EECS 211 Homework 4

Winter 2019

Due:	February 13, 2019 at 11:59 PM
Partners:	Yes; register on GSC before submission

Purpose

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

Preliminaries

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

```
$ curl $URL211/hw/hw04.tgz | tar zvx
⋮
$ cd hw04
```

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

```
$ make all
⋮
cc -o build/irv build/irv.o build/lib211.o build/...
$
```

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:

While this is the second part of a two-part assignment, you are free to choose the same partner, a new partner, or no partner. If you work with a partner, you are free to use either your HW3 libvc.c, their libvc.c, or a solution libvc.so (as explained below) to test your code.

This homework assignment must be completed on Linux using the T-Lab or Wilkinson Lab machines. Each time you login to work on EECS 211, you need to run the *dev* command (as set up in Lab 1).

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.

Orientation

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

- Functions on ballots are declared in src/ballot.h, and you must implement them in src/ballot.c. Unit tests for these functions should be written in test/test_ballot.c.
- Functions on collections of ballots, including the IRV algorithm, are declared in src/ballot_box.h, and you must implement them in src/ballot_box.c. Unit tests for these functions should be written in text/test_ballot_box.c.
- The `main()` function implementing the *irv* program is already written for you in src/irv.c.

The code in src/ballot.c and src/ballot_box.c depends on *libvc*, but the src/libvc.c included in the starter code is incomplete. You may place your own libvc.c in your src directory, or you may edit your Makefile to use our solution libvc.so instead. To do so, you must change the definitions of two Make variables:

- Change the definition of `LDFLAGS` to `-L$(TOV_PUB)/lib -lvc` from blank.
- Remove the line containing "`build/libvc.o \`" from the definition of `COMMON_OBJ`.

Together, these changes will cause your code to be linked against our shared library, rather than depending on your own src/libvc.c.

The .so extension stands for *shared object*. As the standard format for libraries on Unix systems, it's essentially a collection of functions from one or more .o files that other programs can call.

Make targets

The project provides a Makefile with several targets:

target	description
all	builds everything * †
test	builds and runs all tests †
build/test_ballot	builds the ballot tests
build/test_ballot_box	builds the ballot box tests
build/irv	builds the *irv* program
clean	removes all build products †

* default † phony

Specifications

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

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 **char**s that are not 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 is terminated by EOF.

Like *count* from HW3, 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. Note, however, that if you are using our solution libvc.so, its compiled-in `MAX_CANDIDATES` value will continue to be 16, even if you change the value in your own src/libvc.h.

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.

```
$ build/irv
Abbott
Borden
Campbell
%
Campbell
Abbott
Borden
%
Borden
Campbell
Abbott
%
Abbott
Borden
Campbell
%
Campbell
Abbott
Borden
^D
CAMPBELL
$
```

src/ballot.c

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

tially 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, intended to represent a single voter's ranked-choice ballot.

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

This 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 eor standardizing candidate names.

- `ballot_t ballot_create(void)` allocates a new, 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`.

  Ownership: The caller takes ownership of the result.

  Errors: Exits with error code 2 if memory cannot be allocated.

- `void ballot_destroy(ballot_t ballot)` deallocates all memory associated with `ballot`. `ballot` may be `NULL`, in which case this function does nothing.

  Ownership: Takes ownership of `ballot`.

  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` with `clean_name` and adds it to the end of the ballot.

  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.

  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.

- `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` by incrementing the count of the leading (first active) candidate. 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_t` (meaning `vc_update` returns `NULL`) then it exits with error code 4.

- `ballot_t read_ballot(FILE* inf)` reads a single ballot from input file handle `inf`. It reads one name per line until reaching either EOF or a percent sign on a line by itself, and it standardizes each candidate name using the `clean_name` function (described below) before storing it in the ballot.

  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 if memory cannot be allocated.

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

- `void print_ballot(FILE* outf, ballot_t)` prints a ballot to the given file handle in a human-readable format. This function is implemented for you.

  Ownership: Both arguments are borrowed transiently.

- `void clean_name(char* name)` 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.

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.

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.

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.

src/ballot_box.c

In this file, you will implement a collection of owned ballots as a linked list. 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, 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 implemented for you already), 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: 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. Takes a pointer to the `ballot_box_t`, or in other words, a `struct bb_node**`, and updates it in place.

  This function is already implemented for you.

  Preconditions:

  - `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.

- `void bb_eliminate(ballot_box_t bb, const char* candidate)` eliminates all votes for the given candidate.

  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 result and must release it with `vc_destroy`.

  Errors:

  – Exits with a non-zero error code if `vc_update` cannot allocate memory.
  – Calls `count_ballot`, which exits with error code 4 if it cannot allocate memory.

- `ballot_box_t read_ballot_box(FILE* inf)` reads ballots from the given file handle until there are none left to read.

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

  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)` runs the IRV algorithm and returns the name of the winner as an owned string.

  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.

Hints

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

Name standardization

Function `clean_name` 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_charset` 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.

In ISL+λ terms, `(clean_name s)` is like `(map toupper (filter isalpha s))`, but by modifying `s`. See *isalpha*(3) and *toupper*(3).

To do so, you need to track two positions in the same string, which I 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.

As before, you can use two **`char`**`*`s that both move, or one fixed **`char`**`*` and two `size_t` offsets that move. Note that mixed approaches tend to be confusing, if even they work at all.

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

Testing

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 `assert_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.

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

1. Create a subdirectory Resources in your project directory.
2. In the Resources directory, create files containing the text you want the functions to read. 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 that takes a filename as a `const char*`, opens the file using *fopen*(3), reads the file using the function under test, 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.

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"`.

Ballot representation

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.

Ballot box representation

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 preallocating 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, 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

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 element (`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`.

Ownership strategy

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. The string that `get_irv_winner` returns to its caller comes with ownership and must be freed by the caller. 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.

The IRV algorithm

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, then that candidate is the winner. Otherwise, the candidate with the fewest votes is eliminated, and we proceed to 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.

Deliverables and evaluation

For this homework you must:

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

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

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

As usual, self evaluation will spot-check your test coverage by asking for just a few particular test cases. One of those cases 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
- adherance to the EECS 211 Style Manual.

Submission

Homework submission and grading will use the GSC grading server. You must upload any files that you create or change.

For this homework, that will include src/ballot.c, src/ballot_box.c, test/test_ballot.c, and test/test_ballot_box.c. (You should not need to submit a modified Makefile and you must not modify any of the .h files.

If you work with a partner then you must register your partnership **before submitting** using either the `gsc partner request` and `gsc partner accept` commands or on the GSC website. See the **gsc**(1) manual page for details.

Be careful with partner registration, because once a partner request is accepted, undoing it requires an appeal to the instructor.

Once a partner request is accepted, you and your partner's submissions are joined together. When one partner uploads files or performs self evaluation, the results will be visible to both.