# Arrays and Pointers: Multidimensional Arrays

The C language allows multidimensional arrays. We have already discussed one-dimensional arrays and created one-dimensional array-like variables using dynamic memory allocation. To define a one-dimensional array, we need to use one pair of brackets `[]`. Similarly, we need to use two pairs of brackets `[][]` to define a two-dimensional array, and three pairs of brackets `[][][]` to define a three-dimensional array. For example, the following code declares `a` to be a one-dimensional integer array of dimension `5`. Also, it defines `b` to be a two-dimensional integer array of dimensions `3x5`, where `3` is the first dimension (think about the row dimension of a matrix) and `5` is the second dimension (think about the column dimension of a matrix). Finally, `c` is a three-dimensional integer array of first, second and third dimensions `3, 5` and `7` respectively.

```
int a[5];
int b[3][5];
int c[3][5][7];
```

The value in the `i`-th location (note that the first element in the `0`-th position) of the array `a` can be accessed using `a[i]`, where `i` ranges from `0` to 4. A two-dimensional array can be thought of as a matrix where the element `b[i][j]` is the element in the `i`-th row (note that the first row is the zeroth row) and `j`-th column (note that the first column is the zeroth column). Thus, for `b[i][j]`, we have `i` ranges from `0` to `2` and `j` ranges from `0` to `4`. Similarly, `i` ranges from `0` to `2`, `j` ranges from `0` to `4`, and `k` ranges from `0` to `6` for the element `c[i][j][k]` of the three-dimensional array `c`. Let us concentrate on a two-dimensional array declared as

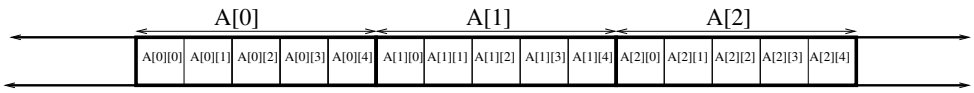


Figure 1: Schematic representation of int A[3][5]

`int A[3][5]`. Although we think of it as a matrix, it is stored row-wise in memory, as shown in the figure. We know that an array name is the address of the first element. In that sense, the array name `A` is the address of the first element `A[0]`, but `A[0]` is a five-element array in which each element can hold an integer. Additionally, since `A[0]` is itself an array name, it is equivalent to the address of the first element `A[0][0]`. Thus, `A=&A[0]` and `A[0]=&A[0][0]`. Observe the output of the following code:

```
int A[3][5];
printf("A=%p      A[0]=%12p\n",A,A[0]);
printf("A+1=%p    A[0]+1=%12p\n",A+1,A[0]+1);
```

```
A=0x16d5571ac       A[0]= 0x16d5571ac
A+1=0x16d5571c0     A[0]+1= 0x16d5571b0
```

As expected, the value of `A`, i.e., `&A[0]` and the value of `A[0]`, i.e., `&A[0][0]` are same which is evident from the figure. However, note that `A+1=&A[0]+1`. Thus, the value of `A+1` is obtained by adding `5*4=20` bytes to the value of `A`. On the other hand, the value `A[0]+1=&A[0][0]+1` is obtained by adding `4` bytes to the value of `A[0]`. These are obvious from the output shown above.

From the above discussion, it is obvious that to assign `10` to `A[2][3]` of the 2D array `int A[3][5]`, we can write any of the following equivalent statements:

```
A[2][3]=10;
*(A[2]+3)=10;
*(*(A+2)+3)=10;
(*(A+2))[3]=10;
*(&A[0][0]+2*5+3)=10;
```

