2-d Arrays
Two Dimensional Arrays

- We have seen that an array variable can store a list of values.
- Many applications require us to store a table of values.

<table>
<thead>
<tr>
<th></th>
<th>Subject 1</th>
<th>Subject 2</th>
<th>Subject 3</th>
<th>Subject 4</th>
<th>Subject 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student 1</td>
<td>75</td>
<td>82</td>
<td>90</td>
<td>65</td>
<td>76</td>
</tr>
<tr>
<td>Student 2</td>
<td>68</td>
<td>75</td>
<td>80</td>
<td>70</td>
<td>72</td>
</tr>
<tr>
<td>Student 3</td>
<td>88</td>
<td>74</td>
<td>85</td>
<td>76</td>
<td>80</td>
</tr>
<tr>
<td>Student 4</td>
<td>50</td>
<td>65</td>
<td>68</td>
<td>40</td>
<td>70</td>
</tr>
</tbody>
</table>
The table contains a total of 20 values, five in each line
  - The table can be regarded as a matrix consisting of four rows and five columns

C allows us to define such tables of items by using two-dimensional arrays
Declaring 2-D Arrays

- General form:
  
  \[ \text{type} \ \text{array\_name \ [row\_size][column\_size]}; \]

- Examples:
  
  int marks[4][5];
  float sales[12][25];
  double matrix[100][100];
Initializing 2-d arrays

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

All of the above will give the 2x3 array

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Accessing Elements of a 2-d Array

- Similar to that for 1-d array, but use two indices
  - First indicates row, second indicates column
  - Both the indices should be expressions which evaluate to integer values (within range of the sizes mentioned in the array declaration)

- Examples:
  
  ```
  x[m][n] = 0;
  c[i][k] += a[i][j] * b[j][k];
  a = sqrt (a[j*3][k]);
  ```
Example

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

A two-dimensional array of 15 elements
Can be looked upon as a table of 3 rows and 5 columns

<table>
<thead>
<tr>
<th></th>
<th>col0</th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>col4</th>
</tr>
</thead>
<tbody>
<tr>
<td>row0</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>row1</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>
How is a 2-d array is stored in memory?

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

- $a[i][j] \Rightarrow$ is allocated memory location at address $x + (i \times c + j) \times k$

<table>
<thead>
<tr>
<th></th>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[0][3]</th>
<th>a[1][0]</th>
<th>a[1][1]</th>
<th>a[1][2]</th>
<th>a[1][3]</th>
<th>a[2][0]</th>
<th>a[2][1]</th>
<th>a[2][2]</th>
<th>a[2][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
int main()
{
    int a[3][5];
    int i, j;

    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
            printf("%u
", &a[i][j]);
        printf("\n");
    }
    return 0;
}
int main()
{
    int a[3][5];
    printf("a = %u\n", a);
    printf("&a[0][0] = %u\n", &a[0][0]);
    printf("&a[2][3] = %u\n", &a[2][3]);
    printf("a[2]+3 = %u\n", a[2]+3);
    printf("*(a+2)+3 = %u\n", *(a+2)+3);
    printf("*(a+2) = %u\n", *(a+2));
    printf("a[2] = %u\n", a[2]);
    printf("&a[2][0] = %u\n", &a[2][0]);
    printf("(a+2) = %u\n", (a+2));
    printf("&a[2] = %u\n", &a[2]);
    return 0;
}

Output

a = 3221224480
&a[0][0] = 3221224480
&a[2][3] = 3221224532
a[2]+3 = 3221224532
*(a+2)+3 = 3221224532
*(a+2) = 3221224520
a[2] = 3221224520
&a[2][0] = 3221224520
(a+2) = 3221224520
&a[2] = 3221224520
How to read the elements of a 2-d array?

