CS 211 Homework 3

Spring 2023

Part 1
Code Due: April 27, 2023, 11:59 PM, Central Time
Self-Eval Due: April 30, 2023, 11:59 PM, Central Time
Maximum allowed Gradescope submissions: 30

Part 2
Code Due: May 04, 2023, 11:59 PM, Central Time
Self-Eval Due: May 07, 2023, 11:59 PM, Central Time
Maximum allowed Gradescope submissions: 30

This assignment builds on HW 2 and is to be completed and submitted in two parts, with each part having its own Gradescope assignment. Each part has its own self-evaluation and you may use your late days for each part and submit each part at a penalty as if they were separate assignments. Part 2 builds on part 1 and so you need to complete part 1 in order to be able to implement and test part 2. All part 1 deliverables must be submitted by the last part 1 deadline (which is the last day you can submit at a penalty) to get credit for those portions. That is, if you miss the last part 1 deadline and complete everything only in time for part 2, your grade for the assignment will only be based on part 2.

See the end of the document for the deliverables for each part.

Advice for reading this document: This document has several components that refer to each other and there several implementation hints towards the end of the document that are referenced throughout the descriptions of the specifications. Make sure to read through the whole document once and absorb the contents so that you are aware of all the information present, even if there are many parts you do not understand as you are reading it. It is likely that your question will be answered (even partially) later in the document. At that point, if you are still unsure of something the assignment is asking, feel free to ask course staff for clarification.

Contents

Purpose

The goal of this assignment is to solidify your C programming skills before moving on to C++.

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:

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 forgot to run *211*.)

```
% cd cs211
% tar -kxvf ~cs211/hw/hw3.tgz
⋮
% cd hw3
```

Preliminaries

If you have correctly downloaded and configured everything then the project should build cleanly (although not all tests will pass):

```
% make
⋮
cc -o irv src/irv.o src/ballot.o src/ballot_box.o sr...
%
```

Background

"Instant-runoff voting (IRV)," according to Wikipedia,

> is a type of preferential voting method used in single-seat elections with more than two candidates. Instead of voting only for a single candidate, voters in IRV elections can rank the candidates in order of preference. Ballots are initially counted for each elector's top choice, losing candidates are eliminated, and ballots for losing candidates are redistributed until one candidate is the top remaining choice of a majority of the voters. When the field is reduced to two, it has become an "instant runoff" that allows a comparison of the top two candidates head-to-head.

For an example of running the IRV algorithm, consider an election in which there are three candidates running and five voters. The initial ballots are as follows:

1. Abbott	1. Campbell	1. Borden	1. Abbott	1. Campbell
2. Borden	2. Abbott	2. Campbell	2. Borden	2. Abbott
3. Campbell	3. Borden	3. Abbott	3. Campbell	3. Borden

We count the first vote on each ballot, after which the vote counts are 2 for Abbott, 2 for Campbell, and 1 for Borden. No one has an outright majority, so the last-place candidate, Borden, is eliminated. Thus, in the second round of counting, the ballots are:

1. Abbott	1. Campbell	1. ~~Borden~~	1. Abbott	1. Campbell
2. ~~Borden~~	2. Abbott	2. Campbell	2. ~~Borden~~	2. Abbott
3. Campbell	3. ~~Borden~~	3. Abbott	3. Campbell	3. ~~Borden~~

On the ballot where Borden was leading, now Campbell takes the lead. We count the first remaining vote on each ballot, giving 3 to Campbell and 2 to Abbott. Campbell has a majority, and is therefore the winner.

Given n candidates, the algorithm may take as many as $n-1$ rounds of counting and elimination to reach a winner.

