Abstract Data Type & ARRAYS

ALGORITHMS & DATA STRUCTURES – I

COMP 221

Abstract Data Type (ADT)

Def. **a collection of related data items**
together with
an associated set of operations

e.g. whole numbers (integers) and arithmetic operators for addition, subtraction, multiplication and division.

e.g. Seats for TFA
Basic operations: find empty seat, reserve a seat,
cancel a seat assignment

Why "abstract?"
Data, operations, and relations are studied
independent of implementation.

What not ***how*** is the focus.

ADT

You drive the car without know how the engine car work and daily you use many items without know how they work, separating the implementation details is called ***abstraction***.

Abstraction focuses on what the engine does and not how it works.

Abstract Data type(ADT): A data type that specifies the logical properties without the implementation details.

Implementation of an ADT

Def. Consists of

storage structures (aka data structures)
to store the data items
and
algorithms for the basic operations.

The storage structures/data structures used in implementations are provided in a language *(primitive* or *built-in)* or are built from the language constructs *(user-defined)*.

In either case, successful software design uses *data abstraction*:

Separating the definition of a data type from its implementation.

C-Style Data Structures: Arrays

*Defn of an **array** as an ADT:*

An ordered set (sequence) with a fixed number of elements, all of the same type,

where the basic operation is

direct access to each element in the array so values can be retrieved from or stored in this element.

Properties:

- Ordered so there is a first element, a second one, etc.
- Fixed number of elements — **fixed capacity**
- Elements must be the same type (and size);

  ∴ use arrays only for homogeneous data sets.
- Direct access: Access an element by giving its location

  — the time to access each element is the same for all elements, regardless of position.

  — in contrast to sequential access (where to access an element, one must first access all those that precede it.)

Declaring arrays in C++

```
element_type array_name[CAPACITY];
```

where

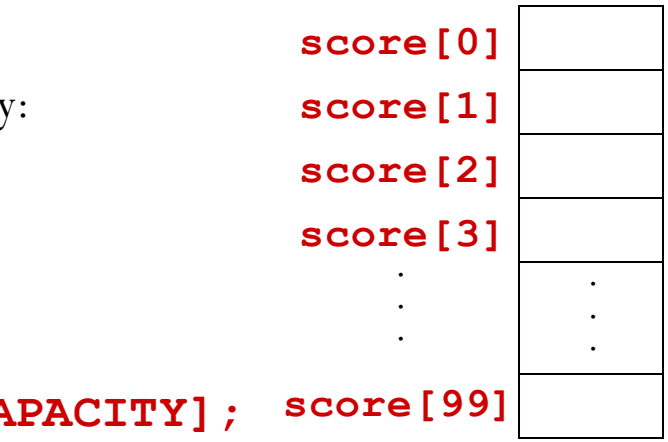
element_type	is any type
array_name	is the name of the array — any valid identifier
CAPACITY	(a positive integer constant) is the number of elements in the array

Can't input the capacity

The compiler reserves a block of consecutive memory locations, enough to hold ***CAPACITY*** values of type ***element_type***.

The elements (or positions) of the array are indexed 0, 1, 2, . . ., ***CAPACITY*** - 1.

e.g., `double score[100];`



★ Better to use a named constant to specify the array capacity:

```
const int CAPACITY = 100;
double score[CAPACITY];
```

★ Can use **typedef** with array declarations; e.g.,

```
const int CAPACITY = 100;
typedef double ScoresArray[CAPACITY];
ScoresArray score;
```

How well does C/C++ implement an array ADT?

As an ADT		In C++
ordered	⟷	indices numbered 0, 1, 2, . . ., **`CAPACITY`** - 1
fixed size	⟷	**`CAPACITY`** specifies the capacity of the array
same type elements	⟷	**`element_type`** is the type of elements
direct access	⟷	subscript operator **`[]`**

Subscript operator

[] is an actual operator and not simply a notation/punctuation as in some other languages. Its two operands are an ***array variable*** and an ***integer index*** (or subscript) and is written

array_name[i]

Here ***i*** is an integer expression with $0 \leq$ ***i*** $\leq$ ***CAPACITY*** $- 1$.

[] returns the ***address*** of the element in location ***i*** in ***array_name***; so ***array_name[i]*** is a **variable**, called an **indexed** (or **subscripted**) **variable**, whose type is the specified ***element_type*** of the array.

Also, it can have an index

This means that it can be used on the left side of an assignment, in input statements, etc. to store a value in a specified location in the array. For example:

```
// Zero out all the elements of score
for (int i = 0; i < CAPACITY; i++)
   score[i] = 0.0;
// Read values into the first numScores elements of score
for (int i = 0; i < numScores; i++)
   cin >> score[i];

// Display values stored in the first numScores elements
for (int i = 0; i < numScores; i++)
   cout << score[i] << endl;
```

Array Initialization

In C++, arrays can be initialized when they are declared.

