DynamicArray Library in C
Documentation

Faruk Alpay

December 29, 2023

1 Introduction

Unlike C++'s standard library, which includes `std::vector` class template, C programming language lacks built-in support for `dynamic arrays`. The provided code implements a dynamic array data structure in C, allowing for the creation, modification, and manipulation of an array whose size can be dynamically adjusted during runtime. Dynamic array is similar to the standard array in C, but it can grow or shrink in size as needed.

2 Code Explanation

2.1 Libaries

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <stdio.h>
```

This project only uses the standard C libraries. These libraries are necessary for the code to run flawlessly and error-free.

2.2 DynamicArray structure

```
typedef struct {
    void* data;
    size_t size;
    size_t capacity;
    size_t element_size;
} DynamicArray;
```

This syntax defines a new data type named `DynamicArray`. Let's break down its components. In `void* data` the array elements are stored in this variable. `size_t size` this variable represents the current number of elements in the dynamic array. In other words it indicates how many elements are currently being used or occupied. `size_t capacity` indicates the maximum number of elements that the array can currently hold without needing to resize the underlying memory. `size_t element_size` this variable denotes the size (in bytes) of each individual element in the dynamic array.

2.3 Function to initialize a DynamicArray

```
void initDynamicArray(DynamicArray* array, size_t element_size) {
    array->data = NULL;
    array->size = 0;
    array->capacity = 0;
    array->element_size = element_size;
}
```

This function initializes a `DynamicArray` instance. The parameters of the function are `DynamicArray* array` and `size_t element_size`. The components of the DynamicArray are then initialized as follows: `array->data` is set to NULL, `array->size` is set to 0, `array->capacity` is set to 0, and `array->element_size` is set to the provided `element_size`. `array->element_size` represents the size of the array's variable type, such as 2 bytes for a short or 4 bytes for an integer. An instance of DynamicArray is created in the main function, and a pointer to this instance is passed into the initDynamicArray function, as shown.

```
DynamicArray myArray;
initDynamicArray(&myArray, sizeof(int));
```

As previously mentioned, An instance of `DynamicArray` named `myArray` is created, and `initDynamicArray` is called to initialize it with the address of `myArray` and the size of an integer (4 bytes).

2.4 Function to free the memory occupied by a DynamicArray

```
void freeDynamicArray(DynamicArray* array) {
    free(array->data);
    array->data = NULL;
    array->size = 0;
    array->capacity = 0;
}
```

This function frees the memory occupied by a `DynamicArray` instance. The only parameter of the function is `DynamicArray* array`. Let's break down each part of the function: Firstly, the function deallocates the memory within the data member, which is a sub-component of the `DynamicArray` instance created in the main program. Then, the function initializes the components as follows: `array->data` is set to NULL, `array->size` is set to 0, `array->capacity` is set to 0.

2.5 Function to clear the DynamicArray

```c
void clearDynamicArray(DynamicArray* array) {
    freeDynamicArray(array);
    initDynamicArray(array, array->element_size);
}
```

This function clears and reinitialize a `DynamicArray` instance. The only parameter of the function is `DynamicArray* array`. Let's break down each part of the function: Firstly, the function calls the `freeDynamicArray` function, previously mentioned, to deallocate the memory occupied by a `DynamicArray` instance, including the memory pointed to by the `data` member. Following that, the function calls the `initDynamicArray` function, also mentioned, to initialize the components of the `DynamicArray` instance to a default state. Notably, the `array->element_size` remains unchanged.

2.6 Function to push an element to the back of the DynamicArray

```c
void pushBack(DynamicArray* array, const void* value) {
    if (array->size >= array->capacity) {
        size_t new_capacity = array->capacity == 0 ? 1 : array->capacity * 2;
        void* new_data = realloc(array->data, new_capacity * array->element_size);

        if (new_data == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(EXIT_FAILURE);
        }

        array->data = new_data;
        array->capacity = new_capacity;
    }

    void* destination = (char*)array->data + array->size * array->element_size;
    memcpy(destination, value, array->element_size);

