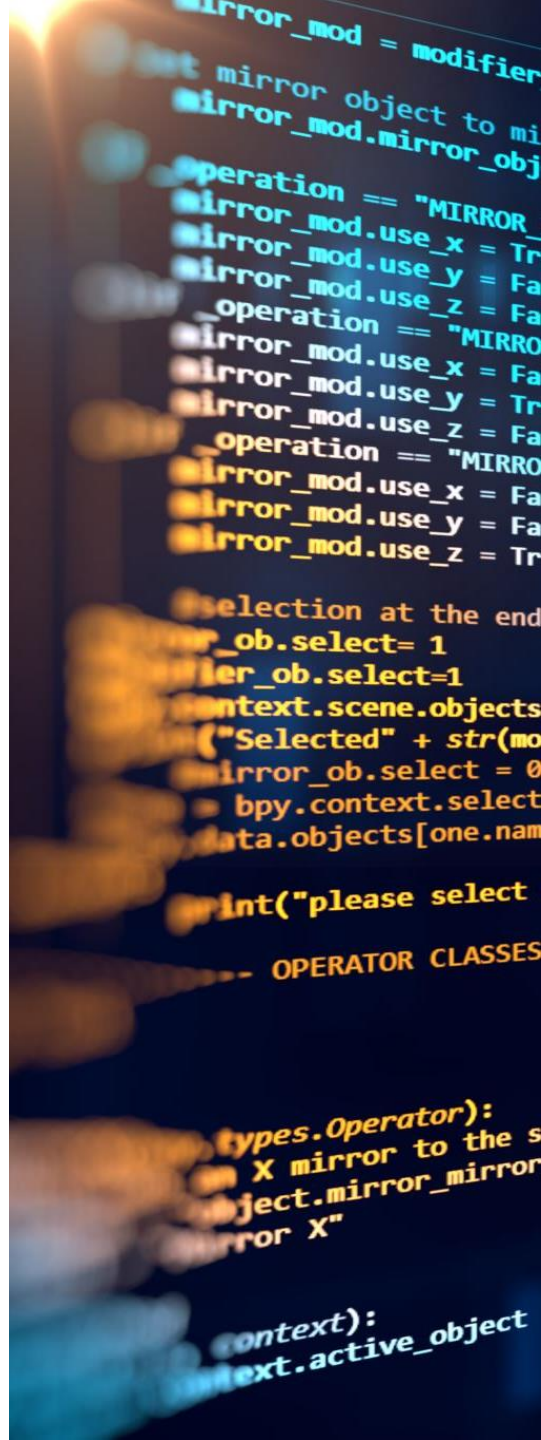
CS 1037
Fundamentals of Computer Science II

C Programming Features

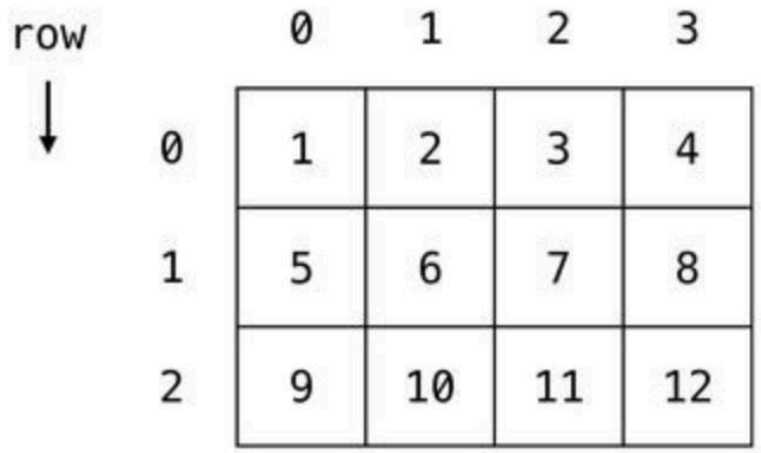
Ahmed Ibrahim



Two-dimensional Arrays

- A two-dimensional array (2D array) organizes data elements in rows and columns, each row having the same number of elements.
- Each element is located in both a row and a column.
- Each column has the same number of elements.
- 2D arrays are often used to represent matrices and rectangular tables.

