# CS:5810
# Formal Methods in Software Engineering

## More Reasoning about Programs with Arrays in Dafny

# Modifying arrays

When a method modifies values accessible through reference parameters (and stored in the heap),

its specification must identify the relevant parts of the heap using *frames*

```
method SetEndPoints(a: array<int>, left: int, right: int)
    requires a.Length != 0
    modifies a
{
    a[0] := left;
    a[a.Length - 1] := right;
}
```

# Modifies clause

*If a method changes the elements of an array a given as a parameter, its specification must include* modifies a

```
method Aliases(a: array<int>, b: array<int>)
    requires 100 <= a.Length
    modifies a
{
    a[0] := 10;
    var c := a;
    if b == a {
        b[10] := b[0] + 1;   // ok since b == a
    }
    c[20] := a[14] + 2;   //ok since c == a
}
```

# Old

*The expression old(E) denotes the value of E on entry to the enclosing method.*

```
method UpdateElements(a: array<int>)
    requires a.Length == 10
    modifies a
    ensures old(a[4]) < a[4]
    ensures a[6] <= old(a[6])
    ensures a[8] == old(a[8])
{
    a[4], a[8] := a[4] + 3, a[8] + 1;
    a[7], a[8] := 516, a[8] - 1;
}
```

# Old

old affects only the heap dereferences in its argument

For example, in

```
method OldVsParameters(a: array<int>, i: int)
returns (y: int)
    requires 0 <= i < a.Length
    modifies a
    ensures old(a[i] + y) == 25
```

only a[i] is interpreted in the pre-state of the method

# New arrays

*A method is allowed to allocate a new array and change the elements of that array without mentioning this array in the modifies clause*

For example,

```
method NewArray() returns (a: array<int>)
    ensures a.Length == 20
{
    a := new int[20];
    var b := new int[30];
    a[6] := 216;
    b[7] := 343;
}
```

# Fresh arrays

```
method Caller() {
    var a := NewArray();
    a[8] := 512;    // error: modification of a not allowed
}
```

To fix error, strengthen specification of NewArray to

```
    method NewArray() returns (a: array<int>)
        ensures fresh(a) && a.Length == 20
```

# Reads clauses

*If a function accesses the elements of an input array a, its specification must include* reads a

```
function IsZeroArray(a: array<int>, lo: int, hi: int): bool
    requires 0 <= lo <= hi <= a.Length
    reads a
    decreases hi - lo
{
    lo == hi || (a[lo] == 0 && IsZeroArray(a, lo + 1, hi))
}
```

# Initializing an array

```
 method InitArray<T>(a: array<T>, d: T)
    modifies a
    ensures forall i :: 0 <= i < a.Length ==> a[i] == d
{
    var n := 0;
    while n != a.Length
        invariant 0 <= n <= a.Length
        invariant forall i :: 0 <= i < n ==> a[i] == d
 }
```

```
{ forall i :: 0 <= i < n + 1 ==> a[i] == d }
n := n + 1
{ forall i :: 0 <= i < n ==> a[i] == d }
```

# Initializing an array

```
 method InitArray<T>(a: array<T>, d: T)
     modifies a
     ensures forall i :: 0 <= i < a.Length ==> a[i] == d
{
     var n := 0;
     while n != a.Length
          invariant 0 <= n <= a.Length
          invariant forall i :: 0 <= i < n ==> a[i] == d
 }
```

```
{ (forall i :: 0 <= i < n ==> a[i] == d) && a[n] == d }
{ forall i :: 0 <= i < n + 1 ==> a[i] == d }
n := n + 1
{ forall i :: 0 <= i < n ==> a[i] == d }
```

# Initializing an array

```
 method InitArray<T>(a: array<T>, d: T)
    modifies a
    ensures forall i :: 0 <= i < a.Length ==> a[i] == d
{
    var n := 0;
    while n != a.Length
        invariant 0 <= n <= a.Length
        invariant forall i :: 0 <= i < n ==> a[i] == d
    {
        a[n] := d;
        n := n + 1;
    }
}
```

# Initializing a matrix

```
method InitMatrix<T>(a: array2<T>, d: T)
    modifies a
    ensures forall i,j :: 0 <= i < a.Length0 &&
                          0 <= j < a.Length1 ==> a[i,j] == d
```

We will need two loops, one nested in the other.

