Two-Dimensional Array
Declaration

A 2-D array within a function is declared as follows:

```c
#define ROW 3
#define COL 5

...... what(....){
    int a[ROW][COL] .... ;

    .........................
}
```
**Logical View**

Logically it may be viewed as a two-dimensional collection of data, three rows and five columns, each location is of type `int`.

**Columns**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
<td>a[0][4]</td>
</tr>
<tr>
<td>1</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
<td>a[1][4]</td>
</tr>
</tbody>
</table>
Memory Mapping

The computer memory is an one-dimensional sequence of bytes. C compiler stores the two-dimensional\textsuperscript{a} object in row-major order in the memory\textsuperscript{b}.

\textsuperscript{a}Multi-dimensional in general.

\textsuperscript{b}It is stored in column-major order in some other programming languages e.g. FORTRAN.
Local 2-D Array

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
</tr>
</tbody>
</table>
```

Row-Major memory mapping

Low Address  Main Memory  High Address

Stack grows (Pentium)
I/O

Data can be read in a 2-D array and data can be printed from a 2-D array, one element at time\(^a\). Consider the following \(3 \times 5\) matrix of real numbers. We can read the matrix in a 2-D array and print it in a C program.

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
-1.0 & -2.0 & -3.0 & -4.0 & -5.0 \\
10.0 & 20.0 & 30.0 & 40.0 & 50.0
\end{bmatrix}
\]

\(^a\)A string can be read as a whole.
```c
#include <stdio.h>
#define MAXROW 50
#define MAXCOL 50
int main() // matRdWr.c
{
    double a[MAXROW][MAXCOL];
    int rows, columns, i, j;

    printf("Enter the number of Rows: ");
    scanf("%d", &rows);
    printf("\nEnter the number of Columns: ");
    scanf("%d", &columns);
    printf("\nEnter row-wise, the elements of the matrix\n");
    for (i = 0; i < rows; i++) {
        for (j = 0; j < columns; j++) {
            printf("Enter element a[%d][%d]: ", i, j);
            scanf("%lf", &a[i][j]);
        }
    }
    // Further processing...
}
```
for(i = 0; i < rows; ++i)
    for(j = 0; j < columns; ++j)
        scanf("%lf", &a[i][j])
    putchar(‘\n’);
printf("The matrix is:\n") ;
for(i = 0; i < rows; ++i) {
    for(j = 0; j < columns; ++j)
        printf("%4.2f ", a[i][j])
    putchar(‘\n’) ;
}
return 0;
Data File

It is tedious to enter data manually. So we use a data file `dataMat` and redirect the input from the file.

```
3 5
1.0 2.0 3.0 4.0 5.0
-1.0 -2.0 -3.0 -4.0 -5.0
10.0 20.0 30.0 40.0 50.0
```
Running the Code

$ cc -Wall matRdWr.c
$ a.out < dataMat

Enter the number of Rows:
Enter the number of Columns:
Enter row-wise, the elements of the matrix
The matrix is:
1.00 2.00 3.00 4.00 5.00
-1.00 -2.00 -3.00 -4.00 -5.00
10.00 20.00 30.00 40.00 50.00
#include <stdio.h>
#define MAXROW 5
#define MAXCOL 5
int main() // init2D.c
{
    int a[MAXROW][MAXCOL], i, j,
    b[MAXROW][MAXCOL] = {{0, 1, 2, 3, 4},
                         {10, 20, 30, 40, 50},
                         {15, 25, 35, 45, 55},
                         {50, 51, 52, 53, 54},
                         {55, 55, 55, 55, 55}},

}
c[MAXROW][MAXCOL] = {{10, 20, 30},
        {40, 50, 60, 70, 80}},

d[] [MAXCOL] = {{2, 4, 6, 8, 0},
        {4, 6, 8, 0, 2} } ,

e[MAXROW] [MAXCOL] = {0, 1, 2, 3, 4,
        5, 6, 7, 8, 9,
        10, 11, 12, 13, 14,
        15, 16, 17, 18, 19,
        20, 21, 22, 23, 24
    },

f[] [MAXCOL] = {2, 4, 6, 8, 0,
        4, 6, 8, 0, 2
    } // ,
//
g[MAXROW] [] = {{0, 1, 2, 3, 4},
    {10, 20, 30, 40, 50},
    {15, 25, 35, 45, 55},
    {50, 51, 52, 53, 54},
    {55, 55, 55, 55, 55},
//
    }
;

printf("\n") ;
printf("Array a[][]\n") ;

for(i = 0; i < MAXROW; ++i) {
    for(j = 0; j < MAXCOL; ++j)
        printf("%d ", a[i][j]) ;
    printf("\n") ;
```c
}   
printf("\n") ;
printf("Array b[][]\n") ;
for(i = 0; i < MAXROW; ++i) {
    for(j = 0; j < MAXCOL; ++j)
        printf("%d ", b[i][j]) ;
    printf("\n") ;
}

