CS 211 Homework 3

Winter 2022

Code Due:	January 27, 2022, 11:59 PM, Central Time
Self-Eval Due:	January 30, 2022, 11:59 PM, Central Time
Partners:	Yes; mark you partner on Gradescope on each submission

Note that this is the first part of a two-part assignment. It may be better to continue with the same partner (and code) for the second part, but you will not be required to.

Purpose

The goal of this assignment is to get you programming with more complex allocation patterns than you have previously.

Preliminaries

Login to the server of your choice and *cd* to the directory where you keep your CS 211 work. Then unarchive the starter code, and change into the project directory:

```
% cd cs211
% tar -kxvf ~cs211/hw/hw03.tgz
⋮
% cd hw03
```

If you have correctly downloaded and configured everything then the project should build cleanly:

```
% make
⋮
cc -o test_vc test/test_vc.o src/libvc.o -l211 -fsaniti...
%
```

This homework assignment must be completed on Linux by logging into a Linux workstation. Each time you login to work on CS 211, you should run *211* to ensure your environment is setup correctly. (If you get an error saying that 211.h doesn't exist, that probably means you missed the step in Lab 1 where you needed to run ~cs211/setup211.)

Introduction

In this project, you will implement a library *vc* for counting votes and a small client program *count* that exercises the library.

An important idea throughout this assignment is to adhere to the specified ownership protocol for managing memory. In the library, you will implement operations for an abstract type `vote_count_t` that points to a mapping from candidate names to their vote counts. A `vote_count_t` object **owns** the strings that hold the names of the candidates, so whoever frees the `vote_count_t` object is responsible for freeing its strings as well.

Orientation

As in previous homeworks, your code is divided into three `.c` files:

Contents

- Most significant functionality will be defined in the "*vc* library," src/libvc.c.
- Tests for those functions will be written in test/test_vc.c.
- The `main()` function that implements the *count* program will be defined in src/count.c.

Function signatures for src/libvc.c are provided for you in src/libvc.h; since the grading tests expect to interface with your code via this header file, **you must not modify src/libvc.h in any way.** All of your code will be written in the three .c files.

Make *targets*

The project also provides a Makefile with several targets:

target	**description**
test	builds everything & runs the tests * &
all	builds everything, runs nothing &
test_vc	builds the unit tests
count	builds the *count* program
clean	removes all build products &

* default   & phony

Target test is the default, which means you can run it by typing `make` alone, with no target name.

Specifications

The project comprises two functional components, which are specified in the next two subsections.

The count *program*

The *count* program [*hint»*] reads candidate names, one per line, from the standard input. It counts the number of occurrences of each candidate name, and when the input ends, it prints a table of candidate names and counts to the standard output, like so:

```
% ./count
kennedy
nixon
nixon
kennedy
kennedy
^D
kennedy                     3
nixon                       2
```

In the terminal, pressing Control-D (only at the beginning of a line) sends the end-of-file signal.

The *count* program is limited in how many different candidates it can handle, and the limit is defined using a C preprocessor macro `MAX_CANDIDATES` in the `src/libvc.h` header file. When *count* is given more different candidates than it can handle, it begins dropping votes. Each time it sees a candidate that it hasn't seen before and doesn't have room for, it prints a message to `stderr`. At the end, if there are any dropped votes, it prints the total count of dropped votes to `stderr` before terminating with exit code 2.

You can print messages to `stderr` using *fprintf*(3). The first argument to the function should be the word stderr, which denotes that you want to print to standard error, and the remaining arguments work just like *printf*(3).

So for example, if `MAX_CANDIDATES` were only 2, it would behave like this:

```
% ./count
perot
bush
clinton
./count: vote dropped: clinton
clinton
./count: vote dropped: clinton
clinton
./count: vote dropped: clinton
bush
^D
perot                          1
bush                           2
./count: 3 vote(s) dropped
[2]% echo $status
2
%
```

I'm using underlining to indicate what the program prints to the standard error.

In *fish*, the special shell variable `$status` contains the exit code of the most recently run command. (Most other shells use `$?` for the exit code.)

If the program fails to allocate memory, it exits with a message printed to `stderr` and an exit code of 1.

The vc *library*

The header src/libvc.h defines one type, intended to represent a mapping from candidate names to vote counts:

```
typedef struct vote_count* vote_count_t;
```

This type is abstract in the sense that other files that include src/libvc.h will know that type `vote_count_t` is a pointer to some struct type, but they won't know anything about the definition of that struct. This means that they can create, manipulate, and destroy `struct vote_count` objects only via the functions declared in the same header.

We will refer to the object that a `vote_count_t` points to as a *vote count map*. The src/libvc.h header declares eight functions for working with vote count maps: two for managing their lifecycles, one for modifying them, and five for querying them. The functions are:

- `vote_count_t vc_create(void)` allocates a new, empty vote count map on the heap, initializes it [*invariant hint»*] , and returns a pointer to it. Every successful call to `vc_create()` allocates a new object that must subsequently be deallocated exactly once using `vc_destroy`.