# Initializing a matrix

```
method InitMatrix<T>(a: array2<T>, d: T)
    modifies a
    ensures forall i,j :: 0 <= i < a.Length0 &&
                          0 <= j < a.Length1 ==> a[i,j] == d
{
    var m := 0;
    while m != a.Length0
        invariant 0 <= m <= a.Length0
        invariant forall i,j :: 0 <= i < m &&
                          0 <= j < a.Length1 ==> a[i,j] == d
}
```

Specification for outer loop (replaces a.Length0 by m)

# Initializing a matrix

These predicates form postcondition of inner loop.

```
{ (forall i,j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d) &&
(forall j :: 0 <= j < a.Length1 ==> a[m,j] == d)}

{ (forall i,j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d)
&& (forall i,j :: i == m && 0 <= j < a.Length1 ==> a[i,j] == d)}

{ forall i,j :: 0 <= i < m + 1 && 0 <= j < a.Length1 ==> a[i,j] == d }
m := m + 1;
{ forall i,j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d }
```

# The inner loop

```
{
    var n := 0;
    while n != a.Length1
        invariant 0 <= n <= a.Length1
        invariant forall i,j :: 0 <= i < m && 0 <= j < a.Length1
                                      ==> a[i,j] == d
        invariant forall j :: 0 <= j < n ==> a[m,j] == d
    {
        a[m,n] := d;
        n := n + 1;
    }
    m := m + 1;
}
```

Loop design technique 8.1

replacing a.Length1 by n

# Incrementing the values in an array

```
method IncrementArray(a: array<int>)
    modifies a
    ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
```

# Incrementing the values in an array

```
 method IncrementArray(a: array<int>)
     modifies a
     ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
     {
         var n := 0;
         while n != a.Length
             invariant 0 <= n <= a.Length
             invariant forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1
         {
             a[n] := a[n] + 1;
             n := n + 1;
         }  // error: second loop invariant not maintained
}
```

# Debugging the verification

```
a[n] := a[n] + 1;
n := n + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1; // error
```

# Debugging the verification

```
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1; // error
n := n + 1;
```

# Debugging the verification

```
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
assert a[n] == old(a[n]) + 1; // error
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
n := n + 1;
```

# Debugging the verification

```
assert a[n] + 1 == old(a[n]) + 1; // error
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
assert a[n] == old(a[n]) + 1;
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
n := n + 1;
```

# Debugging the verification

```
assert a[n] + 1 == old(a[n]) + 1; // error
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
assert a[n] == old(a[n]) + 1;
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
n := n + 1;
```

The verifier tells us that if we can assert the first condition then the verification succeeds.

Need to add invariant:

```
invariant forall i :: n <= i < a.Length ==> a[i] == old(a[i])
```

# Copying an array

```
 method CopyArray(src: array, dst: array)
     requires src.Length == dst.Length
     modifies dst
     ensures forall i ::
               0 <= i < src.Length ==> dst[i] == old(src[i])
{
     var n := 0;
     while n != src.Length
          invariant 0 <= n <= src.Length
          invariant forall i :: 0 <= i < n ==> dst[i] == old(src[i])
          invariant forall i ::
                    0 <= i < src.Length ==> src[i] == old(src[i])
     {  dst[n] := src[n]; n := n + 1;  }
}
```

# Selection sort

```
method SelectionSort(a: array<int>)
    modifies a
    ensures forall i,j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
```