Code structure

Your code will be divided into several `.c` files:

- Functions on ballots are declared in src/ballot.h, and their definitions belong in src/ballot.c. Unit tests for these functions should be written in test/test_ballot.c. These ballot and implementation tests constitute part 1 of this assignment.
- Functions on collections of ballots, including the IRV algorithm, are declared in src/ballot_box.h, and their definitions belong in src/ballot_box.c. Unit tests for these functions should be written in text/test_ballot_box.c. These ballot box implementation tests constitute part 2 of this assignment.
- The `main()` function implementing the *irv* client program is already written for you in src/irv.c. You may however, edit this function to do intermediate manual tests of your ballot library while working on part 1.

The code in src/ballot.c and src/ballot_box.c depends on *libvc*. By default, compile the program with our own shared library solution. If you prefer, you can use your own libvc.c from Homework 2 by copying it into the src/ directory. The build system will automatically use your file if it finds it.

Make *targets*

The project provides a Makefile with several targets you can run your program with:

target	description
test	builds everything & runs the tests * &
all	builds everything, runs nothing &
test_ballot	builds the ballot tests
test_ballot_box	builds the ballot box tests
irv	builds the *irv* 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 overview

The project comprises three functional components, which are specified in the next three sections. These sections describe the expected

functionality and also **link to important hints related to iteration, ownership, and testing later in the document**. The *irv* program contains the `main()` function and so this file uses the libraries described in the second and third sections. The ballot library (src/ballot.c and src/ballot.h) stores and handles information about one ballot. The ballot box library (src/ballot_box.c and src/ballot_box.h) stores and handles information about a collection of multiple ballots.

The three functional components described in the following three sections work together as follows: the *irv* program depends on the ballot box library and calls functions within this library; the ballot box library in turn depends on the ballot library and calls functions within this ballot library as part of its own function implementations.

Specification: The irv *program*

The *irv* program, as shown in the margin to the right, reads ballots from the standard input, standardizes names to remove all non-alphabetic `char`s and converts all `char`s to uppercase letters, runs the IRV algorithm, and prints the name of the winner.

The only other thing that *irv* may print is an out-of-memory error message if `malloc` fails.

The format of the input is as follows. Each candidate's name appears on its own line, with the candidates on each ballot listed in order. Each ballot is terminated by a percent sign (`%`) on a line by itself, except for the last ballot, which may be terminated by the end of the input.

Like *count* from HW 2, the *irv* program is limited in how many different candidates it can handle, and as before, the limit is defined using a C preprocessor macro `MAX_CANDIDATES` in the `src/libvc.h` header file. Similarly to HW 2, if you want to modify this value, you need to pass in the change to *make*. For example, to test with three candidates you would run:

```
% make test SIZE=3
%
```

If *irv* sees more different candidates than it can handle, it exits with error code 3. If *irv* fails to allocate memory, it exits with a message printed to `stderr` and an exit code of 1.

```
% ./irv
Abbott
Borden
Campbell
%
Campbell
Abbott
Borden
%
Borden
Campbell
Abbott
%
Abbott
Borden
Campbell
%
Campbell
Abbott
Borden
^D
CAMPBELL
%
```

Specification: Ballot library (Part 1)

Functions in src/ballot.c

In this file, you will implement a ranked-choice ballot as a heap-allocated struct containing an array of candidate names. At a high

level, when initially added to the ballot, names are *active*, but they may be *eliminated* from the ballot as the IRV algorithm proceeds.

The header src/ballot.h defines one type [*invariant hint»*] , intended to represent a single voter's ranked-choice ballot.

```
typedef struct ballot* ballot_t;
```

By this statement, `ballot_t` is now an alias for the `struct ballot*` data type. This `ballot_t` type is abstract in the sense that other files that include src/ballot.h will know that type `ballot_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 ballot` objects only via the functions also declared in src/ballot.h.

We will refer to the object that a `ballot_t` points to as a *ballot*. The src/ballot.h header declares eight functions for working with ballots: two for managing their lifecycles, two for modifying them, two for querying them, one for reading a ballot from an input stream, and one for formatting a ballot on output stream. Additionally, it declares a function for standardizing candidate names.

