Arrays and Applications

60-141: Introduction to Algorithms and Programming II

School of Computer Science

Term: Summer 2014

Instructor: Dr. Asish Mukhopadhyay

What's an array

- Let $a_0, a_1, \ldots, a_{n-1}$ be a sequence of elements of the *same type*
- An array (data structure) is created by storing these elements contiguously in memory

Nomenclature

- If we name the array a, then a[i] is a reference to the i-th element in the list.
- It is called the *index* or *position* of this element
- Just like a variable, an array has to be declared before use.
- We do this as below:

```
int a[10]; // declares an array of  size 10  of integers
char b[10] // declares an array of size 10 of characters
```

Simple array manipulations

- Initializing an array

```
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int n[ 10 ]; // n is an array of 10 integers
   size_t  i; // counter

   // initialize elements of array n to 0
   for ( i = 0; i < 10; ++i ) {
      n[ i ] = 0; // set element at location i to 0
   } // end for
```

Initializing with an initializer list

- Changes to the previous program

```
// use initializer list to initialize array n
  int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
  size_t   i; // counter
```

Note: int n[10] = {32}; would initialize the first element to 32 and the rest to 0

Specifying array size with a symbolic constant

```
int s[ SIZE ]; // array s has SIZE number of elements
size_t  j; // counter

for ( j = 0; j < SIZE; ++j ) { // set the values
   s[ j ] = 2 + 2 * j;
} // end for

printf( "%s%13s\n", "Element", "Value" );

// output contents of array s in tabular format
for ( j = 0; j < SIZE; ++j ) {
   printf( "%7u%13d\n", j, s[ j ] );
```

Summing array elements

```
#include <stdio.h>
#define SIZE 12

// function main begins program execution
int main( void )
{
   // use an initializer list to initialize the array
   int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
   size_t  i; // counter
   int total = 0; // sum of array

   // sum contents of array a
   for ( i = 0; i < SIZE; ++i ) {
      total += a[ i ];   // same as:  total = total + a[i]
   } // end for

   printf( "Total of array element values is %d\n", total );
} // end main
```

Statistical calculations

- *Problem*: Given a set of 40 responses, on a scale of 1-10, to a survey make a frequency table to summarize the survey

Statistical calculations

```
#include <stdio.h>
#define RESPONSES_SIZE 40 // define array sizes
#define FREQUENCY_SIZE 11

// function main begins program execution
int main( void )
{
   size_t  answer; // counter to loop through 40 responses
   size_t  rating; // counter to loop through frequencies 1-10

   // initialize frequency counters to 0
   int frequency[ FREQUENCY_SIZE ] = { 0 };

   // place the survey responses in the responses array
   int responses[ RESPONSES_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
      1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
      5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
```

Statistical calculations

```
   // for each answer, select value of an element of array responses
   // and use that value as subscript in array frequency to
   // determine element to increment
   for ( answer = 0; answer < RESPONSES_SIZE; ++answer ) {
      ++frequency[ responses [ answer ] ];
   } // end for

   // display results
   printf( "%s%17s\n", "Rating", "Frequency" );

   // output the frequencies in a tabular format
   for ( rating = 1; rating < FREQUENCY_SIZE; ++rating ) {
      printf( "%6d%17d\n", rating, frequency[ rating ] );
   } // end for
} // end main
```

Statistical calculations

```
   printf( "%s%13s\n", "Element", "Value" );

   // output contents of array n in tabular format
   for ( i = 0; i < 10; ++i ) {
      printf( "%7u%13d\n", i, n[ i ] );
   } // end for
} // end main
```

Array of char

- Initialization

char string[] = “first”

- This creates an array of size 6 as below

Array of char

- Reading in a string from input
- Declaration and input

```
char string[20]
scanf("%19s", string)
```

Array of char: Example

```
#include <stdio.h>
#define SIZE 20

// function main begins program execution
int main( void )
{
   char string1[ SIZE ]; // reserves 20 characters
   char string2[] = "string literal"; // reserves 15 characters
   size_t i; // counter

   // read string from user into array string1
   printf( "%s", "Enter a string (no longer than 19 characters): " );
   scanf( "%19s", string1 ); // input no more than 19 characters
```