  Ownership: The caller takes ownership of the result.

  Errors: Returns `NULL` if memory cannot be allocated.

- `void vc_destroy(vote_count_t vc)` deallocates all memory associated with `vc` [*ownership hint»*] . `vc` may be `NULL`, in which case this function does nothing.

  Ownership: Takes ownership of `vc`.

  Errors: If `vc` has already been destroyed or wasn't returned by `vc_create()` in the first place then this function has undefined behavior.

- `size_t* vc_update(vote_count_t vc, const char* name)` **does not update a count.** Rather, it returns a pointer to the count for candidate `name`, so that the caller can use that pointer to update the count. If `name` is already present in `vc` the returned pointer will point to the existing count for candidate `name`; otherwise, `vc` is extended to map `name` to a count of 0 before returning the pointer to that count. [*iteration hint»*] [*ownership hint»*] [*helper hint»*]

  Ownership:

  - Borrows `name` transiently, which means that it does not store it anywhere. (In other words, `vc` must still be valid even after `name` is not.) Instead, it must copy the string contents.
  - Borrows `vc` transiently.
  - The returned pointer is borrowed from `vc` and is valid until `vc` is destroyed.

  Errors:

  - Returns `NULL` if `name` is not present in `vc` and cannot be added because `vc` is full.
  - Prints a message to `stderr` and exits the program with code 1 if we need to allocate a copy of `name` and allocation fails.

- `size_t vc_lookup(vote_count_t vc, const char* name)` looks up the count for candidate `name`; returns 0 if not found. [*iteration hint»*] [*helper hint»*]

  Ownership: Borrows both arguments transiently.

- `size_t vc_total(vote_count_t vc)` returns the total number of votes cast (not counting any dropped votes). [*iteration hint»*]

  Ownership: Borrows `vc` transiently.

- `const char* vc_max(vote_count_t vc)` returns the name of the candidate with the most (non-zero) votes. In case of a tie, returns the candidate who was added to `vc` *earlier*.

  Returns `NULL` if `vc` contains no candidates with more than zero votes. [*iteration hint»*]

  Ownership:

  – Borrows `vc` transiently.

  – The returned pointer is borrowed from `vc` and is valid until `vc` is destroyed.

- `const char* vc_min(vote_count_t vc)` returns the name of the candidate with the fewest (non-zero) votes. In case of a tie, returns the candidate who was added to `vc` *later*.

  Returns `NULL` if `vc` contains no candidates with more than zero votes. [*iteration hint»*]

  Ownership:

  – Borrows `vc` transiently.

  – The returned pointer is borrowed from `vc` and is valid until `vc` is destroyed.

- `void vc_print(vote_count_t vc)` prints a summary of the vote counts on `stdout`. The counts are printed one candidate per line in the order they first were added. The candidate names are left-aligned in a 20-character column, followed by a single space, and then the counts right-aligned in a 9-character column. [*printf(3) reference»*] [*iteration hint»*]

  Ownership: Borrows `vc` transiently.

Note that *libvc* is not responsible for maintaining any information about dropped votes. That counting must be handled by the client program.

Hints

In this section we provide suggestions, such as some useful helper functions and help interpreting the specification.

Representation invariant [*«vc_create() spec*]

If there are n candidates mapped in `vc` then the `candidate` fields of the first n elements of `vc` must contain their names, and the remaining `candidate` fields (if $n <$ `MAX_CANDIDATES`) must be `NULL`. This is so that you know when to stop when searching for a candidate or for a free slot.

To work properly, all of the functions in src/libvc.c must collaborate to maintain each vote count map in a consistent state.

The first n `count` fields, corresponding to the n candidate names, must contain those candidates' counts. It does not matter what the remaining (`MAX_CANDIDATES` $- n$) `count` fields contain (or even whether they are initialized), since they do not store any information until their corresponding `candidate` fields are non-`NULL`.

Iterating over a vote count map

Most of the functions in src/libvc.c need to iterate over the array that their `vote_count_t` argument points to. Be careful, because this iteration requires different termination conditions in different places. In particular, it always needs to stop before `MAX_CANDIDATES`, but often it is also necessary to stop when reaching a `NULL` candidate name.

[*«vc_update() spec*] [*«vc_lookup() spec*] [*«vc_total() spec*] [*«vc_max() spec*] [*«vc_min() spec*] [*«vc_print() spec*]

Ownership strategy [*«vc_destroy() spec*] [*«vc_update() spec*]

A vote count map owns the strings that store the candidate names, but the `vc_update()` function merely borrows the name that it is given. This has several implications:

- In order to store the name of a candidate that it has not yet seen, the implementation of the `vc_update()` function needs to make its own copy of the `name` parameter on the heap.
- Clients of `vc_update()` are free to deallocate or reuse the `name` parameter that they pass to `vc_update()` as soon as `vc_update()` returns.
- Properly deallocating the memory associated with a `vote_count_t` (as in `vc_destroy()`) means deallocating all of the strings it owns.