- `ballot_t ballot_create(void)` allocates a new, empty ballot (i.e., the fields of the allocated ballot should represent an empty ballot) on the heap and returns a pointer to it. Every successful call to `ballot_create()` allocates a new object that must subsequently be deallocated exactly once using `ballot_destroy`. [*hint»*]

  Ownership: The caller takes ownership of the result.

  Errors: Exits with error code 1 if memory cannot be allocated. You can make use of the `mallocb()` helper function available to you in src/helpers.c to allocate memory and handle errors accordingly. (You may want to use this in other parts of your program too, where applicable.)

- `void ballot_destroy(ballot_t ballot)` deallocates all memory associated with `ballot`. `ballot` may be `NULL`, in which case this function should do nothing (your code will need to check for this).

  Ownership: Takes ownership of `ballot` and frees it.

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

- `void ballot_insert(ballot_t ballot, char* name)` standardizes `name` using the `clean_name` function (described below), adds it to the first unused entry in the ballot, and marks it active. [*hint»*]

Ownership:

- Borrows `ballot` transiently.
- Takes ownership of `name`, which means that 1) `name` must have been allocated with `malloc` and owned by the caller prior to the call, and 2) the caller cannot access `name` again after the call.

This means that the borrowing is finished when `ballot_insert` returns.

Errors: Exits with error code 3 if the ballot is full (*i.e.,* adding this name would exceed `MAX_CANDIDATES`).

- `void ballot_eliminate(ballot_t ballot, const char* name)` marks candidate `name`, if present, as no longer active.

  Ownership: Both arguments are borrowed transiently.

One important principle of API design is that what can be done can also be undone. The presence of `ballot_eliminate` without a `ballot_reinstate` to reverse it violates this principle, but the algorithm doesn't require it, so you don't need to implement it.

- `const char* ballot_leader(ballot_t ballot)` returns the first still-active candidate, or `NULL` if no active candidates remain.

  Ownership: The result is borrowed from the `ballot` argument and is valid only as long as the argument is.

- `void count_ballot(vote_count_t vc, ballot_t ballot)` counts a ballot into an existing `vote_count_t vc` passed in by incrementing the count in the map of the leading (first active) candidate in `ballot`. If there is no leading candidate then this function has no effect.

  Ownership: Both arguments are borrowed transiently.

  Errors: If there is no more room in the vote count map (meaning `vc_update` returns `NULL`) then it exits with error code 4.

- `ballot_t read_ballot(FILE* inf)` returns a single ballot read from input file handle `inf` or returns `NULL` if there is no input. In particular, if the first time reading from `inf` in this function indicates end-of-file, this function should return `NULL`. Otherwise, it should build and return a ballot by reading one name per line until reaching either end-of-file or a percent sign on a line by itself; it ensures that each candidate name is standardized using `clean_name()` before storing it in the ballot.

  In C, `FILE*` is the type for representing an open file. To read a line at a time from a `FILE*`, use the *lib211* function *fread_line*(3), which is like *read_line*(3) but takes a `FILE*` argument to read from.

  Ownership:

  - The argument is borrowed transiently.
  - The caller takes ownership of the result and must deallocate it with `ballot_destroy` when finished with it.

  Errors:

  - Exits with an error code 1 if memory cannot be allocated (*fread_line*(3) does this automatically).

  - Exits with error code 3 if the number of names read exceeds `MAX_CANDIDATES`.

- `void print_ballot(FILE* outf, ballot_t)` prints a ballot (name and active status of each entry) to the given file handle in a human-readable format to help you with debugging. For example, you can print each entry's information to a separate line.

  This function is implemented for you.

  Ownership: Both arguments are borrowed transiently.

The `FILE` type is declared in `<stdio.h>`. A `FILE*` representing an open file on disk may be obtained using the *fopen*(3) function and released using the *fclose*(3) function. The console streams `stdin`, `stdout`, and `stderr` are pre-opened `FILE*`s.

- `void clean_name(char* name)` [*hint*»] standardizes argument `name` *in-place* by removing all non-alphabetic `char`s and converting all lowercase letters to uppercase.

