

MultiDimensional Arrays



TwoDimensional Arrays

We have seen that a oneDimensional array can store a list of values.

Many applications require us to store a table of values – implemented as a 2D array.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

TwoDimensional Arrays

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

The table contains 4 rows with 5 elements each.

- The table can be regarded as a **matrix / 2D array** consisting of 4 **rows** and 5 **columns**.

Declaring 2D Arrays

General form:

```
type array_name[row_size][column_size];
```

Examples:

```
int marks[4][5];
```

```
float sales[12][25];
```

```
double matrix[100][100];
```

First index indicates row, second index indicates column.

Both row index and column index start from 0 (similar to what we had for 1D arrays)

Declaring 2D Arrays

```
int m[4][5];
```

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]
Row 1	m[1][0]	m[1][1]	m[1][2]	m[1][3]	m[1][4]
Row 2	m[2][0]	m[2][1]	m[2][2]	m[2][3]	m[2][4]
Row 3	m[3][0]	m[3][1]	m[3][2]	m[3][3]	m[3][4]

Accessing Elements of a 2D Array

Similar to that for 1D array, but use two indices.

- First index indicates row, second index indicates column.
- Both the indices should be expressions which evaluate to integer values.

Examples:

```
x[m][n] = 0;  
c[i][k] += a[i][j] * b[j][k];  
val = sqrt( arr[j*3][k+1] );
```

How is a 2D array stored in memory?

Starting from a given memory location (starting address of the array), the elements are stored **row-wise** in consecutive memory locations.

- **x**: starting address of the array in memory
- **c**: number of columns
- **k**: number of bytes allocated per array element, e.g., sizeof(int)
- **a[i][j]** is allocated memory location at address $x + (i * c + j) * k$

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

Row 0

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

Row 1

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

Row 2

Array Addresses

```
int main()
{
    int a[3][5];
    int i, j;

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

Output

```
3221224480
3221224484
3221224488
3221224492
3221224496

3221224500
3221224504
3221224508
3221224512
3221224516

3221224520
3221224524
3221224528
3221224532
3221224536
```


How to read the elements of a 2D array?

By reading them one element at a time

```
for (i=0; i<nrow; i++)  
    for (j=0; j<ncol; j++)  
        scanf("%f", &a[i][j]);
```

- The ampersand (&) is necessary.
- The elements can be entered all in one line or in different lines.

We can also initialize a 2D array at the time of declaration:

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

How to print the elements of a 2D array?

By printing them one element at a time.

```
for (i=0; i<nrow; i++)  
    for (j=0; j<ncol; j++)  
        printf ("%f  ", a[i][j]);
```

This will print all
elements in one line.

```
for (i=0; i<nrow; i++) {  
    for (j=0; j<ncol; j++)  
        printf ("%f  ", a[i][j]);  
    printf ("\n");  
}
```

This will print the
elements with one row in
each line (matrix form).

Example: Matrix addition

```
int main()
{
    int  a[100][100], b[100][100],
          c[100][100], p, q, m, n;

    printf ("Enter dimensions: ");
    scanf ("%d %d", &m, &n);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &a[p][q]);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &b[p][q]);
```

```
    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            c[p][q] = a[p][q] + b[p][q];

    for (p=0; p<m; p++)
    {
        for (q=0; q<n; q++)
            printf ("%d ", c[p][q]);
        printf ("\n");
    }
    return 0;
}
```

A 2D array means a 1D array of 1D arrays, and so a row pointer

```
#include <stdio.h>
int main ()
{
    int i, j, A[4][5] = { { 7, 14, 3, 16, 6}, {11, 5, 9, 13, 18},
                          { 2, 15, 20, 1, 19}, {10, 4, 12, 17, 8} };

    for (i=0; i<4; ++i) {
        for (j=0; j<5; ++j) printf("%p ", &A[i][j]);
        printf("\n");
    }

    printf("sizeof(A)   = %3lu,    A = %p,    A + 1 = %p\n", sizeof(A),    A,    A + 1);
    printf("sizeof(*A)  = %3lu,   *A = %p,   *A + 1 = %p\n", sizeof(*A),   *A,   *A + 1);
    printf("sizeof(&A)   = %3lu,   &A = %p,   &A + 1 = %p\n", sizeof(&A),   &A,   &A + 1);
    return 0;
}
```

Output

