CS11001/CS11002
Programming and Data Structures (PDS)
(Theory: 3-1-0)

Multi-demensional Arrays

Multi-dimensional Arrays

- One-dimensional arrays are quite able to represent many natural collections.
- There are some other natural collections that may better be conceptualized as 2-dimensional data.
- The first example is a matrix. What else can be a more natural 2-dimensional data other than a matrix whose entries are natural numbers?
- So think of the following 4x5 matrix:

```
1   1   1    1    1
2   3   4    5    6
4   9  16   25   36
8  27  64  125  216
```

2-D Array as 1-D array and vice-versa

- We can write the entries in the row-major order and represent the resulting flattened data as a one-dimensional array:

  1 1 1 1 1 2 3 4 5 6 4 9 16 25 36 8 27 64 125 216
- As long as the column dimension of the matrix is known, the original matrix can be recovered easily from this 1-D array.
- Consider an m-by-n matrix (a matrix with m rows and n columns). It contains a total of mn elements.
- Let us number the rows 0,1,...,m-1 from top to bottom and the columns 0,1,...,n-1 from left to right.
- The entry at position (i,j) then maps to the (ni+j)-th entry of the one-dimensional array.
- On the other hand, the k-th entry of the one-dimensional array corresponds to the (i,j)-th element of the matrix, where i = k / n and j = k % n.

2-D arrays in C

- One-dimensional arrays suffice.
- Still, it is convenient and intuitive to visualize matrices as two-dimensional arrays.
- C provides constructs to define and work with such arrays.
- Of course, the memory of a computer is typically treated as a one-dimensional list of memory cells.
- Any two-dimensional structure has to be flattened using a strategy like that mentioned above.
- C handles this for you. In other words, the abstraction relieves you from the task of doing the index arithmetic explicitly.
- You refer to the (i,j)-th element as the (i,j)-th element. C translates it into the appropriate address in the one-dimensional memory.

Defining a 2-D array in C

- 2-dimensional arrays can be defined like one-dimensional arrays, but with two square-bracketed dimensions. For example, the declaration

```
int matrix[20][10];
```

- This allocates memory for a 20x10 array of int variables.
- The first index (20) indicates the number of rows allocated, whereas the second indicates the number of columns allocated.
- Here is another example:

```
#define MAXROW 50
#define MAXCOL 50
float M[MAXROW][MAXCOL];
```

Initialization

- int mat[4][5] = { { 1, 1, 1, 1, 1 }, /* The zeroth row */
  { 2, 3, 4, 5, 6 }, /* The first row */
  { 4, 9, 16, 15, 25 }, /* The second row */
  { 8, 27, 64, 125, 216 } /* The third row */ };

Rows of a 2-D array of characters can be initialized to constant strings.

- char address[4][100] = { "Department of Foobarnautic Engineering",
  "Indian Institute of Technology",
  "Kharagpur 721302",
  "India" };
- One could have also written, char address []\[100]=…

Accessing the 2-D array

- For a 2-D array A the (i,j)-th element is treated as a variable and can be accessed by the name A[i][j].
- Both the row numbering and the column numbering start from 0.
- For example, the (1,3)-th element of mat is accessed as mat[1][3] and, if initialized as above, stores the int value 5.

Passing 2-D arrays to functions

- 2-D arrays can be passed to functions using a syntax similar to the declaration of 2-D arrays:
- #define ROWDIM 10

  #define COLDIM 12

  int fooray (int A[ROWDIM][COLDIM], int r , int c) { ... }

Passing 2-D arrays to functions

- Here the actual row and column dimensions of the used part of the array A are passed via the parameters r and c.
- It is not mandatory to specify the row dimension ROWDIM.
- But the column dimension COLDIM must be specified, since 2-D to 1-D mapping in memory requires the column dimension. Thus the declaration

  int fooray (int A[][COLDIM], int r , int c) { ... }

  is allowed, whereas the declarations

Passing 2-D arrays to functions

- int fooray (int A[][COLDIM], int r , int c) { ... } is allowed, whereas the declarations

- int fooray (int A[][], int r , int c) { ... } and
- int fooray (int A[ROWDIM][], int r , int c) { ... } are not allowed.

