Lecture Notes on Unbounded Arrays

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 6
September 9, 2010

1 Introduction

Most lectures so far had topics related to all three major categories of learning goals for the course: computational thinking, algorithms, and programming. The same is true for this lecture. With respect to algorithms, we introduce *unbounded arrays* and operations on them. Analyzing them requires *amortized analysis*, a particular way to reason about sequences of operations on data structures. We also briefly talk about *data structure invariants* and *interfaces*, which are crucial computational thinking concepts. To implement unbounded arrays, we will need *structs*, which provide a way to aggregate data of different type, and *pointers*, which allow us to refer to structs allocated in memory.

2 Unbounded Arrays

In the second homework assignment, you were asked to read in some files such as the *Collected Works of Shakespeare* or the *Scrabble Players Dictionary*. What kind of data structure do we want to use when we read the file? In later parts of the assignment we want to look up words, perhaps sort them, so it is natural to want to use an array of strings, each string constituting a word. A problem is that before we start reading we don't know how many words there will be in the file so we cannot allocate an array of the right size! One solution uses either a queue or a stack. We will discuss these in the next two lectures. Another solution is to use an *unbounded array*. While

arrays are a language primitive, unbounded arrays are a *data structure* that we need to implement.

Thinking about it abstractly, an unbounded array should be like an array in the sense that we can get and set the value of an arbitrary element via its index i. We should also be able to add a new element to the end of the array, and delete an element from the end of the array.

We use the unbounded array by creating an empty one before reading a file. Then we read words from the file, one by one, and add them to the end of the unbounded array. Meanwhile we can count the number of elements to know how many words we have read. We trust the data structure not to run out of space unless we hit some hard memory limit, which is unlikely for the kind of task we have in mind, given modern operating systems. When we have read the whole file the words will be in the unbounded array, in order, the first word at index 0, the second at index 1, etc.

The general implementation strategy is as follows. We maintain an array of a fixed length $limit$ and an internal index $size$ which tracks how many elements are actually in the array. When we add a new element we increment $size$, when we remove an element we decrement $size$. The tricky issue is how to proceed when we are already at the limit and want to add another element. At that point, we allocate a new array with a larger limit and copy the elements we already have to the new array. For reasons we explain later in this lecture, every time we need to enlarge the array we *double* its size. Removing and element from the end is simple: we just decrement $size$. There are some issues to consider if we want to shrink the array, but this is optional.

3 Structs

So far we have seen the primitive types of `int` and `bool`. We have also used the types `string` (for strings) and `char` (for ASCII characters). The only compound type we have seen so far is `t[]` for an array of elements of type t.

According to the above implementation sketch, an unbounded array needs to track three forms of data: an integer $limit$, an integer $size$ and an array of strings. We can put these together in a so-called `struct`. We call it `struct ubarray`. We refer to $limit$, $size$ and $strings$ as the *fields* of the struct. It is declared with

```
struct ubarray {
  int limit;                    /* limit > 0 */
  int size;                     /* 0 <= size && size <= limit */
  string[] strings;             /* \length(strings) == limit */
};
```

The general form of a struct definition is

```
struct s {
  t1 f1;
  ...
  tn fn;
};
```

for a struct named s with field $f_1, \ldots, f_n$ of types $t_1, \ldots, t_n$, respectively. Struct names s occupy their own name space and cannot conflict with type names, variable names, or function names. Similarly, field names $f_1, \ldots, f_n$ do not conflict with struct names, type names, variable names, or function names.

4 References and Pointers

In order to explain how members of a struct are accessed, we digress briefly to introduce *pointers*, although we will use them in only a very limited form in this lecture. C0 distinguishes between *small types* and *large types*. Small types have values that can be stored in variables and passed to and from functions. Large types can only be stored in memory. In order to access them we pass *references* or *pointers* to them. Alternatively, we can think of working with their *address* in memory, rather than the values themselves.

Which types are small and large so far? It seems pretty clear that `int`, `bool`, and `char` are explicitly designed to be small. The natural size of a value of type `int` on recent architectures is 32 or 64, although in previous generations of processors it was 8 or 16. In C0 we fix them to be 32. There are only two booleans, so we might expect them to be 1 bit. The processor architecture, however, has a natural word size which is handled most efficiently, so they may actually be implemented to take more space. Similarly, ASCII character values of type `char` should take 7 or 8 bits, although in reality an implementation might allocate more space for them.