```
0x7ffc314fe100 0x7ffc314fe104 0x7ffc314fe108 0x7ffc314fe10c 0x7ffc314fe110
0x7ffc314fe114 0x7ffc314fe118 0x7ffc314fe11c 0x7ffc314fe120 0x7ffc314fe124
0x7ffc314fe128 0x7ffc314fe12c 0x7ffc314fe130 0x7ffc314fe134 0x7ffc314fe138
0x7ffc314fe13c 0x7ffc314fe140 0x7ffc314fe144 0x7ffc314fe148 0x7ffc314fe14c
sizeof(A)   = 80,    A = 0x7ffc314fe100,    A + 1 = 0x7ffc314fe114
sizeof(*A)  = 20,   *A = 0x7ffc314fe100,   *A + 1 = 0x7ffc314fe104
sizeof(&A)   = 8,   &A = 0x7ffc314fe100,   &A + 1 = 0x7ffc314fe150
```

Passing 2D arrays to functions



Passing 2D arrays to functions

Similar to that for 1D arrays.

- 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 starting address of the array in memory (say, **x**)
- Number of bytes per element (say, **k**)
- Number of columns in the array, i.e., the size of each row (say, **c**)

$a[i][j]$ is located at memory
address $x + (i * c + j) * k$


The above three pieces of information must be known to the function.

Example

```
int main()
{
    int  a[15][25],  b[15][25];
    ...
    ...
    add (a, b, 15, 25);
    ...
    ...
}
```

```
void  add (int x[][25], int y[][25],
           int rows, int cols)
{
}

```



We can also write

```
int x[15][25], y[15][25];
```

The first dimension is ignored. But the second dimension *must* be given.

Example: Matrix addition with functions

```
void ReadMatrix (int A[][100], int x, int y)
{
    int i, j;
    for (i=0; i<x; i++)
        for (j=0; j<y; j++)
            scanf ("%d", &A[i][j]);
}
```

```
void AddMatrix (int A[][100], int B[][100], int C[][100], int x, int y)
{
    int i, j;
    for (i=0; i<x; i++)
        for (j=0; j<y; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```


Example: Matrix addition

```
void PrintMatrix (int A[][100], int x, int y)
{
    int i, j;
    printf ("\n");
    for (i=0; i<x; i++)
    {
        for (j=0; j<y; j++)
            printf (" %5d", A[i][j]);
        printf ("\n");
    }
}
```

```
int main()
{
    int  a[100][100], b[100][100],
        c[100][100], p, q, m, n;

    scanf ("%d%d", &m, &n);

    ReadMatrix(a, m, n);
    ReadMatrix(b, m, n);

    AddMatrix(a, b, c, m, n);

    PrintMatrix(c, m, n);
    return 0;
}
```

Example:

```
#include <stdio.h>
int main() {
    int a[15][25], b[15][25], c[15][25];
    int m, n;
    scanf ("%d %d", &m, &n);
    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &a[p][q]);
    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &b[p][q]);
    add (a, b, m, n, c);
    for (p=0; p<m; p++) {
        for (q=0; q<n; q++)
            printf ("%f ", c[p][q]);
        printf ("\n");
    }
}
```

```
void add (int x[][25], int y[][25], int m,
          int n, int z[][25])
{
    int p, q;
    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            z[p][q] = x[p][q] + y[p][q];
}
```

Note that the number of columns has to be fixed in the function definition.

- There is no difference between `void add(int x[][25], ...)` and `void add(int x[15][25], ...)`
- Specifying the first dimension is not necessary, but not a mistake.

Example: Transpose of a matrix

```
#include <stdio.h>

void transpose (int x[][3], int n)
{
    int p, q, t;

    for (p=0; p<n; p++)
        for (q=0; q<n; q++)
        {
            t = x[p][q];
            x[p][q] = x[q][p];
            x[q][p] = t;
        }
}
```

```
main()
{
    int a[3][3], p, q;

    for (p=0; p<3; p++)
        for (q=0; q<3; q++)
            scanf ("%d", &a[p][q]);

    transpose (a, 3);

    for (p=0; p<3; p++)
    {
        for (q=0; q<3; q++)
            printf ("%d  ", a[p][q]);
        printf ("\n");
    }
}
```

Example: Transpose of a matrix

```
#include <stdio.h>

void transpose (int x[][3], int n)
{
    int p, q, t;

    for (p=0; p<n; p++)
        for (q=0; q<n; q++)
        {
            t = x[p][q];
            x[p][q] = x[q][p];
            x[q][p] = t;
        }
}
```

This transpose function is wrong.
Why?

```
main()
{
    int a[3][3], p, q;

    for (p=0; p<3; p++)
        for (q=0; q<3; q++)
            scanf ("%d", &a[p][q]);

    transpose (a, 3);

    for (p=0; p<3; p++)
    {
        for (q=0; q<3; q++)
            printf ("%d  ", a[p][q]);
        printf ("\n");
    }
}
```

