DATA STRUCTURE & ALGORITHMS

Array

Objectives

You should be able to describe:

- One-Dimensional Arrays
- Array Initialization
- Arrays as Arguments
- Two-Dimensional Arrays
- Common Programming Errors

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure
- that specifies:
- • the characteristics of the collection of data
- • the operations that can be performed on the collection
- • It’s *abstract* because it doesn’t specify *how* the ADT will be
- implemented.
- • A given ADT can have multiple implementations

Abstract Data Types (ADT)

- An *abstract data type* is a collection of data structures and operations abstracted into a simple data type.
  - The implementation of an abstract data type is ``hidden" from the rest of the program. How values are represented is therefore less important than what operations are provided for manipulating them.
  - The set of operations define the *interface* to the ADT: accessed to the ADT can be made only through the defined operations.

Abstract Data Types (Cont'd)

- Two major purposes for the use of ADTs:
  - the independence of the use of the ADT from its implementation, which permits modification of the implementation without affecting the execution units where the ADT is used.
  - the maintenance of the integrity of the ADT by restricting access to the operations provided

Abstract Data Types (Cont'd)

- The ideas of ADTs can be used to write modular and reliable programs.

1. define the type of data items to be stored in the ADT,
2. decide the set of valid operations on the ADT,
3. choose an implementation method for the operations, code the operations as procedures/functions, using the defined
4. types for specifying parameters.
5. use only these defined operations to access the ADT.

A Simple ADT: A Bag

- A bag is just a container for a group of data items.
  - analogy: a bag of candy
- The operations supported by our Bag ADT:
  - add(item): add item to the Bag
  - remove(item): remove one occurrence of item (if any) from the Bag
  - contains(item): check if item is in the Bag
  - numItems(): get the number of items in the Bag
  - grab(): get an item at random, without removing it
  - toArray(): get an array containing the current contents of the bag
- Note that we *don't* specify *how* the bag will be implemented

Specifying an ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

```
public interface Bag {
boolean add(Object item);
boolean remove(Object item);
boolean contains(Object item);
int numItems();
Object grab();
Object[] toArray();
}
```

• An interface specifies a set of methods.

- includes only the method headers
- *cannot* include the actual method definitions

Implementing an ADT Using a Class

- To implement an ADT, we define a class:

```
public class ArrayBag implements Bag {
private Object[] items;
private int numItems;
…
public boolean add(Object item) {
…
}
```

- When a class header includes an implements clause, the class must define all of the methods in the interface.

Arrays :
One-Dimension Arrays

- One-Dimension Array(Single-Dimension Array or Vector): a list of related values
  - All items in list have same data type
  - All list members stored using single group name
- Example: a list of grades

  98, 87, 92, 79, 85

One-Dimension Arrays (continued)

- Array declaration statement provides:
  - The array(list) name
  - The data type of array items
  - The number of items in array
- Syntax

  dataType arrayName[numberOfItems]

  - Common programming practice requires defining number of array items as a constant before declaring the array

Arrays as Arguments

- Array elements are passed to a called function in same manner as individual scalar variables
  - Example:

    findMax(grades[2], grades[6]);

- Passing a complete array to a function provides access to the actual array, not a copy
  - Making copies of large arrays is wasteful of storage

Arrays as Arguments (continued)

- Examples of function calls that pass arrays

```
int nums[5];    // an array of five integers

char keys[256];   // an array of 256 characters

double units[500], grades[500];// two arrays of 500
    //doubles
```

- The following function calls can then be made:

```
findMax(nums);
findCharacter(keys);
calcTotal(nums, units, grades);
```

Two-Dimensional Arrays

- Two-dimensional array (table): consists of both rows and columns of elements
- **Example**: two-dimensional array of integers

8		16	9	52
3		15	27	6
14	25	2	10	

- **Array declaration**: names the array val and reserves storage for it

```
int val[3][4];
```

Two-Dimensional Arrays (continued)

