Multi-Dimensional Arrays

December, 2022

Announcement: Lab Exam

- Tierce 2 Exam
  - Recursion to Pointer (till 21 December)
  - Question pattern will be same as Tierce 1 Exam
  - Partial Marks will be given
  - Those who bunk more than 5 Class will not be able to write exam
    - They need to do extra class
    - Need special permission from **DUGC**

Class on Friday (tomorrow) only for limited students

Announcement: Makeup class

- Friday and Saturday: Time 10:30 - 1PM

- It is mainly for those who have not done well (obtained less than 30%) in the Tierce 1 theory and Lab exam

- Class will be conducted in Lab 309

Why Multidimensional Arrays?

- Marks of 800 students in 5 subjects each.
- Distance between cities
- Sudoku

All the above require 2D arrays

- Properties of points in space (Temperature, Pressure etc.)
- Mathematical Plots

> 2D arrays

Multidimensional Arrays

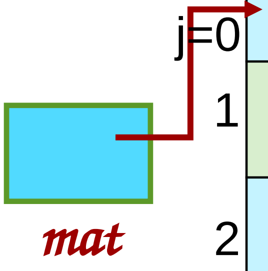
Declaration:

double mat[5][6];

int mat[5][6];

float mat[5][6];

mat is a 5 X 6 matrix of doubles (or ints or floats). It has 5 rows, each row has 6 columns, each entry is of type double.



	i=0	1	2	3	4	5
j=0	2.1	1.0	-0.11	-0.87	31.5	11.4
1	-3.2	-2.5	1.678	4.5	0.001	1.89
2	7.889	3.333	0.667	1.1	1.0	-1.0
3	-4.56	-21.5	1.0e7	-1.0e-9	1.0e-15	-5.78
4	45.7	26.9	-0.001	1000.09	1.0e15	1.0

Accessing Matrix Elements-I

- *(i,j)-th member of mat: mat[i][j] (mathematics: mat(i,j)).*
- *The row and column index start at 0 (not 1).*
- *The following program prints the input matrix.*

```
void print_matrix(double mat[5][6]) {
    int i,j;
    for (i=0; i < 5; i=i+1) {                /* prints the ith row i = 0…4. */
      for (j=0; j < 6; j = j+1) {            /* In each row, prints each of the six
                                                columns  j=0…5 */
              printf(“%f ”, mat[i][j]);
      }
      printf(“\n”);                          /* prints a newline after each row */
    }
}
```

Accessing Matrix Elements-II

- *Code for reading the matrix from the terminal.*
- *The address of the i,j th matrix element is &mat[i][j].*
- *This works without parentheses since the array indexing operator [] has higher precedence than &.*

```
void read_matrix(double mat[5][6]) {
  int i,j;
  for (i=0; i < 5; i=i+1) {
      for (j=0; j < 6; j = j+1) {
          scanf(“%f ”, &mat[i][j]);
      }
    }
}
```

/ read the ith row i = 0..4. */*

/ In each row, read each of the six columns j=0..5 */*

scanf with %f option will skip over whitespace.

So it really doesn’t matter whether the entire input is given in 5 rows of 6 doubles in a row or all 30 doubles in a single line, etc..

Accessing Matrix Elements

```
void read_matrix(double mat[5][6]) {
  int i,j;
  for (i=0; i < 5; i=i+1) {
       for (j=0; j < 6; j = j+1) {
            scanf(“%f ”, &mat[i][j]);
     }
   }
}
```

/* read the ith row i = 0..4. */

/* In each row, read each of the six columns j=0..5 */

Could I change the formal parameter to mat[6][5]? Would it mean the same? Or mat[10][3]?

That would NOT be correct. It would change the way elements of mat are addressed. We will discuss this in details later.

Initializing 2 Dimensional Arrays

We want a[4][3] to be this 4 X 3 int matrix.

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 0 & 1 & 2 \end{matrix}$$

Initialize as

```
int  a[][3] = {
             {1,2,3},
              {4,5,6},
             {7,8,9},
              {0,1,2}
             };
```

Initialization rules:

1. *Most important: values are given row-wise, first row, then second row, so on.*
2. *Number of columns must be specified.*
3. *Values in each row are enclosed in braces {…}.*

Initializing 2 Dimensional Arrays

Initialization rules:

1. *Most important: values are given row-wise, first row, then second row, so on.*
2. *Number of columns must be specified.*
3. *Values in each row are enclosed in braces {…}.*
4. *Number of values in a row may be less than the number of columns specified. Remaining col values set to 0 (or 0.0 for double, '\0' for char, etc.)*

```
int a[][3] = { {1}, {2,3}, {3,4,5} };
```

1	0	0
2	3	0
3	4	5

Passing Two Dimensional Arrays as Parameters

Write a program that takes a two dimensional array of type double[5][6] and prints the sum of entries in each row.

```
void marginals(double mat[5][6]) {
   int i,j; double rowsum;
   for (i=0; i < 5; i=i+1) {
       rowsum = 0.0;
         for (j=0; j < 6; j = j+1) {
                   rowsum = rowsum+mat[i][j]; }
       printf(“%f “, rowsum); }
}
```

Question?

