Exercises — Vector

version #

Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

1 Vector

Files to submit:

- vector/Makefile
- vector/*

Provided files:

- vector/vector.h

Makefile: Your makefile should define at least the following targets:

- all: Generate libvector.a

Authorized functions: You are only allowed to use the following functions:

- malloc(3)
- realloc(3)
- free(3)
- printf(3)
- puts(3)
- putchar(3)

Authorized headers: You are only allowed to use the functions defined in the following headers:

- errno.h
- assert.h
- err.h
- stddef.h

1.1 Goal

Even though linked lists are great for dynamic-length data storage, contiguous arrays are still extremely useful. Sadly, dynamic-length contiguous array manipulation can be hard. In this exercise, you will implement a *vector*, an abstraction of a dynamically-sized array.

```
struct vector
{
    size_t size;      // Number of elements in the vector
    size_t capacity;  // Maximum number of elements in the vector
    int *data;        // The elements themselves
};
```

The structure contains the following fields:

- `size`: the number of elements currently contained in the array
- `capacity`: the maximum number of elements which can currently be contained in the allocated array

- `data`: the data array itself

The vector adopts the following behavior:

- If a vector needs to be expanded, because the size exceeds the capacity, the vector capacity is doubled.
- On the contrary, as soon as the size of the vector is lower than half the capacity, the vector capacity is divided by two to save space.

A header (`vector.h`) containing all the required functions is provided.

You **may** add this header to your submission, and you **must not** modify it.

1.2 Makefile

Your submission must contain a `Makefile` with at least following rule:

- **all**: rule must create a static library `libvector.a` containing the requested functions at the root of the `vector/` directory.

Be careful!

Your are free to create the files you want as long as your Makefile creates the `libvector.a` static library.

If the rule `all` does not create the static library, your code will not be tested.

1.3 vector_init

```c
struct vector *vector_init(size_t n);
```

This function creates and returns a pointer to an empty vector with a capacity of `n`, or `NULL` if an error occurs.

The case with 0 will not be tested.

1.4 vector_destroy

```c
void vector_destroy(struct vector *v);
```

This function destroys the vector `v` and frees the used memory.

1.5 vector_resize

```
struct vector *vector_resize(struct vector *v, size_t n);
```

This function resizes the vector `v` to allow it to contain at least `n` elements. It returns the modified vector if successful, or `NULL` if an error occurs.

The expected behavior is the following:

- if `n` is lower than the current vector size, its content is reduced to the first `n` elements, the following being destroyed.
- if `n` is greater than the current vector capacity, its content is unchanged and the capacity is increased accordingly.
- if `n` is equal to the current vector capacity, the function does nothing and returns the vector unchanged.

The case with 0 will not be tested.

1.6 vector_append

```
struct vector *vector_append(struct vector *v, int elt);
```

This function inserts the integer `elt` at the end of the vector. The function returns the new vector if successful, or `NULL` if an error occurred. If the vector is too small you will have to resize it.

1.7 vector_print

```
void vector_print(const struct vector *v);
```

This function displays every element of the vector on the standard output, separated by commas; followed by a single line break.

For example:

```
1,2,3,42,51\n
```

If the vector is `NULL` or has no element, a line break is only displayed.

1.8 vector_reset

```
struct vector *vector_reset(struct vector *v, size_t n);
```

This function destroys every element of the vector, leaving it with a size of 0. Its capacity is reset to `n` elements. The function returns the new vector if successful, or `NULL` if an error occurs.

1.9 vector_insert

```
struct vector *vector_insert(struct vector *v, size_t i, int elt);
```

This function inserts the element `elt` at the vector index `i`, shifting every following elements (from the index `i` included) by one position to the right. The function returns the new vector if successful, or `NULL` if an error occurs or if the index is invalid. If an error occurs you should not modify the vector. Note that if the index is equal to the vector size, it is **not** an invalid case, but an identical behavior to `vector_append` is expected. Like with `vector_append`, if the vector is too small you will have to resize it.

For example, after a call to `vector_insert(v, 2, 42)`, the vector $v = [1, 2, 3, 4]$ becomes $v = [1, 2, 42, 3, 4]$.

1.10 vector_remove

```
struct vector *vector_remove(struct vector *v, size_t i);
```

This function deletes the element at the index `i` from the vector, shifting every following element (from the index `i+1` included) by one position to the left. The function returns the new vector if successful, or `NULL` if an error occurs or if the index is invalid. If an error occurs you should not modify the vector.

For example, after a call to `vector_remove(v, 2)`, the vector $v = [1, 2, 42, 3, 4]$ becomes $v = [1, 2, 3, 4]$.

It is my job to make sure you do yours.