printf("\n") ;
printf("Array c[][]\n") ;
for(i = 0; i < MAXROW; ++i) {
    for(j = 0; j < MAXCOL; ++j)
        printf("%d ", c[i][j]) ;
```
printf("\n") ;
}

printf("\n") ;
printf("Array d[][]\n") ;
for(i = 0; i < MAXROW; ++i) {
    for(j = 0; j < MAXCOL; ++j)
        printf("%d ", d[i][j]) ;
    printf("\n") ;
}

printf("\n") ;
printf("Array e[][]\n") ;
for(i = 0; i < MAXROW; ++i) {

for(j = 0; j < MAXCOL; ++j)
    printf("%d ", e[i][j]) ;
    printf("\n") ;
}

printf("\n") ;
printf("Array f[][]\n") ;
for(i = 0; i < MAXROW; ++i) {
    for(j = 0; j < MAXCOL; ++j)
        printf("%d ", f[i][j]) ;
    printf("\n") ;
} 
return 0;
int a[10], b[5][3];

We know that ‘a’ is a constant expression whose value is the address of the 0\textsuperscript{th} location of the array a[10]. Similarly a + i is the address of the i\textsuperscript{th} location of the array.

What is ‘b’ and what is its arithmetic?
Consider the following program:

```c
#include <stdio.h>
int main() // 2DArith1.c
{
    int a[10], b[3][5];

