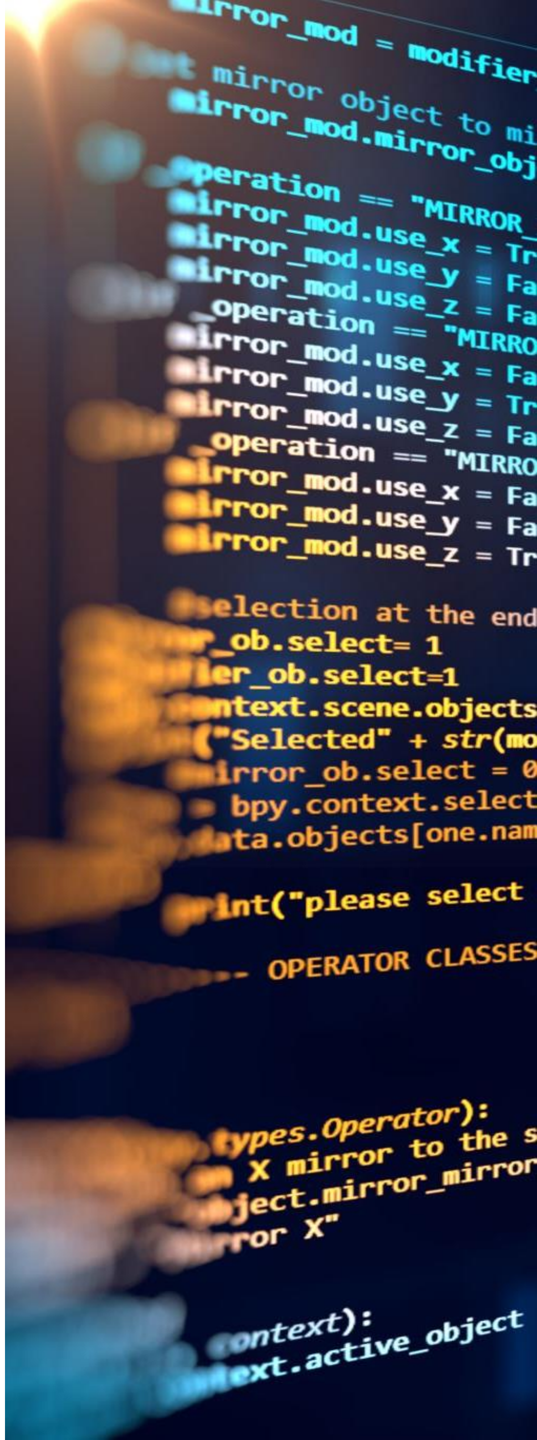
CS 1037
Fundamentals of Computer Science II

