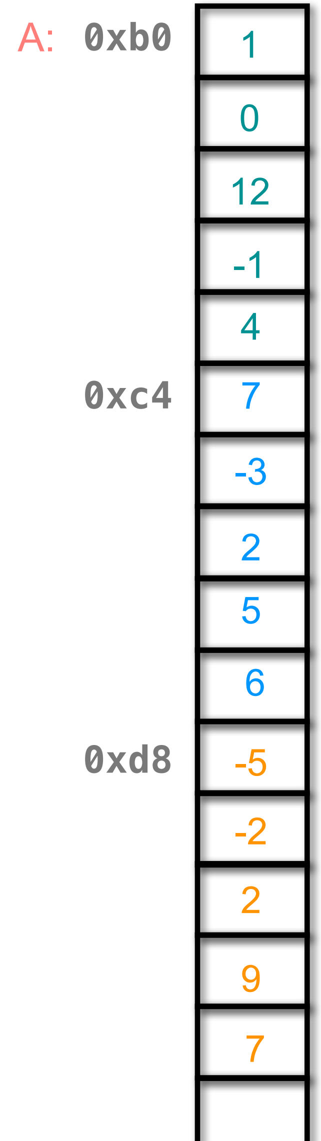
with 3 rows and 5 columns

A:

1	0	12	-1	4
7	-3	2	5	6
-5	-2	2	9	7

But in reality, it has a linear structure.

```
A    :    0xb0
A[0] :    0xb0
0xb0 :    1
0xb4 :    0
0xb8 :    12
0xbc :    -1
0xc0 :    4

A[1] :    0xc4
0xc4 :    7
0xc8 :    -3
0xcc :    2
0xd0 :    5
0xd4 :    6

A[2] :    0xd8
0xd8 :    -5
0xdc :    -2
0xe0 :    2
0xe4 :    9
0xe8 :    7
```



Two-Dimensional Storage Allocation

A 2d array declared in C++ as:

`int A[3][5];`

could be considered as a matrix with 3 rows and 5 columns

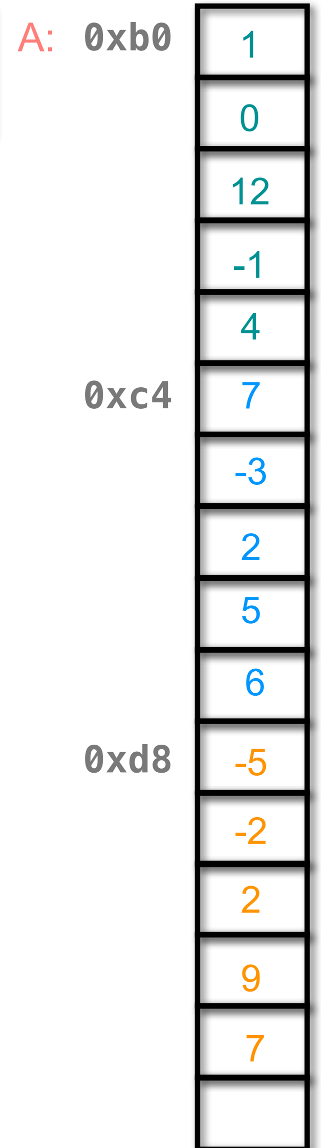
Spoiler Alert:
Row major ordering of elements

A:

1	0	12	-1	4
7	-3	2	5	6
-5	-2	2	9	7

But in reality, it has a linear structure.

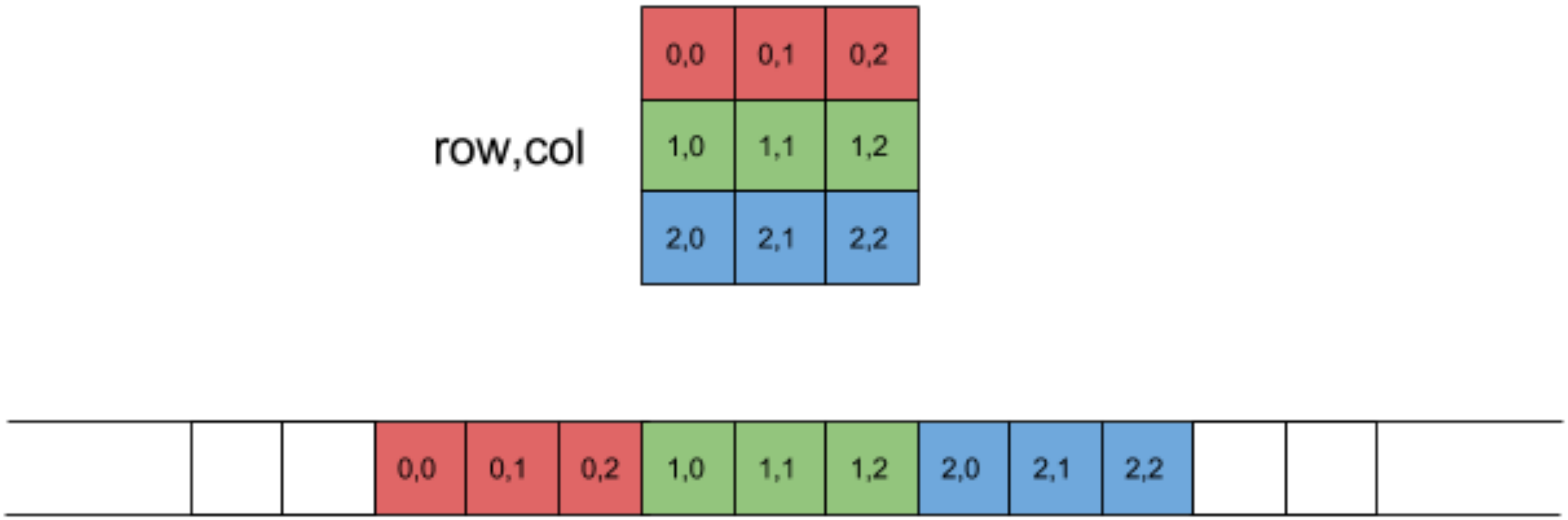
```
A    :    0xb0
A[0] :    0xb0
0xb0 :    1
0xb4 :    0
0xb8 :    12
0xbc :    -1
0xc0 :    4

A[1] :    0xc4
0xc4 :    7
0xc8 :    -3
0xcc :    2
0xd0 :    5
0xd4 :    6

A[2] :    0xd8
0xd8 :    -5
0xdc :    -2
0xe0 :    2
0xe4 :    9
0xe8 :    7
```



