ABSTRACT DATA TYPES

Data Structures and Algorithms, Fall 2023
Waheed Iqbal

Department of Data Science, FCIT
University of the Punjab, Lahore, Pakistan.

Abstract Data Types

- Primitive data types—such as int, char, double, bool are use to store simple data.

- To represent more complex information, we need to combine primitive types and introduce new types.

Primitive Data types in C++

Type	Values	Representation	Operations
boolean	true, false	Single byte	&&, ||, !
char,short, int, long	Integers of varying sizes	Two's complement	+, -, *, /, others
float, double	Floating point numbers of varying sizes and precisions	Two's complement with exponent and mantissa	+, -, *, /, others

Non-Primitive Data Types

- A **class** is a non-primitive user defined *data type*
- The possible ***values*** of a class are called **objects**
- The *data* ***representation*** is a reference (pointer) to a block of storage
- The ***operations*** on the objects are called **methods**
- You can (and must) define your own classes, as well

Abstraction

- Hiding unnecessary details is known as **abstraction**
- Only presenting an **interface** , not the **implementation** part
- An essential element of object oriented programming is **abstraction**

Real Life Abstraction Example

- Human manage complexity through abstraction. For example, people don't think of a car as combination of tens of thousands of parts. But as a single well defined object.

- This abstraction allows humans to drive the car easily without being overwhelmed by the complexity of the parts that form a car.

Real Life Abstraction Example (Cont.)

- User just need to know about the parts and their operations. For example, how to use the steering, breaks, gears, etc. But, not concerned with the Mechanisms of Steering, breaks and gears. To turn left, rotate the steering towards left side.

Abstract Data Types

- An Abstract Data Type (ADT) is:
  - a set of ***values***
  - a set of ***operations***, which can be applied uniformly to all these values
- The word "abstract" refers to the fact that the **data** and the **operations** defined on it are exposed to the user however implementation is hidden from the user.
- Specification of ADTs indicate
  - What the ADT operations are?
  - Not how to **implement** them

What are Data Structures?

- **Implementation of ADT = Data Structures**
- When we talk about data structures, we are talking about the ***implementation*** of a data type
- If I talk about the possible ***values*** of, say, complex numbers, and the ***operations*** I can perform with them, I am talking about them as an ADT
- If I talk about the way the parts ("real" and "imaginary") of a complex number are *stored in memory,* I am talking about a data structure
- An ADT may be implemented in several different ways
  - A complex number might be stored as two separate doubles,
  - or as an array of two doubles, or even in some strange way

Array ADT

- **Collection of data elements**
  - A fixed-size sequence of elements, all of the same type
- **Basic Operations**
  - Direct access to each element in the array by specifying its position so that values can be retrieved from or stored in that position

An Array as an ADT. Let's discuss C++ Arrays

- Fixed size ----------specify the capacity of the array
- Ordered----------indices are numbered 0,1,2,…,capacity-1
- Same type of elements-------specify the element type
- Direct access----------subscript operator[]

In **Python Arrays** are:

- Dynamic Size
- Ordered
- Different type of elements
- Direct access

Array Types

- Array is a **collection of elements** of a specific type
- **One-dimensional array**: only one index is used
- **Multi-dimensional array**: array involving more than one index
- **Static array**: the compiler determines how memory will be allocated for the array
- **Dynamic array**: memory allocation takes place during execution

Example of array

```
# Creating an array
numbers = [1, 2, 3, 4, 5]

# Accessing elements
print("Element at index 2:", numbers[2])  # Outputs: 3

# Modifying elements
numbers[1] = 10
print("Modified array:", numbers)  # Outputs: [1, 10, 3, 4, 5]

# Length of the array
print("Length of the array:", len(numbers))  # Outputs: 5

# Iterating through the array
print("Elements in the array:")
for num in numbers:
    print(num)
```

Array Output Function

```
def display(my_list):
    for value in my_list:
        print(value, end=" ")
    print()
my_list = [1, 2, 3, 4, 5]
display(my_list)# Output?
print(my_list)# Output?
```