C Programming Features

Ahmed Ibrahim



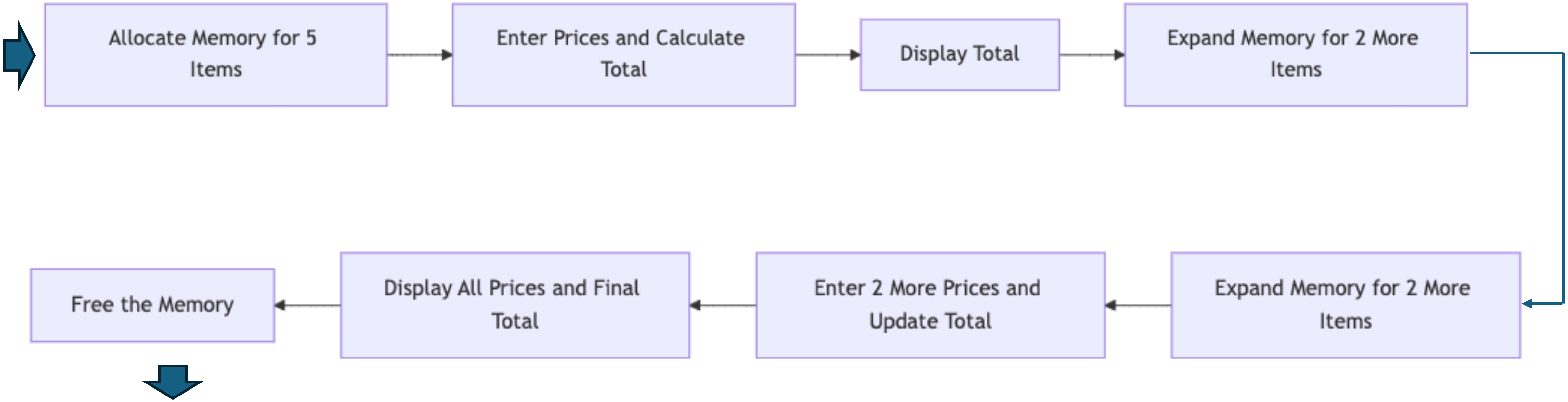
Problem: **Simple Inventory System**

A small electronics store needs a basic inventory system. Create a C program that:

1. Uses malloc() to allocate memory for 5 item prices (integers)
2. Inputs prices for 5 items from the user using scanf()
3. Calculates and displays the total inventory value
4. Uses realloc() to expand the system to accommodate 2 more items
5. Inputs prices for the 2 new items
6. Displays all 7 item prices using pointer arithmetic

Your solution should demonstrate the use of **pointers**, **dynamic memory** allocation (malloc and realloc), **pointer arithmetic**, and proper memory management (free).

Steps for Building a **Simple Inventory System**



Allocate Memory for 5 Items

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *inventory;
    int i;
    long long total = 0; // Use long long for larger sums

    // Seed the random number generator
    srand(time(NULL));

    // Allocate memory for 5 items
    inventory = (int*)malloc(5 * sizeof(int));
    if (inventory == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
return 0;
}
```

Enter Prices and Calculate the Total

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *inventory;
    int i;
    long long total = 0; // Use long long for larger sums

    // Seed the random number generator
    srand(time(NULL));

    // Allocate memory for 5 items
    inventory = (int*)malloc(5 * sizeof(int));
    if (inventory == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Assign random prices for 5 items
    printf("Random prices for 5 items:\n");
    for (i = 0; i < 5; i++) {
        // Generate random values between 1 and 100
        *(inventory + i) = rand() % 100 + 1;
        printf("Item %d: $%d\n", i + 1, *(inventory + i));
        total += *(inventory + i);
    }
return 0;
}
```

Display Total + Expand the Memory

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *inventory;
    int i;
    long long total = 0; // Use long long for larger sums

    // Seed the random number generator
    srand(time(NULL));

    // Allocate memory for 5 items
    inventory = (int*)malloc(5 * sizeof(int));
    if (inventory == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Assign random prices for 5 items
    printf("Random prices for 5 items:\n");
    for (i = 0; i < 5; i++) {
      // Generate random values between 1 and 100
        *(inventory + i) = rand() % 100 + 1;
        printf("Item %d: $%d\n", i + 1, *(inventory + i));
        total += *(inventory + i);
    }

      // Display total inventory value
    printf("Total inventory value: $%lld\n", total);

    // Expand inventory to 7 items using a temporary pointer
    int *temp = (int*)realloc(inventory, 7 * sizeof(int));
    if (temp == NULL) {
        printf("Memory reallocation failed.\n");
        free(inventory); // Free original memory
        return 1;
    }
return 0;
}
```

Data Structures

- A data structure is a collection of data items (also called elements or components) connected in certain structures and accessed by defined logic.
- When we study a data structure, we focus on its definition, how data items are represented, organized, and stored in memory, and how they can be accessed and operated.
- Data structures are used in **algorithms** and **programs** to represent and operate data.

Arrays

- Arrays are fundamental data structures to store a collection of data items (values, elements, records) of the same type in contiguous memory locations.
- Arrays are used to
  - store application data records.
  - store strings (text data).
  - implement other data structures like queues, stacks, hash tales, and heaps.

Concepts of Arrays

- An array is a collection of elements, all of the **same data type**, arranged in a specific **order**.
- Elements are accessed using **index positions** and stored in **contiguous memory locations**.

- **Array Declaration Syntax:**
- data_type array_name[**length**];
  - data_type: Type of elements
  - array_name: Name of the array
  - **length**: Number of elements

Memory Allocation:

- The compiler allocates **length** * sizeof(data_type) bytes for the array.
- sizeof(array_name) returns the total size in bytes.

Concepts of Arrays (cont.)

- Accessing array elements is straightforward using array_name[index], where index is an integer between 0 and the array length minus one.
- For instance, array_name[0] refers to the first element, and array_name[length-1] refers to the last element.
- Since array elements can be accessed directly using a constant number of instructions, **access time is constant**, making arrays highly efficient for **random access**.
- The array_name[index] notation is used to get or set values,
- while &array_name[index] provides the element's address.
- The following code fragment shows

the usage of the bracket notation:

```
float a[5];
a[0] = 10.1;
float b = a[0];
float *ptr = &a[1];
```

Array Initialization

```c
int a[5] = { 4, 1, 7, 2, 1 };
// this declares an int type array of length 5,
// and sets element values 4, 1, 7, 2, 1, respectively
// e.g. a[0] holds value 4, a[4] holds value 1

int a[5] = { 4, 1, 0 };
// this declares an int array of length 5,
// and sets the values 4, 1, 0 for the first three elements, and 0 for the rest elements.

int a[5] = { 0 };
// this declares an int array of length 5, and sets 0 to all elements.

int a[] = { 2, 6, 4 };
// this declares an int array of length 3, and sets element values in order of 2, 6, 4.
```

Question!

Given:

float **vector**[4]; the size of (in Bytes) and length of the vector array are

A. 4, 4

B. 4, 16

C. 16, 4

D. None of these

- The **vector** array is declared as float **vector**[4];.
- This means the **vector** has 4 elements, and each element is of type **float**.
- On most systems, a float occupies 4 bytes.
- Since the array has 4 elements, its total size in bytes is 4 elements × 4 bytes = 16 bytes.

Question!

Given:

float vector[4]; which element will have the **lowest memory address** at runtime?

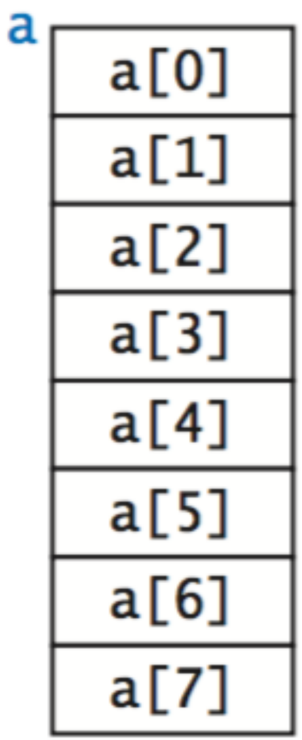
A. vector[0]

B. vector[1]

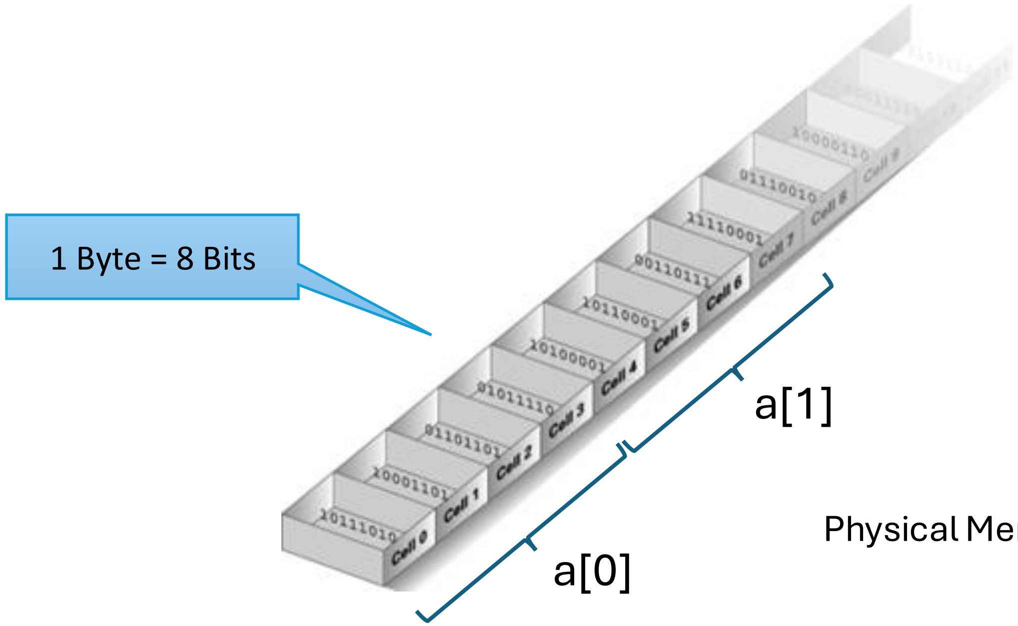
C. vector[2]

D. vector[3]

Recall: Arrays (cont.)



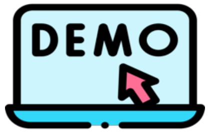
Array of Integers



Physical Memory Layout

Array Traversal

- Array traversal involves accessing or visiting every element of an array. When traversing an array, we can process array elements by printing values, getting addresses, and changing values.



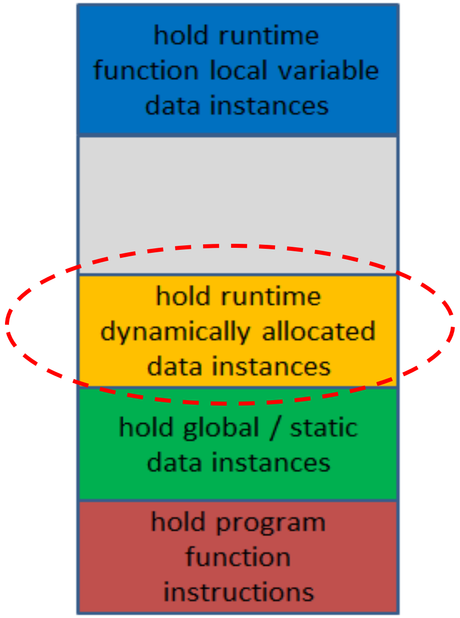