The type of arrays is an interesting case. We might expect `t[]` to be a large type, since each array takes a fixed, but unbounded amount of space.

But we have been passing them as arguments and assigning them to variables without any problems. The reason is that a value of type `t[]` is actually a *reference* to an array that is stored in memory. Such a reference fits within the word size of the machine, since addresses are a basic type that the machine can manipulate. On most recent architectures, an address will take either 32 or 64 bits, depending on the configuration of the compiler and machine. The current environment in use for this class uses 64 bits, although there is no effective way to tell the difference from within a program.

In summary, arrays with elements of type t and length n are allocated in memory, with `alloc_array(t,n)`. Such an allocation returns a *reference* to the array. When two variables of type `t[]` are the *same* reference we say that they *alias*. As a programmer, it is important to be aware of this because assignment to an array variable does *not* copy the contents of the array, but only assigns references. For example, after the operations

```
int[] A = alloc_array(int,5);
int[] B = alloc_array(int,7);
A[2] = 37;
B = A;
B[2] = 11;
```

A and B alias, and we have `A[2] == 11` and also `\length(B) == 5`. In fact, the allocation of B is redundant, and we could just as well have written

```
int[] A = alloc_array(int,5);
int[] B;
A[2] = 37;
B = A;
B[2] = 11;
```

Besides arrays, strings are also manipulated by reference and therefore `string` is a small type. In contrast, structs are *large* types. This means that they are allocated in memory and can *not* be stored in variables or passed to functions. Instead of references, as for arrays, we use explicit *pointers* when passing them to or from functions or assigning them to variables. In fact, because `struct s` is a large type it is an error to try to pass it to a function or directly declare a variable of such a type.

In general, we write `t*` for a pointer to a value of type t in memory. In this lecture we only use it in the form `struct s*`, that is, a pointer to a struct in memory. Structs (and other values) are allocated in memory using

the form `alloc(t)` for a type t. This allocation returns a pointer to the new memory location, and therefore has type `t*`.

If we have a pointer p of type `struct s*` we refer to the field f of the struct using the notation `p->f`. To write to memory we use it on the left-hand side of an assignment, to read from memory with use it as an expression.

We will see these language constructs in action in the next section where we describe the implementation of unbounded arrays.

5 Implementing Unbounded Arrays

As sketched above, an unbounded arrays tracks a limit, a current size, and an underlying (bounded) array.

```
struct ubarray {
  int limit;                    /* limit > 0 */
  int size;                     /* 0 <= size && size <= limit */
  string[] strings;             /* \length(strings) == limit */
};
```

There are some *data structure invariants* that we maintain, although they may be temporarily violated as the elements of the structure are manipulated at a low level. Generally, when we pass a pointer to the data structure or assign it to a variable we expect these invariants to hold. C0, however, has no intrinsic support for ensuring these invariants. Instead, our method is to define a function to test them and then verify adherence to the invariants in contracts as well as loop invariants and assertions. Here, the function `is_uba` serves that purpose.

As a general idiom, when we use structs, we define a new type to stand for a pointer to the struct. This is because most of the time we work with pointers to structs rather than structs themselves. Here we call this new type `uba`.

```
typedef struct ubarray* uba;

bool is_uba (uba L)
//@requires L->limit == \length(L->strings);
{
  return L->limit > 0 && 0 <= L->size && L->size <= L->limit;
}
```

Note that consistency between the `L->limit` and the length of `L->strings` can only be tested in a precondition (or explicit `@assert`) since `\length` can only appear in annotations.

To create a new unbounded array, we allocate a `struct ubarray` and an array of a supplied initial limit.

```
uba uba_new (int initial_limit)
//@requires 1 <= initial_limit;
//@ensures is_uba(\result);
{ assert(1 <= initial_limit,
         "unbounded array must be of initial limit 1 or more");
  { uba L = alloc(struct ubarray);
    string[] Ls = alloc_array(string, initial_limit);
    L->limit = initial_limit;
    L->strings = Ls;
    return L;
  }
}
```

We ascertain in the postcondition that the result adheres to the data structure invariants. Since we are implementing a general-purpose data structure, we do not trust that clients will necessarily adhere to the contracts. We therefore check explicitly that the initial limit is strictly positive and signal an error if it is not. This is a common use of explicit `assert` statements. They will *always* be checked at runtime, while `@assert` annotations will not: they are only verified when dynamic checking of contracts is enabled.

