Lecture Notes on Arrays

15-122: Principles of Imperative Computation
Frank Pfenning, André Platzer

Lecture 3
May 21, 2014

1 Introduction

So far we have seen how to process primitive data like integers in imperative programs. That is useful, but certainly not sufficient to handle bigger amounts of data. In many cases we need *aggregate data structures* which contain other data. A common data structure, in particular in imperative programming languages, is that of an *array*. An array can be used to store and process a fixed number of data elements that all have the same type.

We will also take a first detailed look at the issue of program *safety*. A program is *safe* if it will execute without exceptional conditions which would cause its execution to abort. So far, only division and modulus are potentially unsafe operations, since division or modulus by 0 is defined as a runtime error.[1] Trying to access an array element for which no space has been allocated is a second form of runtime error. Array accesses are therefore potentially unsafe operations and must be proved safe.

With respect to our learning goals we will look at the following notions.

Computational Thinking: Developing contracts that establish the safety of imperative programs.

Developing and evaluating proofs of the safety of code with contracts.

Programming: Identifying, describing, and effectively using arrays and `for`-loops.

[1] as is division of modulus of the minimal integer by -1

In lecture, we only discussed a smaller example of programming with arrays, so some of the material here is a slightly more complex illustration of how to use for loops and loop invariants when working with arrays.

2 Using Arrays

When t is a type, then `t[]` is the type of an array with elements of type t. Note that t is arbitrary: we can have an array of integers (`int[]`), and an array of booleans (`bool[]`) or an array of arrays of characters (`char[][]`). This syntax for the type of arrays is like Java, but is a minor departure from C, as we will see later in class.

Each array has a fixed size, and it must be explicitly allocated using the expression `alloc_array(t, n)`. Here t is the type of the array elements, and n is their number. With this operation, C0 will reserve a piece of memory with n elements, each having type t. Let's try in coin:

```
% coin
C0 interpreter (coin) 0.3.2 'Nickel'
Type '#help' for help or '#quit' to exit.
--> int[] A = alloc_array(int, 10);
A is 0x603A50 (int[] with 10 elements)
-->
```

The result may be surprising: A is an array of integers with 10 elements (obvious), but what does it mean to say A is `0xECE2FFF0` here? As we discussed in the lecture on integers, variables can only hold values of a small fixed size, the *word size* of the machine. An array of 10 integers would be 10 times this size, so we cannot hold it directly in the variable A. Instead, the variable A holds the *address* in memory where the actual array elements are stored. In this case, the address happens to be `0xECE2FFF0` (incidentally presented in hexadecimal notation), but there is no guarantee that the next time you run `coin` you will get the same address. Fortunately, this is okay because you cannot actually ever do anything directly with this address as a number and never need to either. Instead you access the array elements using the syntax `A[i]` where $0 \leq i < n$, where n is the length of the array. That is, `A[0]` will give you element 0 of the array, `A[1]` will be element 1, and so on. We say that arrays are *zero-based* because elements are numbered starting at 0. For example:

```
--> A[0];
```

```
0 (int)
--> A[1];
0 (int)
--> A[2];
0 (int)
--> A[10];
Error: accessing element 10 in 10-element array
Last position: <stdio>:1.1-1.6
--> A[-1];
Error: accessing negative element in 10-element array
Last position: <stdio>:1.1-1.6
-->
```

We notice that after allocating the array, all elements appear to be 0. This is guaranteed by the implementation, which initializes all array elements to a default value which depends on the type. The default value of type `int` is 0. Generally speaking, one should try to avoid exploiting implicit initialization because for a reader of the program it may not be clear if the initial values are important or not.