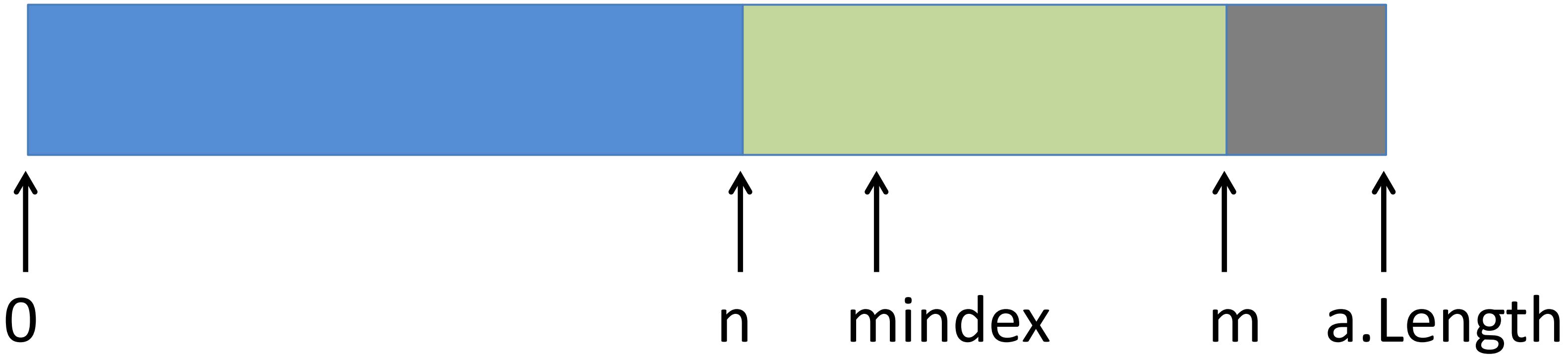
# Selection sort

```
method SelectionSort(a: array<int>)
    modifies a
    ensures forall i,j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
    ensures multiset(a[..]) == old(multiset(a[..]))
```

A multiset is like a set but may contain duplicate elements.

# Selection sort

```
method SelectionSort(a: array<int>)
    modifies a
    ensures forall i,j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
ensures multiset(a[..]) == old(multiset(a[..]))
```



# Implementation

replace constant a.Length in first postcondition with n

use second postcondition as invariant

```
{
    var n := 0;
    while n != a.Length
        invariant 0 <= n <= a.Length
        invariant forall i,j :: 0 <= i < j < n ==> a[i] <= a[j]
        invariant multiset(a[..]) == old(multiset(a[..]))
    …
}
```

# Inner loop

```
var mindex, m := n, n;
while m != a.Length
    invariant n <= m <= a.Length
              && n <= mindex < a.Length
    invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
{
    if a[m] < a[mindex] { mindex := m; }
    m := m + 1;
}
```

# Inner loop

```
var mindex, m := n, n + 1;
while m != a.Length
    invariant n <= mindex < m <= a.Length
    invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
{
    if a[m] < a[mindex] { mindex := m; }
    m := m + 1;
}
```

# Outer loop

```
{
    var mindex, m := n, n + 1;
    while m != a.Length
        invariant n <= mindex < m <= a.Length
        invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
    {
        if a[m] < a[mindex] { mindex := m; }
        m := m + 1;
    }
    a[n], a[mindex] := a[mindex], a[n];
    n := n + 1;                    // error
}
```

# Outer loop

```
{
    var mindex, m := n, n + 1;
    while m != a.Length
        invariant n <= mindex < m <= a.Length
        invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
    {
        if a[m] < a[mindex] { mindex := m; }
        m := m + 1;
    }
    a[n], a[mindex] := a[mindex], a[n];
    assert forall i,j :: 0 <= i < j < n ==> a[i] <= a[j];  // ok
    n := n + 1;
 }
```

# Outer loop

```
invariant forall i,j :: 0 <= i < n <= j < a.Length ==> a[i] <= a[j]
{
    var mindex, m := n, n + 1;
    while m != a.Length
        invariant n <= mindex < m <= a.Length
        invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
    {
        if a[m] < a[mindex] { mindex := m; }
        m := m + 1;
    }
    a[n], a[mindex] := a[mindex], a[n];
    assert forall i,j :: 0 <= i < j < n ==> a[i] <= a[j];  // ok
    n := n + 1;
}
```