Getting and setting an element of an unbounded array is straightforward. However, we do have to verify that the array access is *in bounds*. This is stricter than checking that it is within the allocated array (below `limit`), because everything beyond the current size should be considered to be *undefined*. These array elements have not yet been added to the array, so reading or writing them is meaningless. We show only the operation of writing to an unbounded array, `uba_set`.

```
void uba_set(uba L, int index, string s)
//@requires is_uba(L);
//@requires 0 <= index && index < L->size;
{ assert (0 <= index && index < L->size,
          "unbounded array index out of bounds");
  L->strings[index] = s;
}
```

More interesting is the operation of adding an element to the end of an unbounded array. For that we need a function to resize an unbounded array. This function takes an unbounded array L and a new limit `new_limit`. It is required that the new limit is strictly greater than the current size, to make sure we have enough room to preserve all current elements and one more for the next one to add. We also stipulate that the size does not change by stating `L->size == \old(L->size)` in the postcondition.

```
void uba_resize(uba L, int new_limit)
//@requires is_uba(L);
//@requires new_limit > L->size;
//@ensures is_uba(L);
//@ensures L->limit == new_limit && L->size == \old(L->size);
//@ensures L->size < L->limit;
{
  string[] Ks = alloc_array(string, new_limit);
  int i;
  for (i = 0; i < L->size; i++)
    Ks[i] = L->strings[i];
  // L->size remains unchanged
  L->limit = new_limit;
  L->strings = Ks;
}
```

Finally we are ready to write the function that adds an element to the end of an unbounded array. We first check whether there is room for another element and, if not, double the size of the underlying array of strings. The contract just states that the array is valid before and after the operation.

```
void addend(uba L, string s)
//@requires is_uba(L);
//@ensures is_uba(L);
{
  if (L->size == L->limit) uba_resize(L, 2*L->limit);
  //@assert L->size < L->limit;
  L->strings[L->size] = s;
  L->size++;
}
```

We discuss the function that removes an element from an array later in this lecture.

6 Amortized Analysis

It is easy to see that reading from or writing to an unbounded array at a given index is a constant-time operation. However, adding an element to an array is not. Most of the time it takes constant time $O(1)$, but when we have run out of space it take times $O(\mathsf{size})$ because we have to copy the old elements to the new underlying array. On the other hand, it doesn't seem to happen very often. Can we characterize this situation more precisely? This is the subject of *amortized analysis*. Calling the operation to add a new element to an unbounded array an *insert*, we claim:

The cost of n insert operations into an unbounded array is $O(n)$.

How do we prove that? Let's say a simple insert takes c steps. Because we are only concerned with asymptotic analysis we can count this as 1 operation. Similarly, we count the act of copying one element from one array to another as 1 operation. Now performing a sequence of inserts, starting with an empty array of, say, size 4 looks as follows.

call	op's	*size*	*limit*
`addend(L,"a")`	1	1	4
`addend(L,"b")`	1	2	4
`addend(L,"c")`	1	3	4
`addend(L,"d")`	1	4	4
`addend(L,"e")`	5	5	8
`addend(L,"f")`	1	6	8
`addend(L,"g")`	1	7	8
`addend(L,"h")`	1	8	8
`addend(L,"i")`	9	9	16

We have taken 4 extra operations when inserting `"e"` in order to copy `"a"` through `"d"`. Overall, we have performed 21 operations for inserting 9 elements. Would that be $O(n)$ by the time we had inserted n elements?

The idea is to redistribute the cost of the expensive calls and "charge" them to the cheap calls. Another way to look at it is that during a cheap call we put a constant number of operations "in the bank", saving them to withdraw later when we come to an expensive call. If we never need more operations than we have saved in the bank, we have constant amortized time per call to `addend`.

We call the operations we save *tokens*. So a token is like an operation we could have performed and remained in constant time but didn't. Therefore

we can perform one operation per saved token later without violating the linear-time bound for the whole sequence of operations.

In the sequence above, let's save one token for every inexpensive call, and withdraw them when the expensive call arises.

