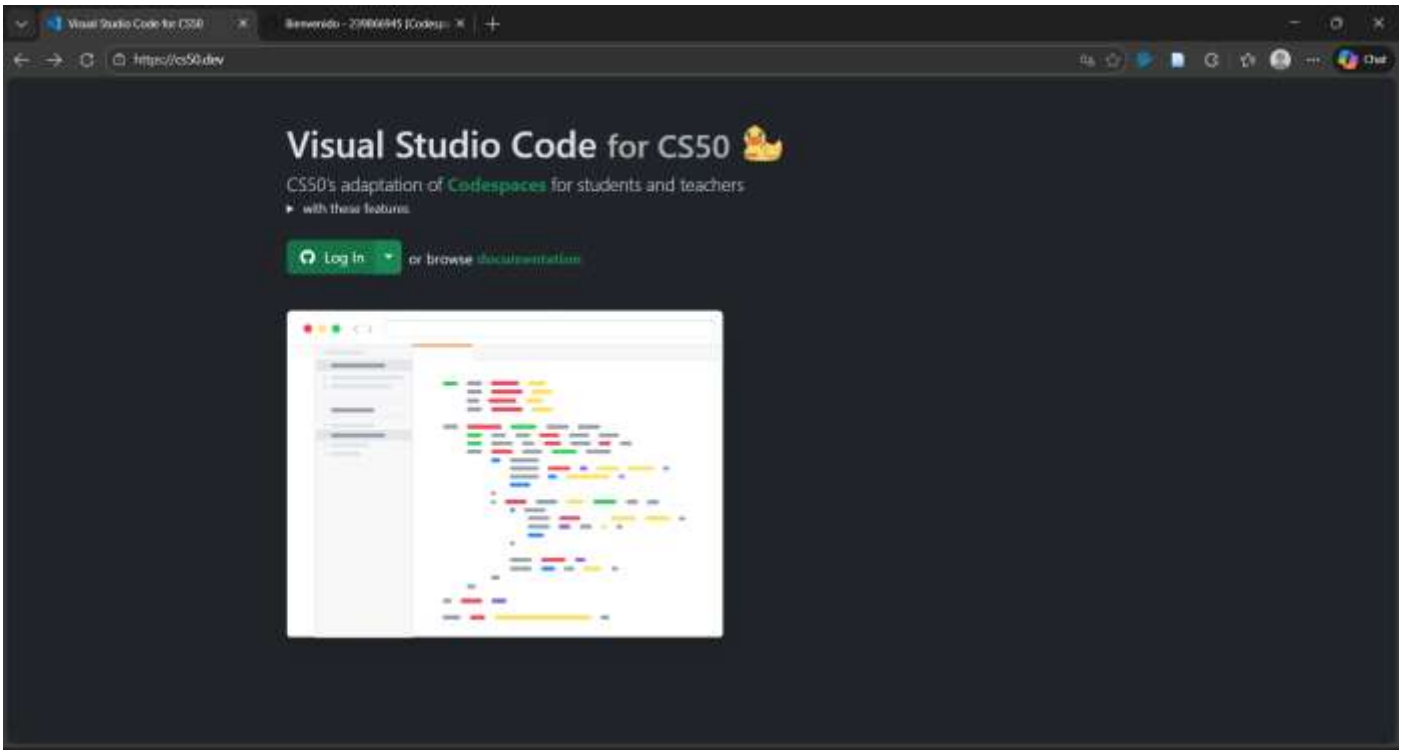
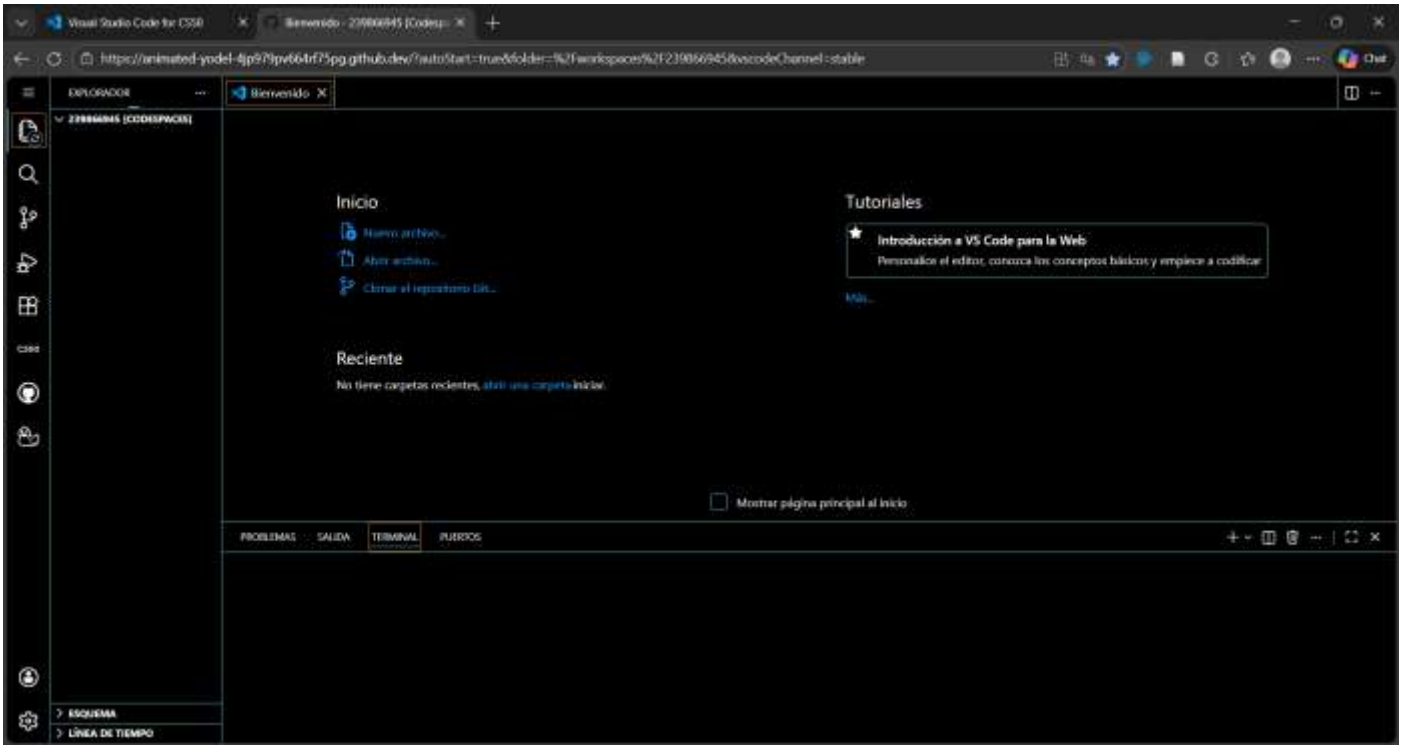
Problem Set 5

