Topic 5. Arrays in C

COMP ENG 2SH4

Principles of Programming

McMaster University, 2015

Instructor: Sorina Dumitrescu

Required Textbook Reading

- Chapter **6**
- Sections: **1-5, 7** – one dimensional arrays
- Section **9, 10** – multidimensional arrays
  Section **8** – searching arrays – covered later

Arrays

- An array is a group of variables
  - of the **same type**,
  - related by the **same name**.
- They are assigned in memory **consecutive locations**.
    - The memory is organized as a sequence of bytes (1 byte = 8 bits)
    - A **memory location** for a variable consists of a number of consecutive bytes as specified by the variable type.
- To refer to an individual element of the array we use the **array name** and a **subscript** (or **index**) representing the position of the element in the array.
- We can use arrays when we need to store and process data which is organized as a **sequence of items**.

Array Definition

- Before using the array we need to **define** it.
- Array **definition**:

  data_type ***array_name*** [***array_size***]

  data_type is the type of elements to be stored in the array

  array_size is the number of elements to be stored in the array
- Effect of definition:
  - A number of consecutive memory locations specified by ***array_size*** are reserved**.**
  - Each location is of appropriate size to hold a variable of type ***data_type.***
- When arrays are defined they are not automatically initialized unless they are static.

Array Definition Example

- Before using the array we need to **define** it.
- Array **definition**:

  data_type ***array_name*** [***array_size***]

  data_type is the type of elements to be stored in the array

  array_size is the number of elements to be stored in the array

- **double** ***aa***[6]; /* defines an array to hold 6 elements of type
  double; the array name is ***aa*** */

  int x[12], y[20]; // defines array **x** to hold 12 **ints**, and array **y** to
  //hold 20 **ints**

  int b[4], j; // defines array b of 4 **ints**, and variable j of type **int**

Arrays

array_size in array definition

- constant expression (C89)
- integral type
- value > 0.

```
int main(void){
    int n=14;
    int a[n];  //syntax  error in C89.  Allowed in C99 and C11
    ……
}
```

How to Change the Array Size in Definition

- In C89 the size of the array in the definition cannot be a variable.
- Use a **symbolic constant** to specify the size of an array
- Use **#define** directive
- To change the size of array change the value specified with define and compile the code.

```
#include <stdio.h>
#define SIZE 10 /* during preprocessing any
      occurrence of SIZE (which is not in a
      string literal) is replaced by 10 */
int main(void){
      int b[SIZE]; … /* some more code */
      return 0;
      }
```

Referring to an Array Element

- Each element of the array can be referred to as:

  ***array_name* [*subscript*]**

- ***subscript*** (or **index**)
  - indicates the position of the element in the array
  - an integer or an integer expression
  - its value should be >= 0 and <= ***array_size*** -1!!!
- Example: **double** a[6];
- Defines an array with 6 elements of type double.
- The elements of the array are referred to as

  a[0], a[1], a[2], a[3], a[4], a[5]

Arrays

- ***array_name*[*subscript*]** is an lvalue (can be used on the left side of an assignment).
- ***array_name*** without any subscript **is not an lvalue**.

- **double** a[2], b[2];

  b[0]=6.0; b[1]=-4.5; b[1]++;
- a = b;   /* syntax error */
- a = { 5.7, 8.9} ;   /* syntax error */

Initializing an Array in a Definition with an Initializer List

```
int c[ 6 ] = {100, 60, 30, 90, 100, 20};
     memory is allocated for an integer array
     with 6 positions; the array elements are
     assigned the values within braces
     correspondingly
```

```
int c[ 6 ] = {100, 60, 80, 40};
     the last two elements of the array will be
     automatically initialized to 0
```

```
int c[   ] = {100, 60, 80, 40};
     The size is not specified within brackets,
     then the number of initializers will
     specify the size
```

Initializing Array Elements with a **for** Loop

```
int c[ SIZE ];
for( i=0; i < = SIZE - 1; i++ )
     c[ i ] = 100;
/* Array c is defined. Its elements are
     initialized to 100. */
```

```
int c[ SIZE ];
for( i=0; i < SIZE; i++ )
     c[ i ] = 100;
// the same effect as above
```

Pay attention to the loop continuation condition!