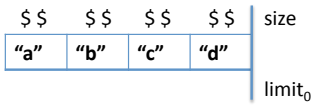
call	op's	saved tokens	spent tokens	total tokens	*size*	*limit*
`addend(L,"a")`	1	1	0	1	1	4
`addend(L,"b")`	1	1	0	2	2	4
`addend(L,"c")`	1	1	0	3	3	4
`addend(L,"d")`	1	1	0	4	4	4
`addend(L,"e")`	5	1	-4	1	5	8
`addend(L,"f")`	1	1	0	2	6	8
`addend(L,"g")`	1	1	0	3	7	8
`addend(L,"h")`	1	1	0	4	8	8
`addend(L,"i")`	9	1	-8	-3	9	16

We see that we spend 4 tokens when adding `"e"` to copy `"a"` through `"d"`, and we add a new one for the insertion of `"e"` itself. Unfortunately we incur a deficit when inserting `"i"`: we did not save enough to show constant amortized time. The argument was too simplistic.

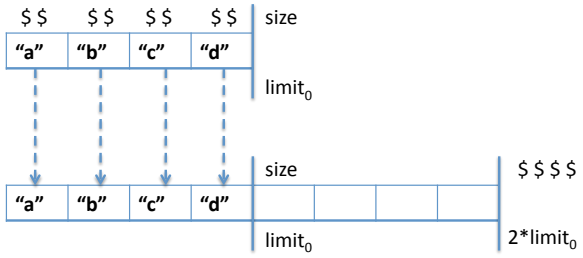
The crucial property we need is that there $k \geq 0$ tokens left just after have we have doubled the size of the array. We think of this as an *invariant* of the computation: it should always be true, no matter how many strings we insert.

To prove this invariant, we must show that it holds the first time we have to double the size of the array, and that it is preserved by the operations.

We when create the array, we give it some initial limit $limit_0$. We run out of space, once we have inserted $limit_0$ tokens, arriving at the following situation.

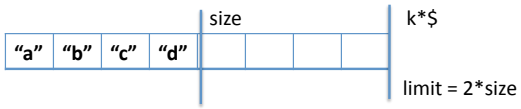


We have accrued $2 * limit_0$ tokens. We have to spend $limit_0$ of them to copy the elements so far, keeping $limit_0 > 0$ in the bank.

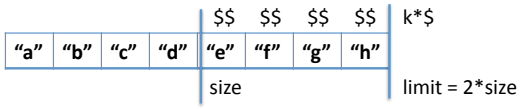


So the invariant holds the first time we double the size.

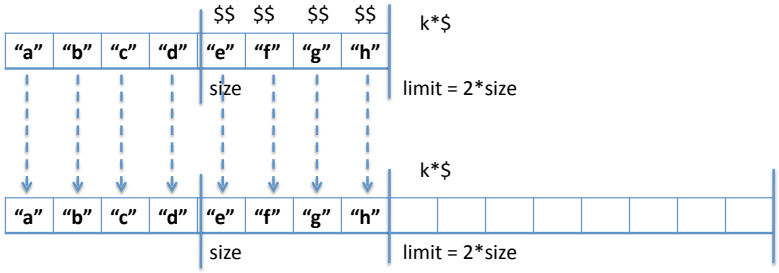
Now assume we have just doubled the size of the array and the invariant holds, that is, we have $k \geq 0$ tokens, and $2 * size = limit$.



After $size$ more inserts we are at $limit$ and added another $2 * size = limit$ tokens.



On the next insert we double the size of the array and copy $limit$ array elements, spending $limit$ tokens.



Our bank account is reduced back to k tokens, but we know $k \geq 0$, preserving our invariant.

Since we only save a constant number of tokens on each operation, in addition to the constant time the operation itself takes, we never perform more operations than a constant times the number of operations. So our claim above is true: any sequence of n operations performs at most $O(n)$ steps. We also say that the insert operation has *constant amortized time*.

This completes the argument. In the example above we get the following numbers.

call	op's	saved tokens	spent tokens	total tokens	*size*	*limit*
`addend(L,"a")`	1	2	0	2	1	4
`addend(L,"b")`	1	2	0	4	2	4
`addend(L,"c")`	1	2	0	6	3	4
`addend(L,"d")`	1	2	0	8	4	4
`addend(L,"e")`	5	2	-4	6	5	8
`addend(L,"f")`	1	2	0	8	6	8
`addend(L,"g")`	1	2	0	10	7	8
`addend(L,"h")`	1	2	0	12	8	8
`addend(L,"i")`	9	2	-8	6	9	16

The number of tokens will now never fall below 6. If we add another 8 elements, we will also put $2 * 8 = 16$ tokens into the bank. We will need to spend these to copy the 16 elements already in the array and we are back down to 6.