    array->size++;
}
```

This function adds an element to the back of a `DynamicArray` instance. The parameters of the function are `DynamicArray* array, const void* value`. Let's break down each part of the function: Firstly, the function checks whether the current `array->size` (total amount of variables in the array) equals or exceeds `array->capacity` (amount of elements it can hold without needing reallocation), indicating that the array is either full or has reached its current `capacity` which is `array->capacity`. In such a case, the subsequent parts of the function are called. It calculates `array->new_capacity` for the array based on whether `array->capacity` is zero or not. If the `array->capacity` is zero (indicating an empty array or uninitialized array), the `array->new_capacity` is set to one. Otherwise, it is set to twice `array->capacity`. After that, the function allocates a new block of memory using realloc for the `void* new_data` pointer. It takes two arguments: The first argument `array->data` is a pointer to the block of memory that holds the array's data. The second argument calculates the new size for the block of memory, which is the product of the `array->new_capacity` and the `array->element_size`. Then, the function checks the value of `new_data` to determine whether the allocation has failed or not. If the allocation has not failed, `array->data` is initialized as `new_data`, and `array->capacity` is initialized as `array->new_capacity`. The if loop is now closed. Then, `void* destination` calculates the memory location where the new element will be added to the array. In other words, two terms are getting summed. The first term `(char*)array->data` is the memory address of the beginning of the array's data. The second term `array->size * array->element_size` is the total amount of variables in the array times size of variable type of the array. Let's assume the starting memory address of the array is 0x00000004. We have an array of 4 integers (for example, 2, 4, 6, 7), where each integer occupies 4 bytes, resulting in a total size of the array being 16 bytes (4 integers * 4 bytes each). In this case, the destination address would be calculated as the starting address plus the offset we need to allocate, which is determined by the total size of the array. This gives the memory location where the new element will be added. Note that it is assumed that Int32 is used, which has a memory size of 4 bytes.

$$\begin{aligned}
\text{Starting address} &= 0x00000004 \\
\text{Total size of the array} &= 4 \text{ integers} \times 4 \text{ bytes per integer} = 16 \text{ bytes} \\
\text{Destination address} &= \text{Starting address} + \text{Total size of the array} \\
&= 0x00000004 + 16 \\
&= 0x00000014
\end{aligned}$$

After that, the `memcpy` function copies the content from the memory location of `value`. Then, this content is overwritten to the `destination` with size of `array->element_size` which is 4 bytes if we assume given element is an integer. Finally, `array->size++` increases the value of size by 1, indicating that a new element has been added to the dynamic array. This way, `array->size` always reflects the current number of elements in the array. The usage of `pushBack` function is given in the example below.

```
DynamicArray intArray;
initDynamicArray(&intArray, sizeof(int));
for (int i = 1; i <= 4; ++i) {
    pushBack(&intArray, &i);
}
```

2.7 Function to trim the capacity of the DynamicArray to match its size

```
void trimToSize(DynamicArray* array) {
    if (array->size < array->capacity) {
        size_t new_capacity = array->size;
        void* new_data = realloc(array->data, new_capacity * array
    ->element_size);

        if (new_data == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(EXIT_FAILURE);
        }

        array->data = new_data;
        array->capacity = new_capacity;
    }
}
```

This function reduces the capacity of a `DynamicArray` instance to match its current size. The only parameter of the function is `DynamicArray* array`. This function is used within other functions. Thus, it's important to understand this function. Firstly, the function checks if the current `array->size` smaller than `array->capacity`. This means, the array is currently not utilizing its full allocated capacity, and there is unused memory. In this case, trimming the capacity ensures that the memory utilization is minimized to match the actual number of elements in the array. Firstly, `size_t new_capacity` is initialized with `array->size`. After that, the function allocates a new block of memory using realloc for the `void* new_data` pointer. It takes two arguments: The first argument `array->data` is a pointer to the block of memory that holds the array's data. The second argument calculates the new size for the block of memory, which is the product of the `array->new_capacity` and the `array->element_size`. Then, the function checks the value of `new_data` to determine whether the allocation has failed or not. If the allocation has not failed, `array->data` is initialized as `new_data`, and `array->capacity` is initialized as `array->new_capacity`. The if loop is now closed. As a result of of the close resemblance between this function and the `pushBack` function, certain portions from that section are included here.

2.8 Function to pop an element from the back of the DynamicArray

```
void popBack(DynamicArray* array) {
    if (array->size > 0) {
        array->size--;
        trimToSize(array);
    }
}
```

This function removes the last element from a `DynamicArray` instance. The only parameter of the function is `DynamicArray* array`. Firstly, the function check whether `array->size` is greater than 0. This is an indication that there is at least one element in the array. In this case, the last element of the array is getting removed and `trimToSize(array)` function is called.

2.9 Function to delete an element at a specific index from the DynamicArray