Examples of Syntax Errors in Initialization

```
int c[ 6 ] = {100, 60, 30, 90, 100,
    20, 90};
    syntax error (more initializers
    than array elements)
```

```
int c[ 5 ] = {100, 60, 30, 90, 100}; //correct
int d[ 5 ] = c;  //syntax error
```

C Has No Array Boundary Checking !

If we refer to an element outside the boundary → ERROR not detected by the compiler

```
int c[ 6 ] = {100, 60, 30, 90, 100, 20};
int a;
a = c[ 6 ];
               program may crash or produce
               incorrect results
```

```
int c[ 6 ] = { 0 };
c[ 10 ] = 10;
               program may crash or produce
               incorrect results
```

C Has No Array Boundary Checking !

- If we refer to an element outside the boundary
  → ERROR not detected by the compiler
- Pay attention to the **loop continuation condition**!

```
int c[ SIZE ];
for( i=0; i < = SIZE; i++ )

     c[ i ] = 100;
```

ERROR

```
int c[ SIZE ];
for( i=0; i < SIZE; i++ )

     c[ i ] = 100;
```

CORRECT

Arrays and Functions

- A function **cannot return** an array.
- A function can have an array as a parameter. In this case the **array name** is **passed** to the function.
- If we need to write a function to process an array, it is sufficient to pass to the function the **array name** and its **size**.
- If a function is passed an array name, then the function is able to **access** and **modify** all array elements in the caller!!

Passing an Array Name to a Function

```
#include <stdio.h>
void reverse( double x_array[], int n); //prototype
int main(void)
{   double my_array[6] = {1.1,3.3,9.7,8.9,4.3,2.0};
    int i;
    reverse( my_array, 6);
    for(i=0; i<6; i++)
      printf( “a[%d]=%.1f\n”, i, my_array[i]);
    return 0;
}
```

When passing the array name to a function

❑The called function has access to the array elements in the caller.

❑The called function can modify the array elements in the caller.

Passing an Array Name to a Function

```
void reverse( double x_array[], int n) {
    int i; double temp;
    for(i=0; i < n/2; i++){
      temp=x_array[i];
      x_array[i]=x_array[n-1-i];
      x_array[n-1-i]=temp;
    }//end for
}//end of function
```

When passing the array name to a function

❑The called function has access to the array elements in the caller.

❑The called function can modify the array elements in the caller.

Function reverse. Variant 2

```
void reverse( double x[], int n) {
    int i,j; double temp;
    /* use i to scan the array from left to
    right; use j to scan the array from right
    to left; swap x[i] and x[j] */
    j=n-1;
    for(i=0; ?? ; i++){
      temp=x[i];
      x[i]=x[j];
      x[j]=temp;
      j--;
    }//end for
}//end of function
```

Function reverse. Variant 2

```
void reverse( double x[], int n) {
    int i,j; double temp;
    /* use i to scan the array from left to
    right; use j to scan the array from right
    to left; swap x[i] and x[j] */
    j=n-1;
    for(i=0; i<j ; i++){
      temp=x[i];
      x[i]=x[j];
      x[j]=temp;
      j--;
    }//end for
}//end of function
```

const Qualifier

- ❑ If we do not want to allow a function to change the elements of the array which is passed to the function, declare the array parameter with **const**

```
#include <stdio.h>

int sum(const int x[], int size); // prototype

int main(void)
{ int a[4] = {1,2,3,4};
  printf(“The sum of array elements is %d\n”,sum(a,4));
  return 0;
}
```

const Qualifier

- If we do not want to allow a function to change the elements of the array which is passed to the function, declare the array parameter with **const**

```
int sum( const int x[], int size) {
 /* function sum can access the array elements in
     the caller, but it is not allowed to modify
     them  */
     int i, temp=0;
     for(i=0; i < size; i++)
       temp += x[i];
     return temp;
} // end of function
```

