Arrays

Arrays are objects on the heap.

Arrays are accessed by pointers.

Arrays are sequences of mutable locations: its *items*.

Arrays have an immutable Length field.

```
var a := new int[5] ;
a[0], a[1], a[2], a[3], a[4] := 9, 4, 6, 3, 8 ;
var k := 0 ;
while( k < a.Length ) { print a[k], "\n" ; k := k+1 ; }
```

Case study: compare and swap

Let's write a method to compare two items (possibly the same) of an int array and swap them so that the smaller comes first.

Because it changes the contents of the array's items, it is a method with *side-effects*.

Up to now we have not seen methods with side effects.

Let's look at this contract in detail:

```
method compareAndSwap( a : array<int>, p : int, q : int )
requires 0 <= p <= q < a.Length
```

The indices have to be valid.

```
modifies a
```

This line declares that the method has the side effect of potentially changing the items of the array a points to.

(When a contract has no modifies clause, it implicitly has no side effects.)

ensures a[p] <= a[q]

The method ensures that after execution $\mathrm{a[p]} \leq \mathrm{a[q]}$.

ensures (a[p] == **old**(a[p]) && a[q] == **old**(a[q]))
|| (a[p] == **old**(a[q]) && a[q] == **old**(a[p]))

The method ensures that the final values of a[p] and a[q] are either the same as their initial values or have been swapped.

- The notation $\mathbf{old}(E)$ means the value that expression E had when the method execution started.
- The parentheses are needed because Dafny puts || and && on the same level of precedence.
- This expression could be written more succinctly as

$$\mathrm{multiset\{a[p], a[q]\}} == \mathrm{old}(\mathrm{multiset\{a[p], a[q]\}})$$

ensures forall i :: 0 <= i < a.Length && i != p && i != q
==> a[i] == **old**(a[i])

This postcondition ensures that all other items are unchanged.

- ==> is Dafny's implication operator
- Note the use of the forall quantifier. In mathematical notation:

$$\forall i \cdot i \in \{0, ..a.\,\mathrm{length}\} \wedge i \notin \{p, q\} \Rightarrow a[i] = \mathbf{old}(a[i])$$

All together the contract looks like this

```
method compareAndSwap( a : array<int>, p : int, q : int )
requires 0 <= p <= q < a.Length
modifies a
ensures a[p] <= a[q]
ensures (a[p] == old(a[p]) && a[q] == old(a[q]))
    || (a[p] == old(a[q]) && a[q] == old(a[p]))
ensures forall i :: 0 <= i < a.Length && i != p && i != q
                        ==> a[i] == old(a[i])
```

Case Study: Selection Sort

We'll write a method that sorts an array of ints in place.

Some predicates

In order to specify it, we will define some predicates.

We'll use Dafn's sequence type in the predicates since they are a little easier to use than arrays

- A sequence is a sequence of values.
- Sequences are values.
- Arrays are objects.

First we need to be able to say that a sequence is sorted.

```
predicate Sorted( s : seq<int>)
{
  forall i,j : int :: 0 <= i <= j < |s| ==> s[i] <= s[j]
}
```

(Note $|s|$ is the length of sequence s.)

Also we need to say that one sequence is a permutation of another.

For this we use Dafny's multiset type.

A multiset (aka bag) is a collection where order is not important, but multiplicity is.

The expression $\mathbf{multiset}(s)$ is the multiset that has the same items as s in the same numbers. For example $\mathbf{multiset}([1,1,2,1,2,3])$ contains 1 thrice, 2 twice, and 3 once. It is equal to $\mathbf{multiset}([1,1,1,2,2,3])$

```
predicate PermutationOf( s : seq<int>, t : seq<int> ) {
  multiset(s) == multiset(t)
}
```

A contract for sorting

Now we can write the contract for SelectionSort

```
method selectionSort( a : array<int> )
modifies a
ensures Sorted( a[..] )
ensures PermutationOf( a[..], old( a[..] ) )
```

a[..] is the sequence of values of items of array a.

