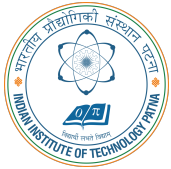
CS1101: Foundations of Programming

Multi-Dimensional Arrays



Dept. of Computer Science & Engineering

Indian Institute of Technology Patna

Two dimensional arrays

- We have seen one dimensional array that can store a list of values
- Many times we need to store values for tables / matrices
- For example, we need to store marks for each student for each assignment

	Assignment 1	Assignment 2	Assignment 3	Assignment 4
Student 1	10	30	63	23
Student 2	15	32	43	34
Student 3	20	22	90	64
Student 4	40	12	88	79
Student 5	12	40	88	79

- C allows us to define such table of items by using two dimensional arrays.
- Each row can viewed as an array.

2-D array declaration

- **General form:** `type array_name[row_size][col_size];`
- **Example:**

```
int marks[5][4];
double val[10][100];
char str[20][100];
```

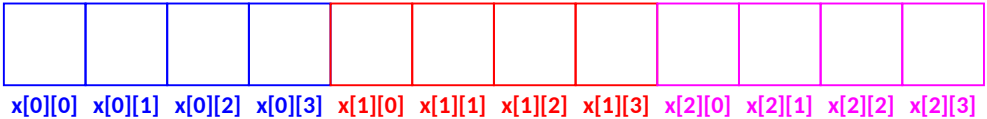
- **First index denotes the row and the second denotes the column**
- **Both row and column index start from 0**
- `marks[i][j]` **indicates the** `(i+1)`**-th row and** `(j+1)`**-th column**

2-D array declaration

- `int x[5][4];`

	col-1	col-2	col-3	col-4
row-1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
row-2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
row-3	x[2][0]	x[2][1]	x[2][2]	x[2][3]
row-4	x[3][0]	x[3][1]	x[3][2]	x[3][3]
row-5	x[4][0]	x[4][1]	x[4][2]	x[4][3]

- **In memory, whole chunk will be contiguous memory**



Accessing elements of a 2-D array

- Similar to 1D array, now we need two indices
- 1st index denotes the row and the second index denotes the column
- Example:

```
x[i][j] = 5;
x[i][j] += a[i][k] * b[k][j];
val = a[i*3][z[m]] * b[k][j];
```

Memory for 2D array

- **Starting from a given memory location, the elements are stored in row-wise in consecutive memory locations**
  - `x`: **starting address of the array in memory**
  - `c`: **number of columns**
  - `k`: **number of bytes allocated per element**

Memory for 2D array

- **Starting from a given memory location, the elements are stored in row-wise in consecutive memory locations**
  - `x`: **starting address of the array in memory**
  - `c`: **number of columns**
  - `k`: **number of bytes allocated per element**
- `x[i][j]` **is allocated memory location at address** `x+(i*c+j)*k`

Array addresses

```
int main(){
   int x[3][5], i, j;
   for(i=0; i<3; i++){
     for(j=0; j<5; j++)
       printf("%p\n",&x[i][j]);
     printf("\n");
   }
   return 0;
}
```

```
0x7ffcf5268ba0
0x7ffcf5268ba4
0x7ffcf5268ba8
0x7ffcf5268bac
0x7ffcf5268bb0

0x7ffcf5268bb4
0x7ffcf5268bb8
0x7ffcf5268bbc
0x7ffcf5268bc0
0x7ffcf5268bc4

0x7ffcf5268bc8
0x7ffcf5268bcc
0x7ffcf5268bd0
0x7ffcf5268bd4
0x7ffcf5268bd8
```

Reading elements of 2D array

- **Need to read one element at a time**

```
for(i=0; i<row; i++)
  for(j=0; j<col; j++)
    scanf("%d",&x[i][j]);
```

- `x[i][j]` **can be thought of as a single integer variable, hence** `&` **is necessary**
- **2D array can be initialized at the time of declaration:**

```
int x[MROWS][MCOLS] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

Printing the elements of a 2D array

- Need to print one element at a time
- One way to print: All elements in one line

