Verification of Immutable Arrays

19CSE205 : PROGRAM REASONING

Dr. Swaminathan J

Assistant Professor

Department of Computer Science and Engineering

AMRITA VISHWA VIDYAPEETHAM

Jul - Dec 2020

Contents

Working with individual array elements

Swapping elements in an array.

File: arrayswap.c

```
/*@




*/
int arrayswap(int arr[], int n, int i, int j) {
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
}
```

Working with individual array elements

Swapping elements in an array.

File: arrayswap.c

```
/*@
    ensures arr[i] == \old(arr[j]);
    ensures arr[j] == \old(arr[i]);

*/
int arrayswap(int arr[], int n, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

Working with individual array elements

Swapping elements in an array.

File: arrayswap.c

```
/*@
    ensures arr[i] == \old(arr[j]);
    ensures arr[j] == \old(arr[i]);

*/
int arrayswap(int arr[], int n, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

```
[kernel] Parsing arrayswap.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals: 2 / 2
    Qed: 1 (0.58ms-2ms)
    Alt-Ergo: 1 (9ms) (12)
```

Try these variations

Note: This is a mutable example of arrays. The array's state changes.

Working with individual array elements

Swapping elements in an array.

File: arrayswap.c

```
/*@
    ensures arr[i] == \old(arr[j]);
    ensures arr[j] == \old(arr[i]);
    assigns arr[i], arr[j];
*/
int arrayswap(int arr[], int n, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

```
[kernel] Parsing arrayswap.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals: 2 / 2
    Qed: 1 (0.58ms-2ms)
    Alt-Ergo: 1 (9ms) (12)
```

Try these variations

1. Specify assigns for arr[i], arr[j]
2. Specify assigns for arr[i] only
3. Specify assigns \nothing

Note: This is a mutable example of arrays. The array's state changes.

Working with individual array elements

Swapping elements in an array.

File: arrayswap.c

```
/*@
    ensures arr[i] == \old(arr[j]);
    ensures arr[j] == \old(arr[i]);
    assigns arr[i], arr[j];
*/
int arrayswap(int arr[], int n, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

```
[kernel] Parsing arrayswap.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals: 2 / 2
    Qed: 1 (0.58ms-2ms)
    Alt-Ergo: 1 (9ms) (12)
```

Try these variations

1. Specify assigns for arr[i], arr[j]
2. Specify assigns for arr[i] only
3. Specify assigns \nothing
4. Run with -wp-rte option. What happens?

Note: This is a mutable example of arrays. The array's state changes.

Working with individual array elements

Swapping elements in an array.

File: arrayswap.c

```
/*@
    ensures arr[i] == \old(arr[j]);
    ensures arr[j] == \old(arr[i]);
    assigns arr[i], arr[j];
*/
int arrayswap(int arr[], int n, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

```
[kernel] Parsing arrayswap.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals: 2 / 2
     Qed: 1 (0.58ms-2ms)
     Alt-Ergo: 1 (9ms) (12)
```

Try these variations

1. Specify assigns for arr[i], arr[j]
2. Specify assigns for arr[i] only
3. Specify assigns \nothing
4. Run with -wp-rte option. What happens?
5. Try fixing it.

Note: This is a mutable example of arrays. The array's state changes.

ACSL constructs for arrays

We list few most used constructs

1. ensures \forall **integer** i; 0 <= i < n ==> a[i] == x; universal quantifier
2. ensures \exists **integer** i; 0 <= i < n && a[i] == x; existential quantifier
3. requires \valid_read(a + (0..n-1)); for immutable arrays
4. requires \valid(a + (0..n-1)); for mutable arrays

Few words of advice

1. Don't expect to get the specification right at the first attempt.
2. Follow the thumb rules and refine the loop invariant.
3. You have to fight errors on war footing. Don't resort to shortcuts.
4. Trying and not getting is far superior to not trying and copying.
5. Don't waste your chance to build confidence on your own abilities.
6. Seek help only after you have tried enough, not due to laziness.

Array traversal with single behavior

Finding the max element of an array.

```
/*@




*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@




    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

1. What are the constraints on n?
2. What are the constraints on array range?

Array traversal with single behavior

Finding the max element of an array.

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));


*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@





    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

3. What is the constraint on the return value?
4. How does it relate to elements of the array?

Array traversal with single behavior

Finding the max element of an array.

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    ensures \forall integer i;
        0 <= i < n ==> \result >= arr[i];
*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@




    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

5. What is the entry span of the loop?
6. What is the exit point of the loop?

Array traversal with single behavior

Finding the max element of an array.

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    ensures \forall integer i;
        0 <= i < n ==> \result >= arr[i];
*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@


      loop invariant 1 <= j <= n;


    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

7. How do we capture the progress made as iterations proceed?
$\rightarrow \forall i \in [0,j)$, max $\geq$ arr[i]

Array traversal with single behavior

Finding the max element of an array.

