DATA STRUCTURES

Linear Array

هذا التشابتر يحتوي على :

Linear Array

2D Array

Pointers

Prof. Khalil Al-Shqeerat

CS Dept. – Qassim University

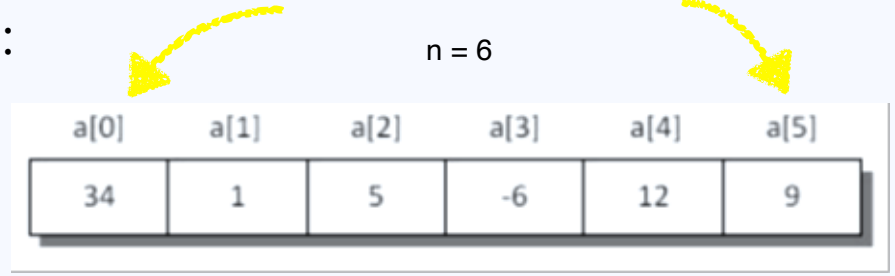
OUTLINE

1. Review
2. Operations on the array
3. Implementation of various array operations
4. Two-Dimensional Arrays
5. Row and Column Arrangement
6. Implementation of matrix operations

REVIEW!

Fixed data size Same data type

- An **Array** is a finite collection of similar elements stored in adjacent memory locations.
  - It contains ***n*** number of elements
    - It is referenced using an index that varies from 0 to n
    - An array ***arr[n]*** contains ***n*** elements
      - where **0** is the lower bound, and **n - 1** is the upper bound.
    - Example:



For example
arr[5] = {1, 2, 3, 4, 5}

arr[0] = 1 // first element

arr[4] = 5 // last element

OPERATIONS ON THE ARRAY

Linear Search : No need for sorted array

Binary Search : Array have to be sorted (Faster)

- There are several operations that can be performed on an array.

	Operation	Description
To display array element	Traversal	Processing each element in the array
search for a specific element	Search	Finding the location of an element with a given value
add element	Insertion	Adding a new element to an array
delete element	Deletion	Removing an element from an array
organise or sort elements	Sorting	Organizing the array elements in some order
combine two arrays	Merging	Combining two arrays into a single array
reverse the order of an array	Reversing	Reversing the elements of an array

IMPLEMENTATION OF VARIOUS ARRAY OPERATIONS

```
#include <iostream>
using namespace std;
const int MAX = 5;
class array
  {
    private:
        int arr[ MAX ];
    public:
        void insert (int pos, int num);
        void del (int pos);
        void reverse();
        void display();
        void search (int num);
  };
```

Array size is 5, its data type is int

location index number content

location index number

content

- A class called “array”.
- It contains an **arr** of 5 elements.
- Five functions defined like
  - **insert(),** add element
  - **del(),** delete element
  - **reverse()** reverse the array order
  - **display(),** display array element (content)
  - and **search()** search for a specific element

CONT. - INSERTING AN ELEMENT

Return type class function parameters

```
void array:: insert (int pos, int num)
{
    int i;
    // shift elements to right
    for (i = MAX - 1; i >= pos; i--)
        arr[i] = arr[i - 1];
    arr[i] = num;
}
```

loop counter

last element — i == -1 or lower (stop)

move to the next element — dec order

insert a value to this location

The **insert()** function takes two parameters:

- The position **pos** at which the new number has to be inserted.
- The number **num** that has to be inserted.

In this function, firstly through a loop, we have shifted the numbers from the specified position,
one place to the right of their existing position. Then we have placed the number **num** at position

CONT. - DELETING AN ELEMENT

function parameter

```
void array:: del (int pos)
  {
     int i;
     // skip to the desired position
     for (i = pos; i < MAX; i++)
           arr[i - 1] = arr[i];
      arr[i - 1] = 0;
   }
```

i equal pos i is lower than array size

The **del()** function deletes the element presented at the given position.
We have shifted the numbers placed after the position from where the number is to be deleted, one place to the left of their existing positions.
The number at position **n-1** is then overwritten with 0.

CONT. - REVERSING THE ENTIRE ARRAY

```
void array:: reverse()
{
    for (int i = 0; i < MAX / 2; i++)
    {
        int temp = arr[i];
        arr[i] = arr[MAX - 1 - i];
        arr[MAX - 1 - i] = temp;
    }
}
```

In **reverse()** function we have reversed the entire array by swapping the elements **arr[0]** with **arr[n-1]** and **arr[1]** with **arr[n-2]** and so on. Note that swapping should continue for **n / 2** times only, irrespective of whether **n** is odd or even.