Array of char: Example

```
// output strings
  printf( "string1 is: %s\nstring2 is: %s\n"
          "string1 with spaces between characters is:\n",
          string1, string2 );

  // output characters until null character is reached
  for ( i = 0; i < SIZE && string1[ i ] != '\0'; ++i ) {
     printf( "%c ", string1[ i ] );
  } // end for

  puts( "" );
} // end main
```

Static Vs non-static array

- Demo

Passing an array as a function parameter

- Example
  - Array Declaration

```
int hourlyTemperatures[HOURS_IN_A_DAY]
```

  - Function Call

```
modifyArray( hourlyTemperatures, HOURS_IN_A_DAY )
```

Passing an array as a function parameter

```
#include <stdio.h>

// function main begins program execution
int main( void )
{
   char array[ 5 ]; // define an array of size 5

   printf( "    array = %p\n&array[0] = %p\n   &array = %p\n",
      array, &array[ 0 ], &array );
} // end main
```

Parameter passing: array Vs array element

- Demo

Protecting array values passed as parameter

- Use the **const** qualifier as in the program below

```
#include <stdio.h>

void tryToModifyArray( const int b[] ); // function prototype

// function main begins program execution
int main( void )
{
   int a[] = { 10, 20, 30 }; // initialize array a

   tryToModifyArray( a );

   printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
} // end main
```

Protecting array values passed as parameter

```
// in function tryToModifyArray, array b is const, so it cannot be
// used to modify the original array a in main.
void tryToModifyArray( const int b[] )
{
  b[ 0 ] /= 2; // error
  b[ 1 ] /= 2; // error
  b[ 2 ] /= 2; // error
} // end function tryToModifyArray
```

Applications: List Search

- Problem:
  - Given a list *L* of, say *n* numbers, and another number *x* search for *x* in the list *L*
- Algorithm
  - Organize list as an array a[n] and search for x by increasing index, starting at a[0]
- This is known as *unordered list search*

C program

```
#include <stdio.h>
#define MAXSIZE 5

int searchList(int list[], int, int); / / function prototype

typedef enum {false, true} bool;
```

Searching an unordered list

```
int main(void)
{
   /* declarations */
   int limit = 0, i, number, x, list[MAXSIZE];

   /*read in list */

   scanf("%d", &number);
   while((limit < MAXSIZE) && number != -1) { // end the input
list with a -1;
            list[limit++] = number;
            scanf("%d", &number);
    }
```

Searching an unordered list

```
/* read in number to search for */
    printf("Input the number to search for: \n");
    scanf("%d", &x);

    /* search list */

    i = searchList(list, limit, x);
    if(i > 0) printf("Element has been found at position %d\n", i);
    else printf("Element is not in list\n");
}
```

Searching an unordered list

```
int searchList(int list[], int lim, int x)
{
   int i =0;
   bool found = false;

   while (i <lim && !found)
      if (list[i] == x) found = true;
      else ++i;

   if (!found) return -1;
   else return i;
}
```

How good is the algorithm ?

- We might have to scan all the elements of the list
- In algorithm parlance, we say it is an O(n) time algorithm, where n is the list size
- Can we do better ?

What if we know more ?