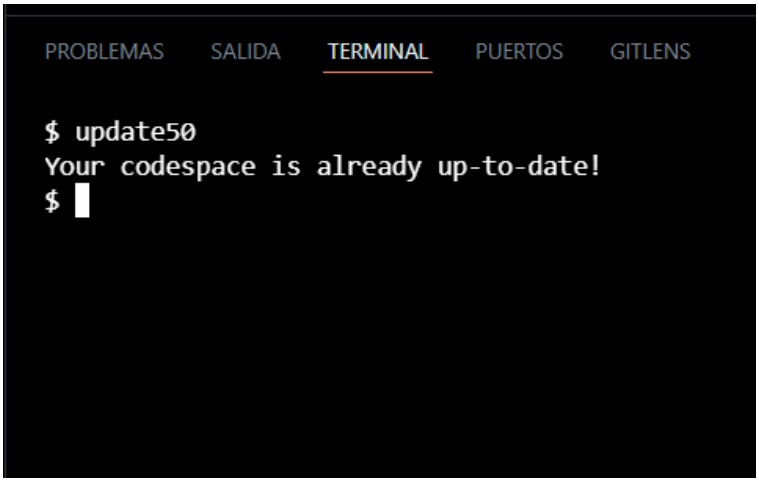
What to Do

1. Log into cs50.dev using your GitHub account





2. Run in your codespace's terminal window to ensure your codespace is up-to-date and, when prompted, click **Rebuild now**update50



3. Submit Inheritance

1. Problem to Solve

A person's blood type is determined by two alleles (i.e., different forms of a gene). The three possible alleles are A, B, and O, of which each person has two (possibly the same, possibly different). Each of a child's parents randomly passes one of their two blood type alleles to their child. The possible blood type combinations, then, are: OO, OA, OB, AO, AA, AB, BO, BA, and BB.