Say, we have read only first 3 rows of mat. We would like to find the marginal sum of the first 3 rows.

Answer:

That’s easy, we can take an additional parameter nrows and run the loop for i=0..(nrows-1) instead of 0..5.

The slightly generalized program would be:

```
void marginals(double mat[5][6], int nrows) {
  int i,j; double rowsum;
  for (i=0; i < nrows; i=i+1) {
     rowsum = 0.0;
       for (j=0; j < 6; j = j+1) {
           rowsum = rowsum+mat[i][j];
     }
    printf("%f ", rowsum);
  }
}
```

In parameter double mat[5][6], C completely ignores the number of rows 5.

It is only interested in the number of cols: 6.

We declared mat to be of type double [5][6].
Does this mean that nrows should be <= 5?
We are not checking for it!

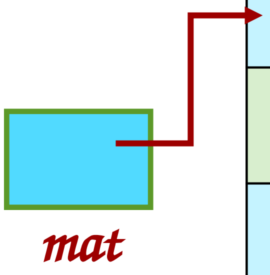
Let's see more examples…

The following program is exactly identical to the previous one.

```
void marginals(double mat[ ][6], int nrows)
 {
   int i,j; int rowsum;
   for (i=0; i < nrows; i=i+1) {
      rowsum = 0.0;
        for (j=0; j < 6; j = j+1) {
            rowsum = rowsum+mat[i][j];
      }
     printf("%f ", rowsum);
   }
}
```

1. Why? because C does not care about the number of rows, only the number of cols.
2. And why is that? We'll have to understand 2-dim array addressing.

This means that the above program works with a k X 6 matrix where k could be passed for nrows.



2.1	1.0	-0.11	-0.87	31.5	11.4
-3.2	-2.5	1.678	4.5	0.001	1.89
7.889	3.333	0.667	1.1	1.0	-1.0
-4.56	-21.5	1.0e7	-1.0e-9	1.0e-15	-5.78
45.7	26.9	-0.001	1000.09	1.0e15	1.0

Why is Number of Columns Required?

- The memory of a computer is a 1D array!
- 2D (or >2D) arrays are "flattened" into 1D to be stored in memory
- In C (and most other languages), arrays are flattened using Row-Major order
  - In case of 2D arrays, knowledge of number of columns is required to figure out where the next row starts.
  - Last n-1 dimensions required for nD arrays

Example calls for marginals

```
void marginals(double mat[ ][6], int nrows);
void main() {
        double mat[9][6];
       /* read the first 8 rows into mat */
    marginals(mat,8);
}
```

✓

```
void marginals(double mat[ ][6], int nrows);
void main() {
      double mat[9][6];
     /* read 9 rows into mat */
    marginals(mat,10);
}
```

UNSAFE

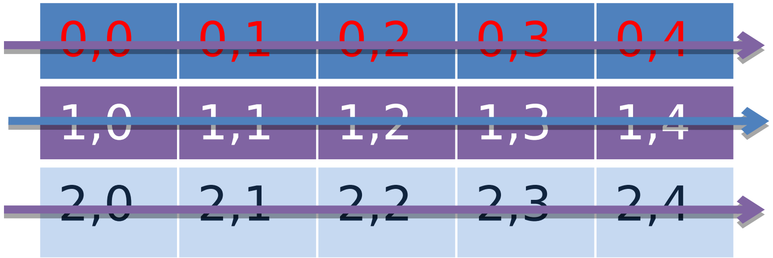
✗

The 10th row of mat[9][6] is not defined. So we may get a segmentation fault when marginals() processes the 10th row, i.e., i becomes 9.

As with 1 dim arrays, allocate your array and stay within the limits allocated.

