Two dimensional arrays

The data of many systems require arrays with two or more dimensions. For example, a digital image requires two dimensional arrays to store the colors. Two indices are used to refer to an element in a 2D array: row and column. A one or more dimensional array lives in memory as a contiguous block of memory.

Declaration:

```
DataType ArrayName[ROWS] [COLUMNS]
```

DataType: data type of array elements
ArrayName: Name of the array
ROWS: number of rows
COLUMNS: number of columns

- Arrays declared this way are called static array, they are created on the stack and their size is fixed at compilation time.
- No need to deal with memory management for these arrays. They get destroyed automatically when the control leaves the block where they defined in.
- The number of memory bytes required to hold this array is ROWS*COLUMNS*sizeof(DataType). For example, the declaration

```
int A[4][3];
```

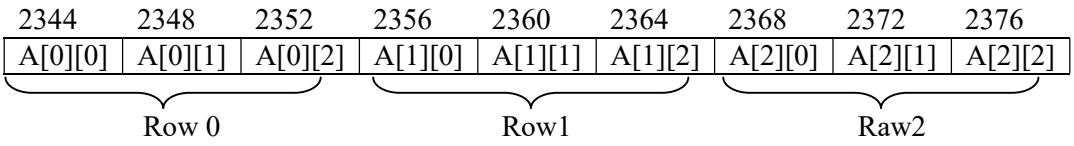
requires a block of memory of size 4*3 = 12 memory locations where each location is size of int bytes. If the sizeof (int) is 4 bytes, then the array requires 48 bytes.

Memory layout

In memory, two or multidimensional arrays are stored the same as one dimensional array in a contiguous memory block. C language is row major, which means the first row is stored, followed by the second row, the third row and so on. The 2D array A:

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]

is stored as



assuming the first element of the array A[0][0] is stored at memory location 2344, the array A data type is int, and the sizeof(int) is 4 bytes, the memory addresses of the rest of elements will be as shown above .

Initialization:

- 2D arrays can be initialized when declared

```
int A[2][3] = {{1,2,3}, {4,5,6}};
```

The inner braces are not required. Or

```
int A[][3] = {{1,2,3}, {4,5,6}};
```

The number of rows can be omitted

- For large arrays they can be initialized using two loops, one to loop over the ROWS and the other to loop over the COLUMNS.

```
for (int r=0; r<ROWS; r++)
     for (int c=0; c<COLUMNS; c++)
          A[r][c] = some_value;
```

Access:

To access an element of a 2D array, we use two subscripts. The first corresponds to the row index and the second corresponds to the column index. The assignment,

```
A[2][3] = 9;
```

assigns 9 to the element at the 3rd row and 4th column, while the assignment,

```
int x = A[1][2];
```

extracts the element at the second row and 3rd column and assigns it to the variable x

Accessing all elements require two loops, one for the rows and the other for the columns.

```
for (int i=0; i<ROWS; i++) {
     for (int j=0; j<COLUMNS; j++){
          sum += A[i][j];
     }
}
```

Passing Array to a function

C++ doesn't allow passing a complete memory block to a function by value, but it allows passing its memory address. This is more efficient in terms of memory and speed. To declare a function to accept an array as parameter, we specify the data type of the array elements and an identifier followed by brackets. For example, the function declaration,

```
int func ( int  param[] )
```

allow the function `func` to accept a parameter of type array of integers. However, the function can't determine the length of the array passed to it using this declaration alone. Therefore, we typically declare another parameter to tell the function the length of the array. For example, the declaration,

```
 int func ( int  param[], int size )
```

allow passing the length of the array in the parameter size. In order to pass the array

```
int arr[100] = {1,2,3,4}
```

assuming it contains only 4 elements, we write the function call as:

```
int x = func(arr, 4)
```

The second parameter 4 tells the function the number of element stored in the array.

Passing multidimensional is similar. The format of passing a two-dimensional array is,

```
void func(int arra[][10], int rows, int columns)
```

Here the first bracket [] is left empty, but the sizes of the following ones must be specified for their respective dimensions. This is necessary in order for the compiler to map the multi-dimensional indices to the corresponding one dimensional index as we explained in class.

Again, we typically pass the lengths of each dimension as other parameters in order to be able to loop over the array elements in the function. As an example, we pass the array declared as:

int a[99][10] = {0}

by writing the function call

func(a, 10, 2)

assuming the array has only 10 rows and two columns in it. But trying to pass the array

int b[99][8]

will produce error because second dimension of the array allocated memory block must be 10.

Example: Assume the temperature in Nablus in the week Jan 1 - Jan 7 has been recorded in the past 5 years as follows:

Day,Month/year	2008	2009	2010	2011	2012
1/1	5	12	8	14	12
2/1	3	11	8	13	11
3/1	7	11	10	14	11
4/1	3	8	11	12	9
5/1	2	7	7	6	11
6/1	8	9	9	4	13
7/1	9	10	5	3	12

To make a trip planning you are asked to write a program to do the following:

1) Calculate the daily average temperature in Nablus in the days Jan 1-Jan 7 over the past five year.
2) Calculate the average temperature of the 7 days for each year in the past 5 years.