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    ensures \forall integer i;
        0 <= i < n ==> \result >= arr[i];
*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@
      loop invariant \forall integer i;
        0 <= i < j ==> max >= arr[i];
      loop invariant 1 <= j <= n;


    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

8. Which variables can be legally assigned within the loop?

Array traversal with single behavior

Finding the max element of an array.

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    ensures \forall integer i;
        0 <= i < n ==> \result >= arr[i];
*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@
      loop invariant \forall integer i;
        0 <= i < j ==> max >= arr[i];
      loop invariant 1 <= j <= n;
      loop assigns j, max;

    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

9. What must be the termination condition for the loop?

Array traversal with single behavior

Finding the max element of an array.

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    ensures \forall integer i;
        0 <= i < n ==> \result >= arr[i];
*/
int arraymax(int arr[], int n) {
    int max = arr[0];
    /*@
      loop invariant \forall integer i;
        0 <= i < j ==> max >= arr[i];
      loop invariant 1 <= j <= n;
      loop assigns j, max;
      loop variant n − j;
    */
    for (int j=1; j<n; j++) {
        if (arr[j] > max)
            max = arr[j];
    }
    return max;
}
```

Try these exercises

1. Find the index of the max element in an array.
2. Find the min element of an array.
3. Find sum of all elements of an array.
4. Reset each element of an array to 0.

Array traversal with two behaviors

Searching for an element in an array.

Function contract

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@






*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@





    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));









*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@





    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:


    behavior found:



*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@



    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:


        ensures \result == 0;
    behavior found:


        ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@





    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
   requires n > 0;
   requires \valid_read(arr + (0..n−1));
   behavior not_found:
      assumes \forall integer i;
         0 <= i < n ==> arr[i] ! = key;
      ensures \result == 0;
   behavior found:


      ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
   /*@





   */
   for (int i=0; i<n; i++) {
      if (arr[i] == key)
         return 1;
   }
   return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:
        assumes \forall integer i;
            0 <= i < n ==> arr[i] != key;
        ensures \result == 0;
    behavior found:
        assumes \exists integer i;
            0 <= i < n && arr[i] == key;
        ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@




    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:
        assumes \forall integer i;
            0 <= i < n ==> arr[i] != key;
        ensures \result == 0;
    behavior found:
        assumes \exists integer i;
            0 <= i < n && arr[i] == key;
        ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@



        loop assigns i;
        loop variant n − i;
    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:
        assumes \forall integer i;
            0 <= i < n ==> arr[i] ! = key;
        ensures \result == 0;
    behavior found:
        assumes \exists integer i;
            0 <= i < n && arr[i] == key;
        ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@


        loop invariant 0 <= i <= n;
        loop assigns i;
        loop variant n − i;
    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:
        assumes \forall integer i;
            0 <= i < n ==> arr[i] ! = key;
        ensures \result == 0;
    behavior found:
        assumes \exists integer i;
            0 <= i < n && arr[i] == key;
        ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@
        loop invariant \forall integer j;

        loop invariant 0 <= i <= n;
        loop assigns i;
        loop variant n − i;
    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
   requires n > 0;
   requires \valid_read(arr + (0..n−1));
   behavior not_found:
      assumes \forall integer i;
         0 <= i < n ==> arr[i] ! = key;
      ensures \result == 0;
   behavior found:
      assumes \exists integer i;
         0 <= i < n && arr[i] == key;
      ensures \result == 1;


*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
   /*@
      loop invariant \forall integer j;
         0 <= j < i ==> arr[j] != key;
      loop invariant 0 <= i <= n;
      loop assigns i;
      loop variant n − i;
   */
   for (int i=0; i<n; i++) {
      if (arr[i] == key)
         return 1;
   }
   return 0;
}
```

Array traversal with two behaviors

Searching for an element in an array.

Function contract

```
/*@
    requires n > 0;
    requires \valid_read(arr + (0..n−1));
    behavior not_found:
        assumes \forall integer i;
            0 <= i < n ==> arr[i] ! = key;
        ensures \result == 0;
    behavior found:
        assumes \exists integer i;
            0 <= i < n && arr[i] == key;
        ensures \result == 1;
    complete behaviors;
    disjoint behaviors;
*/
```

Function with loop invariant

```
int linearsearch(int arr[], int n, int key) {
    /*@
        loop invariant \forall integer j;
            0 <= j < i ==> arr[j] != key;
        loop invariant 0 <= i <= n;
        loop assigns i;
        loop variant n − i;
    */
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return 1;
    }
    return 0;
}
```

Try these exercises

1. Check if every element of an array is equal to its index.
2. Check if an array is sorted.
3. Compare if two arrays are of equal length.
4. Compare if two arrays are equal. i.e. all elements are same.

More readings

Skim through Chapter 4.2 of prescribed text book.

Allan Blanchard, "Introduction to C program proof with Frama-C and its WP plugin", 2020.

Solve exercises listed under Section 4.2.6.