The Correct Version

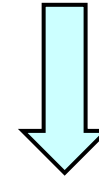
```
void transpose (int x[][3], int n)
{
    int p, q, t;

    for (p = 0; p < n; p++)
        for (q = p; q < n; q++)
        {
            t = x[p][q];
            x[p][q] = x[q][p];
            x[q][p] = t;
        }
}
```

10 20 30

40 50 60

70 80 90



10 40 70

20 50 80

30 60 90

You may start the inner loop with $q = p + 1$

Note

- We have done static allocation of 1D / 2D arrays. Dynamic allocation will possibly be covered later.
- We have studied 1D and 2D arrays only. C allows arrays of higher dimensions as well.
For example, `int a[10][15][20]` is a 3D array (not in syllabus).



Practice problems

1. Write a function that takes an $n \times n$ square matrix A as parameter ($n < 100$) and returns 1 if A is an upper-triangular matrix, 0 otherwise.
2. Repeat 1 to check for lower-triangular matrix, diagonal matrix, identity matrix.
3. Consider an $n \times n$ binary matrix (contains only 0 or 1). Write a function that takes such a matrix and returns 1 if the number of 1's in each row are the same and the number of 1's in each column are the same; it returns 0 otherwise.
4. Write a function that reads in an $m \times n$ matrix A and an $n \times p$ matrix B , and returns the product of A and B in another matrix C . Pass appropriate parameters.
5. Write a function to find the transpose of a non-square matrix A in a matrix B .
6. Repeat the last exercise when the transpose of A is computed in A itself. Use no additional 2D arrays.

For each of the above, also write a main function that reads the matrices, calls the function, and prints the results (a message, the result matrix, etc.)

ADVANCED TOPICS

Pointers equivalent to twoDimensional arrays

Generalization from oneDimensional arrays

Consider the statically allocated 1D array:

```
int A[20];
```

A pointer that can browse through A is declared as:

```
int *p;
```

Such a pointer can be allocated dynamic memory and freed as:

```
p = (int *)malloc(20 * sizeof(int));  
free(p);
```

- What are the analogous pointers for 2D arrays that you have seen earlier?
- How can these pointers be allocated and deallocated their own memory?

What are our 2D arrays?

We have seen two types of 2D arrays:

```
int A[10][20];  
int *B[10];
```

Both these arrays are statically allocated.

- A is an array of arrays, and has no dynamic component.
- B is an array of pointers. Individual pointers in B[] can be dynamically allocated.

As statically allocated arrays, both A and B suffer from the two standard disadvantages:

- Waste of space
- Inadequacy to handle larger than the allocated space

Dynamic versions of A and B overcome these shortcomings.

Dynamic version of A

```
int A[10][20];
```

A pointer matching A should be a pointer to an array of 20 int variables.

But

```
int *p[20];
```

declares an array of 20 int pointers, not a pointer to an array.

Three ways of defining the correct pointer equivalent to A:

Method 1:

```
int (*p)[20];
```

Method 2:

```
typedef int row[20];  
row *p;
```

Method 3:

```
typeof(int [20]) *p;
```

 // Not available in the original C specification

In all the cases, p is a *single* pointer.

Dynamic version of B

```
int B[10];
```

B is an array of 10 int pointers.

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 is *illegal* to set `q = A;` or `p = B;` Expect segmentation fault if you do so (ignoring the warnings issued by the compiler).

Dynamic memory for p

p is a single pointer, and can be allocated and deallocated memory in a single shot.

- Method 1:

```
p = (int (*) [20]) malloc(10 * 20 * sizeof(int));
```

- Method 2:

```
p = (row *) malloc(10 * sizeof(row));
```

- Method 3:

```
p = (typeof(int [20]) *) malloc(10 * 20 * sizeof(typeof(int [20])));
```

Freeing requires only one call.

```
free(p);
```

Dynamic memory for q

First, you allocate the required number of row headers, and then the rows individually.

```
q = (int **)malloc(10 * sizeof(int *));  
for (i=0; i<10; ++i)  
    q[i] = (int *)malloc(20 * sizeof(int));
```

Freeing is also a multi-step process.

```
for (i=0; i<10; ++i) free(q[i]);  
free(q);
```

Note: Free the individual rows *before* freeing the array of row headers.