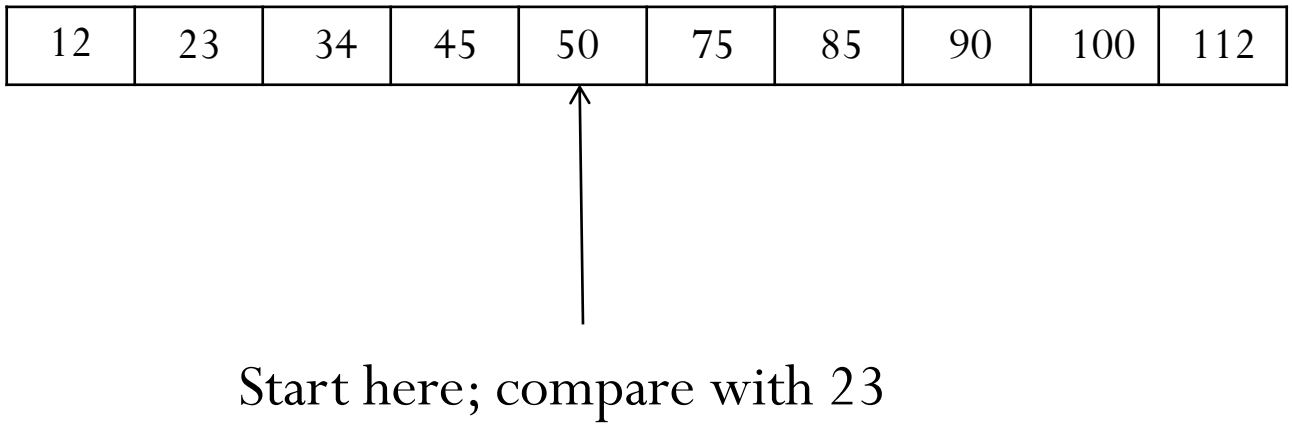
- The numbers in the list are arranged in order, say, increasing ?

12	23	34	45	50	75	85	90	100	112

Looking for 23 in this list

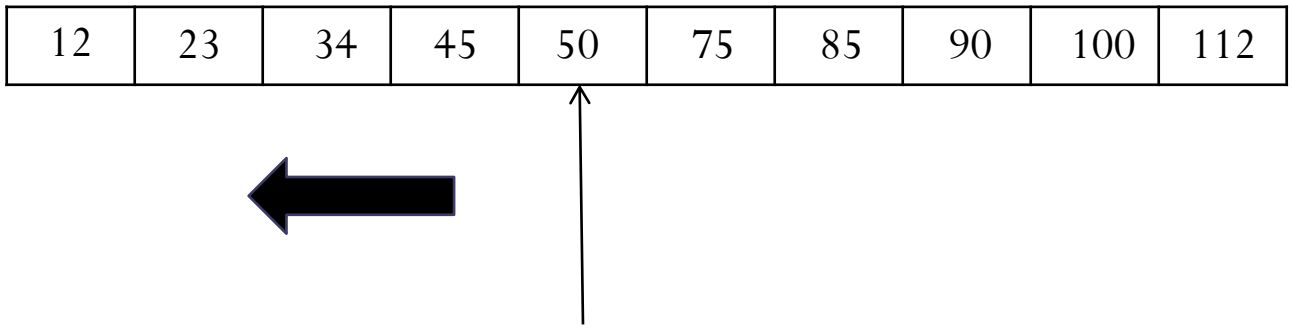
Halving (or binary) search ?

- Start from the middle



Halving search?

- Go left in search of 23
- Forget right half
- Continue recursively



Searching an ordered list

```
int binsearch(int x, int list[], int n)
{
   int low, high, mid;

   low = 0;
   high = n-1;
```

Searching an ordered list

```
while(low <= high){
        mid = (low+high)/2;
        if(x < list[mid])
          high = mid -1;
        else if (x >list[mid])
          low = mid +1;
        else /*found match */
          return mid;
    }
    return -1; /* no match */
}
```

Demo of binary search

- binarySearchDemo (Fig. 6.19 of textbook)

Sorting a list by comparisons

- List L : $a_0, a_2, \ldots, a_{n-1}$ is
  - from a *totally ordered* set
- A *comparison sort* orders *L* by pair-wise comparisons

Bubble Sort

- Algorithm:
  - Given an unsorted list of elements $a_0, a_1, \ldots, a_{n-1}$, the Bubble Sort algorithm makes n-1 passes, moving the largest element in the sub-list $a_0, a_1, \ldots, a_{i-1}$, to the i-th position, for $i = n-1$ down to 1
- Example

1, 8, 3, 7, 5

1, 3, 7, 5, 8

1, 3, 5, 7, 8

1, 3, 5, 7, 8

1, 3, 5, 7, 8

Bubble Sort

```
void bubbleSort(int list[], int lim)
{
   int i, j, temp;

   for( i = 0; i < lim; i++) // lim  passes
      for( j = 0; j < (lim-1)-i; j++)
       if ( list[j] > list[j+1])
        swap( list, j);
}
```

Bubble Sort

```
/* swap adjacent elements in the given list */
void swap(int list[], int i)
{
    int temp;

    temp=list[i];
    list[i] = list[i+1];
    list[i+1] = temp;
}
```

How good is bubble sort ?