```
void deleteAt(DynamicArray* array, size_t index) {
    assert(index < array->size);

    void* delete_location = (char*)array->data + index * array->element_size;
    size_t bytes_to_shift = (array->size - index - 1) * array->element_size;
    memmove(delete_location, (char*)delete_location + array->element_size, bytes_to_shift);

    trimToSize(array);
    array->size--;
}
```

This function removes an element at a specific index from a `DynamicArray` instance. The only parameter of the function is `DynamicArray* array`. Firstly, an `assert` statement checks whether `index` is less than `array->size`. If the condition `index < array->size` is false, it indicates an invalid index, and the program will terminate with an assertion failure. After that, The memory location of the index to be deleted is getting determined and this value is given to `void* delete_location` In other words, two terms are getting summed. The first term `(char*)array->data` is the memory address of the beginning of the array's data. The second term `index * array->element_size` calculates the offset in bytes from the beginning of the array to the specified index. Afterwards, total size, in bytes, of the memory block that needs to be shifted to fill the gap left by deleting an element at a specific index is getting calculated and this value is given to `size_t bytes_to_shift`. In other words, the product of two terms is taken. The first term `array->size - index - 1` calculates the number of elements to the right of the element at the specified index (excluding the element at the index itself). For instance, if you have an array of size 10 and you wish to remove the element at index 3, this expression evaluates to 10 - 3 - 1 = 6, meaning that the element at index 3 has 6 elements to its right. The second term `array->element_size` represents the size of the array's variable. After the pointer that holds the memory address of the element to be deleted is found, the `memmove` function is used to shift the subsequent elements in a `DynamicArray` instance starting from that location, effectively filling the gap left by the deleted element. In other words, the `memmove` function shifts the data in the memory block starting from `delete_location + array->element_size` to `delete_location`, essentially filling the gap left by the deleted element. Finally, as one element is removed from the array, the array size is decremented by one.

2.10 Function to get the element at a specific index

```c
void* getElementAt(const DynamicArray* array, size_t index) {
    assert(index < array->size);
    return (char*)array->data + index * array->element_size;
}
```

This function returns a pointer to the element at a specified index in a `DynamicArray` instance. The parameters of the function are `DynamicArray* array, size_t index`. An `assert` statement is used at the start of the function to make sure that the given index is within the array's valid range. An invalid index is indicated by the condition `index < array->size` being false, in which case the program will terminate with an assertion failure. If the index is valid, the function calculates the memory address of the element at the specified index. It uses pointer arithmetic to determine this address. `(char*)array->data` converts the base address of the array's data to a `char*` pointer to ensure that the pointer arithmetic is done in terms of bytes. `index * array->element_size` calculates the offset in bytes from the beginning of the array to the specified index. The sum of `(char*)array->data` and `index * array->element_size` gives the memory address of the element at the specified index in a `DynamicArray` instance.

2.11 Function to find an element in the array

```c
size_t findElement(const DynamicArray* array, const void* value) {
    for (size_t i = 0; i < array->size; ++i) {
        void* current_element = getElementAt(array, i);
        if (memcmp(current_element, value, array->element_size) ==
    0) {
            return i;
        }
    }
    return SIZE_MAX;
}
```

This function searches for a specific value within a `DynamicArray` instance. The parameters of the function are `const DynamicArray* array, const void* value`. `const DynamicArray* array` is a pointer to a DynamicArray instance that represents the array in which the search is conducted. And `const void* value` is a pointer to the value being searched for within the array. Firstly, the for loop iterates as long as the value of i is less than `array->size`. After each iteration of the loop, the value of i is increased by 1. This ensures that the loop will eventually terminate when i becomes equal to or greater than `array->size`. In other words, the for loop is designed to iterate over the elements of an array. For each element in the loop, a void* `current_element` is defined and initialized with the memory address of the element at the specified index in a `DynamicArray` instance using the `getElementAt(array, i)`. Then, the `memcmp` function is utilized to compare the memory content of `current_element` with the memory content pointed to by value, considering a block of memory equal to `array->element_size`. If `memcmp` returns 0, indicating that the memory blocks are equal, it implies that the content of the current element is equal to the target value. In this case, the function returns the index i at which the value was found. The loop concludes at this point since the desired value has been located. If no match is found throughout the loop, the function returns `SIZE_MAX`. Example usage is shown below.