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```



Syntax of 2D arrays

- A two-dimensional array is declared using the syntax:

  data_type **array_name**[row][col];

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

- This declares a 2D array with **row** * **col** elements organized in row rows and col columns.
- An element at row **i** and column **j** is represented using two subscripts **i** and **j** as **array_name**[**i**][**j**], where 0 <= i <= row-1 and 0 <= j <= col-1.
- **i** and **j** are called row index and column index, respectively.
- The compiler allocates a memory block of size **row** * **col** * sizeof(data_type) for the 2D array.

2D Array Declaration

```
2  // this declares an int type 2D array of 2 rows and 3 columns.
3  int a[2][3];
4
5  // this sets value 2 to element at row 1 and column 2
6  a[1][2] = 2;
7
8  // this gets the value of element at row 1 and column 2
9  int b = a[1][2];
```

2D array rows and columns.

rows/columns	column 0	column 1	column 2
row 0	a[0][0]	a[0][1]	a[0][2]
row 1	a[1][0]	a[1][1]	a[1][2]

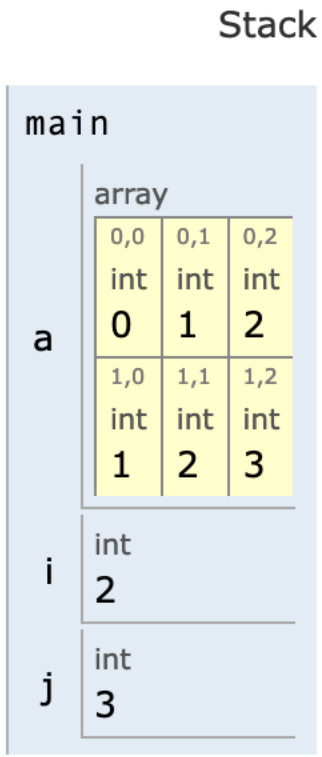
2D Array Traversal

- An example of a 2D array traversal in row-major order

```c
#include <stdio.h>
int main()
{
  int a[2][3], i, j;
  for (i=0; i<2; i++) {
    for (j=0; j<3; j++) {
      a[i][j] = i+j;
      printf("a[%d][%d]: %d, address: %lu\n", i, j, a[i][j],
      &a[i][j]);
    }
  }
  return 0;
}
```

Program Output:

```
a[0][0]: 0, address: 68702702528
a[0][1]: 1, address: 68702702532
a[0][2]: 2, address: 68702702536
a[1][0]: 1, address: 68702702540
a[1][1]: 2, address: 68702702544
a[1][2]: 3, address: 68702702548
```



Memory

Row-Major Order

- Row-major order is a method for storing 2D arrays in linear memory.
- In this ordering, the elements of each row of a 2D array are stored in contiguous memory locations, one after the other.

```
int a[2][3] = {
    {0, 1, 2},
    {1, 2, 3}
};
```

- Row-major order stores the array in memory as:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

- In this method, all elements of the first row (a[0]) are stored first, followed by elements of the second row (a[1]), and so on.

2D Array Initialization

- A 2D array can be initialized similarly to that of a 1-dimensional array.
- Example:

```
// This initializes 2D elements in row-major linear order
int a[2][3]={90, 87, 78, 68, 62, 71};
```

- If the number of items on the right side is less than that of the 2D array, the rest will be set to 0.
- A 2D array can also be initialized in a row-by-row manner as the following:

```
// row 0: {90,87,78}, row 1: {68, 62, 71}
int a[2][3]={{90,87,78},{68, 62, 71}};
```