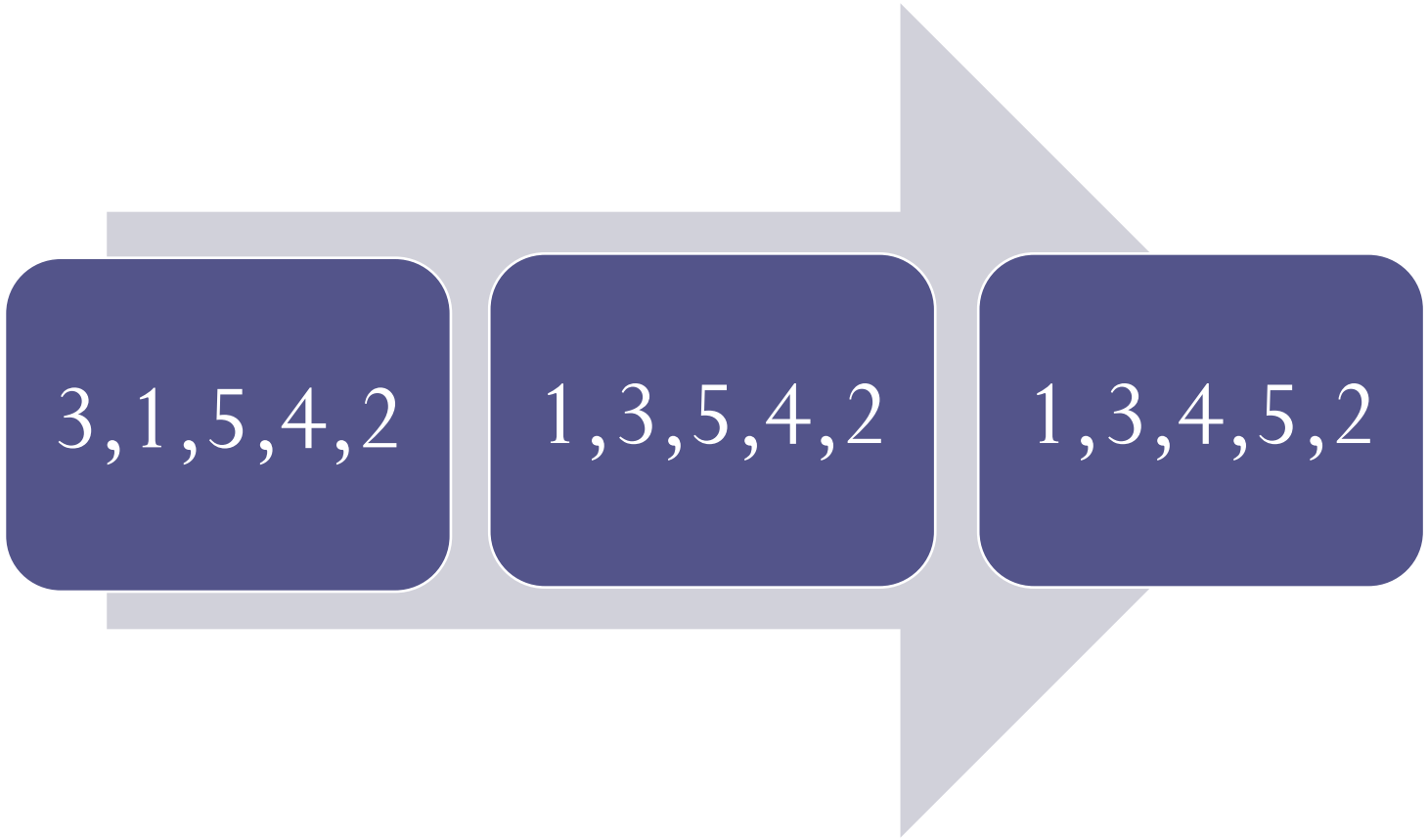
- The sort makes
  - *n*-1 comparisons in the *first* pass
  - *n*-2 comparisons in the *second* pass
  - …
  - *n*-*i* comparisons in the *i*-th pass
  - 1 comparison in the final pass,
- Thus the total numbers of comparisons is:

$$(n-1) + (n-2) + \ldots + (n-i) + \ldots + 1 = O(n^2)$$

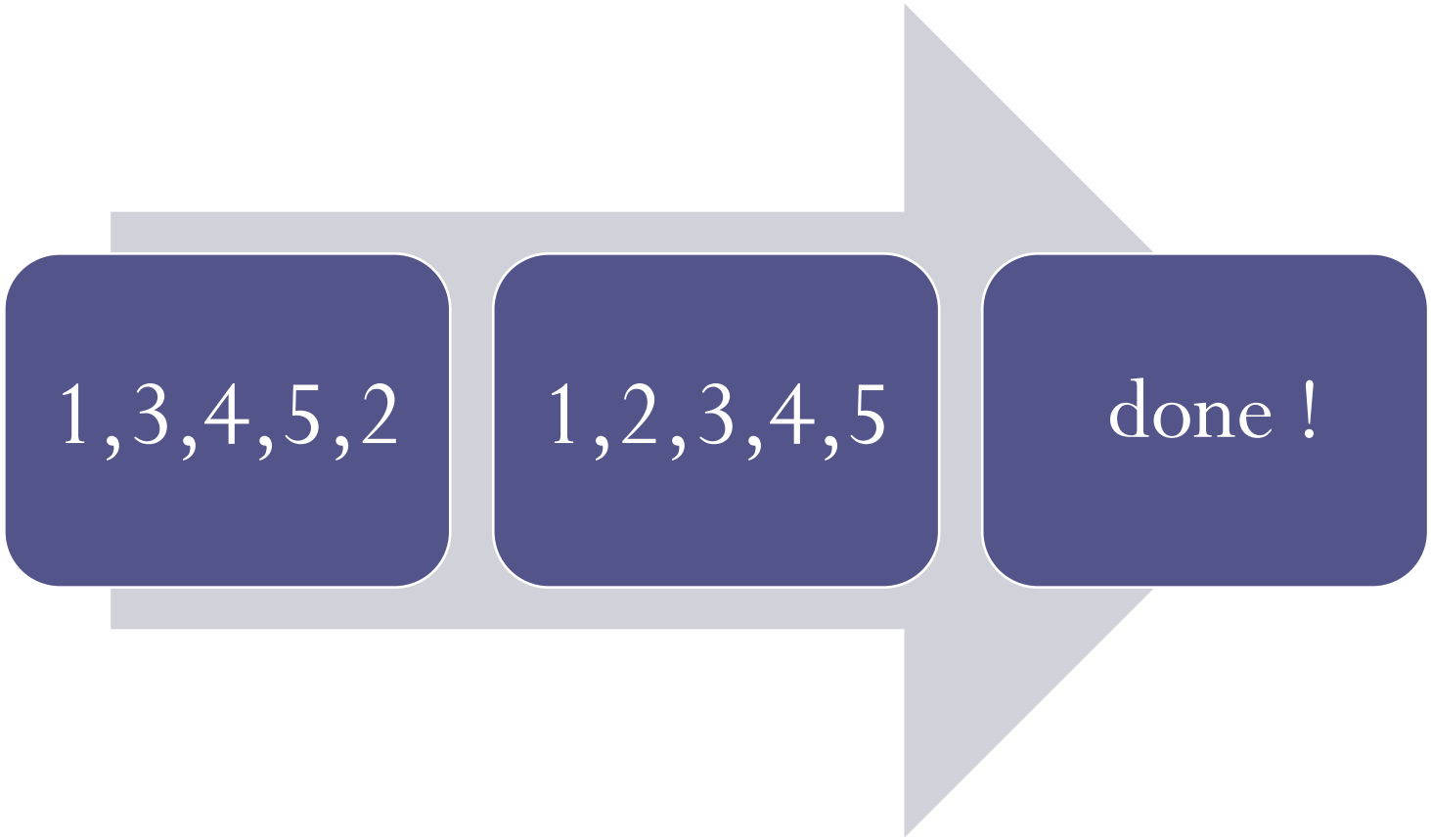
Insertion Sort

- Algorithm
  - From *left to right*, we put each element in the correct order relative to all its preceding elements
  - At the *i*-th step the first *i* elements
    - $a_1, a_2, \ldots, a_i$ are in sorted order
- Example List
  - 3, 1, 5, 4, 2

Example



Example



Pseudo code for Insertion Sort

Algorithm InsertionSort

Input: unsorted array $a[1..n]$, $n \geq 2$
Output: array $a[1..n]$, sorted in ascending order

Step 1. $i \leftarrow 2$;

Step 2. If $(i > n)$ go to Step 7;

Step 3. $temp \leftarrow a[i]$;
$j \leftarrow i$;

Step 4. while $(j > 1$ && $a[i-1] > temp)$
$a[j] \leftarrow a[j-1]$;
$j \leftarrow j - 1$;

Step 5. $a[j] \leftarrow temp$;

Step 6. $i \leftarrow i + 1$;
go to Step 2

Step 7. Output the array $a[1..n]$ and STOP.

How good is insertion sort ?

- The maximum number of comparisons needed to place the *i*-th element in place relative to the previous *i-1* elements is *at most* *i-1*, for $i = 2, 3, \ldots, n\text{-}1$
- Thus the total numbers of comparisons is *at most*:

$$(n\text{-}1) + (n\text{-}2) + \ldots + (n\text{-}i) + \ldots + 1 = O(n^2)$$

Are better algorithms possible ?