Row Major Layout

mat[3][5]



0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

Layout of mat[3][5] in memory

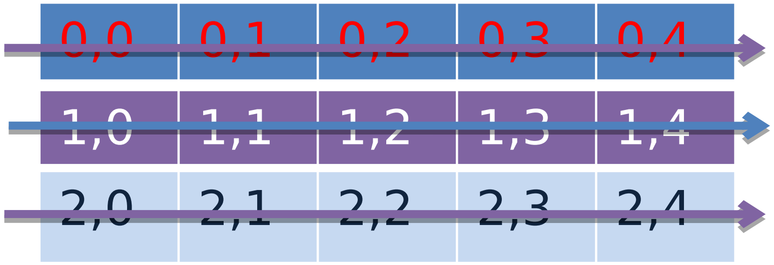
0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4

- *for 2D array* ***mat[M][N]***, *cell [i][j] is stored in memory at location* ***i*N + j*** *from start of mat.*

- *for k-D array arr[N_1][N_2]…[N_k], cell [i_1][i_2]…[i_k] will be stored at location*

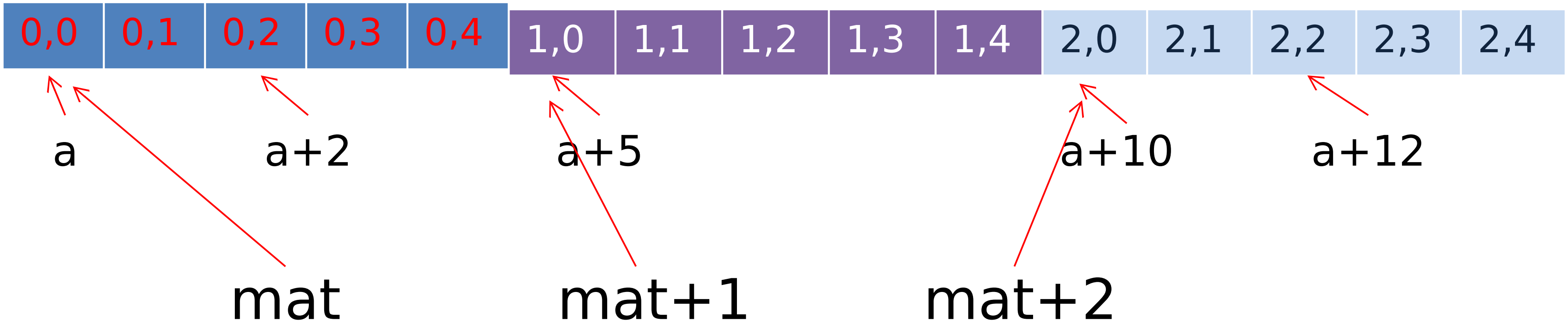
  $i_k + N_k * (i_{k-1} + N_{k-1} * (i_{k-2} + (\dots + N_2 * i_1) \dots))$

mat[3][5]



0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

Layout of mat[3][5] in memory



0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4

- **About C implementation:** a = *mat
- *mat = mat[0], *(mat+1) = mat[1], *(mat+2) = mat[2], ... Each of which stores the address to the start of the corresponding row.
- That is, **mat** POINTS to the beginning of the array.

Array of Strings

- 2D array of char.
- Recall
  - Strings are character arrays that end with a '\0'
  - To display a string we can use printf with the %s placeholder.
  - To input a string we can use scanf with %s. Only reads non-whitespace characters.

Array of Strings: Example

- Write a program that reads and displays the name of few cities of India

```
const int ncity = 4;
const int lencity = 10;

int main(){
  char city[ncity][lencity];
  int i;

  for (i=0; i<ncity; i++){
    -------------------- }

  for (i=0; i<ncity; i++){
    ---------------------}


  return 0;
}
```

Array of Strings: Example

- Write a program that reads and displays the name of few cities of India

```
const int ncity = 4;
const int lencity = 10;

int main(){
  char city[ncity][lencity];
  int i;

  for (i=0; i<ncity; i++){
    scanf("%s", city[i]); }

  for (i=0; i<ncity; i++){
    --------------------- }


  return 0;
}
```

Array of Strings: Example

- Write a program that reads and displays the name of few cities of India

```
const int ncity = 4;
const int lencity = 10;

int main(){
  char city[ncity][lencity];
  int i;

  for (i=0; i<ncity; i++){
    scanf("%s", city[i]); }

  for (i=0; i<ncity; i++){
    printf("%d %s\n", i, city[i]); }

  return 0;
}
```