    printf("a: %p\t\tb = %p\n", a, b);
    printf("a+1: %p\t\tb+1: %p\n", a+1, b+1);
    printf("a+2: %p\t\tb+2: %p\n", a+2, b+2);
    printf("a+3: %p\t\tb+3: %p\n", a+3, b+3);
    return 0;
}
```
Output

$ cc -Wall 2DArith1.c
$ a.out
a: 0xbfec6a90  b = 0xbfec6a50
a+1: 0xbfec6a94  b+1: 0xbfec6a64
a+2: 0xbfec6a98  b+2: 0xbfec6a78
a+3: 0xbfec6a9c  b+3: 0xbfec6a8c

Increment of ‘a’ is by 4-bytes, sizeof(int), but the increment of ‘b’ is by 20-bytes. The question is why?
Row-Major Space Allocation

The answer lies in the row-major memory space allocation of 2-D array by the C compiler.
1-D Array


a+2 a+4 a+6 a+8 a+10
2-D Array

Row-Major memory mapping

Main Memory
Arithmetic of ‘b’

- $b$ is the address of the $0^{th}$ row.
- $b+1$ is the address of the $1^{st}$ row.
- $b+i$ is the address of the $i^{th}$ row.

The size of a row is

$$c \times \text{sizeof}(\text{int})$$

$$= 5 \times \text{sizeof}(\text{int})$$

$$= 5 \times 4 = 20 \text{ bytes}$$

where $c$ is the number of columns.
Arithmetic of ‘b’

The difference between $b + 1$ and $b$ is 20 and that of $b+i$ and $b$ is $20i$.

$b+i$ points to the $i^{th}$ row
Type of ‘b’

‘b’ is a pointer constant of type \texttt{int [] [5]}, a row of five \texttt{int}. If such a pointer is incremented, it goes up by \(5 \times \text{sizeof(int)}\) (number of bytes).

Type \texttt{int [] [5]} is equivalent to \texttt{int (*)[5]}.
Arithmetic of *(b+i)

- If \( b \) is the address of the 0\(^{th} \) row, \( *b \) is the 0\(^{th} \) row itself. A row may be viewed as an 1-D array, so \( *b \) is the address of the 0\(^{th} \) element of the 0\(^{th} \) row.

- Similarly \( b+i \) is the address of the \( i^{th} \) row, \( *(b+i) \) is the \( i^{th} \) row, so \( *(b+i) \) is the address of the 0\(^{th} \) element of the \( i^{th} \) row.
Arithmetic of *(b+i)

- If \( b \) is the address of the 0\(^{th}\) element of the 0\(^{th}\) row, \( b + 1 \) is the address of the 1\(^{st}\) element of the 0\(^{th}\) row.

- Similarly \( b + j \) is the address of the \( j^{th}\) element of the 0\(^{th}\) row.

- The difference between \( b + 1 \) and \( b \) is 20 (bytes) but the difference between \( b + 1 \) and \( b \) is the \texttt{sizeof(int)}, 4 (bytes).
Arithmetic of *(b+i)

- If *(b+i) is the address of the 0^{th} element of the \(i^{th}\) row, *(b+i) + 1 is the address of the 1^{st} element of the \(i^{th}\) row.
- Similarly *(b+i) + j is the address of the \(j^{th}\) element of the \(i^{th}\) row.
- The difference between \(b + i\) and \(b\) is 20\(i\) (bytes), but the difference between *(b + i) + j and *(b+i) is 4\(j\) (bytes).
**2-D Array**

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>b+1</td>
<td>b+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b+1</td>
<td>b+2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Memory Mapping**

- **Row-Major**

  - **Address Calculation**
    - `b + (b+1)*b + 2 + (b+1)*b + 3 + (b+1)*b + 4 + 1`
    - `b + (b+1)*b + 2 + (b+1)*b + 3 + (b+1)*b + 4 + 2`
    - `b + (b+1)*b + 2 + (b+1)*b + 3 + (b+1)*b + 4 + 3`
    - `b + (b+1)*b + 2 + (b+1)*b + 3 + (b+1)*b + 4 + 4`

- **Main Memory**
#include <stdio.h>
int main() // 2DArith2.c
{
    int b[3][5] ;
    printf("b: %p \t *b: %p \n", b, *b) ;
    printf("b+1: %p \t *b+1: %p \n", b+1, *b+1) ;
    printf("b+2: %p \t *(b+2): %p \t *(b+2)+3: %p \n", b+2, *(b+2), *(b+2)+3) ;
    return 0;
}
Output

$ cc -Wall 2DArith2.c
$ a.out
b: 0xbfeb3360 *b: 0xbfeb3360
b+1: 0xbfeb3374 *b+1: 0xbfeb3364
b+2: 0xbfeb3388 *(b+2): 0xbfeb3388
*(b+2)+3: 0xbfeb3394
We know that

- \( b \) is the address of the 0\(^{th} \) row,
- \( b+i \) is the address of the \( i^{th} \) row,
- \( *(b+i) \) is the address of the 0\(^{th} \) element of the \( i^{th} \) row,
- \( *(b+i)+j \) is the address of the \( j^{th} \) element of the \( i^{th} \) row,
We know that

- \((b + i) + j\) is the address of the \(j^{th}\) element of the \(i^{th}\) row,
- \(b[i][j]\) is the \(j^{th}\) element of the \(i^{th}\) row,
- \&b[i][j]\) is the address of the \(j^{th}\) element of the \(i^{th}\) row, so

\[(b + i) + j \text{ is equivalent to } \&b[i][j]\]
We know that \((b+i) + j\) is the address of the \(j^{th}\) element of the \(i^{th}\) row, so

\[\text{***(b + i) + j** and } b[i][j]\]

\[\text{*((*(b + i) + j) is equivalent to } b[i][j]}\]
Equivalences

- \(*((b + i) + j)\) is equivalent to \(b[i][j]\)
- \(*_(b + i) + j\) is equivalent to \&_{b[i][j]}
- \(*_(b[i] + j)\) is equivalent to \(b[i][j]\)
- \(b[i] + j\) is equivalent to \&_{b[i][j]}
- \(*_(b+i))\)[j] is equivalent to \(b[i][j]\)

[We shall use the right-hand side notations]
```c
#include <stdio.h>
int main() // 2DArith3.c
{
    int b[3][5] = {{0,1,2,3,4},
                   {5,6,7,8,9},
                   {10,11,12,13,14}};