- Yes, but you will have to wait for other courses for this

Multidimensional array

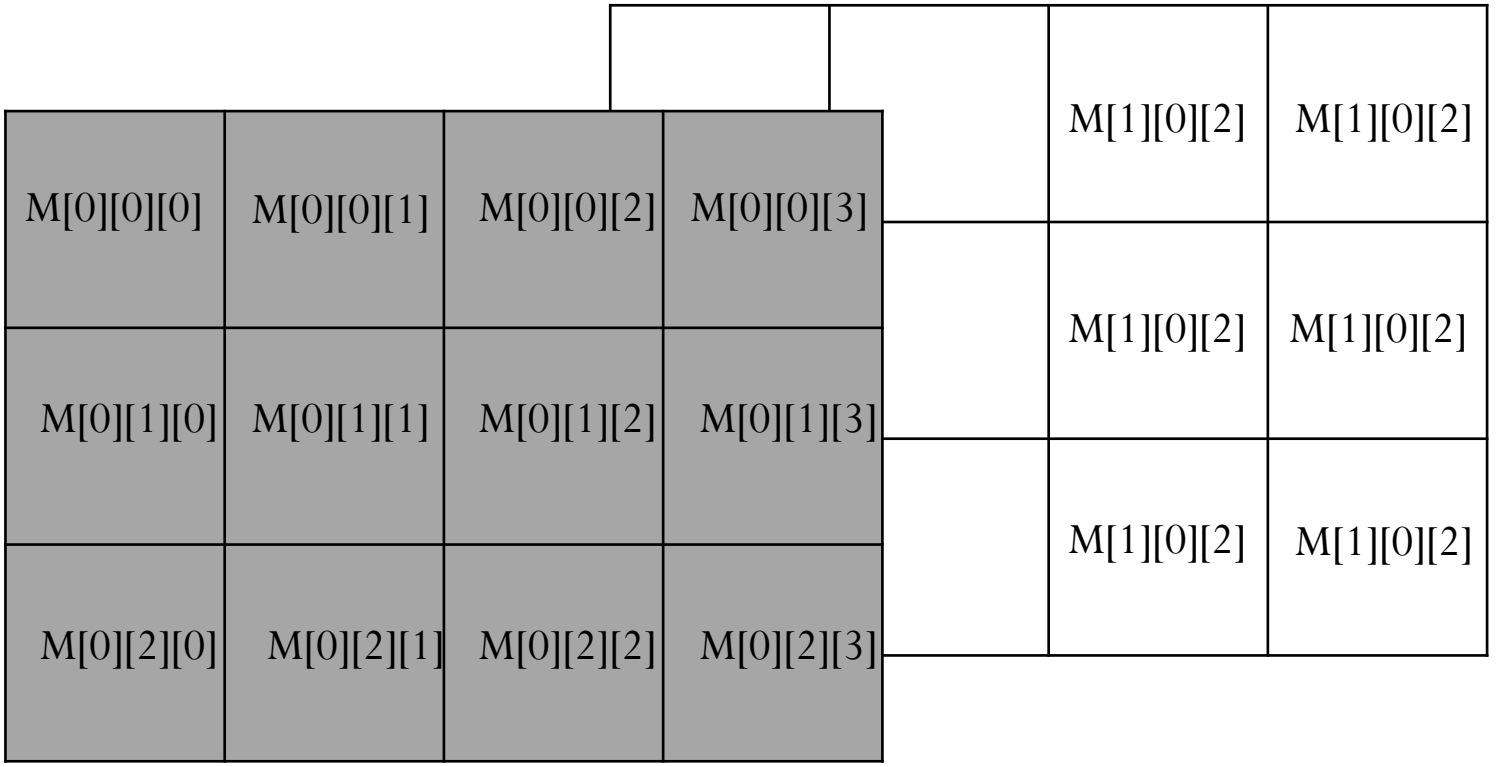
- It's conceivable that each array element a[i] is also an array
- Since all the a[i]'s must be of the same type the declaration `int A[5][6]` is interpreted to mean that we have an array of size 5, each an array of integers of size 6 .

A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	A[1][6]
A[2][1]	A[2][2]	A[2][3]	A[2][4]	A[2][5]	A[2][6]
A[3][1]	A[3][2]	A[3][3]	A[3][4]	A[3][5]	A[3][6]
A[4][1]	A[4][2]	A[4][3]	A[4][4]	A[4][5]	A[4][6]
A[5][1]	A[5][2]	A[5][3]	A[5][4]	A[5][5]	A[5][6]

A two-dimensional array

Multidimensional array

- Since, we can have an array of 2-dimensional arrays, M[2][3][4] is interpreted to mean an array of size 2, each array element a 2-dimensional array of size 3 X 4.
- This can be generalized to any number of dimensions



Multidimensional array manipulations

- Initializing a two-dimensional array

```
#include <stdio.h>
void printArray( int a[][ 3 ] ); // function prototype

int main( void )
{
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } }; // initialize array1, array2, array3
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   puts( "Values in array1 by row are:" );
   printArray( array1 );

   puts( "Values in array2 by row are:" );
   printArray( array2 );

   puts( "Values in array3 by row are:" );
   printArray( array3 );
} // end main
```

Two-dimensional array manipulations

```
// function to output array with two rows and three columns
void printArray( int a[][ 3 ] )
{
   size_t  i; // row counter
   size_t  j; // column counter

   // loop through rows
   for ( i = 0; i <= 1; ++i ) {

      // output column values
      for ( j = 0; j <= 2; ++j ) {
         printf( "%d ", a[ i ][ j ] );
      } // end inner for

      printf( "\n" ); // start new line of output
   } // end outer for
} // end function printArray
```

Two-dimensional array manipulations

- Demo of Fig. 6.22 of text

Passing a md-array as a parameter to a function

- Take a 2d-array for example
  - The parameter declaration must include the number of columns
  - Thus

```
f(int m[5][6]){   }   or
f(int m[][6]){   }    or
f(int (*m([6]){   }
```

  - More generally, only the first dimension is free, all others have to be specified