```
for(i=0; i<row; i++)
  for(j=0; j<col; j++)
    printf("%d ",x[i][j]);
```

- Another way to print: One row in each line

```
for(i=0; i<row; i++){
  for(j=0; j<col; j++)
    printf("%d ",x[i][j]);
  printf("\n");
}
```

Matrix addition

```
int main(){
   int x[10][10], y[10][10], z[10][10];
   int i, j, k, n, m;
   printf("Enter the dimension: ");
   scanf("%d%d",&n,&m);
   // Read x
   for(i=0;i<n;i++)
     for(j=0;j<m;j++)
       scanf("%d",&x[i][j]);
   // Read y
   for(i=0;i<n;i++)
     for(j=0;i<m;j++)
       scanf("%d",&y[i][j]);
```

```
   // z = x + y
   for(i=0;i<n;i++)
     for(j=0;i<m;j++)
       z[i][j]=x[i][j]+y[i][j];
   // print output
   for(i=0;i<n;i++){
     for(j=0;i<m;j++)
       printf("%d ",z[i][j]);
     printf("\n");
   }
   return 0;
}
```

2D array

```
int main(){
   int x[3][4]={{1,2,3,4}, {5,6,7,8}, {9,10,11,12}}, i,j;
   for(i=0;i<3;i++){
     for(j=0;i<4;j++) printf("%p",&x[i][j]);
     printf("\n");
   }
   printf("sizeof(x)  = %3lu,  x = %p,  x+1 = %p\n", sizeof(x),   x,  x+1);
   printf("sizeof(*x) = %3lu, *x = %p, *x+1 = %p\n", sizeof(*x), *x, *x+1);
   printf("sizeof(&x) = %3lu, &x = %p, &x+1 = %p\n", sizeof(&x), &x, &x+1);
   return 0;
}
```

2D array

```
int main(){
   int x[3][4]={{1,2,3,4}, {5,6,7,8}, {9,10,11,12}}, i,j;
   for(i=0;i<3;i++){
     for(j=0;i<4;j++) printf("%p",&x[i][j]);
     printf("\n");
   }
   printf("sizeof(x)  = %3lu,  x = %p,  x+1 = %p\n", sizeof(x),   x,  x+1);
   printf("sizeof(*x) = %3lu, *x = %p, *x+1 = %p\n", sizeof(*x), *x, *x+1);
   printf("sizeof(&x) = %3lu, &x = %p, &x+1 = %p\n", sizeof(&x), &x, &x+1);
   return 0;
}
```

```
0x7fff4e7502e0 0x7fff4e7502e4 0x7fff4e7502e8 0x7fff4e7502ec
0x7fff4e7502f0 0x7fff4e7502f4 0x7fff4e7502f8 0x7fff4e7502fc
0x7fff4e750300 0x7fff4e750304 0x7fff4e750308 0x7fff4e75030c
sizeof(x)  = 48,  x = 0x7fff4e7502e0,  x+1 = 0x7fff4e7502f0
sizeof(*x) = 16, *x = 0x7fff4e7502e0, *x+1 = 0x7fff4e7502e4
sizeof(&x) =  8, &x = 0x7fff4e7502e0, &x+1 = 0x7fff4e750310
```

Passing 2D arrays to functions

- Similar to 1D array. The array contents are not copied into the function, rather the address of the first element is passed
- For calculating the address of an element in a 2D array, the function needs the following:
  - The starting address of the array (say `x`)
  - Number of bytes per element (say `k`)
  - Number of of columns in the array (size of each row) (say `c`)
  - `x[i][j]` is located at address `x+(i*c+j)*k`

Example: passing 2D array

```
int main(){
   int x[10][15], y[10][15];
   ⋮
   add(x, y, 10, 15);
   ⋮
   return 0;
}
```

```
void add(int a[][15], int b[10][15], int r, int c){
   ⋮
   ⋮
}
```

Example: passing 2D array

```
int main(){
   int x[10][15], y[10][15];
   ⋮
   add(x, y, 10, 15);
   ⋮
   return 0;
}
```