Memory Storage

- There are two types of placement for multidimensional arrays in memory:
  - Row major ordering
  - Column major ordering

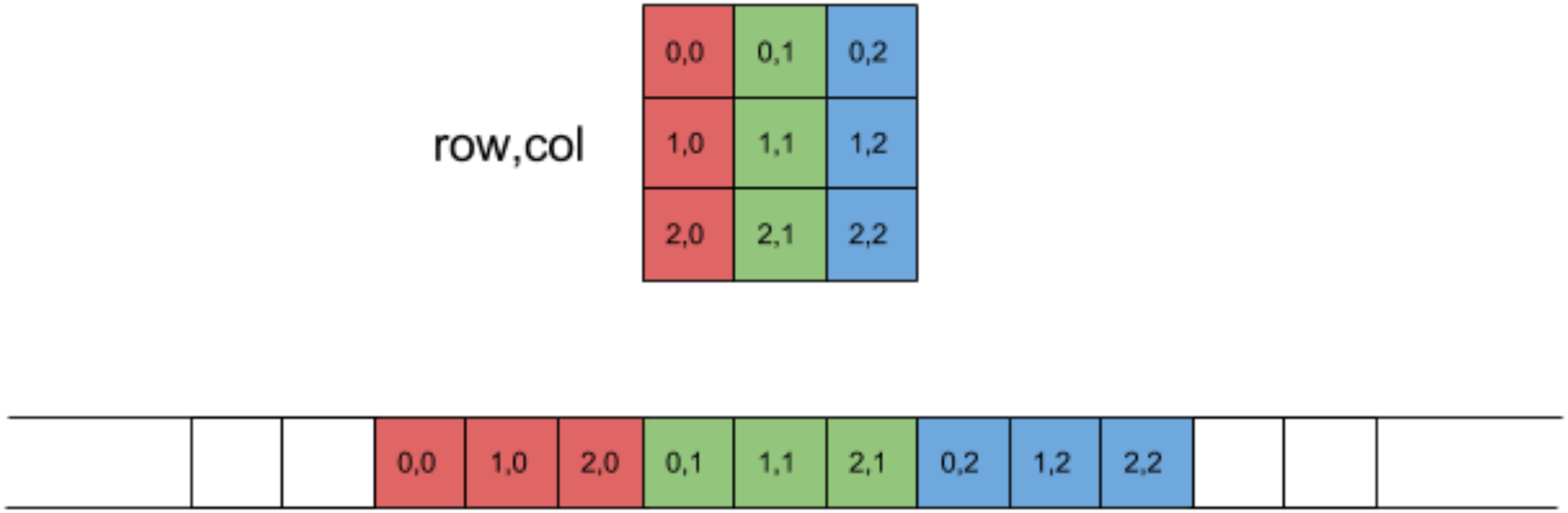
Raw Major Ordering



$$offset = i_{row} * NCOLS + i_{col}$$

https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Column Major Ordering



$$offset = i_{col} * NROWS + i_{row}$$

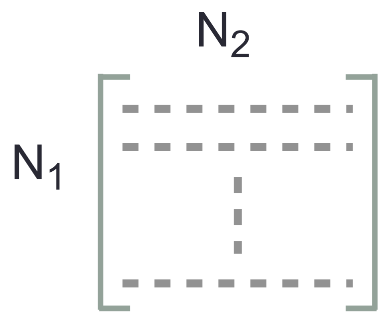
https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Memory Storage

- There are two types of placement for multidimensional arrays in memory:
  - Row major ordering
  - Column major ordering

<u>Example:</u> In an array which is defined as $A[N_1][N_2]$, if the memory address of A[0][0] is α, then what is the memory address of A[i][0] (according to row major ordering)?

$$\alpha + i * N_2$$



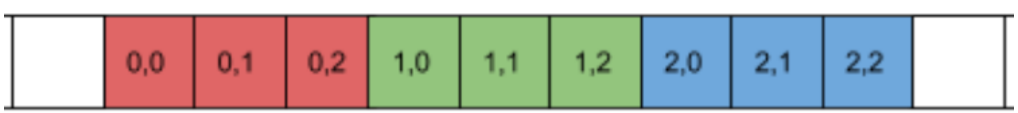
Multi-dimensional Arrays

- In row-major layout of multi-dimensional arrays, the *last* index is the fastest changing.
  - In case of matrices the last index is columns, so this is equivalent to the previous definition.

$$offset = n_d + N_d(n_{d-1} + N_{d-1}(n_{d-2} + N_{d-2}(\ldots + N_2 n_1)\ldots))) = \sum_{i=1}^{d} \left(\prod_{j=i+1}^{d} N_j \right) n_i$$

