Lecture 3
Arrays

15-122: Principles of Imperative Computation (Fall 2018)
Frank Pfenning, André Platzer

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 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

[1] Another runtime error is for division or modulus of the smallest integer by -1.

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) 0.3.3 'Nickel'
Type '#help' for help or '#quit' to exit.
--> int[] A = alloc_array(int, 10);
A is 0xECE2FFF0 (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
-->
```

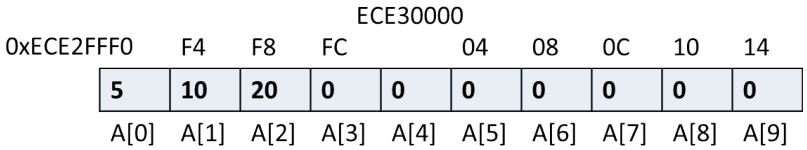
We notice that after allocating the array, all elements appear to be 0. This is guaranteed by the implementation, which initializes all array elements to a default value which depends on the type. The default value of type **`int`** is 0. Generally speaking, one should try to avoid exploiting implicit 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 its 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
-->
```

2 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 i-th 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.

3 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 will expand on our previous example, filling an array with cubes.

```
1 int[] cubes(int n) {
2   int[] A = alloc_array(int, n);
3
4   for (int i = 0; i < n; i++) {
5     A[i] = i * i * i;
6   }
7
8   return A;
9 }
```

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.

The first problem is the safety of the call to **`alloc_array`**, because allocating an array will 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.

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 cubes function, we want to specify the post-condition that the length of the array that the function returns is n.

```
1 int[] cubes(int n)
2 //@requires n >= 0;
3 //@ensures \length(\result) == n;
4 {
5   int[] A = alloc_array(int, n);
6
```

```
7   for (int i = 0; i < n; i++) {
8     A[i] = i * i * i;
9   }
10
11  return A;
12 }
```

4 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 `A[i]`, which would raise an error if `i` were negative or greater than `\length(A)`, because that would violate the bounds of the array.

Because `n` is not modified by the loop, we can use our knowledge that `\length(A) == n` from `A`'s declaration in conjunction with the loop guard `i < n` to conclude that `i` does not violate the upper bound (i.e., that `i < \length(A)` in each iteration of the loop). For the lower bound, we need to specify a loop invariant that ensures $i \geq 0$.

```
7   for (int i = 0; i < n; i++)
8   //@loop_invariant 0 <= i;
9   {
10    A[i] = i * i * i;
11  }
```

Operationally, of course, we can reason that because `i` starts at 0 and only increments on every iteration, `i` can't ever be negative. But in this course we eschew such operational reasoning, instead encoding this information in loop invariant. We know that the loop invariant is true initially by the for loop's declaration: `i` is initially 0, and $0 \leq 0$. We furthermore know that, in an arbitrary iteration of the loop that initially $i < n$ by the loop guard, so $i' = i + 1$ cannot overflow to a negative number and the loop invariant is always preserved.

5 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 cubes.c0
C0 interpreter (coin) 0.3.3 'Nickel'
Type '#help' for help or '#quit' to exit.
--> int[] F;
--> int[] G;
--> F = cubes(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 = cubes(10);
G is 0xF6969A30 (int[] with 10 elements)
-->
```

The first assignment to `F` is as expected: it is the address of an array 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`.

Afterward, however, when we assign `G = F`, then `G` and `F` (as locals) *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
C0 interpreter (coin) 0.3.3 'Nickel'
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.

```
1  int[] array_copy(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures \length(\result) == n;
4  {
5    int[] B = alloc_array(int, n);
6
7    for (int i = 0; i < n; i++)
8    //@loop_invariant 0 <= i;
9    {
10     B[i] = A[i];
11   }
12
13   return B;
14 }
```

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
C0 interpreter (coin) 0.3.3 'Nickel'
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)
-->
```

6 Implementation Note

Internally, arrays are stored in the area of the memory called the *heap*. Memory on the heap is allocated with **`alloc_array`**, which returns the address of an array (and later in this course **`alloc`** which returns a pointer). In C0, memory is not explicitly deallocated, but it is *garbage collected* in the sense that memory that can no longer be accessed from within the running program is freed so that it can be used to satisfy future allocation requests.

In order to check whether array accesses are in bounds, the C0 runtime system must store not only the array data, but also the length of the array. In the running program, this information cannot be accessed directly: given an array we cannot obtain its length. This is mostly in order to simulate safe programming practices in C. For example, when arrays are passed as arguments to functions we usually also pass (a bound on) their length.

However, in *contracts* (that is, function preconditions `@requires`, postconditions `@ensures`, loop invariants `@loop_invariant` and C0 assertions `@assert`) we can refer to the length of an array using the special function `\length`. We have already used this in the examples above. For example, the copy function

```
int[] array_copy(int[] A, int n)
 //@requires 0 <= n && n <= \length(A);
 //@ensures \length(\result) == n;
 ;
```

requires `n` to be smaller than the length of the parameter array `A` and ensures that the result array will have length `n`.

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

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