Automatic vs. Static Local Arrays

- A local array is an array defined inside a function body
- Its name can be used only in the function body
- It has **automatic storage** duration by default:
  - it is created when the function begins execution and
  - it is destroyed when the function is exited.
- If we define a local array using **static** (C99 and C11)
  - it acquires **static storage duration**; it is created before program start up and exists in memory until program ends
  - however, its name can be used only inside the function body
- The elements of a **static** array are automatically initialized to 0 when the array is defined, unless otherwise specified.

Double-Subscripted Arrays

- **Double-subscripted** arrays (or **two-dimensional**, or **2D** arrays) are used to represent tables of values arranged in **rows** and **columns**.
- ***array_name*** [*i*] [*j*] -- the element on row ***i*** and column ***j***
- Array definition:

data_type array_name [***rows_num***] [***columns_num***]

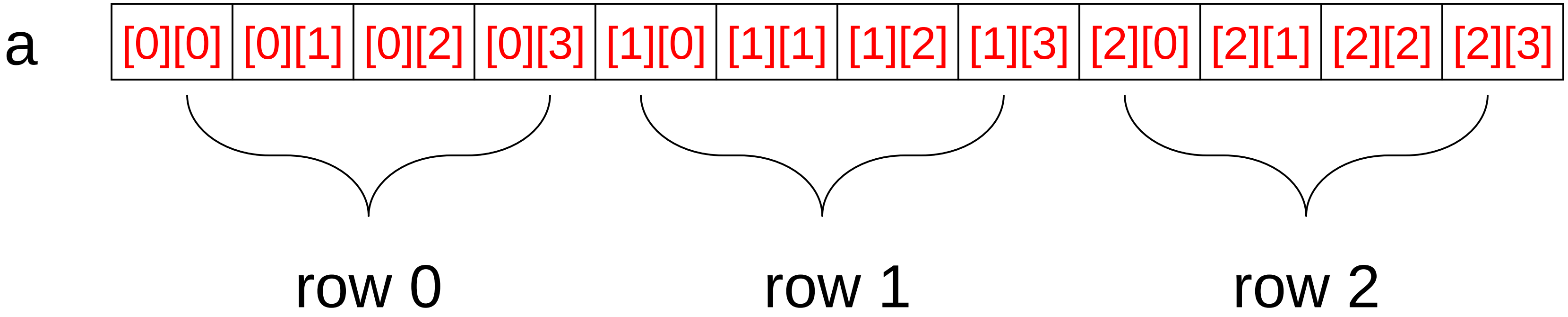
Ex: **int** a[3][4]; **// defines a 3-by-4 array**

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]

In Memory

- Elements in a 2D array are stored consecutively in memory, **row by row**.

```
int a[3][4];
```



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

row 0

row 1

row 2

Definition and Initialization

```
int c[3][4];
for( i=0; i<3; i++ )
   for( j=0; j<4; j++)
     c[i][j] = i+j;
```

```
int c[3][4] = {10,10,10,10,20,20,20,20,30,30,30,30};
```

```
int c[3][4] = {{10,10,10,10},{20,20,20,20},{30,30,30,30}};
```

```
int c[3][4] = {{0,10},{20,20,20,20},{30,30,30}};
```

```
int c[3][4] = {{10},{20,20}};
```

```
int c[3][4] = {{10},{},{30,30}};
```

```
int c[ ][4] = {10,10,10,10,20,20,20,20,30,30,30,30};
```