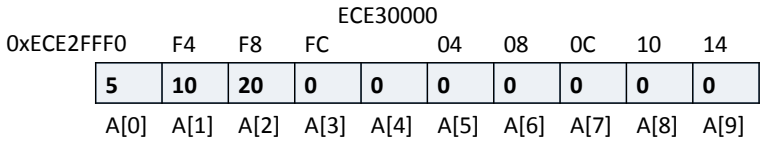
We also observe that trying to access an array element not in the specified range of the array will lead to an error. In this example, the valid accesses are `A[0]`, `A[1]`, . . ., `A[9]` (which comes to 10 elements); everything else is illegal. And every other attempt to access the contents of the array would not make much sense, because the array has been allocated to hold 10 elements. How could we ever meaningfully ask what it's element number 20 is if it has only 10? Nor would it make sense to ask `A[-4]`. In both cases, `coin` and `cc0` will give you an error message telling you that you have accessed the array outside the bounds. While an error is guaranteed in C0, in C no such guarantee is made. Accessing an array element that has not been allocated leads to *undefined behavior* and, in principle, anything could happen. This is highly problematic because implementations typically choose to just read from or write to the memory location where some element *would* be if it had been allocated. Since it has not been, some other unpredictable memory location may be altered, which permits infamous *buffer overflow attacks* which may compromise your machines.

How do we change an element of an array? We can use it on the left-hand side of an assignment. We can set `A[i] = e;` as long as e is an expression of the right type for an array element. For example:

```
--> A[0] = 5; A[1] = 10; A[2] = 20;
```

```
A[0] is 5 (int)
A[1] is 10 (int)
A[2] is 20 (int)
-->
```

After these assignments, the contents of memory might be displayed as follows, where $A =$ `0xECE2FFF0`:



Recall that an assignment (like `A[0] = 5;`) is a statement and as such has an effect, but no value. `coin` will print back the effect of the assignment. Here we have given three statements together, so all three effects are shown. Again, exceeding the array bounds will result in an error message and the program aborts, because it does not make sense to store data in an array at a position that is outside the size of that array.

```
--> A[10] = 100;
Error: accessing element 10 in 10-element array
Last position: <stdio>:1.1-1.6
-->
```

3 Using For-Loops to Traverse Arrays

A common pattern of access and traversal of arrays is for-loops, where an index i is counted up from 0 to the length of the array. To continue the example above, we can assign i^3 to the ith element of the array as follows:

```
--> for (int i = 0; i < 10; i++)
... A[i] = i * i * i;
--> A[6];
216 (int)
-->
```

Characteristically, the exit condition of the loop tests for `i < n` where i is the array index and n is the length of the array (here 10).

After we type in the first line (the header of the for-loop), `coin` responds with the prompt `...` instead of `-->`. This indicates that the expression or statement it has parsed so far is incomplete. We complete it by supplying the body of the loop, the assignment `A[i] = i * i * i;`. Note that no assignment effect is printed. This is because the assignment is part of a loop. In general, `coin` will only print effects of top-level statements such as assignments, because when a complicated program is executed, a huge number of effects could be taking place.

4 Specifications for Arrays

When we use loops to traverse arrays, we need to make sure that all the array accesses are in bounds. In many cases this is evident, but it can be tricky in particular if we have two-dimensional data (for example, images). As an aid to this reasoning, we state an explicit loop invariant which expresses what will be true on every iteration of the loop.

To illustrate arrays, we develop a function that computes an array of the first n Fibonacci numbers, starting to count from 0. It uses the recurrence:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_{n+2} &= f_{n+1} + f_n \quad \text{for } n \geq 0 \end{aligned}$$

When we represent f_n in an array as $A[n]$, we can write the recurrence directly as a loop operating on the array:

```
int[] fib(int n) {
  int[] F = alloc_array(int, n);
  F[0] = 0;
  F[1] = 1;
  for (int i = 0; i < n; i++)
    F[i+2] = F[i+1] + F[i];
  return F;
}
```

This looks straightforward. Is there a problem with the code or will it run correctly? In order to understand whether this function works correctly, we systematically develop a specification for it. Before you read on, can you spot a bug in the code? Or can you find a reason why it will work correctly?

Allocating an array will also fail if we ask for a negative number of elements. Since the number of elements we ask for in `alloc_array(int, n)` is n, and n is a parameter passed to the function, we need to add $n \geq 0$ into the precondition of the function. In return, the function can safely promise to return an array that has exactly the size n. This is a property that the code using, e.g., `fib(10)` has to rely on. Unless the `fib` function promises to return an array of a specific size, the user has no way of knowing how many elements in the array can be accessed safely without exceeding its bounds. Without such a corresponding postcondition, code calling `fib(10)` could not even safely access position 0 of the array that `fib(10)` returns.