Pointers and 2-D arrays

- Like 1-D arrays, 2-D arrays are not copied element-by-element to functions. A pointer is only passed. This implies that changes made to the array elements inside the function are visible outside the function.
- Indeed 2-D arrays are pointers too. However, these pointers are rather distinct in nature from those pointers that represent 1-D arrays. The situation is quite clumsy and confusing.

Example 1: Finding saddle point of a matrix

An element A[i][j] is a saddle point, if it is the least element in the ith row and maximum in the jth column.

```
#include<stdio.h>

main()
{
 int i, j, k;
 int m, n, p, q, min;
 int a[10][10];
 int flag=1;
 // Read dimension and matrix elements
 scanf("%d %d",&m, &n);

 for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    scanf("%d",&a[i][j]);
```

```
for(i=0;i<m;i++)
 {
  min=a[i][0];
  p=i;q=0;
   //finding the min element of ith row
    for(j=0;j<n;j++)
    {
       if(a[i][j]<min)
        {
         min=a[i][j];
         p=i;q=j;
      }
    }
```

```
//checking if min element is max in the column
for(j=0;j<m;j++)
  {
    if(a[j][q]>a[p][q])
      flag=0; //otherwise set flag to 0
  }
    if(flag)
       printf("Saddle point is
   a[%d][%d]=%d\n",p+1,q+1,a[p][q]);
    else{ //it may be there is another min in the
   //row which is the saddle
```

```
for(k=q+1;k<n;k++)
     {
     flag=1;
     if(a[i][k]==min)//if any element is also min, then look into that col. also
      {
        for(j=0;j<m;j++)
        {
         if(a[j][k]>a[i][k])
           flag=0;
        }
       }
      else flag=0;

   if(flag)
         printf("Saddle point is a[%d][%d]=%d\n",i+1,k+1,a[i][k]);
      }
    }
    if(!flag)//the ith row has no saddle point
      printf("No such point in row %d\n",i+1);
  flag=1;
 }
}
```

Example 2: Matrix Multiplication

```
#include<stdio.h>
main()
{
 int a[5][5], b[5][5], c[5][5];
 int m1, n1, m2, n2, i, j;
 void matread(int [][5],int ,int );
 void matwrite(int [][5],int ,int );
 void matmul(int [][5], int [][5], int [][5], int, int, int,
   int);
 scanf("%d %d",&m1,&n1);
 scanf("%d %d",&m2, &n2);
```

Call to read and write functions

```
printf("Enter Matrix A\n");
 matread(a,m1,n1);
 matwrite(a,m1,n1);
 printf("Enter Matrix B\n");
 matread(b,m2,n2);
 matwrite(b,m2,n2);

 matmul(a,b,c,m1,n1,m2,n2);
 matwrite(c,m1,n2);
}
```

Matrix Read and Write

```
void  matread(int a[][5], int m,
     int n)
{
  int i, j;
  for(i=0;i<m;i++)
   for(j=0;j<n;j++)
     {
        scanf("%d",&a[i][j]);
     }

}
```

```
void  matwrite(int a[][5], int m,
     int n)
{
  int i, j;
  printf("m=%d n=%d\n",m,n);

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

}
```

The Matrix Multiplication

```
void  matmul(int a[][5], int b[][5], int c[][5], int m1, int n1, int m2, int n2)
{
  int i, j, k;
printf("Dimension of A: row=%d, col=%d\n",m1, n1);
printf("Dimension of B: row=%d, col=%d\n",m2, n2);

printf("Matrix A:\n");
matwrite(a,m1,n1);
printf("Matrix B:\n");
matwrite(b,m2,n2);


if(n1 != m2)
   printf("Mult not defined\n");
 else{
   for(i=0;i<m1;i++)
      for(j=0;j<n2;j++)
         c[i][j]=0;

   for(i=0;i<m1;i++)
     for(j=0;j<n2;j++)
        for(k=0;k<n1;k++)
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
   }
}
```