```c
int a[5] = { 4, 1, 7, 2, 1 };
int i;

for (i=0; i<5; i++)  printf("%d ", a[i]);
// this prints the array data in forward order: 4 1 7 2 1

for (i=4; i>=0; i--) printf("%d ", a[i]);
// this prints the array data in backward order: 1 2 7 1 4
```

- The array traversal algorithm, a key tool in **array manipulation**, uses a for **loop** with a subscript index that changes from 0 to length-1 for **forward** direction traversal, or from length-1 to 0 for **backward** direction traversal.

Dynamic Arrays



- Arrays can also be created using the dynamic method, namely the **malloc()** function, to allocate memory space for an array.
- **Dynamic arrays** have memory blocks in the heap region at runtime.

```c
#include<stdio.h>
#include<stdlib.h>
int main() {
  int length = 5;
  int *a = (int*) malloc(sizeof(int)*length);
  int i;
  for (i=0; i<length; i++)  {
    a[i] = i;
    printf("a[%d]:%d, address:%lu\n", i, a[i], &a[i]);
  }
  return 0;
}
```

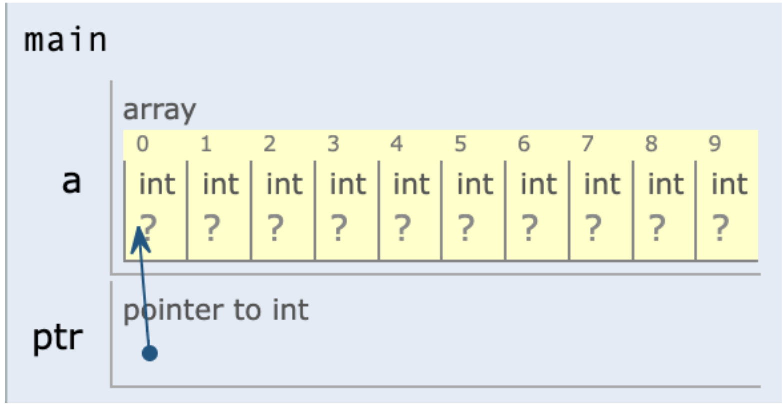
DEMO