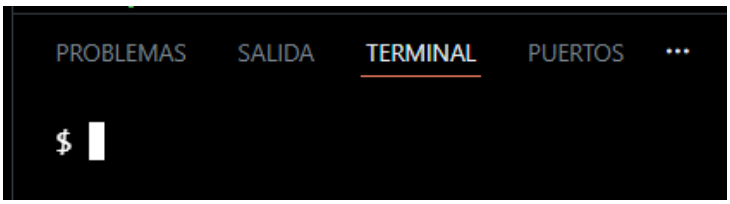
For example, if one parent has blood type AO and the other parent has blood type BB, then the child's possible blood types would be AB and OB, depending on which allele is received from each parent. Similarly, if one parent has blood type AO and the other OB, then the child's possible blood types would be AO, OB, AB, and OO.

In a file called in a folder called , simulate the inheritance of blood types for each member of a family.inheritance.cinheritance

2. Distribution Code

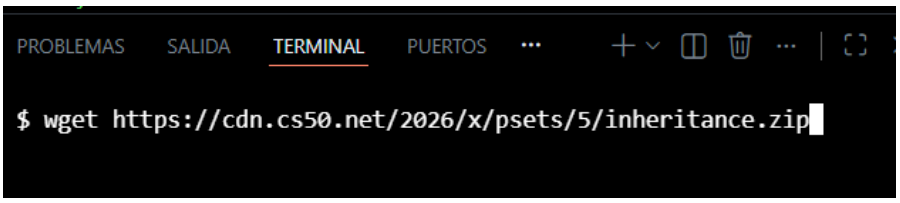
For this problem, you'll extend the functionality of code provided to you by CS50's staff.

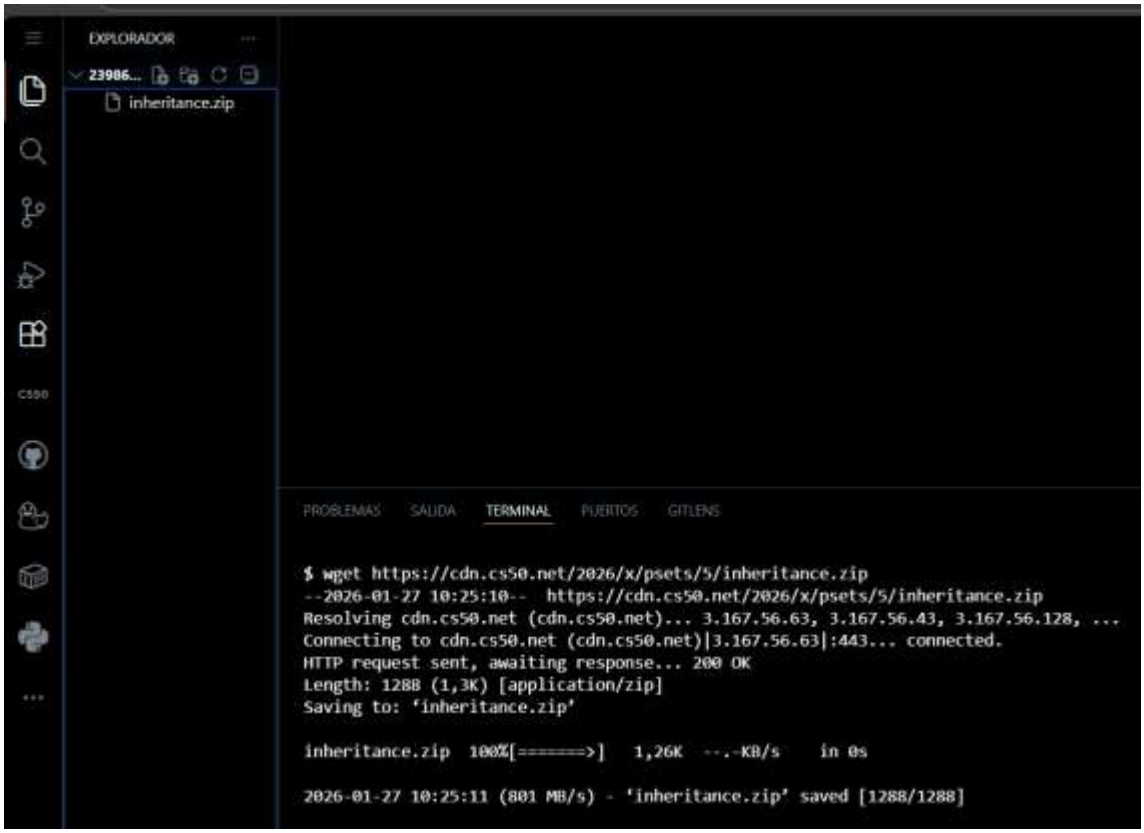
Log into cs50.dev, click on your terminal window, and execute by itself. You should find that your terminal window's prompt resembles the below: cd