CONT. - SEARCHING ARRAY FOR A GIVEN ELEMENT NUM

The **search()** function searches the array for the specified number.
The comparison is carried out until either the list is exhausted or a match is found.
If the match is not found then the function displays the relevant message.

```
void array:: search (int num)
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        if (arr[i] == num)
            cout << endl << "Element" << num<< "is at" << (i + 1) << "th position";
    }
    if (i == MAX)
        cout << endl << "Element" << num << "is absent";
}
```

CONT. - DISPLAYING THE CONTENTS OF AN ARRAY

```
void array:: display()
{
    cout << endl;
    for (int i = 0; i < MAX; i++)
    cout << arr[i] << “”;
}
```

In the **display()** function, the entire array is traversed to display the elements of the array.

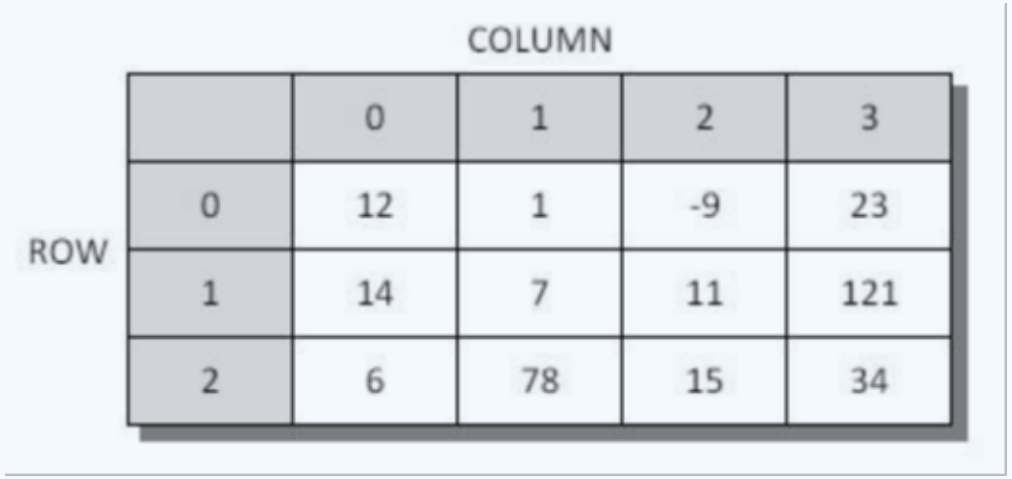
DATA STRUCTURES

2D Array and Pointers

CS Dept. – Qassim University

TWO-DIMENSIONAL ARRAYS

- It is a collection of elements placed in ***m*** rows and ***n*** columns.
  - For example, arr[3][4] is a 2-D array containing 3 rows and 4 columns.



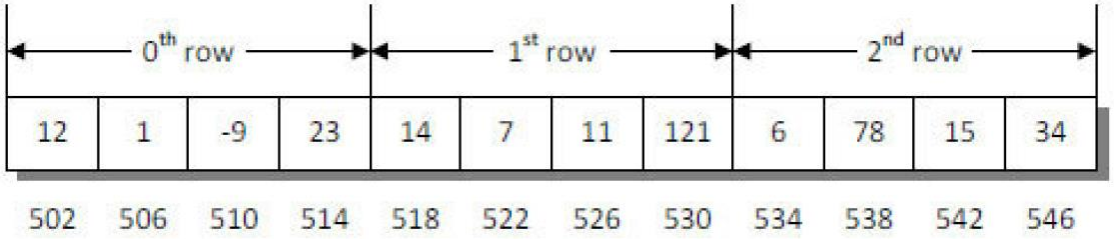
صفوف والأعمدة مجرد طريقة تفكير
الذاكرة تخزن بشكل خطي
في طريقتين:
Row Major (في C)
Column Major
نقدر نوصل لأي عنصر باستخدام:
العنوان الأساسي + حساب الإزاحة

ROW AND COLUMN ARRANGEMENT

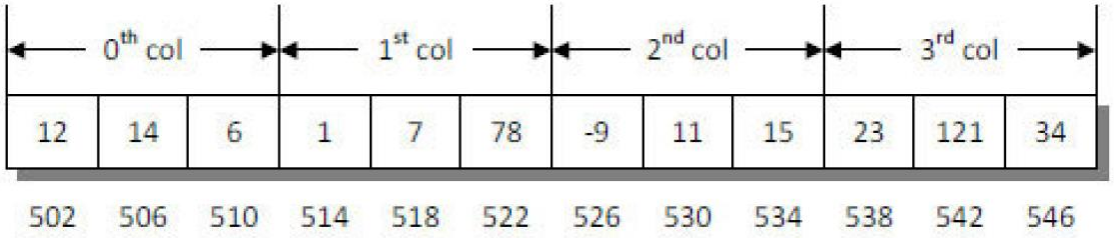
- Rows and columns of a matrix are only a matter of imagination.
  - When a matrix gets stored in memory all its elements are stored linearly since computer's memory can only be viewed as consecutive units of memory locations.
- Two possible arrangements of elements in memory-Row Major Arrangement and Column Major Arrangement.
- Since the array elements are stored in adjacent memory locations, we can access any element of the array once we know the **base address** (starting address) of the array and **number of rows and columns** present in the array.

```
int a[ 3 ][ 4 ] = {
                    { 12, 1, -9, 23 },
                    { 14, 7, 11, 121 },
                    { 6, 78, 15, 34 }
                  } ;
```