Dafny's slice notation

If a is a sequence (or array)

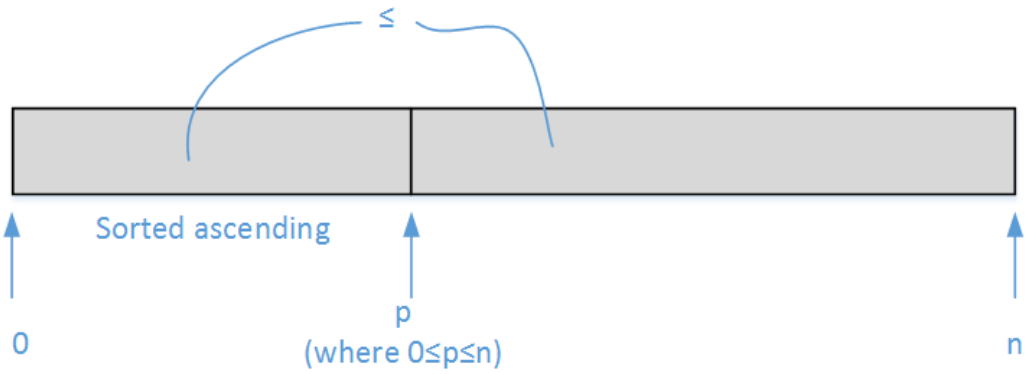
- $a[..]$ is the sequence of (the values of) all items.
- $a[..v]$ is the sequence of (the values of) the first v items of a.
- $a[u..]$ is the sequence of all but (the values of) the first u items of a.
- $a[u..v]$ is the sequence of (the values of) the items $a[i]$ where $u \leq i < v$.

Finding an invariant

So far so good. Now we need an invariant.

The idea of selection sort is that we partition the array into a left side and right side so that

- Every item on the left is $\leq$ every item on the right.
- The left side is sorted.



We also need that the array remains a permutation of its original value.

First let's make a predicate to express the idea that every item on the left side of a sequence is $\leq$ every item on the right side.

```
predicate Partitioned( s : seq<int>, p : int ) {
    forall i, j : int :: 0 <= i < p <= j < |s| ==> s[i] <= s[j] }
```

We are ready to state the invariant

```
invariant 0 <= p <= a.Length
invariant Sorted( a[0..p] )
invariant Partitioned( a[..], p )
invariant PermutationOf( a[..], old( a[..] ) )
```

In the last line **old**(a[..]) refers to the value of a[..] at the start of the method execution.

Writing selection sort

We can start writing the code now

```
method selectionSort( a : array<int> )
contract as above
{
    var p := 0 ;
    while( p != a.Length )
    invariant 0 <= p <= a.Length
    invariant Sorted( a[0..p] )
    invariant Partitioned( a[..], p )
    invariant PermutationOf( a[..], old( a[..] ) ) {
    }
}
```

Now for the loop body: Suppose we have a subroutine that can find the index in a of the smallest item in the segment a[p..a.Length]. Then we can write

```
while( p != a.Length )
invariant as above {
    // Let q be an index of the least value in the right part.
    var q := select(a, p) ;
    assert forall i :: p <= i < a.Length ==> a[q] <= a[i] ;
    a[p], a[q] := a[q], a[p] ;
    p := p+1 ; }
```

Note that a multiple assignment

$$a[p], a[q] := e, f$$

is problematic if $p = q$ and $e \neq f$. The verifier checks that this can not be the case.

A contract for select

The contract and stub implementation for select is

```
method select( a : array<int>, p : int ) returns ( q : int)
requires 0 <= p < a.Length
ensures p <= q < a.Length
ensures Least( a[p..], a[q] ) {
}
```

where Least is a predicate:

```
predicate Least( s : seq<int>, x : int )
{
    forall i :: 0 <= i < |s| ==> x <= s[i]
}
```

Before we implement select, we can verify selectionSort.

Note that the contract of select has no **modifies** clause.

- That means it has no side effects.
- In particular it can change no item of the array.
- The verifier uses this information when checking selectionSort's loop.
- If we were to add **modifies** a to select's contract, the verifier would be unable to verify, for example, that the left part of the array remains sorted when select is called.

Implementing select

The implementation of select is a straight-forward linear search, maintaining that a[q] is the least item in segment a[p..k]

```
method select( a : array<int>, p : int ) returns ( q : int )
requires 0 <= p < a.Length
ensures p <= q < a.Length
ensures Least( a[p..], a[q] )
{
    q := p ;
    var k := p + 1 ;
    while( k < a.Length )
        invariant 0 <= p <= q < k <= a.Length
        invariant Least( a[p..k], a[q] )
    {
        if( a[k] < a[q] ) { q := k ; }
        k := k + 1 ;
    }
}
```