Next execute

```
wget https://cdn.cs50.net/2026/x/psets/5/inheritance.zip
```

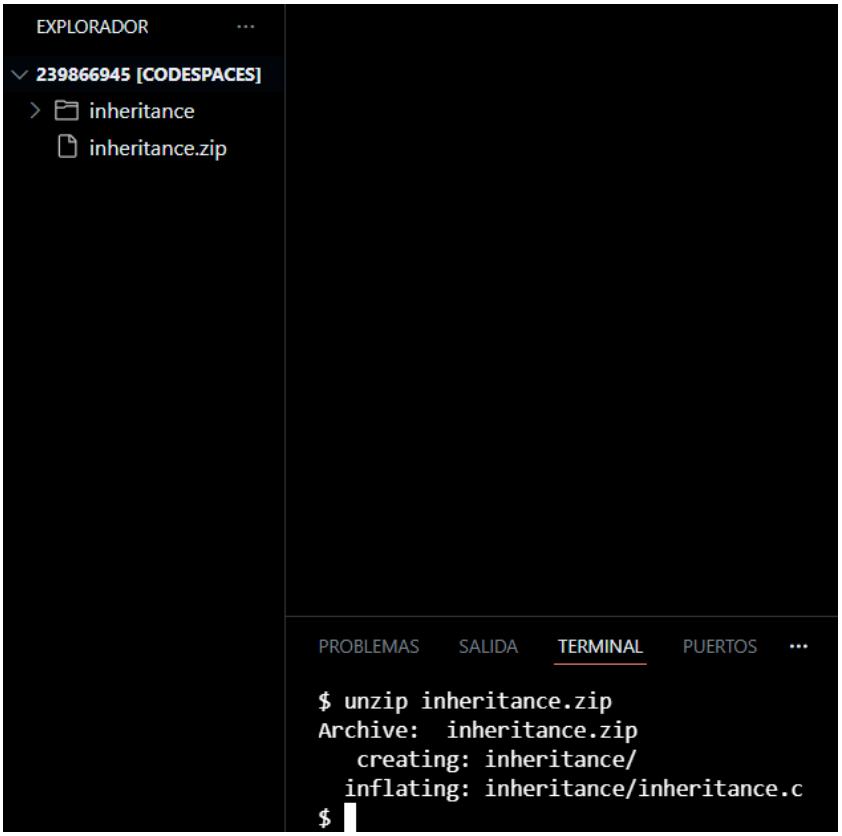




in order to download a ZIP called inheritance.zip into your codespace.

Then execute

```
unzip inheritance.zip
```



to create a folder called inheritance. You no longer need the ZIP file, so you can execute

```
rm inheritance.zip
```

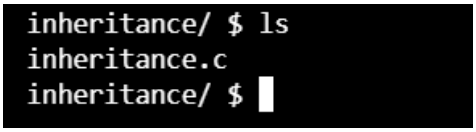
and respond with “y” followed by Enter at the prompt to remove the ZIP file you downloaded.