```
void add(int a[][15], int b[10][15], int r, int c){
   ⋮
   ⋮
}
```

The first dimension is ignored. The second dimension must be given.

Matrix addition with functions

```
void readMatrix(int x[][20], int r, int c){
   int i, j;
   for(i=0;i<r;i++)
     for(j=0;j<c;j++)
       scanf("%d",&x[i][j]);
}
void addMatrix(int x[][20], int y[][20], int z[][20], int r, int c){
   int i, j;
   for(i=0;i<r;i++)
     for(j=0;j<c;j++)
       z[i][j] = x[i][j] + y[i][j];
}
```

Matrix addition with functions

```
void printMatrix(int x[][20], int r, int c){
   int i, j;
   for(i=0;i<r;i++){
     for(j=0;j<c;j++)
       printf("%4d ",x[i][j]);
     printf("\n");
   }
}
```

```
int main(){
   int x[10][20], y[10][20];
   int z[10][20], r, c;
   scanf("%d%d",&r,&c);
   readMatrix(x, r, c);
   readMatrix(y, r, c);
   addMatrix(x, y, z, r, c);
   printMatrix(z, r, c);
   return 0;
}
```

The number of columns has to be fixed in function definition.

There is no difference between — `void add(int x[][20],…)` **or** `void add(int x[10][20],…)`

Specifying the first dimension is not necessary, but not a mistake

Matrix transpose

```
void transpose(int x[][3], int n){
   int i, j, t;
   for(i=0; i<n; i++)
     for(j=0; j<n; j++){
       t = x[i][j];
       x[i][j] = x[j][i];
       x[j][i] = t;
     }
}
```

```
int main(){
   int x[3][3], i, j;
   for(i=0;i<3;i++)
     for(j=0;j<3;j++)
       scanf("%d",&x[i][j]);
   transpose(x, 3);
   for(i=0;i<3;i++)
     for(j=0;j<3;j++)
       printf("%d ",x[i][j]);
   return 0;
}
```

Matrix transpose

```
void transpose(int x[][3], int n){
   int i, j, t;
   for(i=0; i<n; i++)
     for(j=0; j<n; j++){
       t = x[i][j];
       x[i][j] = x[j][i];
       x[j][i] = t;
     }
}
```

```
int main(){
   int x[3][3], i, j;
   for(i=0;i<3;i++)
     for(j=0;j<3;j++)
       scanf("%d",&x[i][j]);
   transpose(x, 3);
   for(i=0;i<3;i++)
     for(j=0;j<3;j++)
       printf("%d ",x[i][j]);
   return 0;
}
```

This is wrong!!!

Matrix transpose: correct version

```
void transpose(int x[][3], int n){
   int i, j, t;
   for(i=0; i<n; i++)
     for(j=i; j<n; j++){
       t = x[i][j];
       x[i][j] = x[j][i];
       x[j][i] = t;
     }
}
```

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

Dynamic allocation of 2D array

- We have seen dynamic memory allocation for 1D array using `malloc` library function

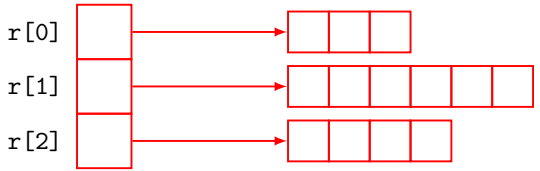
```
int *ptr = (int *)malloc(10 * sizeof(int));
```

- There are many variations
  - Fixed number of rows but variable number of columns
  - Variable number of rows but fixed number of columns
  - Both number of rows and columns variable
- We will consider the first one — fixed number of rows but variable number of columns

Dynamic allocation: Fixed rows, variable columns

- Let us assume that the number of rows are 5
- We can use array of pointers of size 5, where the i-th element if this array a pointer will point to the i-th row of the 2D array