Tokens are a conceptual tool in our analysis, but they don't need to be implemented. The fact that there are always 0 or more tokens during any sequence of operations is an invariant of the data structure, although not quite in the same way as discussed before because it tracks sequences of operations rather than the internal state of the structure. In fact, it would be possible to add a new field to the representation of the array that would count tokens and raise an exception if it becomes negative. That would alert us to some kind of mistake, either in our amortized analysis or in our program. This would, however, incur a runtime overhead even when assertions are not checked, so tokens are rarely, if ever, explicitly implemented.

This kind of analysis is important to avoid serious programming mistakes. For example, let's say we decide to increase the size of the array only by 1 whenever we run out of space. The token scheme above does not work, because we cannot accumulate enough tokens before we need to

copy the array again. And, indeed, after we hit $limit$ the first time, the next sequence of n inserts takes $O(n^2)$ operations, because we copy the array on each step until we reach $2 * limit$.

7 Deleting Elements

Deleting elements from the end of the array is simple, and does not change our amortized analysis, unless we want to shrink the size of the array.

A first idea might be to simply cut the array in half whenever $size$ reaches half the size of the array. However, this cannot work in constant amortized time. The example demonstrating that is an alternating sequence of n inserts and n deletes precisely when we are at the limit of the array. In that case the total cost of the $2 * n$ operations will be $O(n^2)$.

To avoid this problem we cut the size of the array in half only when the number of elements in it reaches $limit/4$. The amortized analysis requires one token to be put aside for any delete operation. Then if $size = limit/2$ just after we doubled the size of the array and have no tokens, putting aside one token on every delete means that we have $size/2 = limit/4$ tokens when we arrive at a size of $limit/4$. Again, we have just enough tokens to copy the $limit/4$ elements to the new, smaller array of size $limit/2$.

The code for `remend` ("remove from end"):

```
string remend(uba L)
//@requires is_uba(L);
//@requires L->size > 0;  /* always check dynamically */
//@ensures is_uba(L);
{ string s;
  assert (L->size > 0, "cannot remove element from empty unbounded array");
  L->size--;
  s = L->strings[L->size];
  L->strings[L->size] = "";     /* avoids a space leak */
  if (4*L->size <= L->limit && L->limit > 1) uba_resize(L, L->limit/2);
  return s;
}
```

One side remark about the assignment `L->strings[L->size] = ""`. In C0, we do not have any explicit memory management. Storage will be reclaimed and used for future allocation when the garbage collector can see that data are no longer accessible from the program. If we remove an element from an unbounded array, but keep the element in the array, the

garbage collector can not determine that we will not access it again, because the reason is rather subtle and lies in the bounds check for `uba_get`. In order to allow the garbage collector to free the space occupied by the strings stored in the array, we therefore overwrite the array element with the empty string `""`, which is the default element for strings.

8 Interfaces

So far we have shown how to implement and analyze unbounded arrays. From a programming perspective, clients of this data structure should not have to know the exact details of the implementation. It is an important computational thinking principle to hide implementation details from the clients and just present an *interface* with the operations on the data structure. We also note down the complexity of the operations, since they are an important guide to potential clients.

```
struct ubarray;
typedef struct ubarray* uba;

uba uba_new(int initial_limit);           /* O(1) */
int uba_size(uba L);                      /* O(1) */
string uba_get(uba L, int index);         /* O(1) */
void uba_set(uba L, int index, string s); /* O(1) */
void addend(uba L, string s);             /* O(1), amortized */
string remend(uba L);                     /* O(1), amortized */
```

Unfortunately, C (and, by association, C0) does not provide a way to enforce that clients do not incorrectly exploit details of the implementation of a data structure. Higher-level languages such as Java or ML have interfaces and data abstraction as one of their explicit design goals. In this course, the use of interfaces is a matter of programming discipline. As we discuss further data structures we generally focus on the interface first, before writing any code. This is because the interface often guides the selection of an implementation technique and the individual functions.