Now type

```
cd inheritance
```

followed by Enter to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

Execute ls by itself, and you should see and see a file named inheritance.c.



If you run into any trouble, follow these same steps again and see if you can determine where you went wrong!

3. Implementation Details

Complete the implementation of inheritance.c, such that it creates a family of a specified generation size and assigns blood type alleles to each family member. The oldest generation will have alleles assigned randomly to them.

- The create_family function takes an integer (generations) as input and should allocate (as via malloc) one person for each member of the family of that number of generations, returning a pointer to the person in the youngest generation.
    - For example, create_family(3) should return a pointer to a person with two parents, where each parent also has two parents.
    - Each person should have alleles assigned to them. The oldest generation should have alleles randomly chosen (as by calling the random_allele function), and younger generations should inherit one allele (chosen at random) from each parent.
    - Each person should have parents assigned to them. The oldest generation should have both parents set to NULL, and younger generations should have parents be an array of two pointers, each pointing to a different parent.

4. Hints

- Understand the code in inheritace.c

Take a look at the distribution code in .inheritance.c

Notice the definition of a type called . Each person has an array of two , each of which is a pointer to another struct. Each person also has an array of two , each of which is a (either , , or) .person parents person alleles char 'A' 'B' 'O'

```
// Each person has two parents and two alleles
typedef struct person
{
   struct person *parents[2];
   char alleles[2];
} person;
```

Now, take a look at the function. The function begins by “seeding” (i.e., providing some initial input to) a random number generator, which we’ll use later to generate random alleles. main

```
// Seed random number generator
srandom(time(0));
```

The function then calls the function to simulate the creation of structs for a family of 3 generations (i.e. a person, their parents, and their grandparents). main create_family person

```
// Create a new family with three generations
person *p = create_family(GENERATIONS);
```

We then call to print out each of those family members and their blood types. print_family

```
// Print family tree of blood types
print_family(p, 0);
```

Finally, the function calls to any memory that was previously allocated with. free_family free malloc

```
// Free memory
free_family(p);
```

The and functions are left to you to write! create_family free_family

```
// Simulate genetic inheritance of blood type
#define _DEFAULT_SOURCE
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Each person has two parents and two alleles
typedef struct person
{
   struct person *parents[2];
   char alleles[2];
} person;

const int GENERATIONS = 3;
const int INDENT_LENGTH = 4;

person *create_family(int generations);
void print_family(person *p, int generation);
void free_family(person *p);
char random_allele();

int main(void)
{
   // Seed random number generator
   srandom(time(0));

   // Create a new family with three generations
   person *p = create_family(GENERATIONS);

   // Print family tree of blood types
   print_family(p, 0);

   // Free memory
   free_family(p);
}
```

- Complete the create family function

The function should return a pointer to a who has inherited their blood type from the number of given as input.create_familypersongenerations

- Notice first that this problem poses a good opportunity for recursion.
    - To determine the present person's blood type, you need to first determine their parents' blood types.
    - To determine those parents' blood types, you must first determine their parents' blood types. And so on until you reach the last generation you wish to simulate.

To solve this problem, you'll find several TODOs in the distribution code.

First, you should allocate memory for a new person. Recall that you can use to allocate memory, and to get the number of bytes to allocate. malloc sizeof(person)

```
// Allocate memory for new person
person *new_person = malloc(sizeof(person));
```

```
38 // Create a new individual with `generations`
39 person *create_family(int generations)
40 {
41     // TODO: Allocate memory for new person
42     person *new_person = malloc(sizeof(person));
43
```

Next, you should check if there are still generations left to create: that is, whether .generations > 1

If , then there are more generations that still need to be allocated. We've already created two new parents, and , by recursively calling . Your function should then set the parent pointers of the new person you created. Finally, assign both for the new person by randomly choosing one allele from each parent. generations > 1 parent0 parent1 create_family create_family alleles