For referring to the length of an array, C0 contracts have a special function `\length(A)` that stands for the number of elements in the array `A`. Just like the `\result` variable, the function `\length` is part of the contract language and cannot be used in C0 program code. Its purpose is to be used in contracts to specify the requirements and behavior of a program. For the Fibonacci function, we want to specify the postcondition that the length of the array that the function returns is n.

```
int[] fib(int n)
//@requires n >= 0;
//@ensures \length(\result) == n;
{
  int[] F = alloc_array(int, n);
  F[0] = 0;
  F[1] = 1;
  for (int i = 0; i < n; i++) {
      F[i+2] = F[i+1] + F[i];
    }
  return F;
}
```

5 Loop Invariants for Arrays

By writing specifications, we should convince ourselves that all array accesses will be within the bounds. In the loop, we access $F[i]$, which would raise an error if i were negative, because that would violate the lower bounds of the array. So we need to specify a loop invariant that ensures $i \geq 0$.

```
int[] fib(int n)
//@requires n >= 0;
//@ensures \length(\result) == n;
{
  int[] F = alloc_array(int, n);
  F[0] = 0;
  F[1] = 1;
  for (int i = 0; i < n; i++)
    //@loop_invariant i >= 0;
    {
      F[i+2] = F[i+1] + F[i];
    }
  return F;
}
```

Clearly, if $i \geq 0$ then the other array accesses `F[i+1]` and `F[i+2]` also will not violate the lower bounds of the array, because $i + 1 \geq 0$ and $i + 2 \geq 0$. Will the program work correctly now?

The big issue with the code is that, even though the code ensures that no array access exceeds the lower bound 0 of the array F, we do not know whether the upper bounds of the array i.e., `\length(F)`, which equals n, is always respected. For each array access, we need to ensure that it is within the bounds. In particular, we need to ensure $i < n$ for array access `F[i]` and the condition $i + 1 < n$ for array access `F[i+1]` and the condition $i + 2 < n$ for `F[i+2]`. But this condition does not work out, because the loop body also runs when $i = n - 1$, at which point $i + 2 = (n - 1) + 2 = n + 1 < n$ does not hold, because we have allocated array F to have size n.

We can also easily observe this bug by using `coin`.

```
% coin fibc.c0 -d
Coin 0.2.9 'Penny'(r10, Fri Jan  6 22:08:54 EST 2012)
Type ‘#help’ for help or ‘#quit’ to exit.
--> fib(5);
Error: accessing element 5 in 5-element array
Last position: fibc.c0:11.7-11.30
               fib from <stdio>:1.1-1.7
```

Consequently, we need to stop the loop earlier and can only continue as long as $i + 2 < n$. Since the loop condition in a for loop can be any boolean expression, we could trivially ensure this by changing the loop as follows:

```
int[] fib(int n)
//@requires n >= 0;
//@ensures \length(\result) == n;
{
  int[] F = alloc_array(int, n);
  F[0] = 0;
  F[1] = 1;
  for (int i = 0; i+2 < n; i++)
    //@loop_invariant i >= 0;
    {
      F[i+2] = F[i+1] + F[i];
    }
  return F;
}
```

Since it can be more convenient to see the exact bounds of a for loop, we can replace the loop condition $i + 2 < n$ by $i < n - 2$ since both are equivalent. It does not make much difference, which one we use, but the latter can be more intuitive to determine how long a loop iterates to complete.

```
int[] fib(int n)
//@requires n >= 0;
//@ensures \length(\result) == n;
{
  int[] F = alloc_array(int, n);
  F[0] = 0;
  F[1] = 1;
  for (int i = 0; i < n-2; i++)
    //@loop_invariant i >= 0;
    {
      F[i+2] = F[i+1] + F[i];
    }
  return F;
}
```

This program looks good and will behave well after a number of tests. Is it correct? Before you read on, find an answer yourself.

When we verify the previous program, we suddenly realize that there are two array accesses for which we have not yet convinced ourselves that they will access within bounds. The two array accesses `F[0]` and `F[1]` before the loop. And in fact, they may fail when we run coin.