  - For a matrix (2D):

$$offset = n_2 + N_2 \cdot n_1$$



- the *last* index is the fastest changing

https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

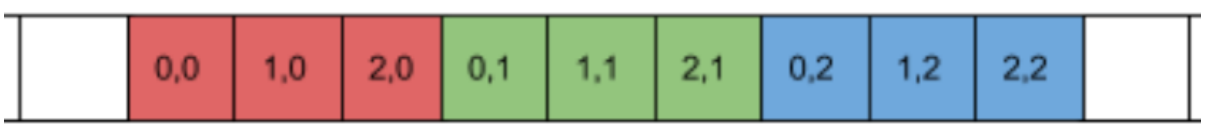
Multi-dimensional Arrays

- In column-major layout of multi-dimensional arrays, the *first* index is the fastest changing.
  - In case of matrices the first index is rows, so this is equivalent to the previous definition.

$$offset = n_1 + N_1(n_2 + N_2(n_3 + N_3(\ldots + N_{d-1}n_d)\ldots))) = \sum_{i=1}^{d}(\prod_{j=1}^{i-1} N_j)n_i$$

  - For a matrix (2D):

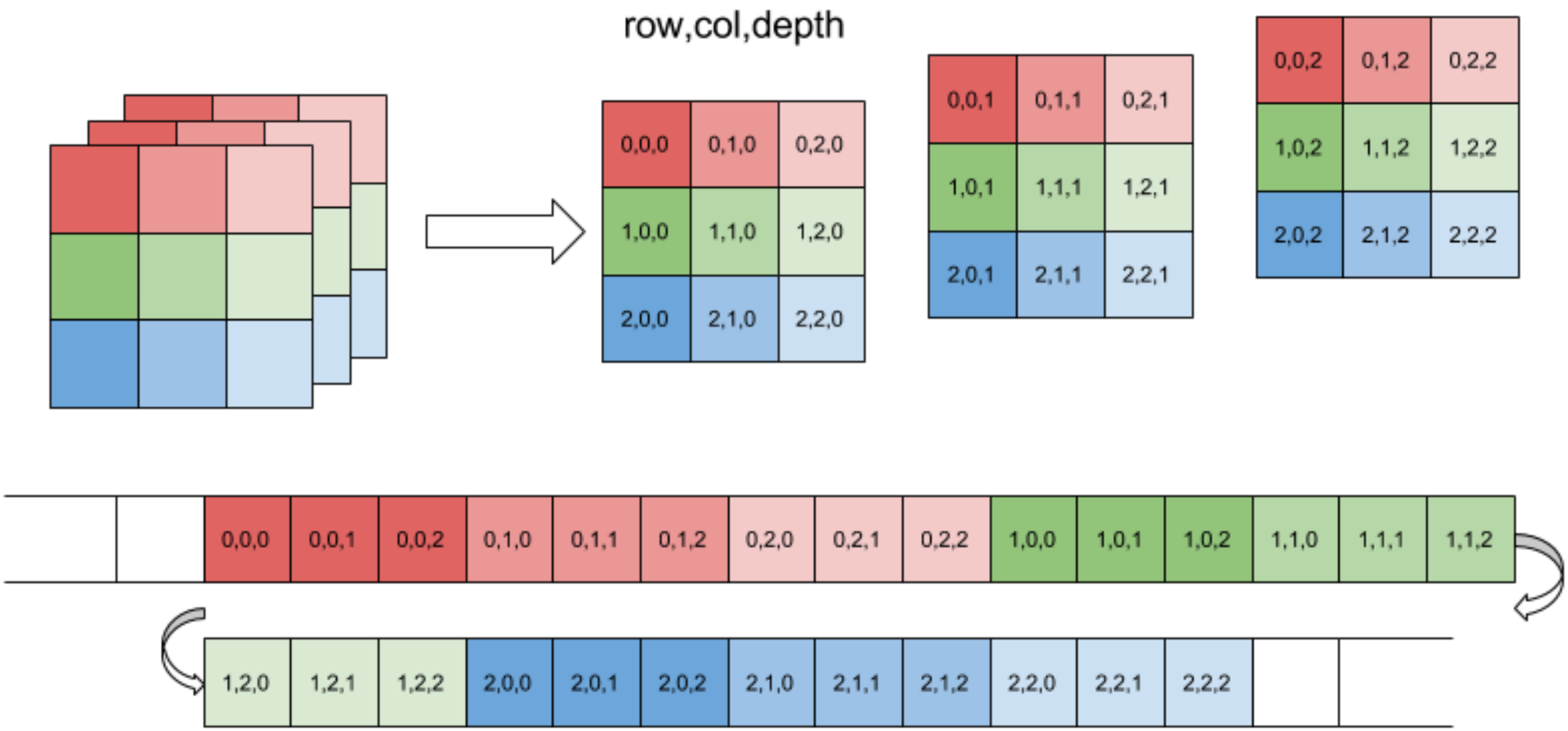
$$offset = n_1 + N_1 \cdot n_2$$



- the *first* index is the fastest changing

https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

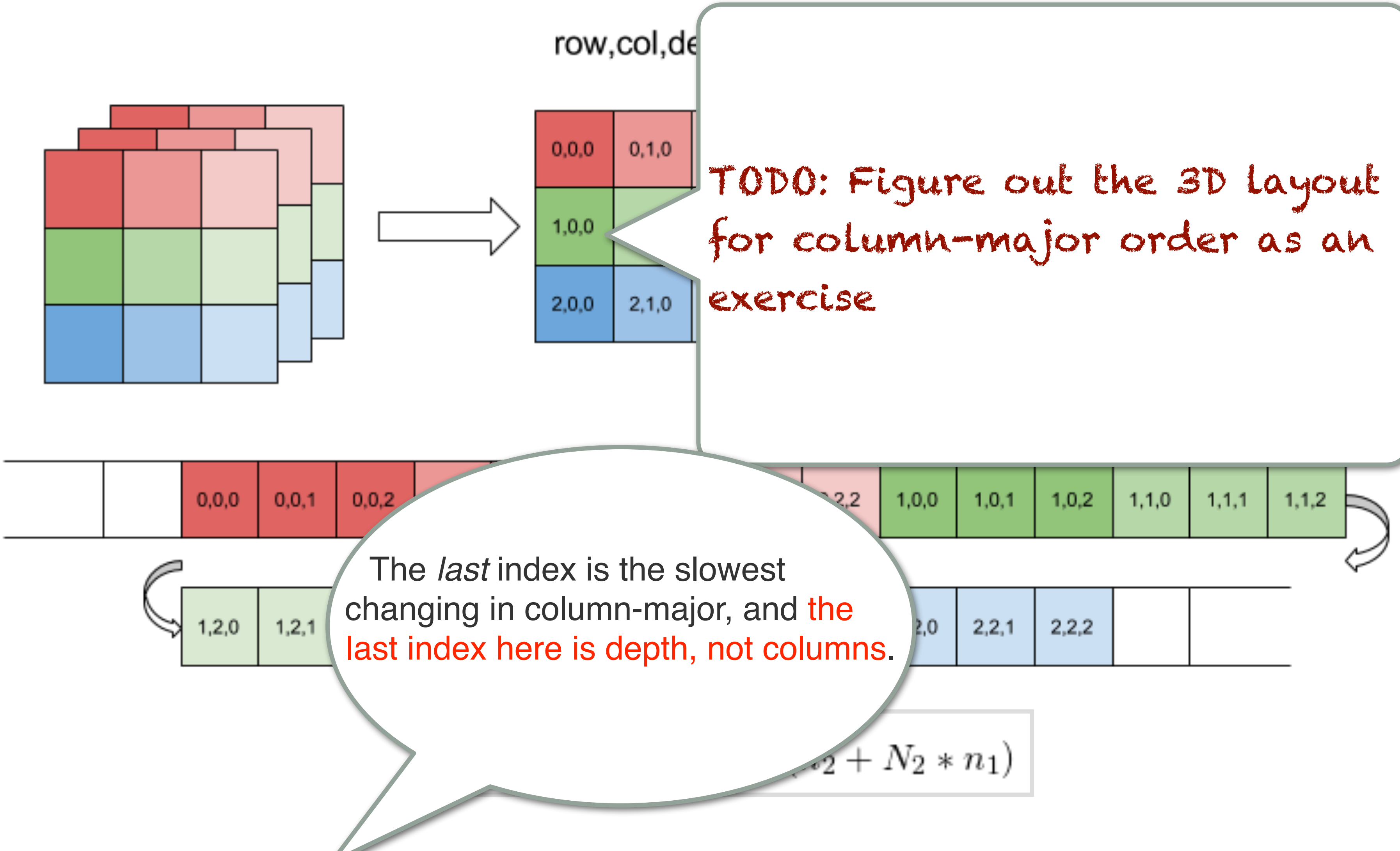
3D row-major order layout