  Ownership: The argument is borrowed transiently.

  Implementation hint: This function is specified to transform a string "in place," which means that it doesn't allocate, but modifies the `char`s in the the string it is given. Such an approach was not possible for `expand_charseq` because the string often gets longer. But when all we want to do is filter and/or map `char`s one by one, doing it in place is straightforward.

  To do so, you need to track two positions in the same string, which we will call the source and the destination. We consider each source character in turn until the source position reaches the terminating `'\0'`. To retain and map a source character, we convert it, store the result at the destination, and then advance both positions. To remove a character, we merely advance the source position. Notice that as we remove `char`s, the destination position falls behind the source position, but it can never get ahead, which means we are never in danger of overwriting the source before we get there.

Alternatively, you can use two `char*`s (addresses in the string) that both move forward, or one fixed `char*` and two `size_t` offsets that move throughout the string, although we recommend the latter.

  Converting a character should be done with the *isalpha*(3) and *toupper*(3) functions found in the `ctype.h` library.

  When the loop terminates, we must store a terminating `'\0'` at the destination position of the string before returning.

Ballot representation invariant

[*«ballot spec intro*] [*«ballot_create() spec*] [*«ballot_insert() spec*]

Unlike `vote_count_t`, which was defined as a pointer to an array, `ballot_t` is a pointer to a *struct containing an array*.

A ballot `b` contains `b->length` candidates in the first `b->length` elements of the `b->entries` array, so that prefix of elements must be initialized. For each entry `i < b->length`, `b->entries[i].name` is a non-null pointer to an owned string that has been standardized so that

all of its `char`s are uppercase letters. The `active` field in each entry indicates whether the associated candidate is still in the running or has been eliminated.

The `ballot_t` type uses a different invariant than `vote_count_t` to keep track of how many entries it contains. Rather than storing `NULL` pointers in the candidate names of all unused slots, the `length` field stores the count of how many slots are in use. The remaining `MAX_CANDIDATES - length` elements should be left uninitialized until they are needed to store additional candidates.

This means that iterating over a ballot is simpler than iterating over a vote count map, because the loop condition only needs one comparison—`i < ballot->length`—instead of two like the *libvc* functions did.

Manually testing the ballot library from the irv *program*

The code in src/irv.c is only meant to interact with the ballot box library which in turns interacts with the ballot library (details in the next two sections). However, as you are working on part 1, you may want to manually test your functions and behavior in src/ballot.c by calling them from a client program with a `main()` function.

For the purposes of testing part 1 manually (separately from your unit and integration tests), you may modify irv.c to comment out the given code and you can add code that calls your ballot library functions. You can then run the *irv* program to test out the functionality you want. Make sure to comment out your code and uncomment the original code once you are working on part 2 and plan to run the intended *irv* program.

Specification: Ballot box library (Part 2)

Functions in src/ballot_box.c

In this file, you will implement a collection of owned ballots as a linked list [hint»] . This collection, which we will call a *ballot box*, is the main data structure on which the IRV algorithm, also defined in this file, will operate.

The header src/ballot_box.h defines one type [hint»] , intended to represent a whole ballot box.

```
typedef struct bb_node* ballot_box_t;
```

This type is abstract in the sense that other files that include src/ballot_box.h will know that type `ballot_box_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 modify, query, and destroy ballot box objects only via the functions also declared in src/ballot_box.h. However, unlike the other abstract types we've implemented, the null pointer is a valid `ballot_box_t`, representing the empty ballot box.

The src/ballot_box.h header declares six functions for working with

ballot boxes: one for managing their lifecycles, two for modifying them, one for querying them, one for reading a ballot box from a file or input stream, and the IRV algorithm itself.

- `void bb_destroy(ballot_box_t bb)` deallocates the memory associated with a ballot box, including all of its ballots (which it owns). `bb` may be null. [*ownership hint»*] [*iteration hint»*]

  Ownership: Takes ownership of the argument in order to release its resources.