We can also easily exhibit this bug with coin on either `fibe.c0` or `fibd.c0`

```
% coin fibe.c0 -d
Coin 0.2.9 'Penny'(r10, Fri Jan  6 22:08:54 EST 2012)
Type '#help' for help or '#quit' to exit.
--> fib(5);
0xFF4FF780 (int[] with 5 elements)
--> fib(2);
0xFF4FF760 (int[] with 2 elements)
--> fib(1);
Error: accessing element 1 in 1-element array
Last position: fibe.c0:7.3-7.12
               fib from <stdio>:1.1-1.7
--> fib(0);
Error: accessing element 0 in 0-element array
Last position: fibe.c0:6.3-6.12
               fib from <stdio>:1.1-1.7
-->
```

To solve this issue we add tests that only run them if the array is big enough to contain that entry.

See `fibf.c0`

```
int[] fib(int n)
//@requires n >= 0;
//@ensures \length(\result) == n;
{
  int[] F = alloc_array(int, n);
  if (n > 0) F[0] = 0;             /* line 0 */
  if (n > 1) F[1] = 1;             /* line 1 */
  for (int i = 0; i < n-2; i++)
    //@loop_invariant i >= 0;
    {
      F[i+2] = F[i+1] + F[i];      /* line 2 */
    }
  return F;
}
```

6 Proving Correctness: Loop Invariants

The loop invariant states a property that must be true **just before the exit condition is tested**. Variable i is initialized to 0 with `i = 0` when the for loop begins. Clearly, i is incremented each time around the loop (with the step statement `i++` which is the same as `i = i+1`), so i will always be greater or equal to 0. Let us prove this precisely.

Init: When we enter the loop for the first time, the for loop initialization assigns i = 0 so $i \geq 0$.

Preservation: Assume that $i \geq 0$ (the loop invariant) when we enter the loop, we have to show it still holds after we traverse the loop body once. We obtain the next value of the loop by executing `i = i+1` so the new value of i, written i', will only be bigger, so it must still be greater of equal to 0, thus $i' = i + 1 \geq 0$.

A subtle point: we are in two's complement arithmetic, but $i + 1$ cannot overflow since i is bounded from above by $n - 2$.

7 Proving Correctness: Array Bounds

Now we have verified the loop invariant but still need to verify that all array accesses are guaranteed to be in bounds, otherwise the program still would not run correctly.

1. In line 0 we assign to `F[0]`. If the length of the array F (which is n) were 0, this would be out of bounds. But we check that $n > 0$ in the if statement so that the assignment only takes place if there is at least one element in the array, labeled `F[0]`.

2. Similarly, in line 1 we access `F[1]`, but this is okay because we only access it if $n > 1$.

3. In line 2, we access `F[i+2]`, `F[i+1]` and `F[i]`. By the loop invariant we know that $i + 2$, $i + 1$, and i are all greater than or equal to 0, because $i \geq 0$. Since we only enter the loop body if the loop condition $i < n - 2$ holds, and n is the length of array, we also know that $i + 2$ is less than the length of the array (and so are $i + 1$ and i because they are only smaller). So all three accesses must always be in bounds.

In the last case, we do not reason about how the loop operates but rely solely on the loop invariant and loop condition instead. This is crucial, since the loop invariant and condition are supposed to contain all the relevant information about the relevant effect of the loop. In particular, our reasoning about the array accesses does not depend on understanding what exactly the loop does after, say, 5 iterations or where i started and how it evolved since. All that matters is whether we can conclude from the loop invariant $i \geq 0$ and the loop condition $i < n - 2$ that the array accesses are okay. In this way, loop invariants have the effect of entirely localizing our reasoning to one general scenario to consider for the loop body. This is how loop invariants can greatly simplify our understanding of programs and ensure we have implemented them correctly.

Similar effects occur in other scenarios where our understanding of the behavior of loops becomes entirely focused on a local question of a single behavior just by virtue of being able to reason from the loop invariant. Needless to say, before we use a loop invariant in our reasoning about the behavior of the code, we should convince ourselves that the loop invariant is correct by a proof.