Example: Vandermonde matrices

A Vandermonde matrix corresponding to n real-valued elements a_0, a_1, \dots, a_{n-1} is defined as:

1	1	1	...	1
a_0	a_1	a_2	...	a_{n-1}
a_0^2	a_1^2	a_2^2	...	a_{n-1}^2
\vdots	\vdots	\vdots	...	\vdots
a_0^{n-1}	a_1^{n-1}	a_2^{n-1}	...	a_{n-1}^{n-1}

An application works with Vandermonde matrices for $n \leq 100$. A static 2D array would require a total storage of $100 \times 100 = 10,000$ cells. This leads to waste if n is small.

We write a function `genvdm(A,n)` that obtains a_0, a_1, \dots, a_{n-1} from the 1D array A , and returns a pointer to a dynamically allocated array of rows.

The row size must be fixed beforehand. But we can allocate exactly n rows to reduce wastage.

Dynamic memory for storing Vandermonde matrices

```
#include <stdio.h>
#include <stdlib.h>

#define MAXDIM 100

double (*genvdm ( double *A, int n )) [MAXDIM]
{
    double (*p) [MAXDIM] ;
    int i, j;

    p = (double (*) [MAXDIM]) malloc (n * MAXDIM * sizeof(double));
    for (i=0; i<n; ++i) {
        // i is an index in A, and a column in p. j is a row in p.
        p[0][i] = 1;
        for (j=1; j<n; ++j) p[j][i] = p[j-1][i] * A[i];
    }
    return p;
}

void prnvdm ( double M[][MAXDIM], int n )
{
    int i, j;

    for (i=0; i<n; ++i) {
        for (j=0; j<n; ++j) printf("%10.5lf ", M[i][j]);
        printf("\n");
    }
}
```


Storage of Vandermonde matrices (continued)

```
int main ()
{
    double A[MAXDIM], (*V) [MAXDIM];
    int n, i;

    printf("Enter dimension of V: "); scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (i=0; i<n; ++i) scanf("%lf", A+i);
    V = genvdm(A,n);
    prnvdm(V,n);
    exit(0);
}
```

Exercise: Free the 2D
memory allocated to V.

Output

```
Enter dimension of V: 5
Enter 5 elements: -1 0.1 1.1 2.5 3.2
  1.00000    1.00000    1.00000    1.00000    1.00000
 -1.00000    0.10000    1.10000    2.50000    3.20000
  1.00000    0.01000    1.21000    6.25000   10.24000
 -1.00000    0.00100    1.33100   15.62500   32.76800
  1.00000    0.00010    1.46410   39.06250  104.85760
```

Antisymmetric matrices

A symmetric matrix is an $n \times n$ matrix with $a_{ij} = a_{ji}$ for all i, j .

An antisymmetric matrix is an $n \times n$ matrix with $a_{ij} = -a_{ji}$ for all i, j . Since $a_{ii} = -a_{ii}$, the main diagonal is filled by 0. Moreover, the entries below the main diagonal can be obtained from the entries above the main diagonal.

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

We can use a fully dynamic 2D array to store only the elements above the main diagonal.

Compact storage of an antisymmetric matrix

The function `genasm(n)` returns a pointer to the “array” given `n` as input. Let us take $a_{ij} = i - j$.

```
#include <stdio.h>
#include <stdlib.h>

int **genasm ( int n )
{
    int **q, i, j;

    q = (int **)malloc((n-1) * sizeof(int *));
    for (i=0; i<n-1; ++i) {
        q[i] = (int *)malloc((n-i-1) * sizeof(int));
        for (j=i+1; j<n; ++j) q[i][j-i-1] = i-j;
    }
    return q;
}
```

Storage of antisymmetric matrices (continued)

```
void prnasm ( int *U[], int n )
{
    int i, j;
    for (i=0; i<n; ++i) {
        for (j=0; j<i; ++j) printf("%3d ", -U[j][i-j-1]);
        printf(" 0 ");
        for (j=i+1; j<n; ++j) printf("%3d ", U[i][j-i-1]);
        printf("\n");
    }
}

int main ()
{
    int **U, n;
    printf("Enter dimension (n): "); scanf("%d", &n);
    U = genasm(n);
    prnasm(U,n);
    exit(0);
}
```

Exercise: Free the 2D memory allocated to U.

Output

```
Enter dimension (n): 5
0  -1  -2  -3  -4
1   0  -1  -2  -3
2   1   0  -1  -2
3   2   1   0  -1
4   3   2   1   0
```

Four types of 2D arrays

Declaration	Number of rows	Number of columns
<code>int A[10][20];</code>	Static	Static
<code>int (*p)[20];</code>	Dynamic	Static
<code>int *B[10];</code>	Static	Dynamic
<code>int **q;</code>	Dynamic	Dynamic