Array Pointer

- An array can be viewed as a derived data type. An *array pointer* is a pointer pointing to an array.
- An array pointer can be declared by syntax: data_type (*ptr_name)[**k**];.
- Then ptr_name is a pointer to an array of type data_type of length **k**.
- For example:

  int (*ptr)[**10**];

- declares array pointer ptr, which points to an int array of length **10** elements.



Array Traversal by Pointers

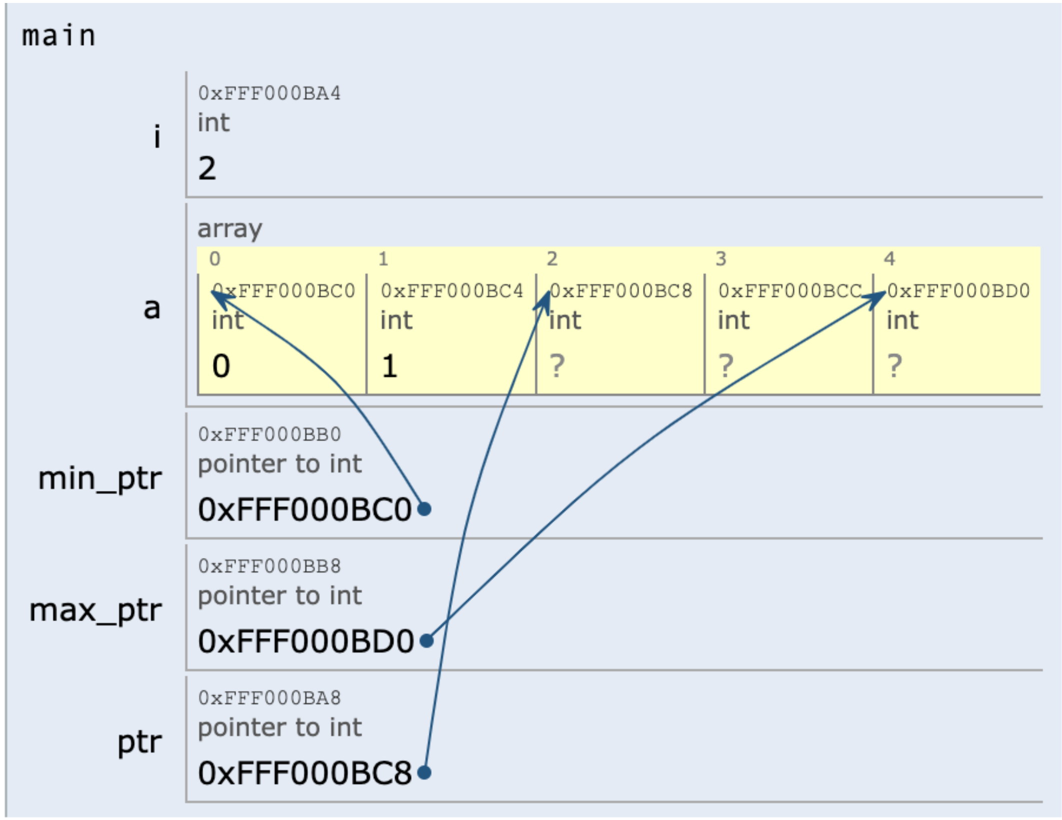
```c
#include <stdio.h>

int main()
{
    int i = 0;
    int a[5];
    int *min_ptr = &a[0], *max_ptr = &a[4], *ptr;

    // Initialize the array using pointers
    for (ptr = min_ptr; ptr <= max_ptr; ptr++) {
        *ptr = i;
        i++;
    }

    // Display the array values and addresses in reverse order
    for (ptr = max_ptr; ptr >= min_ptr; ptr--) {
        printf("value: %d, address: %lu\n", *ptr, ptr);
    }

    return 0;
}
```



DEMO

Question!

Given:

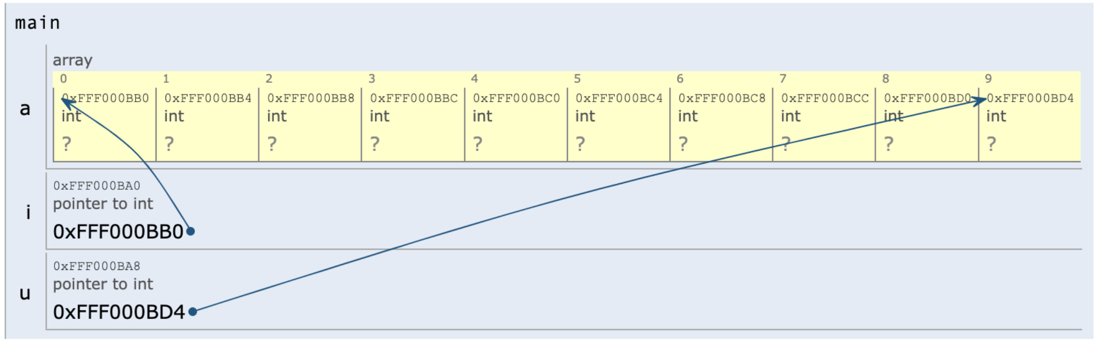
```c
#include <stdio.h>