Using Arrays in Applications

Arrays

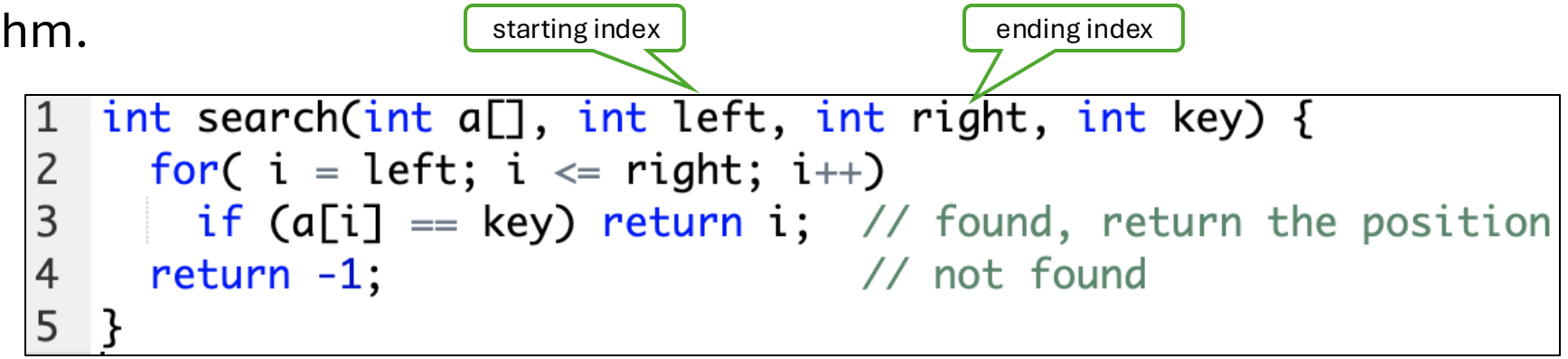
- Arrays are declared with a data type and **maximum length**.
- Suitable for storing collections of application data records.

Basic Array Operations

- **Traverse** – Visit every data record in the array
- **Insert** – Add a new data record
- **Delete** – Remove a data record
- **Search** – Find a data record by key value
- **Sort** – Arrange data in ascending or descending order

Searching 1D Array

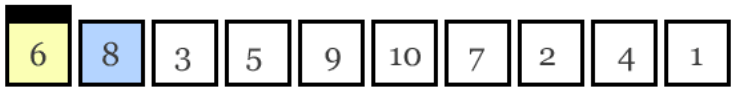
- Searching by a key means finding the position of an element that matches the key value. It returns the **position** if found; otherwise returns -1.
- A simple searching algorithm is to traverse the array with key checking and return when found (Linear Searching).
- The following function is a simple implementation of the simple searching algorithm.



Sorting 1D Array Elements

```
3  // Function to perform Selection Sort
4  void selectionSort(int array[], int n) {
5      int i, j, minIndex, temp;
6
7      // Traverse the array
8      for (i = 0; i < n - 1; i++) {
9          // Assume the minimum element is at index i
10         minIndex = i;
11
12         // Find the minimum element in the unsorted part
13         for (j = i + 1; j < n; j++) {
14             if (array[j] < array[minIndex]) {
15                 minIndex = j;
16             }
17         }
18
19         // Swap the found minimum element with
20         // the element at index i
21         if (minIndex != i) {
22             temp = array[i];
23             array[i] = array[minIndex];
24             array[minIndex] = temp;
25         }
26     }
27 }
```

- **Selection sort**: Repeatedly selects the smallest (or largest) element from the unsorted part of the array and swaps it with the first unsorted element.
- The average **time complexity** for the selection sort is n^2 operations.
- Use Case: Suitable for small datasets or when memory space is limited.



Yellow is smallest number found
Blue is current item
Green is sorted list

Selection Sort Animation, thanks to Xybernetics for the gif

Question!

Given:

```
float mat[2][3];
```

How many **bytes** are allocated for the 2D array mat?

A) 6

B) 12

C) 24

D) 48

P.S. A float typically takes 4 bytes of memory.

Question!

Given:

```
float mat[2][3];
*p = &mat[0][0];
```

Which of the following pointer expressions correctly accesses the element mat[1][2]?

A) *(p+2)

B) *(p+3)

C) *(p+4)

D) *(p+5)


thank
you

References

- Data Structures Using C, second edition, by Reema Thareja, Oxford University Press, 2014.