```c
int main() {
    // Initialize a DynamicArray of integers
    DynamicArray intArray;
    initDynamicArray(&intArray, sizeof(int));

    // Populate the array with some values
    for (int i = 0; i < 10; ++i) {
        pushBack(&intArray, &i);
    }

    // Value to search for
    int targetValue = 5;

    // Find the index of the target value in the array
    size_t index = findElement(&intArray, &targetValue);

    // Display the result
    if (index != SIZE_MAX) {
        printf("Value %d found at index %zu.\n", targetValue, index
    );
    }
    else {
        printf("Value %d not found in the array.\n", targetValue);
    }

    // Free memory used by the DynamicArray
    free(intArray.data);

    return 0;
}
```

Output: Value 5 found at index 5.

2.12 Function to erase the first occurrence of an element from the array

```c
void eraseElement(DynamicArray* array, const void* value) {
    size_t index = findElement(array, value);
    if (index != SIZE_MAX) {
        deleteAt(array, index);
    }
}
```

This function searches for a specific value in a `DynamicArray` instance. If the value is found, it deletes the element at the corresponding index, effectively erasing that element from the array. If the value is not found, no action is taken, and the function completes without modifying the array. `eraseElement` provides a more user-friendly interface by allowing users to specify the value of the element that is being removed. Internally, it uses `findElement` to locate the index of the specified value and then calls deleteAt to remove the element at that index.

2.13 Function to insert an element at a specific index

```c
void insertAt(DynamicArray* array, size_t index, const void* value) {
    assert(index <= array->size);

    if (array->size >= array->capacity) {
        size_t new_capacity = array->capacity == 0 ? 1 : array->capacity * 2;
        void* new_data = realloc(array->data, new_capacity * array->element_size);

        if (new_data == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(EXIT_FAILURE);
        }

        array->data = new_data;
        array->capacity = new_capacity;
    }

    void* insert_location = (char*)array->data + index * array->element_size;
    size_t bytes_to_shift = (array->size - index) * array->element_size;
    memmove((char*)insert_location + array->element_size, insert_location, bytes_to_shift);

    memcpy(insert_location, value, array->element_size);

    array->size++;
}
```

This function inserts a new element at a the specified index in a `DynamicArray` instance. It resizes the array, shifts existing elements to make room, and then copies the new element to the designated position. The parameters of the function are `DynamicArray* array, size_t index, const void* value`. Firstly, an `assert` statement checks that the specified index is within the valid range of the array. If it's not, the program will terminate with an error message. Then, within this if statement, the code assesses whether the current size of an `DynamicArray` instance is greater than or equal to its capacity, and if this condition holds true, it initiates a resizing operation. A detailed explanation of this is available where the `pushBack` function is explained. After that, the memory address for the new element's insertion is computed by adding `(char*)array->data`, denoting the array's starting memory address, to `index * array->element_size`, which determines the offset from the beginning based on the given index and this value is given to `void* insert_location`. After this operation the number of bytes to shift existing elements to make room for the new element is computed by multiplying `array->size - index`, denoting the number of elements after the insert index. And `array->element_size` which represents the size of the array's variable type. Then, `memmove` function is used to shift existing elements. The destination address is calculated by `(char*)insert_location + array->element_size`. The source is `insert_location`. And the size is `bytes_to_shift`. After that, `memcpy` function is used to copy the new element to the calculated insert location. `insert_location` is the destination address, `value` is the source address and `array->element_size` specifies the number of bytes to copy. Finally, the size of the array is incremented by one. As, all of these were thoroughly covered in the explanation of the other functions, a more concise summary was provided.

2.14 Function to Reserve Capacity in a DynamicArray

```
void reserve(DynamicArray* array, size_t new_capacity) {
    if (new_capacity > array->capacity) {
        void* new_data = realloc(array->data, new_capacity * array->element_size);

        if (new_data == NULL) {
            fprintf(stderr, "Memory reallocation failed\n");
            exit(EXIT_FAILURE);
        }