void main()
{
 int a[10],*i,*u=&a[9];
 for(i=a; i < u; i++)
   *i=1;
}
```

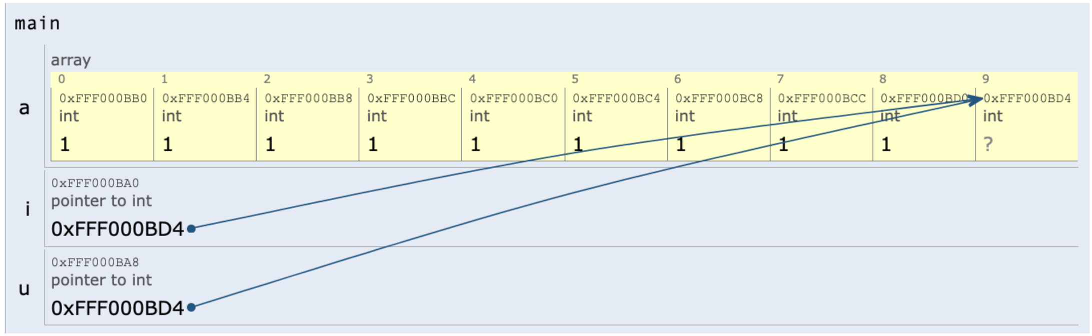
This sets value 1 to every element of array a.

A. True **B.** False

Start



End



Pointer Array

- A **pointer array** (also called an **array of pointers**) is an array of <u>pointer-type elements</u>.
- A pointer array is declared by syntax:

  data_type *ptr_array_name[**k**].

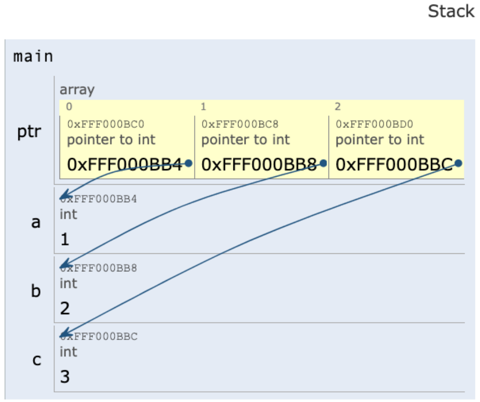
- Here, ptr_array_name is an array of pointers. The array element ptr_array_name[**index**] is a pointer pointing to a data_type variable. *ptr_array_name[**index**] gets the value it points to.

```c
#include <stdio.h>

int main() {
  int *ptr[3];
  int a=1, b=2, c=3;
  ptr[0]=&a;
  ptr[1]=&b;
  ptr[2]=&c;
  printf("%d", *ptr[2]); // output: 3

  return 0;
}
```

Memory:



Key Differences

- The table below highlights the differences between an **array pointer** and a **pointer array** in C programming.

	Array pointer	Pointer array
Aspect	int (*ptr)[size]	int *arr[size]
Declaration	int (*ptr)[5]	int *arr[5]
Meaning	ptr is a pointer to an array of 5 integers.	arr is an array of 5-pointers to integers.
Memory Layout	Points to a single array in memory.	Contains multiple pointers, each pointing to different memory locations.
Access	(*ptr)[i] accesses the array element.	arr[i] accesses the i-th pointer.
Use Case	When you need a pointer to an entire array.	When you need an array that holds multiple pointers.

Stop and think

How do you apply pointer arrays in applications?

Pointer Array (cont.)

- An auxiliary pointer array can represent the same type of data objects stored at different locations.
- Working on the pointer array prevents us from working directly on the data objects.
- This is useful when sorting data objects without changing their original locations, which can be time-consuming.
- We create a pointer array where each element points to a data object.
- Sorting is then performed on the pointer array using the pointed data objects for comparisons.
- The sorted pointer array provides the sorted information of the original data objects.

Passing arrays to functions

- When input data are stored in an array, and a data processing algorithm is implemented in a function, we must pass the array to the function. For example, we need to pass an array to a **sorting function**.
- There are three primary methods to pass an array to a function:
  - Passing an **individual** array **element** to a function is like **passing by value**; its value is copied, and it won't be changed by the function.
  - Passing an array **element by reference** copies its address, allowing the function to access and modify it.
  - **Passing the array name** to a function passes the array's address (the address of the first element), enabling access to all elements.

Passing arrays to functions (cont.)