- By reading them one element at a time
  
  ```c
  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
How to print the elements of a 2-d array?

- By printing them one element at a time
  
  ```c
  for (i=0; i<nrow; i++)
      for (j=0; j<ncol; j++)
          printf ("\n \%f", a[i][j]);
  ``
  
  - The elements are printed one per line

  ```c
  for (i=0; i<nrow; i++)
      for (j=0; j<ncol; j++)
          printf ("\%f", a[i][j]);
  ``
  
  - The elements are all printed on the same line
Contd.

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

The elements are printed nicely in matrix form
Example: Matrix Addition

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

    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++)
    {
        printf("\n");
        for (q=0; q<n; q++)
            printf("%d   ", c[p][q]);
    }
    return 0;
}
```
Passing 2-d Arrays as Parameters

- Similar to that for 1-D 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 2-d array, we need:
  - The starting address of the array in memory
  - Number of bytes per element
  - Number of columns in the array
- The above three pieces of information must be known to the function
Example Usage

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

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

We can also write

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

But at least 2nd dimension must be given
Dynamic Allocation of 2-d Arrays

- Recall that address of [i][j]-th element is found by first finding the address of first element of i-th row, then adding j to it
- Now think of a 2-d array of dimension [M][N] as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k-th row can be found by adding 1 to the starting address of (k-1)-th row)
- This is done by allocating an array p of M pointers, the pointer p[k] to store the starting address of the k-th row
Contd.

- Now, allocate the M arrays, each of N elements, with p[k] holding the pointer for the k-th row array.
- Now p can be subscripted and used as a 2-d array.
- Address of p[i][j] = *(p+i) + j (note that *(p+i) is a pointer itself, and p is a pointer to a pointer)
Dynamic Allocation of 2-d Arrays

```c
int **allocate (int h, int w)
{
    int **p;
    int i, j;

    p = (int **) malloc(h*sizeof (int *) );
    for (i=0;i<h;i++)
        p[i] = (int *) malloc(w * sizeof (int));
    return(p);
}

void read_data (int **p, int h, int w)
{
    int i, j;
    for (i=0;i<h;i++)
        for (j=0;j<w;j++)
            scanf ("%d", &p[i][j]);
}
```

Allocate array of pointers

Allocate array of integers for each row

Elements accessed like 2-D array elements.
### void print_data (int **p, int h, int w)

```c
{
    int i, j;
    for (i=0; i<h; i++) {
        for (j=0; j<w; j++)
            printf("%5d ", p[i][j]);
        printf("\n");
    }
}
```

### int main()

```c
{
    int **p;
    int M, N;
    printf("Give M and N \n");
    scanf("%d%d", &M, &N);
    p = allocate (M, N);
    read_data (p, M, N);
    printf("\nThe array read as \n");
    print_data (p, M, N);
    return 0;
}
```
void print_data (int **p, int h, int w)
{
    int i, j;
    for (i=0; i<h; i++)
    {
        for (j=0; j<w; j++)
        {
            printf ("%5d ", p[i][j]);
            printf ("\n");
        }
    }
}

int main()
{
    int **p;
    int M, N;
    printf ("Give M and N\\n");
    scanf ("%d%d", &M, &N);
    p = allocate (M, N);
    read_data (p, M, N);
    printf ("The array read as \\
");
    print_data (p, M, N);
    return 0;
}
Memory Layout in Dynamic Allocation

```c
int main()
{
    int **p;
    int M, N;
    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;
}

int **allocate (int h, int w)
{
    int **p;
    int i, j;

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>31535120</td>
<td>31535128</td>
<td>31535136</td>
</tr>
<tr>
<td>31535152</td>
<td>31535156</td>
<td>31535160</td>
</tr>
<tr>
<td>31535184</td>
<td>31535188</td>
<td>31535192</td>
</tr>
<tr>
<td>31535216</td>
<td>31535220</td>
<td>31535224</td>
</tr>
</tbody>
</table>

Starting address of each row, contiguous (pointers are 8 bytes long)

Elements in each row are contiguous