- `void bb_insert(ballot_box_t* bbp, ballot_t ballot)` adds a ballot to a ballot box at the front of the list. This function takes a pointer to the `ballot_box_t`, or in other words, a `struct bb_node**`, and updates it in place.

  `bbp` is a pointer to the head of the linked list where the head is of type `ballot_box_t`. To add a new ballot to the head of the linked list, this function allocates space for a new `balllot_box_t` object (i.e., pointer to a new node) and sets its `ballot` field to the new ballot. This new `ballot_box_t` node object should now replace the original head of the linked list by making `bbp` point to this new `ballot_box_t` object. This new `ballot_box_t`' `next` field should then be set to the old linked list head.

  Preconditions (these are assumed; you shouldn't check for them):

  – `bbp` is non-null.

  – `*bbp` is initialized (but may be null).

  Ownership:

  – Borrows `bbp` transiently, but takes ownership of the old values of `*bbp`, in the sense that any other references to `*bbp` are invalidated after the call.

  – Takes ownership of `ballot`; thus, the caller must own `ballot` before the call, and must not access it again after `bb_insert` returns.

  Errors: Calls `perror("bb_insert")` and `exit(1)` on out-of-memory (via `mallocb`).

- `void bb_eliminate(ballot_box_t bb, const char* candidate)` eliminates all votes for the given candidate in the ballot box. [*iteration hint»*]
  Ownership: Borrows both arguments transiently.

- `vote_count_t bb_count(ballot_box_t bb)` creates a new `vote_count_t` and uses it to count each ballot's leading candidate (*i.e.,* the candidate returned by `ballot_leader`, if any).

Ownership:

- Borrows the argument transiently.
- The caller takes ownership of the vote count map result and must release it with `vc_destroy`.

Errors:

- Exits with error code 4 if `vc_create` cannot allocate memory.
- Calls `count_ballot`, which exits with error code 4 if the vote count map is full.

- `ballot_box_t read_ballot_box(FILE* inf)` [*testing hint»*] reads ballots from the given file handle using `read_ballot` until there are none left to read.

  Precondition: `inf` must be open for reading, as by *fopen*(3). This will be done for you.

  Ownership:

  - Borrows the argument transiently.
  - The caller takes ownership of the result and must release it with `bb_destroy`.

  Errors: Calls `read_ballot` and `bb_insert`, which exit with a non-zero error code if they cannot allocate memory.

- `char* get_irv_winner(ballot_box_t bb)` [*hint»*] runs the IRV algorithm and returns the name of the winner to be owned by the caller. The algorithm is described in detail in the hints section here.

  Ownership:

  - Borrows the argument transiently.
  - The caller takes ownership of the result and must free it.

  Errors: Returns `NULL` if there are no votes and thus no winner. Exits with error code 1 if memory cannot be allocated. Using the helper functions provided will exit correctly if necessary.

Ballot box representation [*«spec*]

We represent a ballot box as linked list of `struct bb_node`s containing owned `ballot_t`s:

```
struct bb_node
{
    ballot_t         ballot;
    struct bb_node* next;
};
```

Using a linked list allows us to expand smoothly to accomodate any number of ballots (within the limits of memory) without either pre-allocating an array to some limit or implementing dynamic array growth.

Unlike the other pointer-to-struct types we have seen, `ballot_box_t` uses the null pointer as a valid representation. In particular, `NULL` is how we represent the empty ballot box (this empty ballot box is defined as a constant variable named `empty_ballot_box` in `ballot_box.c`, and we only allocate nodes when there are ballots to store.

Linked lists often use a null pointer to represent the empty list.

When non-null, the *head pointer* of a `ballot_box_t` owns the entire list—all of the ballots and all of the list nodes. This means that `bb_destroy` must deallocate all of the ballots and all of the list nodes. And this means that isn't advisable for client code to hold onto pointers to nodes deeper in the list.