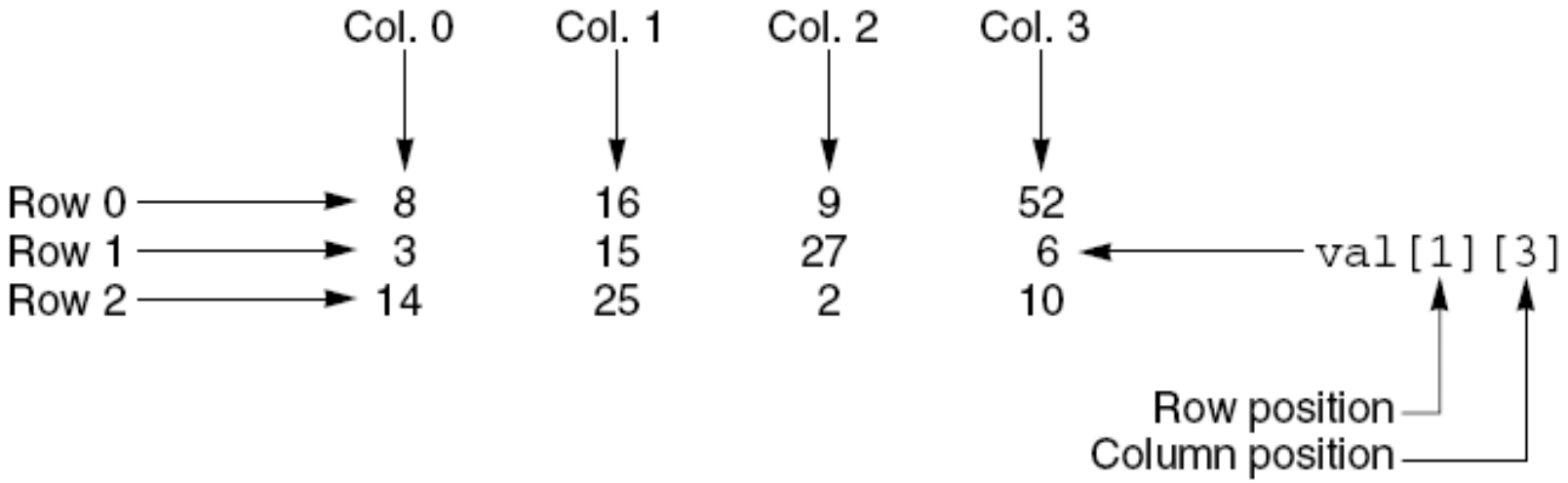
- Locating array elements
  - val[1][3] uniquely identifies element in row 1, column 3
- Examples using elements of val array:

```
price = val[2][3];
val[0][0] = 62;
newnum = 4 * (val[1][0] - 5);
sumRow = val[0][0] + val[0][1] + val[0][2] + val[0]
   [3];
```

  - The last statement adds the elements in row 0 and sum is stored in sumRow

Two-Dimensional Arrays (continued)

FIGURE 8.9 *Each Array Element Is Identified by Its Row and Column Position*



2-Dimensional Arrays Initialization

- can be done within declaration statements (as with single-dimension arrays)
- Example:

```
int val[3][4] = { {8,16,9,52},
                  {3,15,27,6},
                  {14,25,2,10} };
```

  - First set of internal braces contains values for row 0, second set for row 1, and third set for row 2
  - Commas in initialization braces are required; inner braces can be omitted

2-Dimensional Arrays Processing

- Processing two-dimensional arrays: nested for loops typically used
  - Easy to cycle through each array element
    - A pass through outer loop corresponds to a row
    - A pass through inner loop corresponds to a column
  - Used Nested for loop to multiply each val element by 10 and display results

2-Dimensional Arrays as Arguments

- Prototypes for functions that pass two-dimensional arrays can omit the row size of the array

  - Example ():

    Display (int nums[ ][4]);