```c
#include <stdio.h>
void f(int num);
void main()
{
  int a[5] = {1, 2, 3, 4, 5};
  f(a[3]);
}
void f(int num)
{
  printf("%d", num);
}
```

Passing an **individual** array **element**

```c
#include <stdio.h>
void f(int*);
void main()
{
  int a[5] = {1, 2, 3, 4, 5};
  f(&a[3]);
  printf(" %d", a[3]);
}
void f(int *num)
{
  printf("%d", *num);
  *num = 10;
}
```

Passing an array **element by reference**

```c
#include <stdio.h>
void f(int n, int arr[]);
void main()
{
  int a[5] = {1, 2, 3, 4, 5};
  f(5, a);  //  or f(5, &a[0]);
  printf("\n");
  int i;
  for(i=0; i<5; i++)
    printf("%d ", a[i]);

}
void f(int n, int arr[])
{
  int i;
  for(i=0; i<n; i++)
    printf("%d  ", arr[i]);

  for(i=0;i<n;i++)
    arr[i] = arr[i]*arr[i];
}
```

Passing the array name

```c
#include <stdio.h>
void f(int n, int arr[]);
void main()
{
  int a[5] = {1, 2, 3, 4, 5};
  f(5, a);  //  or f(5, &a[0]);
  printf("\n");
  int i;
  for(i=0; i<5; i++)
    printf("%d ", a[i]);

}
void f(int n, int arr[])
{
  int i;
  for(i=0; i<n; i++)
    printf("%d  ", arr[i]);

  for(i=0;i<n;i++)
    arr[i] = arr[i]*arr[i];
}
```

```c
#include <stdio.h>
void f(int, int *);
void main()
{
  int a[5] = {1, 2, 3, 4, 5};
  f(5, a);  //  or f(5, &a[0]);
  printf("\n");
  int i;
  for(i=0; i<5; i++)
    printf("%d ", a[i]);

}
void f(int n, int *p)
{
   int i;
   for(i=0; i<n; i++)
     printf("%d  ", *(p+i));

   for(i=0;i<n;i++)
     *(p+i) = *(p+i)* *(p+i);
}
```

Summary

- You learned that arrays are fundamental data structures to store a collection of data objects of the same type.
- Array elements are stored in contiguous memory space.
- Array elements can be accessed efficiently using the position index.
- Arrays can be operated efficiently by pointers.

Two-dimensional Arrays

Introduction

- The arrays learned so far are one-dimensional (1D), having only one subscript index.
- Here, we will introduce two-dimensional arrays with 2 subscript indexes.

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}
};
```

col →

row ↓

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	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 <u>compiler allocates a memory block of size</u> **row** * **col** * sizeof(data_type) for the 2D array.

2D Array Declaration

```
// this declares an int type 2D array of 2 rows and 3 columns.
int a[2][3];

// this sets value 2 to element at row 1 and column 2
a[1][2] = 2;

// this gets the value of element at row 1 and column 2
int b = a[1][2];
```

2D array rows and columns.

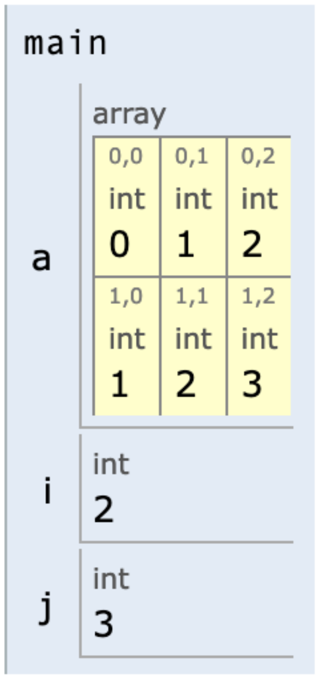
rows/columns	column 0	column 1	column 2
row 0	a[0][0]	a[0][1]	a[0][2]
row 1	a[1][0]	a[1][1]	a[1][2]

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;
}
```

Stack



Memory

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
```

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.

