```
/*      dynArr.c: Dynamic Array implementation. */
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "dynamicArray.h"

struct DynArr
{
        TYPE *data;             /* pointer to the data array */
        int size;               /* Number of elements in the array */
        int capacity;   /* capacity ofthe array */
};

/* ************************************************************************
        Dynamic Array Functions
************************************************************************ */

/* Initialize (including allocation of data array) dynamic array.

        param:  v               pointer to the dynamic array
        param:  cap     capacity of the dynamic array
        pre:    v is not null
        post:   internal data array can hold capacity elements
        post:   v->data is not null
*/
void initDynArr(DynArr *v, int capacity)
{
        assert(capacity > 0);
        assert(v!= 0);
        v->data = malloc(sizeof(TYPE) * capacity);
        assert(v->data != 0);
        v->size = 0;
        v->capacity = capacity;

}

/* Allocate and initialize dynamic array.

        param:  cap     desired capacity for the dyn array
        pre:    none
        post:   none
        ret:    a non-null pointer to a dynArr of cap capacity
                        and 0 elements in it.
*/
DynArr* createDynArr(int cap)
{
        DynArr *r;
        assert(cap > 0);
        r = malloc(sizeof( DynArr));
        assert(r != 0);
        _initDynArr(r,cap);
        return r;
}

/* Deallocate data array in dynamic array.

        param:  v               pointer to the dynamic array
        pre:    v is not null
        post:   d.data points to null
        post:   size and capacity are 0
```

```
        post:   the memory used by v->data is freed
*/
void freeDynArr(DynArr *v)
{
        assert(v!=0);

        if(v->data != 0)
        {
                free(v->data);  /* free the space on the heap */
                v->data = 0;    /* make it point to null */
        }
        v->size = 0;
        v->capacity = 0;
}

/* Deallocate data array and the dynamic array ure.

        param:  v               pointer to the dynamic array
        pre:    v is not null
        post:   the memory used by v->data is freed
        post:   the memory used by d is freed
*/
void deleteDynArr(DynArr *v)
{
        assert (v!= 0);
        freeDynArr(v);
        free(v);
}

/* Resizes the underlying array to be the size cap

        param:  v               pointer to the dynamic array
        param:  cap             the new desired capacity
        pre:    v is not null
        post:   v has capacity newCap
*/
void _dynArrSetCapacity(DynArr *v, int newCap)
{
        int i;
        TYPE *oldData;
        int oldSize = v->size;
        oldData = v->data;

        printf("========Resizing========\n");
        /* Create a new dyn array with larger underlying array */
        _initDynArr(v, newCap);

        for(i = 0; i < oldSize; i++){
                v->data[i] = oldData[i];
        }

        v->size = oldSize;
        /* Remember, init did not free the original data */
        free(oldData);

#ifdef ALTERNATIVE
        int i;

        /* Create a new underlying array*/
        TYPE *newData = (TYPE*)malloc(sizeof(TYPE)*newCap);
        assert(newData != 0);
```

```
        /* copy elements to it */

        for(i = 0; i < v->size; i++)
        {
                newData[i] = v->data[i];
        }

        /* Delete the oldunderlying array*/
        free(v->data);
        /* update capacity and size and data*/
        v->data = newData;
        v->capacity = newCap;
#endif
}

/* Get the size of the dynamic array

        param:  v               pointer to the dynamic array
        pre:    v is not null
        post:   none
        ret:    the size of the dynamic array
*/
int sizeDynArr(DynArr *v)
{
        assert(v!=0);
        return v->size;
}

/*      Adds an element to the end of the dynamic array

        param:  v               pointer to the dynamic array
        param:  val             the value to add to the end of the dynamic array
        pre:    the dynArry is not null
        post:   size increases by 1
        post:   if reached capacity, capacity is doubled
        post:   val is in the last utilized position in the array
*/
void addDynArr(DynArr *v, TYPE val)
{

        assert(v!=0);

        /* Check to see if a resize is necessary */
        if(v->size >= v->capacity)
                _dynArrSetCapacity(v, 2 * v->capacity);

        v->data[v->size] = val;
        v->size++;

}

/*      Get an element from the dynamic array from a specified position

        param:  v               pointer to the dynamic array
        param:  pos             integer index to get the element from
        pre:    v is not null
        pre:    v is not empty
        pre:    pos < size of the dyn array and >= 0
        post:   no changes to the dyn Array
        ret:    value stored at index pos
```

```
 */

 TYPE getDynArr(DynArr *v, int pos)
 {
        assert(v!=0);
        assert(pos < v->size);
        assert(pos >= 0);

        return v->data[pos];
 }

 /*     Put an item into the dynamic array at the specified location,
        overwriting the element that was there

        param:  v               pointer to the dynamic array
        param:  pos             the index to put the value into
        param:  val             the value to insert
        pre:    v is not null
        pre:    v is not empty
        pre:    pos >= 0 and pos < size of the array
        post:   index pos contains new value, val
 */
 void putDynArr(DynArr *v, int pos, TYPE val)
 {
        assert(v!=0);
        assert(pos < v->size);
        assert(pos >= 0);
        v->data[pos] = val;
 }

 /*     Swap two specified elements in the dynamic array

        param:  v               pointer to the dynamic array
        param:  i,j             the elements to be swapped
        pre:    v is not null
        pre:    v is not empty
        pre:    i, j >= 0 and i,j < size of the dynamic array
        post:   index i now holds the value at j and index j now holds the value at i
 */
 void swapDynArr(DynArr *v, int i, int  j)
 {
        TYPE  temp;
        assert(v!=0);
        assert(i < v->size);
        assert(j < v->size);
        assert(i >= 0);
        assert(j >= 0);

        temp = v->data[i];
        v->data[i] = v->data[j];
        v->data[j] = temp;

 }

 /*     Remove the element at the specified location from the array,
        shifts other elements back one to fill the gap

        param:  v               pointer to the dynamic array
        param:  idx             location of element to remove
        pre:    v is not null
        pre:    v is not empty
```

```
        pre:    idx < size and idx >= 0
        post:   the element at idx is removed
        post:   the elements past idx are moved back one
*/
void removeAtDynArr(DynArr *v, int idx){
        int i;
        assert(v!= 0);
        assert(idx < v->size);
        assert(idx >= 0);

   //Move all elements up

        /* My loop does not execute when idx == size-1
         * so I don't have to worry about coping an element outside the array
         * into that idx!
         */
   for(i = idx; i < v->size-1; i++){
      v->data[i] = v->data[i+1];
   }

   v->size--;

}

/*      Returns boolean (encoded in an int) demonstrating whether or not the
        dynamic array stack has an item on it.

        param:  v               pointer to the dynamic array
        pre:    v is not null
        post:   none
        ret:    >0  if empty, otherwise 0
*/
int isEmptyDynArr(DynArr *v)
{
        assert(v!= 0);
        return !(v->size);
        /* alternatively:

        if(v->size == 0)
                return 1;
        else return 0;

        */



}


/* Utility function for debugging */
void _printDynArr(struct DynArr *da)
{
        int i;
        for(i=0; i < da->size; i++)
                printf("DA[%d] == %d\n", i, da->data[i]);
}
```