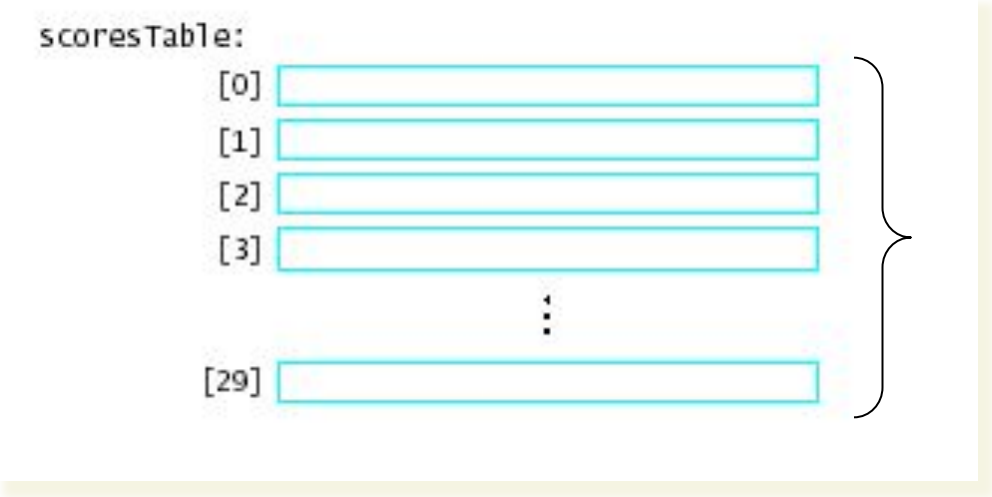
Multidimensional Arrays

- Consider a table of test scores for several different students

	Test 1	Test 2	Test 3	Test 4	Test 5
Student 1	99.0	93.5	89.0	91.0	97.5
Student 2	66.0	68.0	84.5	82.0	87.0
Student 3	88.5	78.5	70.0	65.0	66.5
⋮	⋮	⋮	⋮	⋮	⋮
Student 30	100.0	99.5	100.0	99.0	98.0

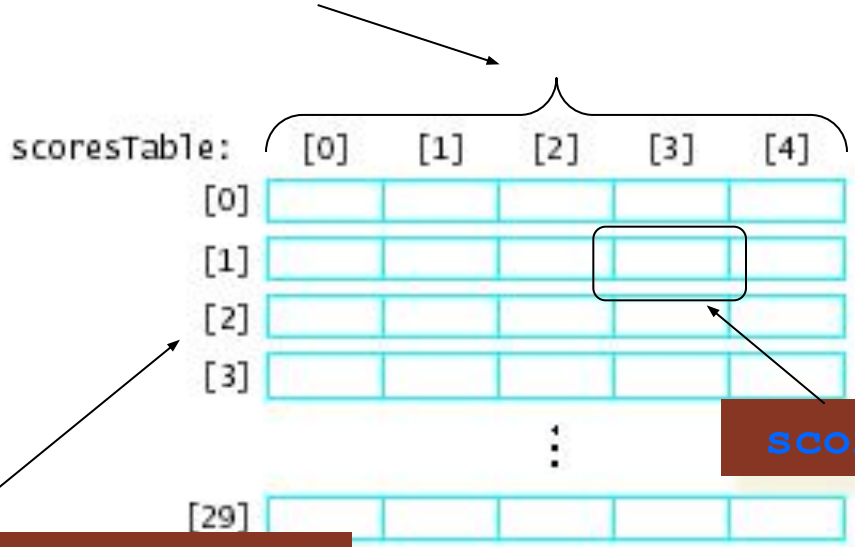
Array of Array Declarations

- An array of arrays
  - An array whose elements are other arrays



Array of Array Declarations

- Each of the rows is itself a one dimensional array of values



`scoresTable [1][3]`

`scoresTable[2]`

is the whole row numbered 2

Multidimensional Arrays

```
# Creating a 2D array (list of lists)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing elements in the 2D array
print("Element at row 2, column 1:", matrix[1][0])  # Outputs: 4
```

Multidimensional Arrays

```
# Creating a 3D array (list of lists of lists)
cube = [
    [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ],
    [
        [10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]
    ],
    [
        [19, 20, 21],
        [22, 23, 24],
        [25, 26, 27]
    ]
]

# Accessing elements in the 3D array
print("Element at layer 2, row 1, column 0:", cube[1][0][0])
```

Types of ADTs

Two broad types of ADTs:

- **Linear ADTs**
  - Arrays
  - Lists
  - Stacks
  - Queues
- **Non Linear ADTs**
  - Trees
  - Graphs
  - Hash Table

Binary Search in Array

- Let's discuss how we may search an element in a sorted array using binary search algorithm!

```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid  # Element found, return its index
        elif arr[mid] < target:
            low = mid + 1  # Search in the right half
        else:
            high = mid - 1  # Search in the left half

    return -1  # Element not found

# Example usage:
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target_element = 7

result = binary_search(sorted_array, target_element)

if result != -1:
    print(f"Element {target_element} found at index {result}")
else:
    print(f"Element {target_element} not found in the array")
```

Identify Common Elements in Sorted Arrays

- Given two sorted arrays, you need to identify similar elements. Consider the following two arrays:

- A: 13, 27, 35, 40, 49, 55, 59
- B: 17, 35, 39, 40, 55, 58, 60

Next!

Linked Lists, Stacks, and Queues

Credit

- Some of the slides are taken from:
  - *David Matuszek's* (http://www.cis.upenn.edu/~matuszek/) course on Programming Languages & Techniques
  - *Bernard Chen's* lecture notes on Data Structures and Abstract Data Types