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

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

```c
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.

starting index

ending index

```
int search(int a[], int left, int right, int key) {
  for( i = left; i <= right; i++)
    if (a[i] == key) return i;  // found, return the position
  return -1;                    // not found
}
```

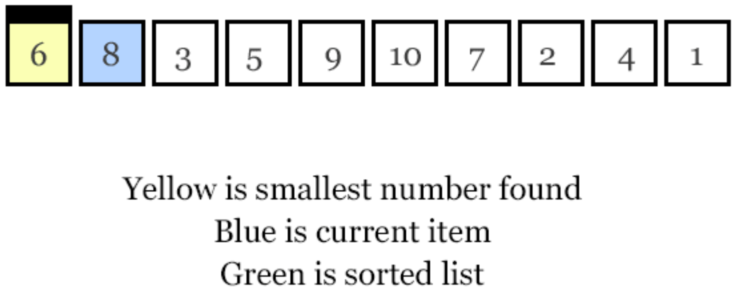
Sorting 2D Array Elements

```c
// Function to perform Selection Sort
void selectionSort(int array[], int n) {
    int i, j, minIndex, temp;

    // Traverse the array
    for (i = 0; i < n - 1; i++) {
        // Assume the minimum element is at index i
        minIndex = i;

        // Find the minimum element in the unsorted part
        for (j = i + 1; j < n; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the found minimum element with
        // the element at index i
        if (minIndex != i) {
            temp = array[i];
            array[i] = array[minIndex];
            array[minIndex] = temp;
        }
    }
}
```



Selection Sort Animation, thanks to [Xybernetics] for the gif

C Advances

Strings

- Almost all messages or text information we see on computer screens are represented internally in a computer as strings.
- **What is a string?** a **string** is data consisting of a sequence of characters, e.g., "hello,world."
- In C program language, a **string** is a sequence of non-null characters followed by a null character, and stored in a char array.
- The **null character** is the character with ASCII code 0, represented by '\0'.
- The **length** of a string is defined as the number of non-null characters.
- The index of a character in a string is the position number from the beginning of the string.
- For example, the length of the string "hello,world" is 11. The index of letter h is 0, and the index of letter w is 6.

String Manipulation

- This code demonstrates how to create and manipulate a string in C using a character array.
- The array str is manually initialized with the characters 'H', 'e', 'l', 'l', 'o', followed by the null character '\0' to mark the end of the string.
- The for loop iterates through the array of elements to display each character’s index, value, and memory address.
- The condition if (str[i] == '\0') checks for the end of the string using the null character '\0'.

```c
#include <stdio.h>

int main() {
    char str[10] = "Hello";

    // output: Hello
    printf("%s\n", str);

    int i;
    for (i = 0; i < 10; i++) {
      // Correct null character check
        if (str[i] == '\0')
          break;
        else
          printf("index: %d, char: %c, code: %d, address: %lu\n",
            i, str[i], str[i], &str[i]);
    }
    return 0;
}
```

```
Hello
index: 0, char: H, code: 72, address: 68702702542
index: 1, char: e, code: 101, address: 68702702543
index: 2, char: l, code: 108, address: 68702702544
index: 3, char: l, code: 108, address: 68702702545
index: 4, char: o, code: 111, address: 68702702546
```

```c
#include <stdio.h>

int main() {
    char str[10];
    str[0] = 'H';
    str[1] = 'e';
    str[2] = 'l';
    str[3] = 'l';
    str[4] = 'o';
    str[5] = '\0';

    // output: Hello
    printf("%s\n", str);

    int i;
    for (i = 0; i < 10; i++) {
      // Correct null character check
        if (str[i] == '\0')
          break;
        else
          printf("index: %d, char: %c, code: %d, address: %lu\n",
            i, str[i], str[i], &str[i]);
    }
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    char str[10] = "Hello";

    // output: Hello
    printf("%s\n", str);

    int i;
    for (i = 0; i < 10; i++) {
      // Correct null character check
        if (str[i] == '\0')
          break;
        else
          printf("index: %d, char: %c, code: %d, address: %lu\n",
            i, str[i], str[i], &str[i]);
    }
    return 0;
}
```

Thank you

References

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