- Remember, to access a variable via a pointer, you can use arrow notation. For example, if is a pointer to a person, then a pointer to this person's first parent can be accessed by . p p->parents[0]
- You might find the function useful for randomly assigning alleles. This function returns an integer between and , or . In particular, to generate a pseudorandom number that is either or , you can use the expression . random( ) 0 RAND_MAX 32767 0 1 random( ) % 2

```
// Create two new parents for current person by recursively calling create_family
person *parent0 = create_family(generations - 1);
person *parent1 = create_family(generations - 1);

// Set parent pointers for current person
new_person->parents[0] = parent0;
new_person->parents[1] = parent1;

// Randomly assign current person's alleles based on the alleles of their parents
new_person->alleles[0] = parent0->alleles[random() % 2];
new_person->alleles[1] = parent1->alleles[random() % 2];
```

```
44     // If there are still generations left to create
45     if (generations > 1)
46     {
47         // Create two new parents for current person by recursively calling create_family
48         person *parent0 = create_family(generations - 1);
49         person *parent1 = create_family(generations - 1);
50
51         // TODO: Set parent pointers for current person
52         new_person -> parents[0] = parent0;
53         new_person -> parents[1] = parent1;
54         // TODO: Randomly assign current person's alleles based on the alleles of their parents
55         new_person-> alleles[0] = parent0-> alleles[random()%2];
56         new_person-> alleles[1] = parent1-> alleles[random()%2];
57     }
58
59     // If there are no generations left to create
```

Let's say there are no more generations left to simulate. That is, . If so, there will be no parent data for this person. Both parents of your new person should be set to , and each should be generated randomly. generations == 1 NULL allele

```
// Set parent pointers to NULL
new_person->parents[0] = NULL;
new_person->parents[1] = NULL;

// Randomly assign alleles
new_person->alleles[0] = random_allele();
new_person->alleles[1] = random_allele();
```

```
59      // If there are no generations left to create
60      else
61      {
62          // TODO: Set parent pointers to NULL
63          new_person->parents[0] = NULL;
64          new_person->parents[1] = NULL;
65          // TODO: Randomly assign alleles
66          new_person->alleles[0] = random_allele();
67          new_person->alleles[1] = random_allele();
68      }
```

Finally, your function should return a pointer for the that was allocated. Person

```
// Return newly created person
return new_person;
```

```
70      // TODO: Return newly created person
71      return new_person;
72  }
```

❖ Complete the free_family function

The function should accept as input a pointer to a , free memory for that person, and then recursively free memory for all of their ancestors. free_family person

- Since this is a recursive function, you should first handle the base case. If the input to the function is , then there's nothing to free, so your function can return immediately. NULL
- Otherwise, you should recursively both of the person's parents before ing the child. free free

The below is quite the hint, but here's how to do just that!

```
// Free `p` and all ancestors of `p`.
void free_family(person *p)
{
  // Handle base case
  if (p == NULL)
  {
    return;
  }

  // Free parents recursively
  free_family(p->parents[0]);
  free_family(p->parents[1]);

  // Free child
  free(p);
}
```

```c
// Free `p` and all ancestors of `p`.
void free_family(person *p)
{
    // TODO: Handle base case
    if(p==NULL){
        return;
    }
    // TODO: Free parents recursively
    free_family(p->parents[0]);
    free_family(p->parents[1]);
    // TODO: Free child
    free(p);
}
```

5. **How to Test**

Upon running , your program should adhere to the rules described in the background. The child should have two alleles, one from each parent. The parents should each have two alleles, one from each of their parents. ./inheritance