Note that the parentheses in the above expressions are necessary. In the absence of parentheses in the second expression, it becomes `*A[2]+3=10`, which is equivalent to `A[2][0]+3=10`, which results in an error. If we ignore the inner parentheses in the third expression, then it is `*(*A+2+3)=10`≡`*(A[0]+5)=10`≡`A[0][5]=10`≡`A[1][0]=10`. Note that `A[0][5]` ≡ `*(A[0]+5)` ≡ `*(&A[0][0]+5)`. Thus, we skip `5` places after the address of the first element, which is `A[1][0]`. Thus, `10` is assigned to `A[1][0]`. Similarly, we can explain the other expressions. Also, observe the last statement. To access the element `A[2][3]`, we note that it is in the 3-rd row and 4-th column. So two rows precede the 3-rd row, which stores `2*5=10` elements since each row has `5` elements. Additionally, `3` elements are also stored before `A[2][3]`. Thus, in total we have `2*5+3=13` elements before `A[2][3]`. Hence, we start at the address `&A[0][0]` of the first element and skip `13` places to get the address of the element `A[2][3]`. The second dimension (column dimension) is very important. Without it, we don't know how many elements were stored before it. Similarly, `A[1][4]` ≡ `*(&A[0][0]+1*5+4)`. For this reason, when we send the address of a two-dimensional array to a function, we also need to send the second dimension. Without the second dimension, it is impossible to access the elements of the two-dimensional array inside the function. Similar remarks hold for three and higher-dimensional arrays.

**Initialization:** There are a few ways in which a multi-dimensional array can be initialized. The declaration

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

initializes the first row with `3,2,1,-1`, the second row with `5,4,0,0` (two missing entries are initialized with zeros), and the third row with `8,7,6,0` (one missing entry initialized with zero). If we declare

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

then first row is initialized with `3,2,1,-1`, second row with `5,4,8,7` and the third row with `6,0,0,0`. Note that the array `a` has `12` elements, out of which only `9` are provided. They are

filled row-wise, and the remaining ones are initialized to zeros. Hence, the declaration
`int a[3][4]={0};`
initializes all the elements to zero since the given `0` is assigned to `A[0][0]` and the rest are initialized to zero. Also, the declaration
`int a[3][4]={{3,2,1,-1},{5,4,8,7}};`
initializes the first row with `3,2,1,-1`, second row with `5,4,8,7` and the third row (missing row) with `0,0,0,0`. Finally, the declaration
`int a[][4]={{3,2,1,-1},{5,4,8}};`
defines the array `a` to be of dimensions `2x4` with first row `3,2,1,-1` and second row `5,4,8,0`. Further, the declaration
`int a[][4]={3,2,1,-1,5,4,8,6,7};`
defines the array `a` to be of dimensions `3x4` with first row `3,2,1,-1`, second row `5,4,8,6`, and the third row `6,0,0,0`.

**Reading and writing a 2D array:** If the dimensions of the 2D array are large, it is best to read them from a file (which will be covered later). For a small array, we can read and print it row by row. The following code segment implements these:

```
int i,j,A[3][5];
   for(i=0;i<3;i++)
{
 printf("Enter 5 elements for row %d :",i);
 for(j=0;j<5;j++)
 {
  scanf("%d",&A[i][j]);
 }
}



printf("The 2D array is:\n");
for(i=0;i<3;i++)
{
 for(j=0;j<5;j++)
 {
  printf("%8d",A[i][j]);
 }
 printf("\n"); // new line after each row
}
```

Below is a sample input-output:

```
Enter 5 elements for row 0 :5 7 8 2 4
```

```
Enter 5 elements for row 1 :-7 4 3 2 1
Enter 5 elements for row 2 :1 2 3 4 5
The 2D array is:
       5       7       8       2       4
      -7       4       3       2       1
       1       2       3       4       5
```