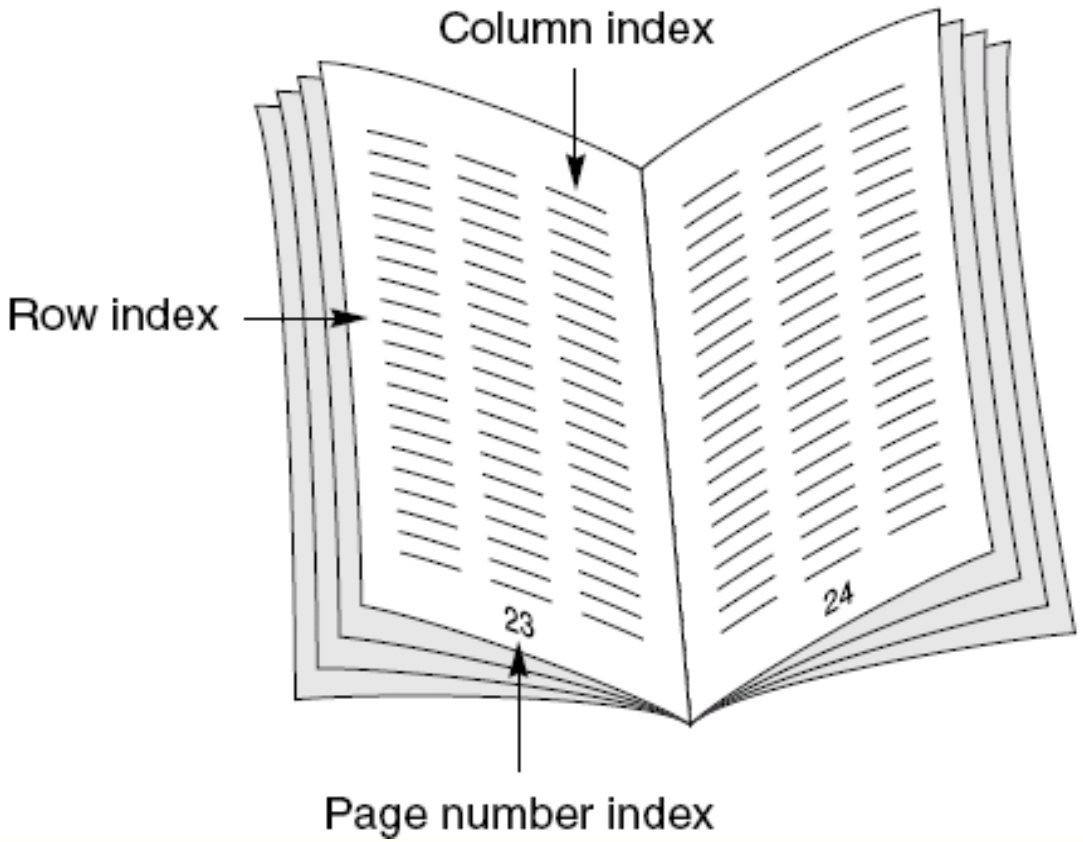
  - Row size is **optional** but column size is **required**

Larger-Dimension Arrays

- Arrays with more than two dimensions allowed but not commonly used
- Example: int response[4][10][6]
  - First element is response[0][0][0]
  - Last element is response[3][9][5]
- A three-dimensional array can be viewed as a book of data tables
  - First subscript (rank) is page number of table
  - Second subscript is row in table
  - Third subscript is desired column

Larger-Dimension Arrays (continued)

FIGURE 8.12 *Representation of a Three-Dimensional Array*



Common Programming Errors

- Forgetting to declare an array
  - Results in a compiler error message equivalent to “invalid indirection” each time a subscripted variable is encountered within a program
- Using a subscript that references a nonexistent array element
  - For example, declaring array to be of size 20 and using a subscript value of 25
  - Not detected by most C++ compilers and will probably cause a runtime error

Common Programming Errors (continued)

- Not using a large enough counter value in a for loop counter to cycle through all array elements
- Forgetting to initialize array elements
  - Don’t assume compiler does this

Summary

- Single-dimensional array: a data structure that stores a list of values of same data type
  - Must specify data type and array size
  - `int num[100];` creates an array of 100 integers
- Array elements are stored in contiguous locations in memory and referenced using the array name and a subscript
  - For example, `num[22]`

Summary (continued)

- Two-dimensional array is declared by listing both a row and column size with data type and name of array
- Arrays may be initialized when they are declared
  - For two-dimensional arrays you list the initial values, in a row-by-row manner, within braces and separating them with commas
- Arrays are passed to a function by passing name of array as an argument