    printf("b[2][3]: %d\n", b[2][3]);
    printf("(*(b+2))\[3\]: %d\n", (*(b+2))\[3\]);
    printf("*(*(b+2)+3): %d\n", *(*(b+2)+3));
}
```
printf("&b[2][3]: %p\n", &b[2][3]);
printf("*(b+2)+3: %p\n", *(b+2)+3);
printf("b[2]+3: %p\n", b[2]+3);
return 0;
}
Output

```bash
$ cc -Wall 2DArith3.c
$ a.out
b[2][3]: 13
(*(b+2))[3]: 13
*(*(b+2)+3): 13
&b[2][3]: 0xbfe94c44
*(b+2)+3: 0xbfe94c44
b[2]+3: 0xbfe94c44
```
Calculation of the Address of $b[i][j]$

Given the declaration `int b[3][5]`, the C compiler can calculate the address of the $j^{th}$ element of the $i^{th}$ row by the following formula:

$$b + k(5i + j)$$

where $k = \text{sizeof}(\text{int})$. Other than the value of row and column indices the compiler needs the starting address $b$, the number of columns 5 and the size of the data type.
#include <stdio.h>
#define COL 5
int main() // 2DArith4.c
{
    int b[3][COL], i=1, j=2;
    printf("&b[%d][%d]=%p\n",i,j,&b[i][j]);
    printf("(int*)(b+%d)+%d=%p\n",i,j,(int*)(b+i)+j);
    printf("(int)b+%d*('%d*%d+%d)=0x%x\n",sizeof(int),
            COL, i,j,(int)b+sizeof(int)*(COL*i+j));
    return 0;
}
Output

$ cc -Wall 2DArith4.c
$ a.out

&b[1][2]=0xbff6104c
(int *)(b+1)+2=0xbff6104c
(int)b+4*(5*1+2)=0xbff6104c
Consider the declaration `int a[10],`

- the array name is a pointer constant.

- the formal parameter: `int x[]` or `int *x` is a pointer variable of the corresponding type, where the address of an array location is copied.

- These two information are sufficient for the compiler to compute the address of `x[i]`. 
Formal Parameter for 2-D Array

Consider the declaration $type \ b[ROW][COL]$. C compiler needs the starting address $b$, the data type $type$, and the number of columns $COL$ inside a called function to calculate the address of $x[i][j]$ (values of $i$ and $j$ are information local to the function).
Formal Parameter for 2-D Array

The formal parameter looks like

\[ \ldots \textbf{but}(\textit{type} \ x[\ldots]) \ldots \]

where \( x \) is a variable of type \( \textit{type} \ [\ldots] \).
Consider the real matrices \( [a_{ij}]_{p \times q} \) and \( [b_{ij}]_{q \times r} \). The product matrix \( [c_{ij}]_{p \times r} = [a_{ij}]_{p \times q} \times [b_{ij}]_{q \times r} \), where \( c_{ij} = \sum_{k=1}^{q} a_{ik} \times b_{kj} \), for all \( i, 1 \leq i \leq p \) and all \( j, 1 \leq j \leq r \). We can store the matrices in 2-D array and multiply.
```c
#include <stdio.h>
#define MAXROW 50
#define MAXCOL 50
void matMult( // matMult.c
      double matA[][MAXCOL],
      double matB[][MAXCOL],
      double matC[][MAXCOL],
      int rowA, int colA, int colB
  ) {
  int i, j, k ;
  
  for(i = 0; i < rowA; ++i)
    