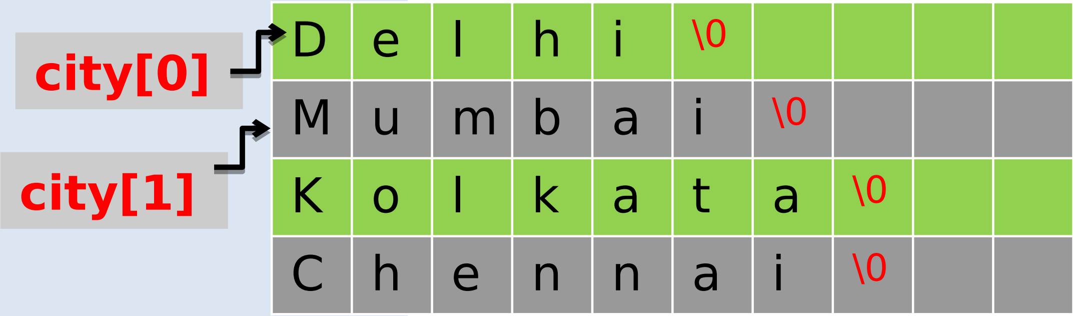
Array of Strings: Example

- Write a program that reads and displays the name of few cities of India

```
const int ncity = 4;
const int lencity = 10;

int main(){
   char city[ncity][lencity];
   int i;

   for (i=0; i<ncity; i++){
      scanf("%s", city[i]);
   }

   for (i=0; i<ncity; i++){
      printf("%d %s\n", i, city[i]);
   }
   return 0;
}
```

INPUT

```
Delhi
Mumbai
Kolkata
Chennai
```



D	e	l	h	i	\0				
M	u	m	b	a	i	\0			
K	o	l	k	a	t	a	\0		
C	h	e	n	n	a	i	\0		

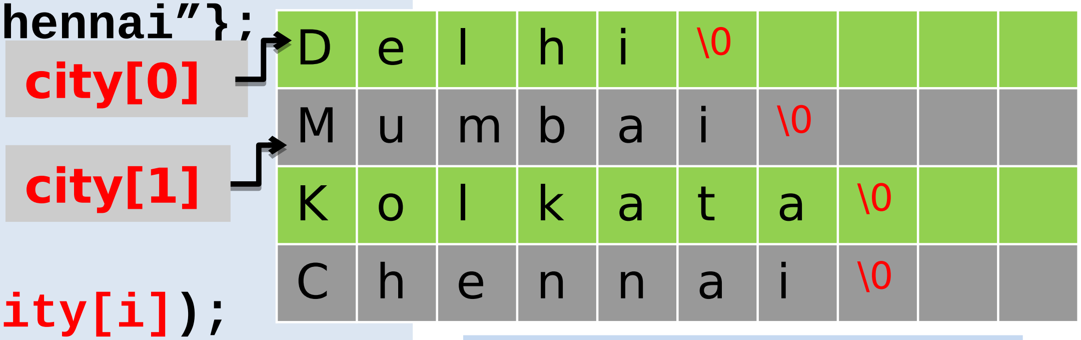
OUTPUT

```
0 Delhi
1 Mumbai
2 Kolkata
3 Chennai
```

Array of Strings: Example

- *List initialization is also allowed:*

```
const int ncity = 4;
const int lencity = 10;

int main(){
   char city[][lencity] = {“Delhi”,
    “Mumbai”, “Kolkata”, “Chennai”};
   int i;


   for (i=0; i<ncity; i++){
      printf("%d %s\n", i, city[i]);
   }
   return 0;
}
```



city[0]

city[1]

D	e	l	h	i	\0				
M	u	m	b	a	i	\0			
K	o	l	k	a	t	a	\0		
C	h	e	n	n	a	i	\0		

OUTPUT

```
0 Delhi
1 Mumbai
2 Kolkata
3 Chennai
```

Search: Array of Strings

- Given an array of strings, find a particular string, say “Delhi”.

```
const int ncity = 4;
const int lencity = 10;

int main() {
   char city[][lencity] = {"Delhi", "Mumbai","Kolkata",
"Chennai"};

   int i,k;

   k = searchstring(city, 4, “Delhi");
      if (k == -1) {printf("City not found");}
      else {printf("The city is at index %d \n", k)}
   return 0;
}
```

Function Searchstring

```
int searchstring(char arrstrings[][lencity], int size, char
key[]) {
    int i;

    for (i=0; i < size; i++) {
        if (strcmp(arrstrings[i], key) == 0) {
            return i;
        }
    }
    return -1;
}
```

Binary Search

- Given a sorted array
- Search in array
- While array is not size 1
  - Split the array in two
  - If key > end of left array
    - Search in right array
  - Else
    - Search in left array

Search

- How many steps needed to search for a *key* in an array of size N?
    - N (simple search)
- What if the array is sorted?
    - Log N (binary search)