$$offset = n_3 + N_3 \cdot (n_2 + N_2 \cdot n_1)$$

https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

3D row-major order layout

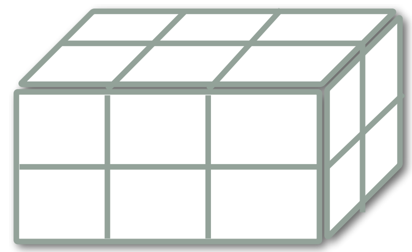


https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Memory Storage

- For a three-dimensional array $A[N_1][N_2][N_3]$

what is the memory storage like?



- Example: char y[2][3][2]

which slice?

which row?

which column?

- Assuming row-major order, what is the memory address of y[1][2][0], if the memory address of y[0][0][0] α?

Memory Storage

Suppose the memory address of a[0][0][0] is α;
the memory address of `a[i][0][0]` is:

$$\alpha + i * N_2 * N_3$$

Therefore, the memory address of `a[i][j][k]` becomes:

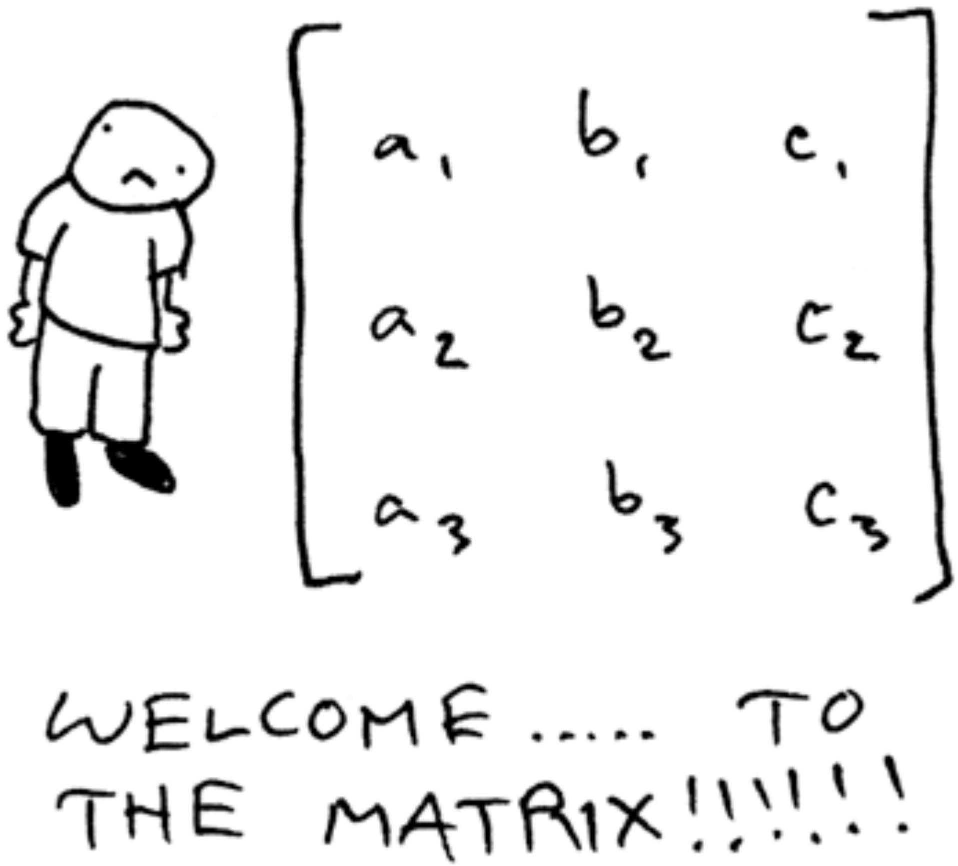
$$\alpha + i * N_2 * N_3 + j * N_3 + k$$