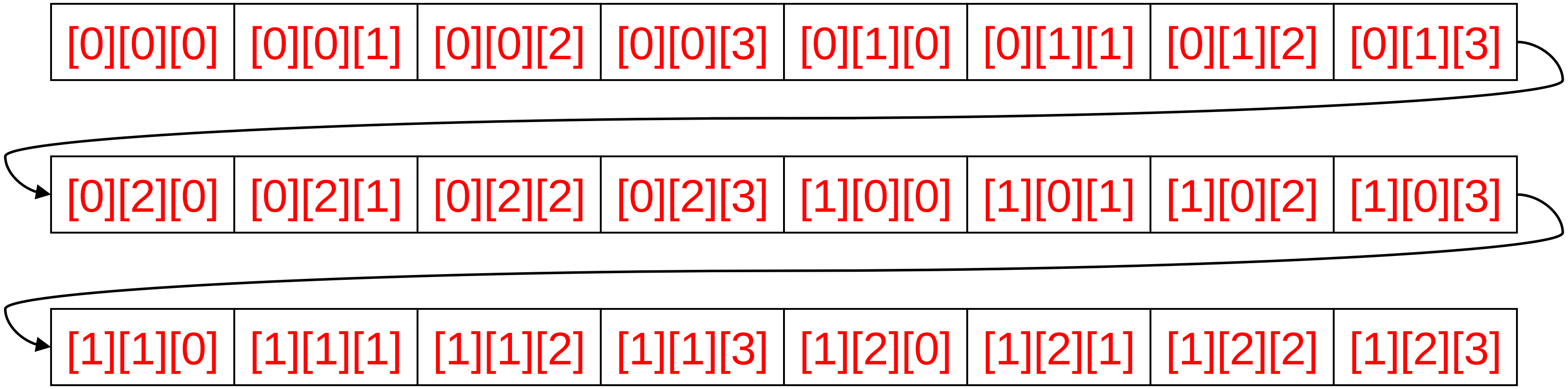
```
int c[ ][4] = {{0,10},{},{30,30,30}};
```

Multiple Subscripted Arrays

- Definition of an **N-dimensional** array (an array with N subscripts):

```
data_type variable_name[int_exp1][int_exp2]...[int_expN];
```

- Elements in an N dimensional array are stored consecutively in memory.
- The later dimension changes faster than earlier dimension.
- `int a[2][3][4]; /* array definition */`



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

Matrix Transpose

```c
/* the function modifies N-by-N matrix mat such that the
new matrix represents the transpose of the old one */
//incomplete code
#define N 10
void transpose(double mat[][N])
{




} // end function
```

- When passing a multi-D array to a function (passing the name of the array), the function is able to access and modify the array elements in the caller.
- In the parameter list the size of the first dimension is not required, but all others are required (have to be constant expressions in C89).

Matrix Transpose

```
/* the function modifies N-by-N matrix mat such that the
new matrix represents the transpose of the old one */
//incomplete code
#define N 10
void transpose(double mat[][N])
{
   int i,j;
   for(i=?;i<?;i++){
     for(j=?;j<?;j++){
         //swap mat[i][j] and mat[j][i]


     } //end for j
   } //end for i
} // end function
```

Matrix Transpose

```
/* the function modifies N-by-N matrix mat such that the
new matrix represents the transpose of the old one */
//incomplete code
#define N 10
void transpose(double mat[][N])
{
   int i,j; double temp;
   for(i=?;i<?;i++){
      for(j=?;j<?;j++){
         //swap mat[i][j] and mat[j][i]
         temp = mat[i][j];
         mat[i][j] = mat[j][i];
         mat[j][i] = temp;
      } //end for j
   } //end for i
} // end function
```

Matrix Transpose

```
/* the function modifies N-by-N matrix mat such that the
new matrix represents the transpose of the old one */
//incomplete code
#define N 10
void transpose(double mat[][N])
{
   int i,j; double temp;
   for(i=0;i<N;i++){
     for(j=0;j<i;j++){
        //swap mat[i][j] and mat[j][i]
        temp = mat[i][j];
        mat[i][j] = mat[j][i];
        mat[j][i] = temp;
     } //end for j
   } //end for i
} // end function
```