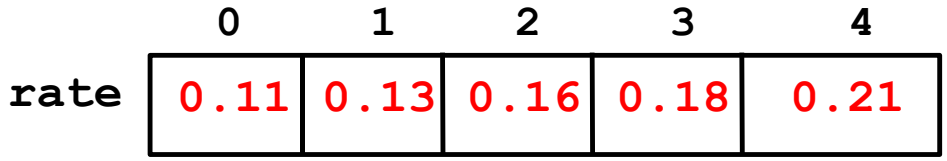
an array literal

Numeric arrays:

```
element_type num_array[CAPACITY] = {list_of_initial_values};
```

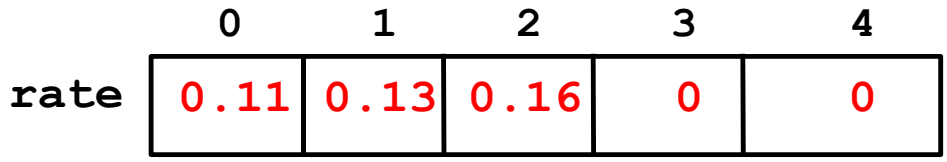
Example:

```
double rate[5] = {0.11, 0.13, 0.16, 0.18, 0.21};
```



Note 1: If fewer values supplied than array's capacity, remaining elements assigned 0.

```
double rate[5] = {0.11, 0.13, 0.16};
```



What's an easy way to initialize an array to all zeros?

Note 2: It is an error if more values are supplied than the declared size of the array. How this error is handled, however, will vary from one compiler to another.

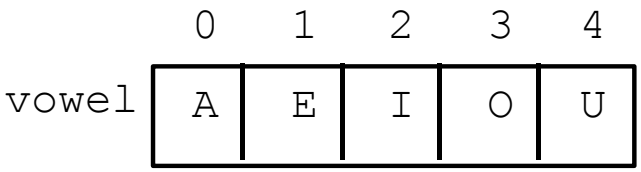
Note 3: If no values supplied, array elements are undefined (i.e., garbage values).

Character arrays:

Character arrays may be initialized in the same manner as numeric arrays.

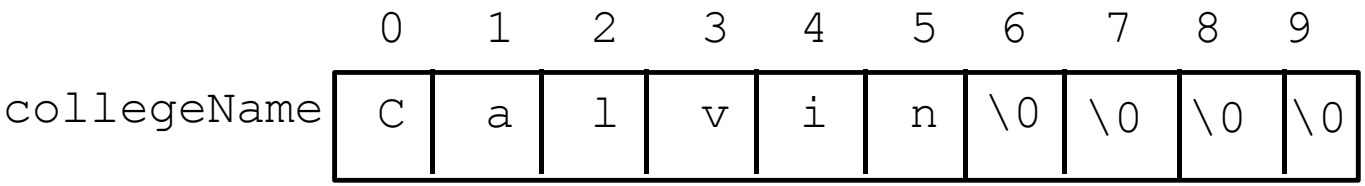
```
char vowel[5] = {'A', 'E', 'I', 'O', 'U'};
```

declares `vowel` to be an array of 5 characters and initializes it as follows:



Note 1: If fewer values are supplied than the declared size of the array, the zeroes used to fill uninitialized elements are interpreted as the null character `'\0'` whose ASCII code is 0.

```
const int NAME_LENGTH = 10;
char collegeName[NAME_LENGTH]={'C', 'a', 'l', 'v', 'i', 'n'};
```



Note 2: Character arrays may be initialized using **string constants**. For example, the following declaration is equivalent to the preceding:

```
char collegeName[NAME_LENGTH] = "Calvin";
```

Note 3: The null character `'\0'` (ASCII code is 0) is used as an **end-of-string mark.**

Thus, character arrays used to store strings should be declared large enough to store the null character. If it is not, one cannot expect some of the string functions and operations to work correctly.

If a character array is initialized with a string constant, the end-of-string mark is added automatically, provided there is room for it.

```
char collegeName[7] = {'C', 'a', 'l', 'v', 'i', 'n', '\0'};
char collegeName[7] = "Calvin";
```

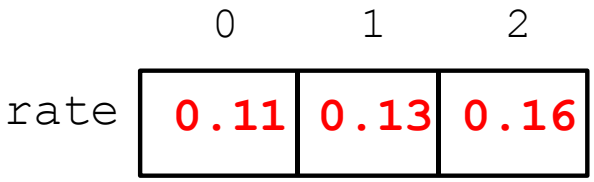
Initializations with no array size specified

The array capacity may be omitted in an array declaration with an initializer list.