Row Major Arrangement



Column Major Arrangement



Note: Each integer occupies four-bytes

ROW MAJOR ARRANGEMENT - FORMULA

- Row Major Arrangement Formula is used to calculate the memory address of a given array index location.
  - The Location of element A[i, j] can be obtained by evaluating the expression:
    - LOC (A [i, j]) = base_address + W [N * i + j]
      - base_address = address of the first element in the array.
      - W = Word size, means a number of bytes occupied by each element of an Array.
      - M = Number of rows in the array.
      - N = Number of columns in the array.

القانون يعتمد على:
عدد الأعمدة (N)
موقع العنصر (i, j)
نحسب كم عنصر نتحرك
ثم نحولها لبايت باستخدام W

ROW MAJOR ARRANGEMENT - EXAMPLE

- Suppose we want to calculate the address of element **121** in the array above.
- It can be calculated as follow:

base_address = 502, **W**= 4, **M**=3, **N**=4, **i**=1, **j**=3

LOC (a [i, j]) = base_address + W * [(N*i) + j]

LOC (a[1, 3]) = 502 + 4 *[4*(1) + 3]

= 502 + 4 * [4 + 3]

= 502 + 4 * 7

= 502 + 28

= 530

COLUMN MAJOR ARRANGEMENT - FORMULA

- Column Major Arrangement Formula is used to calculate the memory address of a given array index location.
  - The Location of element A[i, j] can be obtained by evaluating the expression:
    - LOC (A [i, j]) = base_address + W [M * j + i]
    - **Example:** to calculate the address of element **121**
      - LOC (a[1, 3]) = 502 + 4 *[3*(3) + 1]
        = 502 + 4 * [9 + 1]
        = 502 + 4 * 10
        = 502 + 40
        = 542

COMMON MATRIX OPERATIONS

- Common matrix operations are addition, multiplication and transposition.

IMPLEMENTATION OF MATRIX OPERATIONS

```
#include <iostream>
using namespace std;
const int MAX = 3;
class matrix
    {
       private:
            int mat[MAX][MAX];
       public:
            matrix(); void create();
            void display();
            void matadd (matrix &m1, matrix &m2);
            void matmul (matrix &m1, matrix &m2);
            void transpose (matrix &m1);
    };
```

INITIALIZING MATRIX WITH 0

```
matrix:: matrix()
{
	for (int i = 0; i < MAX; i++)
	{
		for (int j = 0; j < MAX; j++)
			mat[i] [j] = 0;
	}
}
```

CREATING A MATRIX

```
void matrix:: create()
{
      int n;
      for (int i = 0; i < MAX; i++)
      {
            for (int j = 0; j < MAX; j++)
            {
                  cout << endl << "Enter the element:";
                  cin >> n;
                  mat[i][j] = n;
            }
      }
}
```

The **create()** function creates a 2-D array containing 3 rows and 3 columns

DISPLAYING THE CONTENTS OF MATRIX

```
void matrix:: display()
{
      for (int i = 0; i < MAX; i++)
      {
            for (int j = 0; j < MAX; j++)
                  cout << mat[i][j] << “”;
            cout << endl;
      }
}
```

The **display()** function displays the elements of the matrix.

ADDING TWO MATRICES

```
void matrix:: matadd (matrix &m1, matrix &m2)
{
      for (int i = 0; i < MAX; i++)
      {
            for (int j = 0; j < MAX; j++)
                  mat[i][j] = m1.mat[i][j] + m2.mat[i][j];
      }
}
```

The function **matadd()** adds the elements of two matrices **m1.mat** and **m2.mat**.
The result is stored in matrix **mat** of the object that has called the **matadd()** function.

MULTIPLYING TWO MATRICES

```
void matrix:: matmul (matrix &m1, matrix &m2)
{
    for (int k = 0; k < MAX; k++)
    {
        for (int i = 0; i < MAX; i++)
        {
            for (int j = 0; j < MAX; j++)
                mat[k][i] += m1.mat[k][j] * m2.mat[j][i];
        }
    }
}
```

The function **matmul()** multiplies the elements of matrix **m1.mat** with the elements of matrix **m2.mat** and stores the result into the matrix **mat** of the object that called the function.

