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

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

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

We can also write

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

But at least 2nd 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;
}
```

Practice Problems

1. Write a function that takes an 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 an 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.)

Structures

What is a Structure?

- Used for handling a group of logically related data items
  - Examples:
    - Student name, roll number, and marks
    - Real part and complex part of a complex number
- Helps in organizing complex data in a more meaningful way
- The individual structure elements are called members

Defining a Structure

```
struct tag {
        member 1;
        member 2;
        :
        member m;
      };
```

- struct is the required C keyword
- tag is the name of the structure
- member 1, member 2, … are individual member declarations
- Do not forget the ; at the end!

Contd.

- The individual members can be ordinary variables, pointers, arrays, or other structures (any data type)
  - The member names within a particular structure must be distinct from one another
  - A member name can be the same as the name of a variable defined outside of the structure
- Once a structure has been defined, the individual structure-type variables can be declared as:

  struct tag var_1, var_2, …, var_n;

Example

- A structure definition

```
struct student {
        char name[30];
        int  roll_number;
        int  total_marks;
        char dob[10];
    };
```

- Defining structure variables:

$$\underbrace{\text{struct student}}_{\text{A new data-type}}\ \text{a1, a2, a3;}$$

A Compact Form

- It is possible to combine the declaration of the structure with that of the structure variables:

```
struct tag {
            member 1;
            member 2;
            :
            member m;
        }  var_1, var_2,…, var_n;
```

- Declares three variables of type struct tag
- In this form, tag is optional

Accessing a Structure

- The members of a structure are processed individually, as separate entities
  - Each member is a separate variable
- A structure member can be accessed by writing

  variable.member

  where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure
- Examples:

  a1.name, a2.name, a1.roll_number, a3.dob

Example: Complex number addition

```
struct  complex
        {
            float  real;
            float  img;
        };
int main()
{
    struct complex a, b, c;
    scanf ("%f %f", &a.real, &a.img);
    scanf ("%f %f", &b.real, &b.img);
    c.real = a.real + b.real;
    c.img = a.img + b.img;
    printf ("\n %f + %f j", c.real, c.img);
    return 0;
}
```

Defines the structure

Declares 3 variable of type struct complex

Accessing the variables is the same as any other variable, just have to follow the syntax to specify which field of the Structure you want

Operations on Structure Variables

- Unlike arrays, a structure variable can be directly assigned to another structure variable of the same type

  a1 = a2;

  - All the individual members get assigned
- Two structure variables cannot be compared for equality or inequality

  if (a1 == a2)…… ← this cannot be done

Arrays of Structures

- Once a structure has been defined, we can declare an array of structures

  $\underbrace{\text{struct student}}_{\textbf{type name}}$ class[50];

  - The individual members can be accessed as:

    class[i].name

    class[5].roll_number

Example: Reading and Printing Array of Structures

```
int main()
{
        struct complex A[100];
        int n;
        scanf("%d", &n);
        for (i=0; i<n; i++)
              scanf("%f%f", &A[i].real, &A[i].img);
        for (i=0; i<n; i++)
              printf("%f + i%f\n", A[i].real, A[i].img);
}
```

Arrays within Structures

- A structure member can be an array

```
struct  student
{
    char  name[30];
    int  roll_number;
    int  marks[5];
    char  dob[10];
}  a1, a2, a3;
```

- The array element within the structure can be accessed as:

  a1.marks[2], a1.dob[3],…

Structure Initialization

- Structure variables may be initialized following similar rules of an array. The values are provided within the second braces separated by commas
- An example:

```
struct complex a={1.0,2.0}, b={-3.0,4.0};
```

```
a.real=1.0;   a.img=2.0;
b.real=-3.0;  b.img=4.0;
```

Parameter Passing in a Function

- Structure variables can be passed as parameters like any other variables. Only the values will be copied during function invocation

```
int chkEqual(struct complex a, struct complex b)
{
        if ((a.real==b.real) && (a.img==b.img))
            return 1;
        else return 0;
}
```

Returning Structures

- It is also possible to return structure values from a function. The return data type of the function should be as same as the data type of the structure itself

```
struct complex add(struct complex a, struct complex b)
{
    struct complex tmp;

    tmp.real = a.real + b.real;
    tmp.img = a.img + b.img;
    return(tmp);
}
```

Direct arithmetic operations are not possible with structure variables

Defining Data Type: using typedef

- One may define a structure data-type with a single name

```
typedef struct newtype {
        member-variable1;
        member-variable2;
              .
        member-variableN;
 } mytype;
```

- mytype is the name of the new data-type
  - Also called an alias for struct newtype
  - Writing the tag name newtype is optional, can be skipped
  - Naming follows rules of variable naming

typedef : An example

```
typedef struct {
        float real;
        float imag;
      }  _COMPLEX;
```

- Defined a new data type named _COMPLEX. Now can declare and use variables of this type

```
_COMPLEX a, b, c;
```

More about typedef

- Note: typedef is not restricted to just structures, can define new types from any existing type
- Example:
  - typedef int INTEGER
  - Defines a new type named INTEGER from the known type int
  - Can now define variables of type INTEGER which will have all properties of the int type

```
INTEGER a, b, c;
```

The earlier program using typedef

```
typedef struct{
                float real;
                float img;
        } _COMPLEX;

_COMPLEX add(_COMPLEX a, _COMPLEX b)
{
        _COMPLEX tmp;

        tmp.real = a.real + b.real;
        tmp.img = a.img + b.img;
        return(tmp);
}
```

Contd.

```
void print (_COMPLEX a)
{
     printf("(%f, %f) \n",a.real,a.img);
}

int main()
{
    _COMPLEX x={4.0,5.0}, y={10.0,15.0}, z;

     print(x);
     print(y);
     z = add(x,y);
     print(z);
     return 0;
}
```

Output

```
(4.000000, 5.000000)
(10.000000, 15.000000)
(14.000000, 20.000000)
```

Practice Problems

1. Extend the complex number program to include functions for addition, subtraction, multiplication, and division
2. Define a structure for representing a point in two-dimensional Cartesian co-ordinate system. Using this structure for a point
    1. Write a function to return the distance between two given points
    2. Write a function to return the middle point of the line segment joining two given points
    3. Write a function to compute the area of a triangle formed by three given points
    4. Write a main function and call the functions from there after reading in appropriate inputs (the points) from the keyboard

3. Define a structure STUDENT to store the following data for a student: name (null-terminated string of length at most 20 chars), roll no. (integer), CGPA (float). Then
   1. In main, declare an array of 100 STUDENT structures. Read an integer n and then read in the details of n students in this array
   2. Write a function to search the array for a student by name. Returns the structure for the student if found. If not found, return a special structure with the name field set to empty string (just a '\0')
   3. Write a function to search the array for a student by roll no.
   4. Write a function to print the details of all students with CGPA > x for a given x
   5. Call the functions from the main after reading in name/roll no/CGPA to search