        array->data = new_data;
        array->capacity = new_capacity;
    }
}
```

This function is responsible for ensuring that a `DynamicArray` instance has sufficient capacity to accommodate a specified new capacity. If the requested capacity is greater than the current capacity, it attempts to reallocate memory, and if successful, it updates `array->data` to point to the new memory block and sets `array->capacity` to the new capacity. The parameters of the function are DynamicArray* array, size_t new_capacity. Here, size_t new_capacity represents the new capacity to reserve for an DynamicArray instance. Firstly, the if loop checks whether the requested new capacity is greater than the current capacity of the an DynamicArray instance. If this condition holds true, Memory is reallocated for a DynamicArray instance, and the newly allocated memory is assigned to the pointer void* new_data. Then, the function checks the value of new_data to determine whether the allocation has failed or not. If the allocation has not failed, array->data points to the newly allocated memory represented by new_data and array->capacity is initialized as array->new_capacity. Note that, when you use the reserve function, a specific portion of memory is allocated. Subsequent insertions or deletions in the dynamic array utilize this allocated portion. The function does not allocate a new portion until the entire allocated space is used. However, it does not fill in unused spaces.

2.15 Function to Resize a DynamicArray

```
void resize(DynamicArray* array, size_t new_size) {
    if (new_size < array->size) {
        array->size = new_size;
        trimToSize(array);
    }
    else if (new_size > array->size) {
        reserve(array, new_size);
        array->size = new_size;
    }
}
```

This function adjusts the size of the dynamic array to the specified new_size. The parameters of the function are DynamicArray* array, size_t new_size. If new_size is smaller than the current size, it initializes array->size with new_size and it trims the array's capacity using trimToSize. If new_size is larger, it ensures that the array has enough capacity by calling reserve. Finally, it updates the array->size to match the specified new_size. The handling of unused spaces during array enlargement is the primary difference between the reserve and resize functions. Although resize fills in empty spaces during an enlargement, the reserve function does not. In particular, as the array grows, any additional elements are appended to the end and the array expands even more.

2.16 Function to concatenate two DynamicArrays

```
void concatenate(DynamicArray* destination, const DynamicArray* source) {
    size_t new_size = destination->size + source->size;
    if (new_size > destination->capacity) {
        size_t new_capacity = new_size;
        void* new_data = realloc(destination->data, new_capacity * destination->element_size);

        if (new_data == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(EXIT_FAILURE);
        }

        destination->data = new_data;
        destination->capacity = new_capacity;
    }

    void* destination_ptr = (char*)destination->data + destination->size * destination->element_size;
    const void* source_ptr = source->data;
    size_t bytes_to_copy = source->size * source->element_size;
    memcpy(destination_ptr, source_ptr, bytes_to_copy);

    destination->size = new_size;
}
```

This function appends the elements of the source array to the end of the destination array. It dynamically reallocates memory, copies the elements, and updates the size and capacity of the destination array accordingly. The parameters of the function are `DynamicArray* destination, const DynamicArray* source`. Firstly, `new_size` is calculated by summation of `destination->size` and `source->size`. After that, the if loop checks whether `new_size` is greater than `destination->capacity`. In other words, it checks whether the array needs to be resized to accommodate the new size. If `new_size` is larger than the current capacity of the destination array, it implies that the destination array does not have enough capacity to accommodate the new size. In this case, a `size_t new_capacity` variable is declared and initialized with the value of `new_size`, which represents the calculated new capacity for the destination array after concatenation with the source array. Following this, the memory is dynamically reallocated using the `realloc` function for `void* new_data`. The source pointer is `destination->data`, and the size argument is `new_capacity * destination->element_size`. This size represents the total amount of memory, in bytes, needed for the memory block after resizing the destination array to accommodate the combined elements of both arrays. Then, the function checks the value of `new_data` to determine whether the allocation has failed or not. If the allocation has not failed, `destination->data` points to the newly allocated memory represented by `new_data`, and `destination->capacity` is initialized as `new_capacity`. The if loop is now closed. Now, `void* destination_ptr` is initialized with the memory address where new elements can be added at the end of the existing data in the destination array. It is calculated by summing the memory address of the beginning of the destination array's data and the total amount of memory occupied by the elements in the destination array. Afterwards, `const void* source_ptr` is initialized with the memory address of the starting point of the data in the source array, denoted by `source->data`. Followingly, `size_t bytes_to_copy` is initialized with `source->size * source->element_size` represents the total amount of memory occupied by the elements in the source array. Then `memcpy` function is used to copy a block of memory from `source_ptr` to `destination_ptr`. The destination starting address is represented by `destination_ptr`, the source starting address is represented by `source_ptr`, and the number of bytes to copy is determined by `bytes_to_copy`. Finally, the size of the destination array is initialized to its newly calculated size.