OBTAINING TRANSPOSE OF MATRIX

```
void matrix:: transpose (matrix &m1)
{
    for (int i = 0; i < MAX; i++)
    {
        for (int j = 0; j < MAX; j++)
            mat[i][j] = m1.mat[j][i];
    }
}
```

The function transposes a matrix.
The transpose of a matrix is obtained by interchanging the rows with corresponding columns of a given matrix.
The transposed matrix is stored in matrix **mat** of the object

EXAMPLES (TRY THIS CODE)

```
matrix mat1;
cout << endl << “Enter elements for first matrix:” << endl;
mat1.create();
matrix mat2;
cout << endl << “Enter elements for second matrix:” << endl;
mat2.create();
cout << endl << “First matrix:” << endl;
mat1.display();
cout << endl << “Second matrix:” << endl;
mat2.display();
matrix mat3;
```

EXAMPLES (TRY THIS CODE)

```
mat3.matadd (mat1, mat2);
cout << endl << “After Addition:” << endl;
mat3.display();
matrix mat4;
mat4.matmul (mat1, mat2);
cout << endl << “After Multiplication:” << endl;
mat4.display();
matrix mat5;
mat5.transpose (mat1);
cout << endl << “Transpose of first matrix:” << endl;
mat5.display();
```

POINTER REVIEW!

- When a variable is declared, the memory needed to store its value is assigned a specific location in memory (**its memory address**).
- The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator.

أولاً: ماذا يحدث عند تعريف متغير؟
عندما نكتب:
int var;
الكمبيوتر:
يحجز مكان في الذاكرة
يعطيه عنوان (Memory Address)
مثلاً (افتراضياً):
موجود في العنوان 1776 var
عنوان المتغير (&)
نحصل على عنوان أي متغير باستخدام:
&var
لو var في العنوان 1776
إذن:
&var = 1776

POINTER REVIEW!

For example:

```
foo = &var;
```

- let's assume, in order to help clarify some concepts, that var is placed during runtime in the memory address 1776.
- In this case, consider the following code statements, what would be the results?

```
var = 25;

foo = &var;

bar = var;
```

var = 25 → تحمل قيمة 25
foo = &var → تحمل عنوان var
bar = var → تساوي قيمة var

POINTER REVIEW!

- The variable that stores the address of another variable is what in C++ is called a pointer.
- They can be used to access the variable they point to directly.
- This is done by preceding the pointer name with the dereference operator (*)

baz = *foo

- The declaration of pointers follows this syntax:

type * name;

في الـ Pointers
بدون نجمة
foo
يعطيك العنوان

مع نجمة
*foo
يعطيك القيمة داخل هذا العنوان

TRY THIS CODE

```
int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n'
```

الحالة	نستخدم & ؟
متغير عادي	نعم
مصفوفة	لا
نسخ pointer	لا
داينميك	لا

TRY THIS CODE

```
int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;
  p1 = &firstvalue;                  // p1 = address of firstvalue
  p2 = &secondvalue;                 // p2 = address of secondvalue
  *p1 = 10;                          // value pointed to by p1 = 10
  *p2 = *p1;                         // value pointed to by p2 = value pointed to by p1
  p1 = p2;                           // p1 = p2 (value of pointer is copied)
  *p1 = 20;                          // value pointed to by p1 = 20
```

firstvalue = 10
secondvalue = 20

TRY THIS CODE

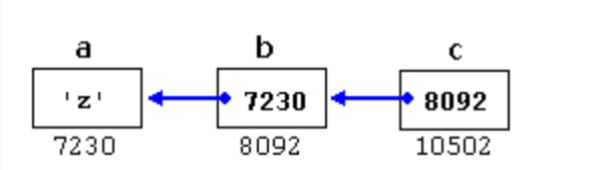
```
int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
```

numbers = [0, 20, 30, 40, 50]

POINTERS TO POINTERS

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

```
c is of type char** and a value of
*c is of type char* and a value of
**c is of type char and a value of
```



NULL POINTERS

- Besides a memory address, there is one additional value that a pointer can hold: **a null value**.
- **A null value** (often shortened to null) is a special value that means something has no value.
- When a pointer is holding a null value, it means the pointer is not pointing at anything.

```
int * p = 0;
int * q = nullptr;
int * r = NULL;
if (p == nullptr) // explicit test for equivalence
        cout << "p is null \n";
    else
        cout << "p is non-null \n";
```