In this case, the number of elements in the array will be
the number of values in the initializer list.

Example:

```
double rate[] = {0.11, 0.13, 0.16};
```



Note: This explains the brackets in constant declarations such as

```
const char IN_FILE[] = "employee.dat";
```

Addresses

When an array is declared, the address of the first byte (or word) in the block of memory associated with the array is called the ***base address*** of the array.

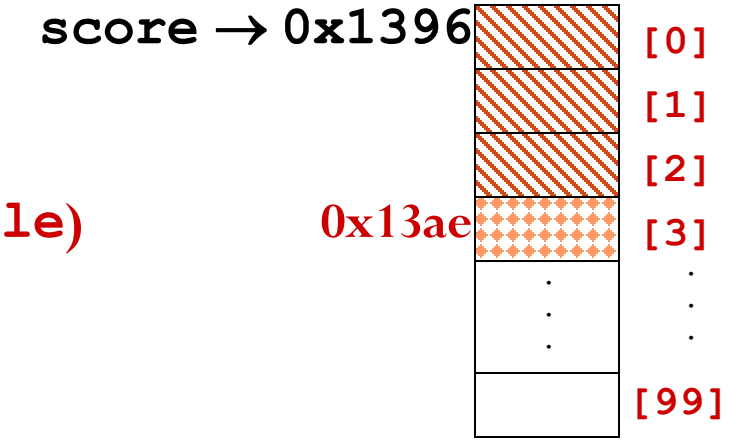
Each array reference must be translated into an ***offset*** from this base address.

For example, if each element of array **`score`** will be stored in 8 bytes and the base address of **`score`** is **`0x1396`**. A statement such as

```
cout << score[3] << endl;
```

requires that array reference **`score[3]`** be translated into a memory address:

`score[3]` → **`0x1396 + 3 * (sizeof double)`**
= **`0x1396 + 3 * 8`**
= **`0x13ae`**



The contents of the memory word with this address 0x13ae can then be retrieved and displayed.

An ***address translation*** like this is carried out each time an array element is accessed.

The value of ***array_name*** is actually the *base address* **of *array_name***

array_name + *index* is the address of ***array_name[index]***.

An array reference ***array_name[index]***

is equivalent to ****(array_name + index)***

***** is the *dereferencing* operator

****ref*** returns the **contents of the memory location with address *ref***

For example, the following statements are equivalent:

```
cout << score[3] << endl;
cout << *(score + 3) << endl;
```

C-Style Multidimensional Arrays

Example: A table of test scores for several different students on several different tests.

p.52

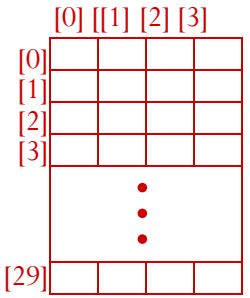
	Test 1	Test 2	Test 3	Test 4
Student 1	99.0	93.5	89.0	91.0
Student 2	66.0	68.0	84.5	82.0
Student 3	88.5	78.5	70.0	65.0
:	:	:	:	:
:	:	:	:	:
Student-n	100.0	99.5	100.0	99.0

For storage and processing, use a **two-dimensional array**.

Declaring Two-Dimensional Arrays

Standard form of declaration:

```
element_type array_name[NUM_ROWS][NUM_COLUMNS];
```



Example:

```
const int NUM_ROWS = 30,
          NUM_COLUMNS = 4;
double scoresTable[NUM_ROWS][NUM_COLUMNS];
```

or

```
typedef double TwoDimArray [NUM_ROWS][NUM_COLUMNS];
TwoDimArray scoresTable;
```

Initialization

- List the initial values in braces, row by row;
- May use internal braces for each row to improve readability.

Example:

```
double rates[2][3] = {{0.50, 0.55, 0.53},  // first row
                      {0.63, 0.58, 0.55}}; // second row
```

Processing Two-Dimensional Arrays

- Remember: Rows (and) columns are numbered from zero!!
- Use *doubly-indexed variables:*

  `scoresTable[2][3]` is the entry in row 2 and column 3

  row index column index

Counting from 0

- Use *nested loops* to vary the two indices, most often in a **rowwise** manner.

```
int numStudents, numTests, i, j;

cout << "# students and # of tests? ";
cin >> numStudents >> numTests;

cout << "Enter test scores for students\n";
for (i = 0; i < numStudents; i++)
{
   cout << '#' << i + 1 << ':';
   for (j = 0; j < numTests; j++)
      cin >> scoresTable[i][j];
}
```

THANK YOU