Below is a program that reads a matrix from the keyboard and calculates its transpose. We use functions to implement these. We use `#define` directive to declare the dimensions. Note that in each function, we are passing `A` or `B`, which are by definition addresses of the first element of arrays. Thus, `A=&A[0]` and $A[0]$ is an array of `N` elements. Thus, we are actually passing an address, as evidenced by the argument of `transmat()`. However, the compiler also allows this to be declared as in `matread()` and `matprint()` for our convenience.

```
#include<stdio.h>
#define M 10
#define N 10
void matread(double [M][N],int *,int *);
void matprint(double [][N],int,int,char);
void transmat(double (*)[N],double (*)[N],int,int);
int main()
{
 double A[M][N],B[M][N]; //B =A^T
 int i,j,m,n;            // m,n actual row and col dims
 matread(A,&m,&n);
 matprint(A,m,n,'A');
 transmat(A,B,m,n);
 matprint(B,n,m,'B');
 return 0;
}
void matread(double A[M][N],int *m,int *n)
{
 int i,j,p,q;
 printf("Enter the row & col dims :");
 scanf("%d%d",&p,&q);
 *m=p;
 *n=q;
 for(i=0;i<p;i++)
 {
  printf("Enter %d elements of row %d:",q,i);
```

```
  for(j=0;j<q;j++)
  {
   scanf("%lf",&A[i][j]);
  }
 }
 return;
}
void matprint(double A[][N],int m,int n,char s)
{
 int i,j;
 printf("The matrix %c is:\n",s);
 for(i=0;i<m;i++)
 {
  for(j=0;j<n;j++)
  {
   printf("%10.2f",A[i][j]);
  }
  printf("\n");
 }
 return;
}
void transmat(double (*A)[N],double (*B)[N],int m,int n)
{
 int i,j;
 for(i=0;i<n;i++)
 {
  for(j=0;j<m;j++)
  {
   B[i][j]=A[j][i];
  }
 }
 return;
}
```

**Listing 56: C Program "transm.c"**

Below is a sample input-output:

```
Enter the row & col dims :3 4
Enter 4 elements of row 0:-1.0 2 4.6 5.0
```

```
Enter 4 elements of row 1:2.20 3.5 -6.5 8
Enter 4 elements of row 2:1.5 2.5 -4.5 3.0
The matrix A is:
     -1.00      2.00      4.60      5.00
      2.20      3.50     -6.50      8.00
      1.50      2.50     -4.50      3.00
The matrix B is:
     -1.00      2.20      1.50
      2.00      3.50      2.50
      4.60     -6.50     -4.50
      5.00      8.00      3.00
```

**Dynamic 2D array like variable:** We know that a 2D array such as `int A[3][4]` is stored row-wise contiguously. Thus, `3*4=12` consecutive places are reserved where each place can hold an integer. Note that in the program given in Listing 56, the array has dimensions `10x10`. Thus, `100` places are reserved but only `3*4=12` is used for `m=3,n=4`. Thus, `88` places are not used.

Note that in `int A[3][4]`, only `12` places are reserved for `A[0][0],A[0][1],`$\cdots$, `A[2][3]`. We think of `A[0]` being the first row, `A[1]` the second row, and A[2] the third row, but as such, there is no storage for `A[0], A[1], A[2]`. Also, `A[0]=&A[0][0]`, i.e., `A[0]` can be thought of as an address of an `int` variable. Unlike `A[0], A[1], A[2]`, we create physical variables of three components where each component can hold the address of an `int` variable. This we do by the following code:

```
int **B;
B=(int **)calloc(3,sizeof(int *));
```

Thus, 3 contiguous spaces have been created in the memory where each space can hold the address of an integer variable. Also, `B` holds the address of the first location. Now `B[0]=*B` is the first space, `B[1]=*(B+1)` is the second space and `B[2]=*(B+2)` is the third space. Now we create space for 4 elements, where each element can hold an integer. The address of its first location is assigned to `B[0]`.

```
B[0]=(int*)calloc(4,sizeof(int));
```

Then, `B[0][0]=*(B[0]+0)`, `B[0][1]=*(B[0]+1)`, `B[0][2]=*(B[0]+2)`, and `B[0][3]=*(B[0]+3)` respectively denote these places where each place can hold an integer. We do similar assignment for `B[1]` and `B[2]`. Consequently, `B` behaves like a 2D array, but it is just an address of an address variable of type `int`. Also, note that these `12` places are not contiguous. In the case of a 2D array, we have `12` contiguous spaces allocated. Here, we have 3 blocks of spaces, each