```
for(j = 0; j < colB; ++j) {
    matC[i][j] = 0.0;
    for(k = 0; k < colA; ++k)
        matC[i][j] += matA[i][k]*matB[k][j];
}

void readMatrix(
    char *name,
    double x[][MAXCOL],
    int row, int col
) {
    int i, j;

    printf("Enter the matrix %s:\n", name);
printf("Row-by-row\n")
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        scanf("%lf", &x[i][j]);
}

void writeMatrix(
    char *name,
    double x[][MAXCOL],
    int row, int col
    ) {
    int i, j;
    printf("The matrix %s:\n", name);
    for(i = 0; i < row; ++i) {
        for(j = 0; j < col; ++j)
        }
printf("%6.2f ", x[i][j]);
printf("\n") ;
}
}

int main() // matMult.c data matData
{

double aMat[MAXROW][MAXCOL],
bMat[MAXROW][MAXCOL],
cMat[MAXROW][MAXCOL];
int aRow, aCol, bCol;

printf("Enter the row and column numbers of A\n");
scanf("%d%d", &aRow, &aCol);
readMatrix("A", aMat, aRow, aCol);
printf("Enter the column numbers of B\n");
scanf("%d", &bCol);
readMatrix("B", bMat, aCol, bCol);
writeMatrix("A", aMat, aRow, aCol);
writeMatrix("B", bMat, aCol, bCol);
matMult(aMat, bMat, cMat, aRow, aCol, bCol);
writematrix("C", cMat, aRow, bCol);
return 0;
$ cc -Wall matMult.c
$ a.out < matData

Enter the row and column numbers of A
Enter the matrix A:
Row-by-row
Enter the column numbers of B
Enter the matrix B:
Row-by-row
The matrix A:
1.00 2.00 3.00 4.00
5.00 6.00 7.00 8.00
The matrix B:
0.00 2.00 3.00
4.00 0.00 6.00
The matrix \( C \):

\[
\begin{bmatrix}
7.00 & 8.00 & 0.00 \\
1.00 & 5.00 & 6.00 \\
33.00 & 46.00 & 39.00 \\
81.00 & 106.00 & 99.00 \\
\end{bmatrix}
\]
Type of $x$ in readMatrix()

Consider the prototype
...... readMatrix(..., int $x[][50]$, ..)

- $x$ is single a variable of type pointer to an int i array of 50-locations,
- we can equivalently write
  .... readMatrix(..., int (*$x$)[50], ..).
  Increment of $x$ is a jump by $50 \times \text{sizeof(int)}$ bytes.
- The parenthesis is essential, otherwise in
  ...... readMatrix(..., int *$x[50]$, ..), $x$ is a pointer to an int pointer
#include <stdio.h>
#define MAXROW 10
#define MAXCOL 50
void what(int x[][MAXCOL], int (*y)[MAXCOL]){
    printf("x: \%u\tx+1: \%u\n",
           (unsigned)x, (unsigned)(x+1)) ;
    printf("y: \%u\ty+1: \%u\n",
           (unsigned)y, (unsigned)(y+1)) ;
}
int main() // 2DArith5.c
{
    int a[MAXROW][MAXCOL] ;
printf("a: %u\ta+1: %u\n",
       (unsigned)a, (unsigned)(a+1)) ;
what(a,a) ;
return 0;
}
$ cc -Wall 2DArith5.c
$ a.out
a: 3220066416 a+1: 3220066616
x: 3220066416 x+1: 3220066616
y: 3220066416 y+1: 3220066616