The memory address of $\texttt{a[i}_1\texttt{][i}_2\texttt{][i}_3\texttt{]...[i}_n\texttt{]}$ is:

$$\alpha + \sum_{j=1}^{n} i_j a_j \qquad \text{where,} \qquad a_j = \prod_{k=j+1}^{n} N_k \quad 0 \le j \le n-1$$

$$a_n = 1$$

Lower/Upper Triangular Matrix
Band Matrix
Sparse Matrix



a1 b1 c1
a2 b2 c2
a3 b3 c3
WELCOME ..... TO
THE MATRIX!!!!!!

Lower Triangular Matrix

Triangular matrix

Upper triangular matrix

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \ldots & u_{1n} \\ 0 & u_{22} & u_{23} & & \vdots \\ 0 & 0 & u_{33} & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \ldots & \ldots & 0 & u_{nn} \end{bmatrix}$$

Lower triangular matrix

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \ldots & 0 \\ l_{21} & l_{22} & 0 & & \vdots \\ l_{31} & l_{32} & l_{33} & & \vdots \\ \vdots & & & \ddots & 0 \\ l_{n1} & \ldots & \ldots & \ldots & l_{nn} \end{bmatrix}$$

Lower Triangular Matrix

- Does the definition of a special data structure for triangular matrix provide any benefits over a typical matrix in terms of memory and processing time?
- We can insert the items in a single dimensional array:
-

ALT	a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	a_{33}
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

- Number of items in the array becomes:

$$1+2+\ldots+(n-1)+n=\frac{n(n+1)}{2}$$

$$a = \begin{bmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Lower Triangular Matrix

- How can we find the position of u[i][j] in the array?
  - Answer: i=1, there is one item in the 0^{th} row, 2 items in the 1^{st} row.
  - i=2, there is one item in the 0^{th} row, 2 items in the 1^{st} row, 3 items in the 2^{nd} row.
  - Therefore the address of u[i][j] in the array is calculated as below:

$$k = \sum_{t=0}^{i}(t) + (j) = (0 + 1 + 2 + \ldots + i) + (j)$$

$$= \frac{i(i+1)}{2} + (j)$$

Lower Triangular Matrix

```
void main(void){
    int alt[MAX_SIZE];
    int i, n;
    cin>>n; //matrix size
    readtriangularmatrix(alt,n);
    for(i=0; i<=n*(n+1)/2-1; i++)
        cout<< alt[i]<<" ";

    i=gettriangularmatrix(3,0,n);
    if(i==-2)
        cout<<"\n invalid index\n";
    else if(i==-1)
        cout<<"\n access to the upper triangular\n";
    else
        cout<<"\n  the position in 'alt' matrix:"<<i<<" value:"<<
alt[i]<<"\n";
```

Lower Triangular Matrix

```
void readtriangularmatrix(int alt[], int n)
{
      int i, j, k;
      if(n*(n+1)/2 > MAX_SIZE){
          cout<<"\n invalid array size \n";
          exit(-1);
      }
      else
         for(i=0; i<=n-1; i++){
             k=(i+1)*i/2;
             for(j=0; j<=i; j++)
                    cin>>alt[k+j];
         }
}
```

```
int gettriangularmatrix(int i, int j, int n){

    if(i<0 || i>=n || j<0 || j>=n){
            //invalid index;
            return -2;
     }
     else if(i>=j) //valid index
            return (i+1)*i/2+j;
     else return -1; //outside of the
triangular; value is zero
}
```

Upper Triangular Matrix

Triangular matrix

Upper triangular matrix

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & . & . & . & u_{1n} \\ 0 & u_{22} & u_{23} & & & & . \\ 0 & 0 & u_{33} & & & & . \\ . & & & . & & & . \\ . & & & & . & & . \\ . & & & & & . & . \\ 0 & . & . & . & . & 0 & u_{nn} \end{bmatrix}$$

Lower triangular matrix

$$L = \begin{bmatrix} l_{11} & 0 & 0 & . & . & . & 0 \\ l_{21} & l_{22} & 0 & & & & . \\ l_{31} & l_{32} & l_{33} & & & & . \\ . & & & . & & & . \\ . & & & & . & & . \\ . & & & & & . & 0 \\ l_{n1} & . & . & . & . & . & l_{nn} \end{bmatrix}$$

Upper Triangular Matrix

- How can we find the position of u[i][j] in the array?

