Vector

Entire Assignment due **2025-02-10 23:59**
Graded files:

- vector.c
- sstring.c

Learning Objectives

The learning objectives for Vector are:

- Implementing a C++ style Vector in C
- Using Dynamic/Heap Memory
- Using malloc(), free(), calloc(), realloc()
- Function Pointers
- OOP in C
- String manipulation in C

Groundwork For Shell

You are an intern at Macrohard, where you'll be writing a shell for everyone on your team.

These projects will take you several weeks, and your mentor has decided on the following timetable:

- Week A: Vector and Sstream
- Week B+C: Shell

The *Shell* is a terminal. Like all good terminals, your shell will need to remember what processes are running.

However, after hearing tales of your talent, and with vectors being all the rage, other team leads have asked for vectors that they can use in their own projects. One option would be to write a vector for each team. However, being a good programmer, you know that code duplication is bad. Also, you're a lazy programmer, so you want to write as little code as possible to accomplish everything. You decide to implement a generic vector, something that every team can use with minimal changes.

Vector

A vector is an array that grows as a user adds and removes items from it. (Since CS 225 was a prerequisite, you probably knew all of that already.) However, your vector will need to be feature-rich enough for someone to easily create a document from it, or anything else the other sneaky teams want for their projects.

Your implementation should go in `vector.c`, which is the only file that will be sent to your team lead for review. As an intern looking to become a full-time employee, you should create test cases in `vector_test.c` to show you are a responsible programmer. Your mentor has left notes in `vector.h` guiding your implementation.

In case a fellow employee asks what you learned in CS 225, here's some review:

- Lectures
- Array Resizing
- Lecture Recording

Since this vector is generic, it will have to call custom constructor, destructor, and default constructor functions when objects are added or removed. (How is this better than a single function which handles all possible types?) Thus, your vector structure will contain pointers to your constructor or destructor routines, and you can initialize the vector by passing pointers to these functions as parameters.

What you'll end up with is a useful general-purpose vector, capable of dynamically expanding. (No more fixed-sized buffers!). If you get confused about the callback typedefs (i.e. what a copy constructor is) take a look at `callbacks.h`.

Note: Remember that vector size (the number of actual objects held in the vector) and capacity (size of the storage space currently allocated for the vector) are two different things. Refer to documentation in vector.h and vector.c for more information.

sstring

sstring is a wrapper around C-strings which makes dealing with strings easier with higher-level functions. We have not specified the definition of the sstring struct, and left that up to you! You can find the specification for the sstring functions in the `sstring.h` header file.

Managing memory

Remember, `man` is man's best friend. Since you're working with dynamic memory, there are quite a few functions you should be familiar with before you proceed. `malloc()`, `free()`, `realloc()`, `calloc()`, `memmove()` are your friends; don't shy away!

- `man 3 malloc`
- `man 3 free`
- ...there's a pattern here

Undefined behavior

Undefined behavior is a scenario or edge case for which there is no documentation describing how the code should react. For example, man pages do not describe what happens when you feed `NULL` into `strcmp()`. Your mentor will not answer questions like "What if my user wants an element past the end of the vector?", because that is undefined behavior.

So, for the entirety of this MP, you should use `assert()` statements to check that your user is passing valid input to your function before operating on the input. For example, if you were implementing `strcmp(const char *s1, const char *s2)`, then your code might look like this:

```
#include <assert.h>

strcmp(const char *s1, const char *s2) {
    assert(s1 != NULL && s2 != NULL);
    // Compare the two strings
    .
    .
    .
    return rv;
}
```

‼️ **A note about asserts:** `SIGABRT` is a signal that will kill any process. It can be called by writing `abort()`, and is often used by `libc` implementations when unrecoverable errors occur. For example if you damage `malloc`'s internal structure via heap overflow, `malloc` will call `abort()` and your program will exit immediately. Processes will also be terminated via `SIGABRT` if an `assert` fails. `SIGABRT` is also known as "signal 6" - if you see this pop up on the autograder, your implementation is either incorrect (or failing your own `asserts`) and causing the autograder to crash.

`INITIAL_CAPACITY`

`INITIAL_CAPACITY` is the starting capacity of the vector, initialized to 8. Do not modify this field, as doing so may cause you to fail tests.

Callbacks

A set of callback functions representing copy constructors, default constructors, and destructors for different types have been provided in `callbacks.h`. Please read `callbacks.h` and `callbacks.c` carefully, especially the function documentation, to understand when to use each one accordingly. The function pointers defined in the vector struct can store function pointers to the defined callback functions, which allows for flexibility and emulation of Object-Oriented Programming in C. You can also learn more about callbacks and how to use them here or here.

Writing test cases

Just to emphasize how important test cases are, this lab spec will repeat itself and remind you that as good programmers, you are expected to write your own test cases for sstring and vector.