8 Aliasing

We have seen assignments to array elements, such as `A[0] = 0;`. But we have also seen assignments to array variables themselves, such as

```
int[] A = alloc_array(int, n);
```

What do they mean? To explore this, we separate the declaration of array variables (here: F and G) from assignments to them.

```
% coin -d fibf.c0
Coin 0.2.9 'Penny'(r10, Fri Jan  6 22:08:54 EST 2012)
Type '#help' for help or '#quit' to exit.
--> int[] F;
--> int[] G;
--> F = fib(15);
F is 0xF6969A80 (int[] with 15 elements)
--> G[2];
Error: uninitialized value used
Last position: <stdio>:1.1-1.5
--> G = F;
```

```
G is 0xF6969A80 (int[] with 15 elements)
--> G = fib(10);
G is 0xF6969A30 (int[] with 10 elements)
-->
```

The first assignment to F is as expected: it is the address of an array of Fibonacci numbers with 15 elements. The use of G in `G[2]`, of course, cannot succeed, because we have only declared G to have a type of integer arrays, but did not assign any array to G.

Afterwards, however, when we assign `G = F`, then G and F (as variables) *hold the same address*! Holding the same address means that F and G are *aliased*. When we make the second assignment to G (changing its value) we get a new array, which is in fact smaller and definitely no longer aliased to F (note the different address). Aliasing (or the lack thereof) is crucial, because modifying one of two aliased arrays will also change the other. For example:

```
% coin
Coin 0.2.9 'Penny'(r10, Fri Jan  6 22:08:54 EST 2012)
Type `#help' for help or `#quit' to exit.
--> int[] A = alloc_array(int, 5);
A is 0xE8176FF0 (int[] with 5 elements)
--> int[] B = A;
B is 0xE8176FF0 (int[] with 5 elements)
--> A[0] = 42;
A[0] is 42 (int)
--> B[0];
42 (int)
-->
```

C0 has no built-in way to copy from one array to another (ultimately we will see that there are multiple meaningful ways to copy arrays of more complicated types). Here is a simple function to copy arrays of integers.

```
/* file copy.c0 */
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);
```

```
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i;
    B[i] = A[i];
  return B;
}
```

For example, we can create B as a copy of A, and now assigning to the copy of B will not affect A. We will invoke coin with the `-d` flag to make sure that if a pre- or post-condition or loop invariant is violated we get an error message.

```
% coin copy.c0 -d
Coin 0.2.9 'Penny'(r10, Fri Jan  6 22:08:54 EST 2012)
Type `#help' for help or `#quit' to exit.
--> int[] A = alloc_array(int, 10);
A is 0xF3B8DFF0 (int[] with 10 elements)
--> for (int i = 0; i < 10; i++) A[i] = i*i;
--> int[] B = array_copy(A, 10);
B is 0xF3B8DFB0 (int[] with 10 elements)
--> B[9];
81 (int)
--> A[9] = 17;
A[9] is 17 (int)
--> B[9];
81 (int)
-->
```

Exercises

Exercise 1 *Write a function* `array_part` *that creates a copy of a part of a given array, namely the elements from position i to position j. Your function should have prototype*

```
int[] array_part(int[] A, int i, int j);
```

Develop a specification and loop invariants for this function. Prove that it works correctly by checking the loop invariant and proving array bounds.

Exercise 2 *Write a function* `copy_into` *that copies a part of a given array $source$, namely n elements starting at position i, into another given array $target$, starting at position j. Your function should have prototype*

```
int copy_into(int[] source, int i, int n, int[] target, int j);
```

As an extra service, make your function return the last position in the target array that it entered data into. Develop a specification and loop invariants for this function. Prove that it works correctly by checking the loop invariant and proving array bounds. What is difficult about this case?

Exercise 3 *Write a function* `can_copy_into` *that returns an integer indicating how many elements, starting from position i, of an array $source$ of a given length n can be copied safely into a part of a given array $target$ starting at position j, into another given array, starting at position j. Your function should have prototype*

```
int can_copy_into(int[] source, int i, int[] target, int j, int n);
```

Develop a specification and loop invariants for this function. Prove that it works correctly by checking the loop invariant and proving array bounds. The number returned by `can_copy_into` *should be compatible with the specification of* `copy_into`*. Which calls to* `copy_into` *are guaranteed to work correctly after a call of*

```
int r = can_copy_into(source, i, target, j, n);
```

Exercise 4 *Can you develop a reasonable (non-degenerate) and useful function with the following prototype? Discuss.*

```
int f(int[] A);
```