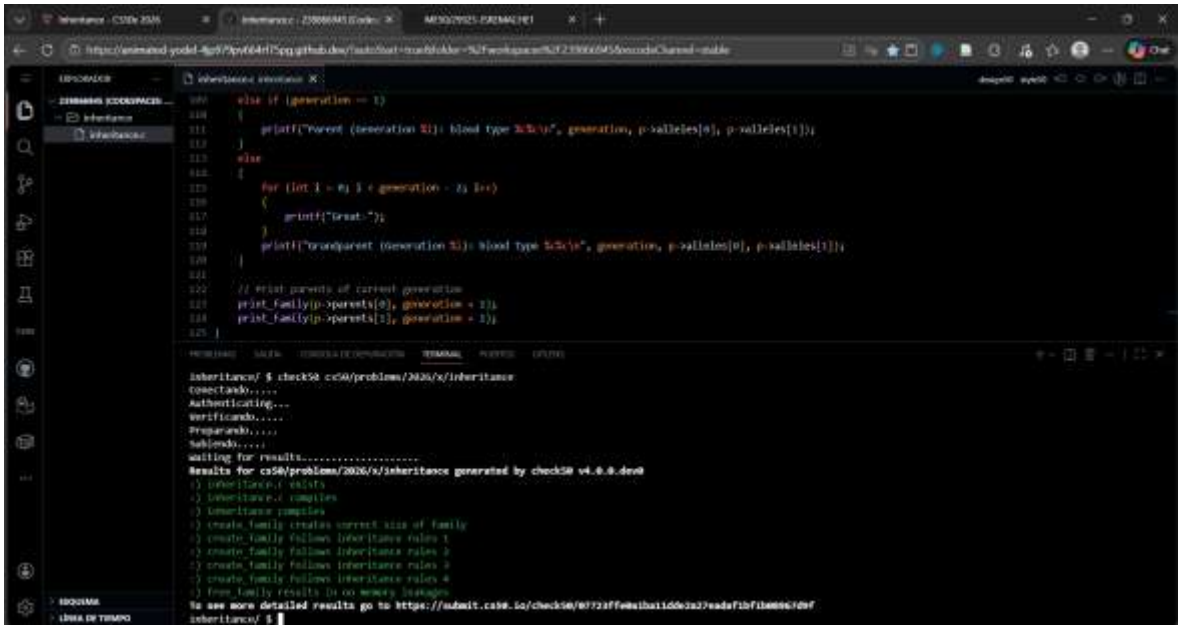
For example, in the example below, the child in Generation 0 received an O allele from both Generation 1 parents. The first parent received an A from the first grandparent and a O from the second grandparent. Similarly, the second parent received an O and a B from their grandparents.

```
$ ./inheritance
Child (Generation 0): blood type OO
  Parent (Generation 1): blood type AO
    Grandparent (Generation 2): blood type OA
    Grandparent (Generation 2): blood type BO
  Parent (Generation 1): blood type OB
    Grandparent (Generation 2): blood type AO
    Grandparent (Generation 2): blood type BO
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  GITLENS
inheritance/ $ ./inheritance
Child (Generation 0): blood type BB
    Parent (Generation 1): blood type BB
        Grandparent (Generation 2): blood type BB
        Grandparent (Generation 2): blood type BO
    Parent (Generation 1): blood type BO
        Grandparent (Generation 2): blood type BB
        Grandparent (Generation 2): blood type OA
```

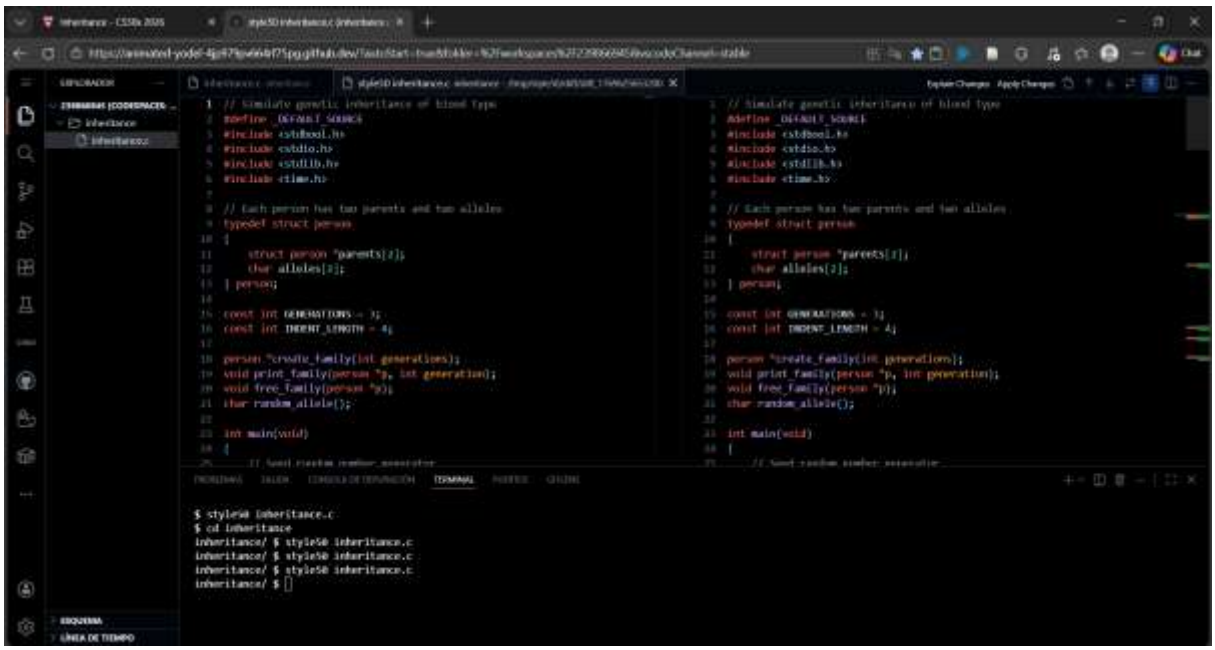
Correctness

```
check50 cs50/problems/2026/x/inheritance
```