- Lower =>

$$k = \sum_{t=0}^{i} (t) + (j) = (0 + 1 + 2 + \ldots + i) + (j)$$

$$= \frac{i(i+1)}{2} + (j)$$

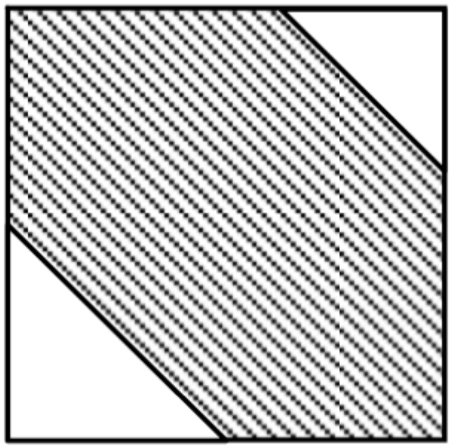
- Upper =>

Band Matrix

$$\begin{bmatrix} B_{11} & B_{12} & 0 & \cdots & \cdots & 0 \\ B_{21} & B_{22} & B_{23} & \ddots & \ddots & \vdots \\ 0 & B_{32} & B_{33} & B_{34} & \ddots & \vdots \\ \vdots & \ddots & B_{43} & B_{44} & B_{45} & 0 \\ \vdots & \ddots & \ddots & B_{54} & B_{55} & B_{56} \\ 0 & \cdots & \cdots & 0 & B_{65} & B_{66} \end{bmatrix}$$

Matrix (n, a) : n by n matrix, non-zero entries are confined to a diagonal band, comprising the main diagonal and zero or more diagonals (a-1) on either side.

Band Matrix



b

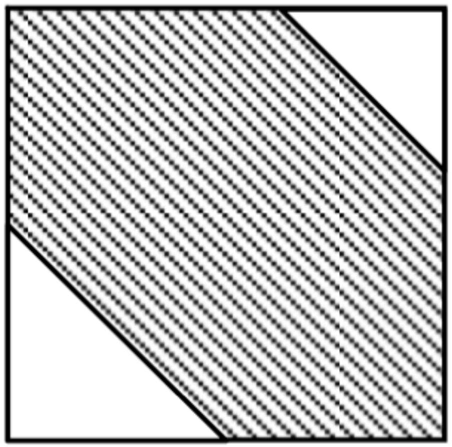
$$A = \begin{bmatrix} d_{00} & d_{01} & 0 & 0 \\ d_{10} & d_{11} & d_{12} & 0 \\ d_{20} & d_{21} & d_{22} & d_{23} \\ 0 & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

4x4

a

n=4
a=3
b=2

Band Matrix



b

$$A = \begin{bmatrix} d_{00} & d_{01} & 0 & 0 \\ d_{10} & d_{11} & d_{12} & 0 \\ d_{20} & d_{21} & d_{22} & d_{23} \\ 0 & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

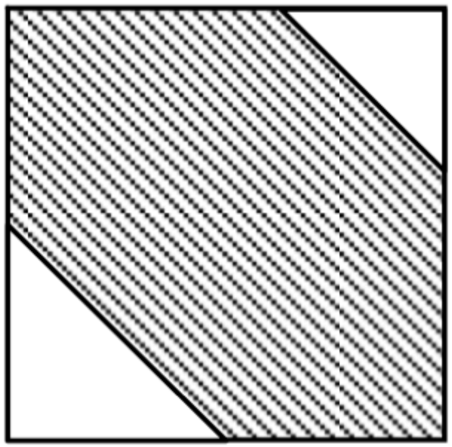
4x4

a

n=4
a=3
b=2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
d_{20}	d_{31}	d_{10}	d_{21}	d_{32}	d_{00}	d_{11}	d_{22}	d_{33}	d_{01}	d_{12}	d_{23}

Band Matrix



b

$$A = \begin{bmatrix} d_{00} & d_{01} & 0 & 0 \\ d_{10} & d_{11} & d_{12} & 0 \\ d_{20} & d_{21} & d_{22} & d_{23} \\ 0 & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

a

4x4

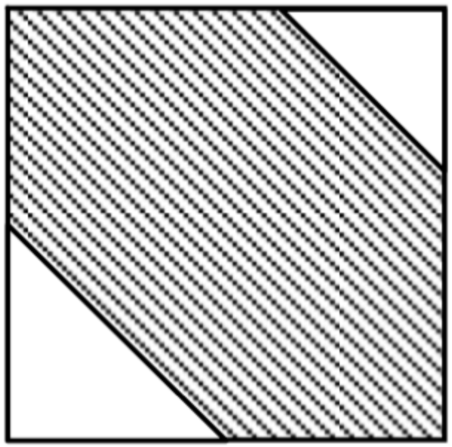
n=4
a=3
b=2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
d_{20}	d_{31}	d_{10}	d_{21}	d_{32}	d_{00}	d_{11}	d_{22}	d_{33}	d_{01}	d_{12}	d_{23}

$$n + (n-1) + (n-2) + \ldots + (n-(a-1))$$ # of elements below triangle (including diagonal)

$$(n-1) + (n-2) + \ldots + (n-(b-1))$$ # of elements above triangle

Band Matrix



b

$$A = \begin{bmatrix} d_{00} & d_{01} & 0 & 0 \\ d_{10} & d_{11} & d_{12} & 0 \\ d_{20} & d_{21} & d_{22} & d_{23} \\ 0 & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

4x4

a

n=4
a=3
b=2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
d_{20}	d_{31}	d_{10}	d_{21}	d_{32}	d_{00}	d_{11}	d_{22}	d_{33}	d_{01}	d_{12}	d_{23}

$$n + (n-1) + (n-2) + \ldots + (n-(a-1))$$

\# of elements on and below the diagonal

$$(n-1) + (n-2) + \ldots + (n-(b-1))$$

\# of elements above the diagonal

$$n \cdot (a+b-1) - \frac{a \cdot (a-1)}{2} - \frac{b \cdot (b-1)}{2}$$

Total # of elements

Band Matrix

- What is the number of items in the array?
  - Number of items **on** and **below** the diagonal:

$$n + (n-1) + (n-2) + \ldots + n - (a-1)$$

  - Number of items **above** the diagonal:

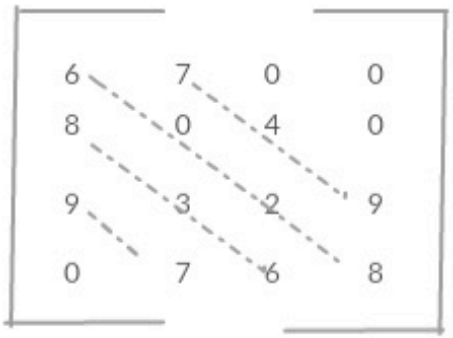
$$(n-1) + (n-2) + \ldots + n - (b-1)$$

  - Sum of these becomes:

$$Sum = n + (n-1) + (n-2) + \ldots + n - (a-1) + (n-1) + (n-2) + \ldots + n - (b-1)$$

$$= n(a+b-1) - \frac{(a-1)a}{2} - \frac{(b-1)b}{2}$$

Band Matrix



$$A = \begin{bmatrix} d_{00} & d_{01} & 0 & 0 \\ d_{10} & d_{11} & d_{12} & 0 \\ d_{20} & d_{21} & d_{22} & d_{23} \\ 0 & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

1

-2 -1 0

n=4
a=3
b=2

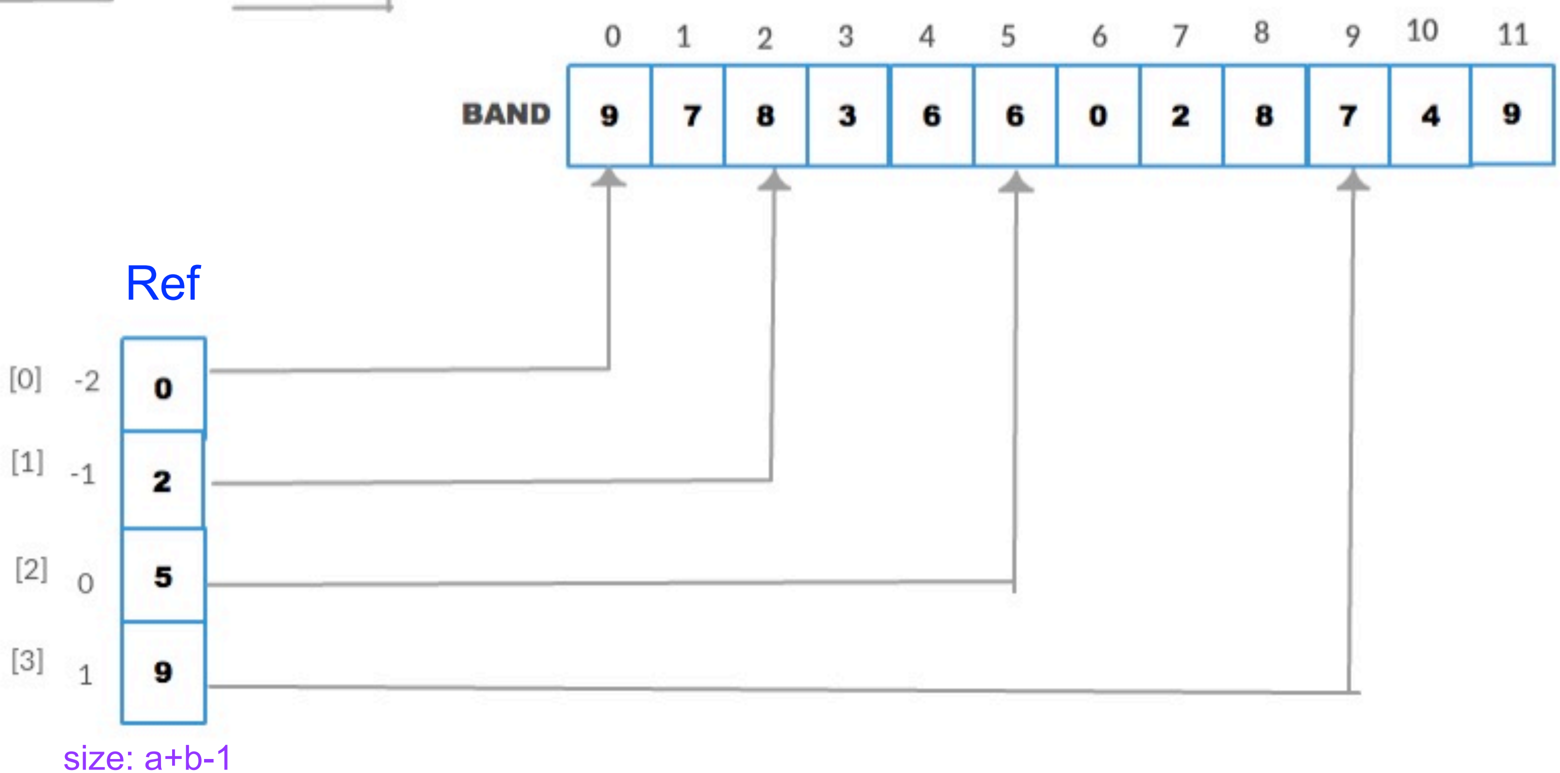
d_{ij} -> j-i

d_{20} => d_{31} -> 2
-2: below the main diagonal

[1-a, b-1]

Fetch from Lower triangle: BAND[Ref[j-i+a-1]+j]

Fetch from Upper triangle: BAND[Ref[j-i+a-1]+i]



Band Matrix

```
void main(void){
   int band[MAX_SIZE];
   int search[MAX_SIZE];

   int i, n, a, b;
   cin>>n; cout<<"n:"<<n;
   cin>>a; cout<<"a:"<<a;
   cin>>b; cout<<"b:"<<b;

   buildbandmatrix(band,search,a,b);

   for(i=0; i<=n*(a+b-1)-a*(a-1)/2-b*(b-1)/2-1; i++)
      cout<<band[i]<<" ";

   cout<<endl;
   for(i=0; i<=a+b-2; i++)
      cout<<search[i]<<" ";

   i=getbandmatrix(3,3,n,a,b,search);

   if(i==2)
      cout<<"\n invalid index";
   else if(i==1)
      cout<<"\n item to be searched: 0";
   else
      cout<<"\n item to be searched: "<<i<<"->"<< band[i];
```

