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

	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

Contd.

- 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:

  type   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

1	2	3
4	5	6

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

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

A two-dimensional array of 15 elements

Can be looked upon as a table of 3 rows and 5 columns

	col0	**col1**	**col2**	**col3**	**col4**
row0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

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] ➔ is allocated memory location at address $x + (i * c + j) * k$

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

Row 0 **Row 1** **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 2-d 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

How to print the elements of a 2-d array?

- By printing them one element at a time

```
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

```
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

```
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

```
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];

But at least 2^{nd} 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];
}
```

```
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;
}
```

Dynamic Allocation of 2-d Arrays

```
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);
  }
```

Allocate array of pointers

Allocate array of integers for each row

```
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]);
 }
```

Elements accessed like 2-D array elements.

Contd.

```
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 ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
}
```

Contd.

```
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");
      }
}
```

```
Give M and N
3 3
1 2 3
4 5 6
7 8 9
 The array read
as
    1     2     3
    4     5     6
```

```
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 ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
}
```

Memory Layout in Dynamic Allocation

```
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

```
3 3
31535120  31535128  31535136


31535152  31535156  31535160
31535184  31535188  31535192
31535216  31535220  31535224
```

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

Elements in each row are contiguous

Practice Problems

1. Write a function that takes a n x 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. Write a function that takes as parameter an m x n matrix A (m, n < 100) and returns the transpose of A (modifies in A only).
4. Consider a n x n matrix containing 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
5. Write a function that reads in an m x n matrix A and an n x p matrix B, and returns the product of A and B in another matrix C. Pass appropriate parameters.

For each of the above, also write a main function that reads the matrices, calls the function, and prints the results (a message, the transposed matrix etc.). Practice each with both static and dynamica-allocated 2-d arrays