Solution:

It is natural to represent the data in a 2D array, where each element corresponds to the recorded temperature on one day in a particular year. Each column is a record of the temperature for a week of a particular year. Note that we can switch the role of the rows and columns.

```
#include <iostream>

using namespace std;

void computeAvgTemp(float T[][6], int r, int c) {

/* sum the first c-1 elements in each row and place the sum in the
last cell of the row also, sum the first r-1 elements in each column
and place the sum in the last cell of the column
*/

      for (int i=0; i<r-1; i++) {
            for (int j=0; j<c-1; j++) {
                  T[i][c-1] += T[i][j];
                  T[r-1][j] += T[i][j];
```

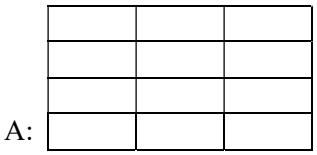
```
            }
      }

      // compute the average of each row
      for (int i=0; i<r-1; i++) {
            T[i][c-1] /= (c-1);
      }

      // compute the average of each column
      for (int i=0; i<c-1; i++) {
             T[r-1][i] /= (r-1);
      }
}

int main() {
      float temp[8][6] = { {5, 12, 8, 14, 12, 0},
                {3, 11, 8, 13, 11, 0},
                {7, 11, 10, 14, 11, 0},
                {3, 8, 11, 12, 9, 0},
                {2, 7, 7, 6, 11, 0},
                {8, 9, 9, 4, 13,0},
                {9, 10, 5, 3, 12, 0},
      {0}};
      // we added extra column and extra row to store the averages

      computeAvgTemp(temp, 8, 6);

      int r = 8, c=6;
      for (int i=0; i<r-1; i++) {
            cout << "day " << i+1 << "   " << temp[i][c-1] << endl;
      }

      for (int i=0; i<c-1; i++) {
            cout << "Year " << 2008+i << "   " << temp[r-1][i] << endl;
      }

      return 1;
}
```

Dynamically allocating 2D array

There are four cases to simulate a 2-dimensional array in c++.

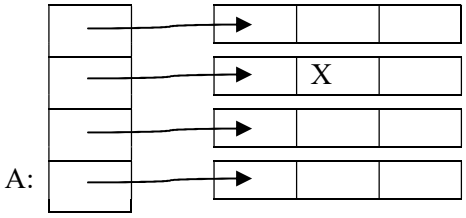
1) 2D array is an array of arrays. In this case both dimensions must be specified at compilation time. The array is organized in memory as one dimensional array where rows are stored in a contiguous memory one after another as we explained in previous note. as an example:

```
int A[4][3] ;
refer to element  at i, j using:
subscript:  A[i][j]
pointer:   *(* (A+i)+j)
```



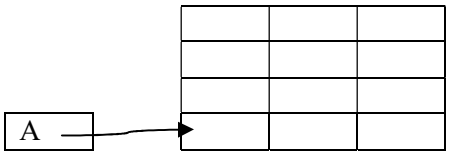
2) Array of pointer, another way to simulate a two dimensional array is to use an array of pointers where each element of the array is a pointer. See figure for example,

```
int *A[4];
for  (int i=0; i<4; i++) {
        A[i] = new int[3];
}

refer to element at i, j
subscript:   A[i][j]
pointer:     *(*(A+i)+j)

Release allocated memory:
for (int i=0; i<4; i++) {
        delete [] A[i];
}
```

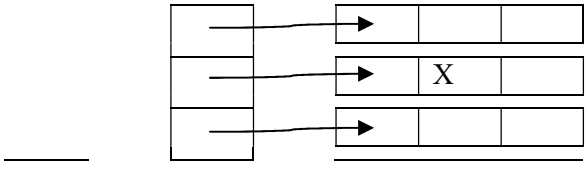


3) Pointer to array, in this case, each row has a fixed number of columns at compilation time, the number of rows is dynamically allocated (we didn't cover this in class yet). Example,

```
int (*A)[4];  // notes the parenthesis ()
A = new int[3][4];

refer to element at i, j
subscript: A[i][j]
pointer: *(*(A+i)+j)
```



4) Pointer to Pointer, in this case, we first a allocate memory for one dimensional array of pointers and then we allocated memory for each element.




```
int **A = new int*[4];
for (int i=0; i<4; i++) {
        A[i] = new int[3];
}
```

refer to element at i, j
subscript: A[i][j]
pointer: *(*(A+i)+j)

Release memory:

```
for (int i=0; i<4; i++) {
        delete [] A[i];   // free allocated memory for each element
}
delete [] A;  // free allocated memory for the array of pointers
```