Band Matrix

```
void buildbandmatrix(int band[], int search[], int n, int a, int b){

    int i, k, itemnum;

    if(n*(a+b-1)-a*(a-1)/2-b*(b-1)/2 > MAX_SIZE){

        cout<<“\n not enough memory“;
        exit(-1);
    }
    else{
        itemnum=0;
        for(i=-a+1; i<=b-1; i++){ //for each diagonal
            search[i+a-1]=itemnum;

            for(k=0; k<= n-abs(i)-1; k++)   //for the current diagonal
                cin>>band[search[i+a-1]+k];
            itemnum = itemnum+(n-abs(i));
        }
    }
}
```

Band Matrix

```
void getbandmatrix(int i, int j, int n, int a, int b, int search[]){

    if(i>=n || i<0 || j>=n || j<0){  //index overflow
        cout<<"\n invalid index\n";
        return -2;
    }
    else{
        if(j>i)   //above the diagonal
            if(j-i<b)  //above the upper band
                return(search[a-1+j-i]+i);  //yes
            else  //no
                return -1;
        else if(i-j<a) //below or on the diagonal
                return(search[j-i+a-1]+j);
        else //not on the band
                return -1;
}
```

Sparse Matrix

- Most of the elements are zero.
- It wastes space.

Sparsity: the fraction of zero elements.

Basic matrix operations:

1. Creation
2. Addition
3. Multiplication
4. Transpose

A	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

Sparse Matrix
Data Structure

```
#define MAX_TERMS 101
typedef struct{
            int col;
            int row;
            int value;
           }term;
term a[MAX_TERMS];
```

- a[0].row: row index
- a[0].col: column index
- a[0].value: number of items in the sparse matrix

- Rows and columns are in ascending order!

Sparse Matrix

Bookkeeping the parameters:
# of rows, # of cols, # of elms

A	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

	Row	Column	Value
A[0]	6	6	8
A[1]	0	0	15
A[2]	0	3	22
A[3]	0	5	-15
A[4]	1	1	11
A[5]	1	2	3
...			
A[8]	5	2	28

Matrix Transpose

- Replacement of rows and columns in a matrix is called the transpose of the matrix:

$$A = \begin{bmatrix} 1 & 3 \\ 0 & 4 \end{bmatrix} \qquad A' = \begin{bmatrix} 1 & 0 \\ 3 & 4 \end{bmatrix}$$

- The item a[i][j] becomes a[j][i].

Matrix Transpose

```
void transpose(term a[],term b[])
{
   int n,i,j,currentb;
   n=a[0].value;  //number of items
   b[0].row=a[0].col;   //number of rows
   b[0].col=a[0].row;   //number of columns
   b[0].value=n;

   if(n>0){
      currentb=1;
      for(i=0; i<a[0].col; i++)
          for(j=1; j<=n; j++) //find the ones with col i in a
              if(a[j].col==i){
                     b[currentb].row=a[j].col;
                     b[currentb].col=a[j].row;
                     b[currentb].value=a[j].value;
                     currentb++;
               }
    }
}
```

Question: What is the complexity of this method?

Matrix Transpose

```
void transpose(term a[],term b[])
{
   int n,i,j,currentb;
   n=a[0].value;  //number of items
   b[0].row=a[0].col;   //number of rows
   b[0].col=a[0].row;   //number of columns
   b[0].value=n;

   if(n>0){
      currentb=1;
      for(i=0; i<a[0].col; i++)
          for(j=1; j<=n; j++) //find the ones with col i in a
              if(a[j].col==i){
                     b[currentb].row=a[j].col;
                     b[currentb].col=a[j].row;
                     b[currentb].value=a[j].value;
                     currentb++;
               }
    }
}
```

Question: What is the complexity of this method? O(cols*n)

Question: What is the complexity of this method for a full matrix?

Matrix Transpose

```
void transpose(term a[],term b[])
{
   int n,i,j,currentb;
   n=a[0].value;  //number of items
   b[0].row=a[0].col;   //number of rows
   b[0].col=a[0].row;   //number of columns
   b[0].value=n;

   if(n>0){
      currentb=1;
      for(i=0; i<a[0].col; i++)
          for(j=1; j<=n; j++) //find the ones with col i in a
              if(a[j].col==i){
                     b[currentb].row=a[j].col;
                     b[currentb].col=a[j].row;
                     b[currentb].value=a[j].value;
                     currentb++;
               }
    }
}
```

Question: What is the complexity of this method? O(cols*n)

Question: What is the complexity of this method for a full matrix? O($cols^2$*rows)

Fast Transpose

```
#define MAX_TERM 101
typedef struct{
            int row;
            int col;
            int value;
    } term;
term a[MAX_TERM];


void fastTranspose(term a[], term b[])
{

   int ItemNum[MAX_COL], StartPos[MAX_COL];
   int i,j,ColNum=a[0].col,TermNum=a[0].value;
   b[0].value=TermNum;
   if(TermNum>0){ //does the item exist?
         for(i=0;i<ColNum;i++)
              ItemNum[i]=0;
         for(i=1;i<=TermNum;i++)
              ItemNum[a[i].col]++;
         StartPos[0]=1; // start from index 1
         for(i=1;i<ColNum;i++)
               StartPos[i]=StartPos[i-1]+ItemNum[i-1];
         for(i=1;i<=TermNum;i++){
              j=StartPos[a[i].col]++;
              b[j].row=a[i].col;
              b[j].col=a[i].row;
              b[j].value=a[i].value;
         }
     }
}
```

Fast Transpose

- Execute the fastTranspose method.
- **Question:** What is the complexity of the method?
- Compare its complexity with the previous transpose method.