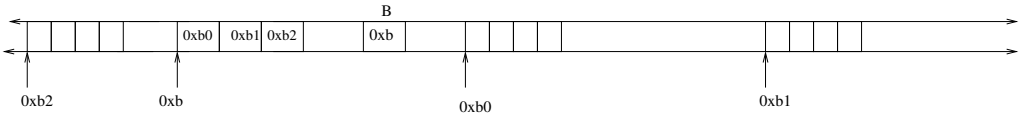


Figure 2: Schematic diagram. The addresses are hypothetical and for illustrative purposes.

with 4 elements that can hold integers. Further, we have a block of 3 spaces that can hold addresses of integers and a space for `B` that holds the address of an address of type `int` (see the figure above).

Below, we implement the same program given in Listing 56 using the ideas developed. We have the same input-output. However, we create the spaces as needed, and hence, there is no unused space. Finally, we free the allocated space. Note that we first free the spaces used for rows, and then the spaces pointed to by `double **` variables.

```
#include<stdio.h>
#include<stdlib.h>
double  **matread(int *,int *);
void matprint(double **,int,int,char);
double **transmat(double **,int,int);
int main()
{
 double **A,**B; //B =A^T
 int i,j,m,n;            // m,n actual row and col dims
 A=matread(&m,&n);
 matprint(A,m,n,'A');
 B=transmat(A,m,n);
 matprint(B,n,m,'B');
 for(i=0;i<m;i++)
 {
  free(A[i]);
 }
 free(A);
 for(i=0;i<n;i++)
 {
  free(B[i]);
 }
 free(B);

 return 0;
}
```

```
double **matread(int *m,int *n)
{
 int i,j,p,q;
 double **C;
 printf("Enter the row & col dims :");
 scanf("%d%d",&p,&q);
 *m=p;
 *n=q;
 C=(double **)calloc(p,sizeof(double *));
 if(C==NULL)
 {
  printf("Problem in calloc for C: stop\n");
  exit(1);
 }
 for(i=0;i<p;i++)
 {
  C[i]=(double *)calloc(q,sizeof(double));
  if(C[i]==NULL)
  {
   printf("Problem in calloc for C[i]: stop\n");
   exit(1);
  }
 }


 for(i=0;i<p;i++)
 {
  printf("Enter %d elements of row %d:",q,i);
  for(j=0;j<q;j++)
  {
   scanf("%lf",&C[i][j]);
  }
 }
 return C;
}
void matprint(double **A,int m,int n,char s)
{
 int i,j;
 printf("The matrix %c is:\n",s);
```

```
 for(i=0;i<m;i++)
 {
  for(j=0;j<n;j++)
  {
   printf("%10.2f",A[i][j]);
  }
  printf("\n");
 }
 return;
}

double **transmat(double **A,int m,int n)
{
 int i,j;
 double **C;
 C=(double **)calloc(n,sizeof(double *));
 if(C==NULL)
 {
  printf("Problem in calloc for C: stop\n");
  exit(1);
 }
 for(i=0;i<n;i++)
 {
  C[i]=(double *)calloc(m,sizeof(double));
  if(C[i]==NULL)
  {
   printf("Problem in calloc for C[i]: stop\n");
   exit(1);
  }
 }

 for(i=0;i<n;i++)
 {
  for(j=0;j<m;j++)
  {
   C[i][j]=A[j][i];
  }
 }
 return C;
```

```
}
```

**Listing 57: C Program "dtransm.c"**

```
Enter the row & col dims :3 4
Enter 4 elements of row 0:-1.0 2 4.6 5.0
Enter 4 elements of row 1:2.20 3.5 -6.5 8
Enter 4 elements of row 2:1.5 2.5 -4.5 3.0
The matrix A is:
     -1.00      2.00      4.60      5.00
      2.20      3.50     -6.50      8.00
      1.50      2.50     -4.50      3.00
The matrix B is:
     -1.00      2.20      1.50
      2.00      3.50      2.50
      4.60     -6.50     -4.50
      5.00      8.00      3.00
```

Return to Lecture Topics