Strategy for the count *program* [*«spec*]

The *count* program should start by allocating a vote count map, terminating with an error message on `stderr` and exit code of 1 if allocation fails. (Use the predefined `OOM_MESSAGE` as your format string.)

Next, it should to read a line at a time using *read_line*(3) until end-of-file. Each string returned by *read_line*() is a candidate name and should be counted in the vote count map, unless calling `vc_update()` indicates that the vote count map is full. (Use `DROP_MESSAGE` to format the required warning when dropping a vote.) Don't forget to free each string allocated by *read_line*().

Once there are no more votes to count, it should print the vote summary and deallocate the vote count map.

Finally, if any votes were dropped, print a final warning (use `FINAL_MESSAGE`) before terminating with exit code 2. Of course, if no votes were dropped, the exit code should be 0.

Helper functions [*«vc_update() spec*] [*«vc_lookup() spec*]

You may factor the required functions however you like, but when writing our solution, we found the following helper functions to be, well, helpful:

The *storage class* `static` makes a function definition local to the `.c` file it is written in, so `static` should be applied to all helper functions.

```
// Returns a pointer to the first element of `vc`
// whose `candidate` matches `name`, or NULL if not found.
static struct vote_count*
vc_find_name(vote_count_t vc, const char* name);

// Returns a pointer to the first element of `vc` whose
// `candidate` is NULL, or NULL it's full.
static struct vote_count*
vc_find_empty(vote_count_t vc);

// Clones a string onto the heap, printing a message to
// stderr and exiting with code 1 if malloc() fails.
static char*
strdup_or_else(const char* src);
```

Reference

Alignment using printf(3) [*«vc_print() spec*]

For printing the table of counts, you will want to use *printf*(3)'s padding and alignment capabilities. In particular:

- A field may be padded to n characters by adding the number n between the `%` and the type specifier (*e.g.*, `s`, `d`, or `zu`). For example, `"%8d"` formats an `int` using (at least) eight characters.
- By default, fields are padded with spaces on the left, in order to right align them. Using a negative number will left align the field instead. For example, `"%-8d"` will format `int`s left-aligned in an eight-character column.

Testing with automatically generated names

For testing *libvc*'s behavior when full, you will need to generate `MAX_CANDIDATES + 1` different candidate names. And your tests should still work when I redefine `MAX_CANDIDATES`. So you need a method to automatically generate name strings.

The easiest way to do this is with *snprintf*(3). It's a lot like *printf*(), but instead of printing to `stdout`, it takes a `char*` and writes the characters into the array that points to.

In addition to the buffer to format into, *snprintf*() takes an upper limit on the number of characters to store; an older function, *sprintf*(3), does not take such a limit. Why might that be a bad idea? (Hint: it might go past the end of the array!)

An easy way to use *snprintf*() is to first create a sufficiently large `char` array as a local variable. Then, inside a loop keep track of the number of iterations. You can write to that array with *snprintf*() and the iteration count to generate a new candidate name, and then call *vc_update*() to add them. You can reuse the same `char` array for each call to *vc_update*().

See the snprintf reference for more information.

Deliverables & evaluation

For this homework you must:

1. Implement the specification for the *vc* library in src/libvc.c.
2. Implement the specification for the *count* program in src/count.c.
3. Add more test cases to test/test_vc.c in order to test the eight functions that you defined in src/libvc.c.

The file test/test_vc.c contains two test cases in order to give you an idea how to write them, but you need to add many more tests. Try to cover all the possibilities, because for this week's self evaluation we will spot-check your test coverage by asking for just a few particular test cases. You can't anticipate which we'll ask about, so you should try to cover everything. Note that you should have test cases for edge cases, even if the autograder has tests for them as well.

Grading will be based on:

- the correctness of your implementations with respect to the specifications,
- the presence of sufficient test cases to ensure your code's correctness, and
- adherence to the CS 211 Style Manual.

Submission

Homework submission and grading will use Gradescope. You must include any files that you create or change. For this homework, that will include src/libvc.c, src/count.c, and test/test_vc.c. (You must not modify Makefile or src/libvc.h.)

Per the syllabus, if you engaged in arms-length collaboration on this assignment, you must cite your sources. You may write citations either in comments on the relevant code, or in a file named README.txt that you submit along with your code. See the syllabus for definitions and other details.

Submit using the command-line tool `submit211`. You can run the command with the `--help` flag to see more details. The tool will ask you to log in with your Gradescope credentials, so make sure you've created an account!

To submit the necessary files for this homework, you will run something that looks like:

```
% submit211 submit --hw hw03 src/libvc.c src/count.c test/test_vc.c
```

Remember that those are relative paths to the files you want to submit. So make sure to change them to make sense for whatever directory you are running the command from. You can also add any additional files you want to upload, like README.txt, to the end of the command.

Partners

If you work with a partner, then you MUST register your partnership on Gradescope. From the Gradescope course page, click on the assignment, then at the bottom click the button labeled "Group Members". A dropdown menu will allow you to select the name of your partner, then click save to send them an email. Group members can also be removed by the same process. For more details see the help page on Gradescope.

Please remember to mark your partner on each submission.