```
int *r[5], i, c;
for(i=0; i<5; i++){
  scanf("%d",&c);
  r[i] = (int *)malloc(c * sizeof(int));
}
```



Statically allocated pointer array

Dynamically allocated memory

Pointer equivalent to 2D array

- **Consider a statically allocated 1D array:** `int x[10];`
- **A pointer that can browse through** `x` **is declared as** `int *p;`
- **Such pointer can be allocated** `p = (int*)malloc(10*sizeof(int));` **and deallocated** `free(p);`
- **What are the analogous pointer for 2D arrays? How can these pointer be allocated and deallocated?**

Pointer equivalent to 2D array

- **Two of the ways to define 2D arrays:** `int x[3][4];` **or** `int *y[3];`
- **Both these arrays are statically allocated**
  - `x` **is an array of arrays and has no dynamic component**
  - `y` **is an array of pointers. Individual pointer** `y[]` **can be allocated dynamically**
- **Statically allocated arrays have following limitations**
  - **Waste of space**
  - **Unable to handle larger arrays than the allocated space**
- **Dynamic version** `x` **and** `y` **can remove these shortcomings**

Dynamic version of 2D array - 1

- **Consider** `int x[3][4];`
- **A pointer matching** `x` **should be pointer to an array with 4** `int` **variables**
- `int *p[4];` **— declares an array of 4 pointers, not a pointer to array**
- **Possible ways for defining correct pointer equivalent to** `x` **are**
  - **Method-1:** `int (*p)[4];`
  - **Method-2:** `typedef int row[4]; row *p;`
  - `p` **is a single pointer**

Dynamic version of 2D array - 2

- Consider `int *y[4];` — `y` is an array of 4 `int` pointer
- The equivalent pointer is a pointer to an `int` pointer — `int **q;`
- A 2D array declared by `q` is fully dynamic
  - The number of rows can be decided during the run of the program
  - The size of each row can also be decided individually during the run
- Note: It will be illegal to set `q = x` or `p = y`. May lead to segmentation fault.
  (`int x[3][4]; int (*p)[4];`)

Dynamic memory for p

- p is single pointer and can be allocated in a single shot (`int (*p)[4];`)
- **Method-1:** `p = (int (*)[4])malloc(3 * 4 * sizeof(int));`
- **Method-2:** `p = (row *)malloc(3 * sizeof(row));`
- **Freeing requires only one call** `free(p);`

Dynamic memory for q

- First we need to allocate required number of row headers then the rows individually

```
q = (int **)malloc(3 * sizeof(int *));
for(i=0; i<3; i++)
  q[i] = (int *)malloc(4 * sizeof(int));
```

- Freeing is also multi-step process (individual row needs to be freed first)

```
for(i=0; i<3; i++) free(q[i]);
free(q);
```

Example: 2D Matrix allocation

```
int **allocate (int h, int w){
   int **p, i, j;
   p = (int **)malloc(h*sizeof (int *));
   for (i=0; i<h; i++)
     printf("%10d", &p[i]);
   printf("\n");
   for (i=0;i<h;i++)
     p[i] = (int *)malloc(w*sizeof(int));
   return(p);
}
```

```
int main(){
   int **p, M, N, i, j;
   printf ("Give M and N \n");
   scanf ("%d%d", &M, &N);
   p = allocate (M, N);
   for (i=0;i<M;i++){
     for (j=0;j<N;j++)
       printf ("%10d", &p[i][j]);
     printf("\n");
   }
   return 0;
}
```

Four types of 2D arrays

Declaration	Number of rows	Number of columns
`int a[5][10];`	static	static
`int (*b)[10];`	dynamic	static
`int *c[5];`	static	dynamic
`int **c;`	dynamic	dynamic

Practice problems

- **Write a function that takes $n \times n$ square matrix as input ($n < 100$) and returns 1 if the matrix is a lower triangular one.**
- **Repeat above for upper triangular matrix, diagonal matrix, identity matrix, anti-symmetric matrix**
- **Consider a $n \times n$ matrix filled with 0 or 1. Write a function that takes such a matrix as input and returns 1 if the number of 1's in each row and each column are the same. It returns 0 otherwise.**
- **Write a function that takes two matrices as input and compute the product of two matrices and store it in a third matrix. The size of the input matrices are $n \times m$ and $m \times p$**
- **write a function that transpose a non-square matrix A in a matrix B**