Iterating over a linked list [«bb_destroy() spec] [«bb_eliminate() spec]

To iterate over a linked list you need a node pointer `current` to keep track of your position in the list, starting at the head pointer (meaning the value of the `ballot_box_t`, which is either null or points to the first node). The loop termination condition is when `current` is null. Otherise, `current` points to a node, which contains an ballot (`current->ballot`) and a pointer to the next node. To advance along the list, assign the pointer to the next node to `current`:

```
current = current->next;
```

A special case of iterating over a linked list is deallocating the list, in which case the assignment above does not suffice. In `bb_destroy`, care must be taken to save each `current->next` in a temporary variable before freeing each `current`.

Hints

In this section we provide suggestions, including descriptions of some algorithms and more help interpreting the specification.

Ownership strategy [«bb_destroy() spec]

A ballot box owns all of its ballots, and the ballots, in turn, own all of the candidate name strings. This implies that `bb_destroy(bb)` must free all of `bb`'s nodes and call `ballot_destroy` on all of `bb`'s ballots; and it implies that `ballot_destroy(ballot)` must in turn free all of `ballot`'s candidate names before freeing `ballot`.

Unlike `vc_update`, which takes a borrowed string, `ballot_insert` takes ownership of the string that it is passed. This makes sense because `ballot_insert` always (except in error cases) needs ownership of the string, whereas `vc_update` only needs ownership when encountering a candidate name that is not yet in the given vote count map. This contract has implications for `ballot_insert`'s caller: the caller must pass a string that it owns (which implies heap allocation by the caller this time). And because the caller gives up ownership, it must not access or attempt to deallocate the string after `ballot_insert` returns.

This ownership transfer also implies that `ballot_insert` never needs to allocate.

Finally, the `get_irv_winner` function also has an ownership situation you may find puzzling. When `get_irv_winner` returns a string, it transfers ownership of the string to the caller and the caller must free the string. Why? To implement the IRV algorithm, `get_irv_winner` must create and destroy a vote count map for each round of counting. In the last round of counting, the winner is the candidate name returned by `vc_max`, which is a string borrowed from the vote count map. Destroying the vote count map before returning is `get_irv_winner`'s responsibility as owner, but once `vc_destroy` is called, the old result of `vc_max` is no longer valid! Thus, once the winner is determined, `get_irv_winner` must make a copy of the winner string to return, and it must make that copy before it deallocates the vote count map.

This means copying the heap-allocated string, not just aliasing it by copying the pointer.

The IRV algorithm [«spec]

The IRV algorithm takes a ballot box as input and returns the name of the winner of the election. The algorithm proceeds in rounds as follows:

- In each round, starting with an empty vote count map, we count every ballot in the ballot box into the vote count map. This means incrementing the count for the leading (first active) candidate on each ballot.
- If one candidate has a majority, meaning more than half the total cast votes in the current round, then that candidate is the winner.
- If no votes were cast then there is no winner, so per the spec the result is `NULL`.
- Otherwise, the candidate with the fewest votes is eliminated from the ballot box, and then we repeat this process for the next round.

Note that the above description of the algorithm does not describe the necessary resource management, so it is up to you to combine the algorithm description in this section with the discussion of ownership in the previous section.

Note also that the algorithm as stated is ambiguous because it doesn't specify how to break ties for the fewest votes. But our particular specification of `vc_min`, which breaks ties for elimination by returning the most recently added candidate, completely determines all decisions, including the elimination step.

Testing [«read_ballot_box() spec]

You will need to test your code thoroughly, both to ensure its correctness and for self evaluation.

File test/test_ballot.c contains one test case for some of the ballot functions, which may help give you an idea how to use the abstraction and test it further.

One important thing to test is the interaction between a ballot and a vote count map as implemented by `count_ballot`. You should complete function `test_ballot_with_vc` to test this scenario: Create a ballot that initially ranks three candidates (henceforth A, B, and C). Starting with a fresh vote count map, count the ballot once and confirm one vote for A and none for the others. Count again and confirm all the votes. Eliminate B, count again, and confirm that A has gone to 3 while the others remain at zero. Then eliminate A, count, and confirm a first vote for C. Eliminate C, so that the ballot has no active candidate, and confirm that counting the ballot again has no effect on the counts.

