Array ADT

So far we have looked at Integer, String, Stack, and List ADTs.

ADT Array:

Elements: A component data type is defined and all elements are of that type (homogeneous).

Structure: A linear index type is specified and a 1-1 correspondence exists between the index type and component type

Array ADT Continued

Domain: All possible index values with all combinations of associated component values.

Operations:

1) Copy array element value (e.g value = a[i])
   results: The i^{th} component of a is copied into value
   requires: ?

Array ADT Continued

2) Update array element (e.g. a[i] = value)
results: The i^{th} component of a is assigned value
requires: ?

3) Array copy (e.g. a = b)
results: All elements from b are copied into their respective positions in a

C Arrays

```
int a [100];
```

```
a[i]   is   a + (i * sizeof (int) );
```

a is a constant pointer

Arrays and Pointers

```
int x, y;
int *array[2];


x = 1;
y = 2;
array[0] = &x;
array[1] = &y;
```

Dynamic Arrays

- What is the difference between:

```
int a[10]
```

- and

```
int* psArray = (int *) malloc(10 * sizeof(int));
```

Dynamic Arrays

- Dynamically sized arrays can be resized.
- How would we double the size of the array created below:

```
int* psArray = (int*) malloc(sizeof(int) * n)
```

Multi-dimensional Arrays p

- Obviously, we can extend the array ADT to include multidimensional arrays. The only real change is the structure which becomes something like:

- component-type array[index1][index2]
- component-type array[row][column]

Array Mapping Function (AMF)

- The only real challenge in implementing arrays is how to map a multi-dimensional array into linear space.
- Two- dimensional AMF by rows:
  - right most index varies the fastest
- Consider: `int a[10][5];`

```
a[i][j] = base(a) + (i * 5 + j) * sizeof (int);
```

More AMF

- What is the AMF for each of the following assuming a row-major mapping?

1. `double a[100];`

2. `int b[5][10][15];`

Arrays and Pointers

```
int x;
int array[2][3];


x = 1;
array[0][1] = x;
array[1][2] = 9;
```

Iterator

Design Pattern

Used to traverse all elements in a container

keep track of a current pointer in the container (state!)

first()

hasNext()

next()

last()

Generally used in Object Oriented Languages but can be applied to any data structure.

C arrays do not provide this interface.