```
#ifndef __VECTOR_H__
#define __VECTOR_H__

/**
 * @brief Create a Generic Vector data type
 * that stores pointer to user provided elements of generic type
 * The Vector is heap allocated and can grow and shrink on demand.
 *
 * @author Author MuhammadZ (muhammadz@experis.co.il)
 */

#include <stddef.h>  /* size_t */

typedef struct Vector Vector;
typedef int     (*VectorElementAction)(void* _element, size_t _index, void* _context);

typedef enum Vector_Result {
        VECTOR_SUCCESS,
        VECTOR_UNITIALIZED_ERROR,                               /**< Uninitialized
vector error    */
        VECTOR_ALLOCATION_ERROR,                                /**< realloc error on
grow/shrink   */
        VECTOR_INDEX_OUT_OF_BOUNDS_ERROR
        /* Add more as needed by your implementation */
} VectorResult;

/**
 * @brief Dynamically create a new vector object of given capacity and
 * @param[in] _initialCapacity - initial capacity, number of elements that can be stored
initially
 * @param[in] _blockSize - the vector will grow or shrink on demand by this size
 * @return Vector * - on success / NULL on fail
 *
 * @warning if _blockSize is 0 the vector will be of fixed size.
 * @warning if both _initialCapacity and _blockSize are zero function will return NULL.
 */
Vector* VectorCreate(size_t _initialCapacity, size_t _blockSize);

/**
 * @brief Dynamically deallocate a previously allocated vector
 * @param[in] _vector - Vector to be deallocated.
 * @params[in] _elementDestroy : A function pointer to be used to destroy all elements
in the vector
 *             or a null if no such destroy is required
 * @return void
 */
void VectorDestroy(Vector** _vector, void (*_elementDestroy)(void* _item));

/**
 * @brief Add an Item to the back of the Vector.
 * @param[in] _vector - Vector to append integer to.
 * @param[in] _item - Item to add.
 * @return success or error code
 * @retval VECTOR_SUCCESS on success
 * @retval VECTOR_....
 * (cover all possibilities)
 */
VectorResult VectorAppend(Vector* _vector, void* _item);

/**
 * @brief Delete an Element from the back of the Vector.
 * @param[in] _vector - Vector to delete integer from.
 * @param[out] _pValue - pointer to variable that will receive deleted item value
 * @return success or error code
 * @retval VECTOR_SUCCESS on success
 * @retval VECTOR_....
 * (cover all possibilities)
 * @warning _item can't be null. this will be assertion violation
 */
VectorResult VectorRemove(Vector* _vector, void** _pValue);
```

```
/**
 * @brief Get value of item at specific index from the the Vector
 * @param[in] _vector - Vector to use.
 * @param[in] _index - index of item to get value from. the index of first elemnt is 1
 * @param[out] _pValue - pointer to variable that will recieve the item's value.
 * @return success or error code
 * @retval VECTOR_SUCCESS on success
 * @retval VECTOR_.... (cover all possibilities)
 *
 * @warning Index starts from 1.
 */
VectorResult VectorGet(const Vector* _vector, size_t _index, void** _pValue);

/**
 * @brief Set an item at specific index to a new value.
 * @param[in] _vector - Vector to use.
 * @param[in] _index - index of an existing item.
 * @param[in] _value - new value to set.
 * @return success or error code
 * @retval VECTOR_SUCCESS on success
 * @retval VECTOR_.... (cover all possibilities)
 *
 * @warning Index starts from 1.
 */
VectorResult VectorSet(Vector* _vector, size_t _index, void*  _value);

/**
 * @brief Get the number of actual items currently in the vector.
 * @param[in] _vector - Vector to use.
 * @return  number of items on success 0 if vector is empty or invalid
 */
size_t VectorSize(const Vector* _vector);

/**
 * @brief Get the current capacity of the  vector.
 * @param[in] _vector - Vector to use.
 * @return  capacity of vector
 */
size_t VectorCapacity(const Vector* _vector);


/**
 * @brief Iterate over all elements in the vector.
 * @details The user provided _action function will be called for each element
 *          if _action return a zero for an element the iteration will stop.
 * @param[in] _vector - vector to iterate over.
 * @param[in] _action - User provided function pointer to be invoked for each element
 * @param[in] _context - User provided context, will be sent to _action
 * @returns number of times the user functions was invoked
 * equevallent to:
 *      for(i = 1; i < VectorSize(v); ++i){
 *             VectorGet(v, i, &elem);
 *             if(_action(elem, i, _context) == 0)
 *                                    break;
 *      }
 *              return i;
 */
size_t VectorForEach(const Vector* _vector, VectorElementAction _action, void*
_context);

#endif /* __VECTOR_H__ */
```