Vec and Str - Abstraction and Testing

Lab 3 - COMP211 - Spring 2020

Abstract

In this lab you will implement `Vec` and `Str` abstractions and gain practice with writing unit tests to guide your development, increase the robustness of your implementations and your confidence in it, and protect your project's code from regressions. The implementations of each function in this lab are short, but will require attention to detail. This lab is designed for you to practice test-driven development as discussed in lecture.

The `Vec` abstraction offers generic, dynamically sized array-like storage. The `Str` abstraction is layered atop `Vec` and offers a more convenient and expressive API for specifically dealing with dynamically sized character string values.

These abstractions will serve as base components in our Tar Hell Shell project: **`thsh`**

Accept the following GitHub classroom assignment: https://classroom.github.com/a/nRS9nCpE

With your project repository open, click the green "Clone or Download" button. Copy the HTTPS URL. Back in a `learncli` container, clone this url with the `git clone <URL>` subcommand. Replace `<URL>` with your copied URL. (Right click to paste in Windows.) After cloning, change your working directory to the cloned repository.

Part 0 of 4. Project Familiarization

In this stage of the project you are primarily focused on the library functionality of `Vec` and `Str`. There is no meaningful `main` functionality in this lab. Your work will be in the testing and implementation of an application programming interface (API) specification.

The three files to peruse (and perhaps open side-by-side-by-side in `vim`) are:

1. `include/Vec.h`
2. `src/Vec.c`
3. `test/unit/VecImpl.c`

You do not need to make changes to `include/Vec.h` in this lab. Your job will be to complete the implementations of each of the functions defined in this header file. Before beginning any work, you should read through the comments and function signatures of Vec's header file.

Next, read through the skeleton code provided in `src/Vec.c`. Implementations of `value`, `drop`, `length`, and `ref` are provided. Note the `Vec_ref` function's bounds check and error handling. A design decision of our little library is to exit early on index out of bounds operations.

Part 1 of 4. `Vec` Implementation

Your job in `src/Vec.c` is to implement the following functions declared in `Vec.h`:

1. `Vec_get`
2. `Vec_set`
3. `Vec_equals`
4. `Vec_splice`

The semantics of each function is described in the comments of `Vec.h`.

Before implementing any of these functions beyond a skeleton, you should write unit tests that demonstrate expected behavior but will fail until you implement the function definition. As you're testing the *implementation* of `Vec`, add your tests to `VecImpl.cpp`. Start with the simplest tests you can imagine and add additional tests that are more complex. In implementation tests you are allowed to access the members of the `Vec`. Example tests for the functions provided and a demonstration of testing for out of bounds will help get you started with writing tests. You are encouraged to make commits as you get each test you write to pass.

The `Vec_splice` function, in particular, is best approached in a deliberate test-driven fashion. You should have individual tests for the many possible use cases of this function. Name each use case as meaningfully as possible. You should be able to identify at least eight substantively different use cases to test the correctness of.

A significant portion of the grade for this assignment is based on the quality and thoughtfulness of your tests.

Once you believe your implementation of each function is correct, open `test/unit/VecSpec.cpp` and write tests from the *user's* perspective. These tests should *only* make use of the abstraction's functions defined in `Vec.h` to test expectations from the user's perspective *above* the barrier of abstraction. More specifically, you should not need to access any of the struct's members directly in these tests.

Part 2 of 4. `Vec` Improvements

Once your tests are written, checkout a new branch named `feature-improve-vec`. In this branch your goal is to improve the code in your `Vec` implementation in two speficic ways.

First, rewrite your `Vec_set` function implementation to simply dispatch to `Vec_splice` instead. After rewriting `Vec_set`, you should be able to run your tests and pass them all without changing your tests. If any tests fail that means your rewrite contains a regression you should address before continuing on. Once your improvement is passing all tests, make a commit.

Second, make any other improvements to your code you can think of. Improve the names of variables, refactor redundant code into static helper functions, and so on. After each attempted improvement, rerun your unit tests to convince yourself you have not regressed. After making a refactoring that passes all tests, make a commit! If you try something you need to revert after making a prior known-good commit, just run `git status` and follow the instructions it provides on "discarding changes in the working directory".

Once you believe the *quality* of your code is A++, make a commit, checkout your `master` branch and merge in your `feature-improve-vec` branch without fast-forwarding.

Part 3 of 4. `Str` Implementation

The `Str` data type offers a dynamically allocated, arbitrary-length character array for working with "string" data. It is implemented as a thin layer of abstraction sitting atop `Vec`. Since its item type is concrete (`char`), the function signatures are more ergonomic to use than `Vec`'s generic functions. You will follow the same process implementing and testing `Str` as you did `Vec`, so you should start by reading and thinking through the implications of `include/Str.h` first.

An important *invariant* of a `Str` is that its buffer contents must *always* terminate with a null `\0` character.

To adhere to the principles of *abstraction barriers*, you should *only* rely on the functions the `Vec` type provides to implement `Str`. You will notice in the starter code the `Str_value` function breaks this principle, but you can fix that now that you've correctly implemented `Vec_set`.

For `StrImpl.cpp` tests, you should focus less on testing anything proven in `Vec` and more on what functionality is special to a `Str`. For `StrSpec.cpp` tests, follow a similar strategy of your `VecSpec` tests which demonstrate the functionalities of a `Str` from the user's perspective.

Part 4 of 4. `main.c` Line Echo

Write a simple program in `main.c` that makes use of your `Str` type. It should use a fixed sized `char` buffer and `fgets` to read chunks of a line that get appended to a `Str` value. After a line is input and stored in a `Str`, simply print the contents of the `Str` back out and repeat until end of file is reached. Your program should not leak any memory nor have access errors in `valgrind`.

Grading

- 50% autograded correctness - testing will be slow to come online to incentivize your personal test authorship
- 35% unit test coverage
  - Are your tests well organized?
  - Do your tests cover the important cases of each function?
  - Do your spec tests avoid accessing members directly? Do they "prove" the functionality a user can expect?
- 15% style
  - Did you choose good names for variables?
  - Is your code well formatted and organized?
  - Did you avoid unnecessary repetition of concepts?
  - Are your implementations relatively straightforward and easy to follow?