Unbounded Arrays

To declare arrays we need to know their size before-hand. Unbounded arrays (**UBA**) overcome this shortcoming of arrays. Internally, in the unbounded array struct we use an array `data` which can store at most `limit` many items. The number of items in the array `data` is stored in the variable `size`. The main idea is that when `size==limit`, we have the `data` array filled to capacity. At this point we can not add any more new items without increasing the size of the array `data`, so we create a new array with size equal to `2*limit` and copy all the items in it.

Struct

```
typedef struct uba_header uba;
struct uba_header {
  int size;          // 0 <= size && size < limit
  int limit;         // 0 < limit
  string[] data;     // \length(data) == limit
};
```

Interface (Client Side View)

A client can use the following functions of the library.

```
/*************************** Interface ***************************/

// typedef ______* uba_t;

//returns the number of strings stored in the uba
int uba_len(uba_t A)
/*@requires A != NULL;   @*/
/*@ensures \result >= 0; @*/ ;

//creates and returns a new uba, with given size.
uba_t uba_new(int size)
/*@requires 0 <= size;               @*/
/*@ensures \result != NULL;          @*/
/*@ensures uba_len(\result) == size; @*/ ;

//returns item stored at index i
string uba_get(uba_t A, int i)
/*@requires A != NULL;                @*/
/*@requires 0 <= i && i < uba_len(A); @*/ ;

//stores x at the i^th index in A
void uba_set(uba_t A, int i, string x)
/*@requires A != NULL;                @*/
/*@requires 0 <= i && i < uba_len(A); @*/ ;

//adds a new element at the end in A
void uba_add(uba_t A, string x)
/*@requires A != NULL; @*/ ;

//removes the last element from A
string uba_rem(uba_t A)
/*@requires A != NULL;      @*/
/*@requires 0 < uba_len(A); @*/ ;

// bonus function
void uba_print(uba_t A)
/*@requires A != NULL;      @*/ ;
```

The functions `uba_add` and `uba_rem` are new as compared to the interface of a self-sorting array (**SSA**). One major difference in the function `uba_set` is that we no longer keep the array sorted.

Implementation (Library Side View)

The following functions are library only functions which help in checking the UBA invariants.

Exercise: Go through the functions and write the invariants as mathematical statements.

```
bool is_array_expected_length(string[] A, int length) {
  //@assert \length(A) == length;
  return true;
}

bool is_uba(uba* A) {
  return A != NULL
      && 0 <= A->size && A->size < A->limit
      && is_array_expected_length(A->data, A->limit);
}
```

The following functions are similar to SSA.

```
int uba_len(uba* A)
//@requires is_uba(A);
//@ensures 0 <= \result && \result < \length(A->data);
{
  return A->size;
}

string uba_get(uba* A, int i)
//@requires is_uba(A);
//@requires 0 <= i && i < uba_len(A);
{
  return A->data[i];
}

void uba_set(uba* A, int i, string x)
//@requires is_uba(A);
//@requires 0 <= i && i < uba_len(A);
//@ensures is_uba(A);
{
  A->data[i] = x;
}
```

Creating a new UBA

The following function creates a UBA with given initial size. An explanation about line 7 follows after the code.

```
1  uba* uba_new(int size)
2  //@requires 0 <= size;
3  //@ensures is_uba(\result);
4  //@ensures uba_len(\result) == size;
5  {
6    uba* A = alloc(uba);
7    int limit = size == 0 ? 1 : size*2;
8    A->data = alloc_array(string, limit);
9    A->size = size;
10   A->limit = limit;
11
12   return A;
13 }
```

Ternary Conditional Operator

It uses the ternary conditional operator `?:` on line 7. The conditional operator is of the form

```
1  variable = Expression1 ? Expression2 : Expression3
```

It can be visualized as an if-else statement as follows:

```
1  if(Expression1)
2  {
3      variable = Expression2;
4  }
5  else
6  {
7      variable = Expression3;
8  }
```

Since the operator `?:` takes three operands to work, it is called as a ternary operator.

Exercise: Write Line 7 in function `uba_new` as an if-else statement.

Adding New Elements at the end of UBA

For the `uba_add` function we need a library side helper function `uba_resize` which resizes the `data` array if array capacity is reached.

```
void uba_resize(uba* A, int new_limit)
/* A may not be a valid array since A->size == A->limit is possible! */
//@requires A != NULL;
//@requires 0 <= A->size && A->size < new_limit;
//@requires \length(A->data) == A->limit;
//@ensures is_uba(A);
{
  string[] B = alloc_array(string, new_limit);

  //copy elements of A->data into B.
  for (int i = 0; i < A->size; i++)
    //@loop_invariant 0 <= i;
    {
      B[i] = A->data[i];
    }

  //update limit
  A->limit = new_limit;

  //set data to be the new larger array B.
  A->data = B;
}

void uba_add(uba* A, string x)
//@requires is_uba(A);
//@ensures is_uba(A);
{
  //put x at the last location.
  A->data[A->size] = x;
  (A->size)++; //one item more in A

  //is the array full?
  if (A->size < A->limit) return;

  //if the array requires too much memory, then halt the program.
  //the following non-contract function is also present in C.
  assert(A->limit <= int_max() / 2); // Fail if array would get too big

  //assertion above holds, therefore safe to create a bigger array.
  uba_resize(A, A->limit * 2);
}
```

Rest of the functions

```
string uba_rem(uba* A)
//@requires is_uba(A);
//@requires 0 < uba_len(A);
//@ensures is_uba(A);
{
  (A->size)--;
  string x = A->data[A->size];

  if (A->limit >= 4 && A->size <= A->limit / 4)
    uba_resize(A, A->limit / 2);

  return x;
}

void uba_print(uba* A)
//@requires is_uba(A);
{
  print("[");
  for (int i = 0; i < A->size; i++)
    {
      print(A->data[i]);
      if (i+1 != A->size) print(", ");
    }
  print("]");
}

// Client type
typedef uba* uba_t;
```