File test/test_ballot_box.c contains three test cases written using a function `check_election`, which takes the winner and all ballots as arguments, builds the ballot box, runs the IRV algorithm, and confirms the result. You should probably add more such test cases, but note that this is not enough to test your input routines `read_ballot` and `read_ballot_box`. When called from `irv.c`, `read_ballot_box` (and thus `read_ballot` are passed `stdin`, in order to read from the console. But for testing, you may want to read from actual files. Additionally, by reading entire files of input this way, you can test out much larger ballots or ballot boxes in an easier way for the rest of your tests.

Here is a procedure to set up testing of the input routines on files:

1. Create a subdirectory Resources in your project directory at the same level of src/ and test/.

2. In the Resources directory, create files containing the text you want the functions to read where the text looks exactly like input you

If you create additional files in this directory, do not forget to the submit them to Gradescope too (see submission instructions)!

may enter manually if the program read everything from `stdin`. Use a good naming scheme, with either names describing each scenario (*e.g.,* one_ballot_one_vote.in) or systematic names based on the function to be tested and numbering (*e.g.,* ballot_box_6.in).

3. Write a function in each test program (test_ballot.c and test_ballot_box.c that takes a filename as a `const char*`, opens the file using *fopen*(3), reads the file using the `read...` function in the library under test (`read_ballot` for testing the ballot library and `read_ballot_box` for testing the ballot box library), closes the file using *fclose*(3), and then returns the new object that was read.

4. Add tests that use the functions from step 3 to read the files from step 2 and confirm that the results are as you expect. You can also test individual ballot or ballot box functions on this "read-in" ballot or ballot box object and check that the results are what you expect.

If you run your test programs from the same directory that your `Makefile` is in then you'll be able to refer to files from your code using relative paths such as `"Resources/ballot_4.in"` and `"Resources/ballot_box_6.in"`.

Deliverables & evaluation

For part 1, you must:

1. Complete the eight unimplemented ballot functions and `clean_name` in src/ballot.c, as specified above.

2. Add more test cases to test/test_ballot.c in order to test the functions that you wrote..

For part 2, you must:

1. Complete the five unimplemented ballot box functions and `get_irv_winner` in src/ballot_box.c, as specified above.

2. Add more test cases to test/test_ballot_box.c in order to test the functions that you wrote.

You will be required to fill out a separate self-evaluation for each part separately. As usual, the self evaluation will spot-check your test coverage by asking for just a few particular test cases. One of those cases for part 1 is described in the *Hints* section. You can't anticipate what other cases we may ask about, so you should try to cover everything.

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 part 1 of this homework, that will include src/ballot.c and test/test_ballot.c. For part 2, that will include src/ballot_box.c and test/test_ballot_box.c in addition to src/ballot.c. (You must not modify any other files except when mentioned otherwise in the document.)

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:

To submit the necessary files for part 1, you will run something that looks like:

```
% submit211 submit --hw hw3-p1 src/ballot.c test/test_ballot.c
```

To submit the necessary files for part 2, you will run something that looks like:

```
% submit211 submit --hw hw3-p2 src/ballot.c src/ballot_box.c test/test_ballot_box.c
```

Warning: if you made any tests that use files in Resources/, you **MUST** upload them with the submit command as well. Otherwise you won't be able to reference them in self-eval questions and we won't be able to award you points for tests that rely on those resource files.

If you want to include your own src/libvc.c, be sure to include that in the submission files for both parts as well.

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.

As with homework 2, some tests will be hidden to you and will only become visible once the assignment closes. You may submit to each part on Gradescope a maximum of 30 times each. Therefore, to use your submissions carefully, you are encouraged to test locally as thoroughly as possible.