Style

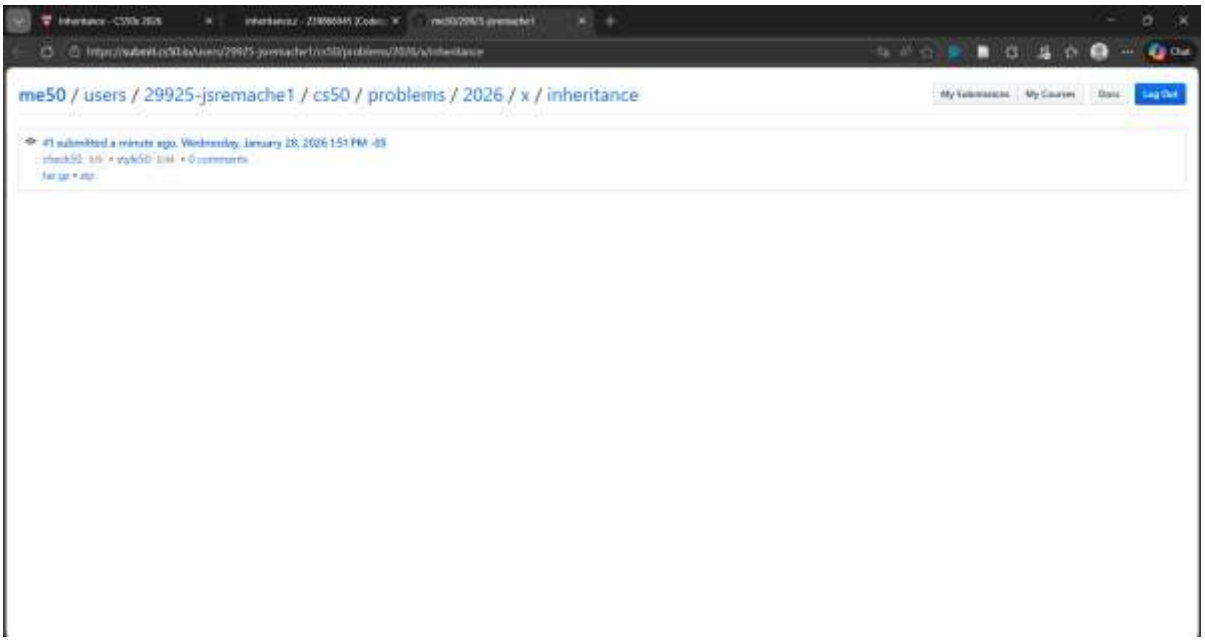
```
style50 inheritance.c
```



6. How to Submit

In your terminal, execute the below to submit your work, answering the prompts that come up as well.

```
submit50 cs50/problems/2026/x/inheritance
```



4. Submit Speller

    1. **Problem to Solve**

        For this problem, you'll implement a program that spell-checks a file, a la the below, using a hash table.

    2. **Distribution Code**

        For this problem, you'll extend the functionality of code provided to you by CS50's staff.

        Log into cs50.dev, click on your terminal window, and execute cd by itself. You should find that your terminal window's prompt resembles the below:

        ```
        $
        ```

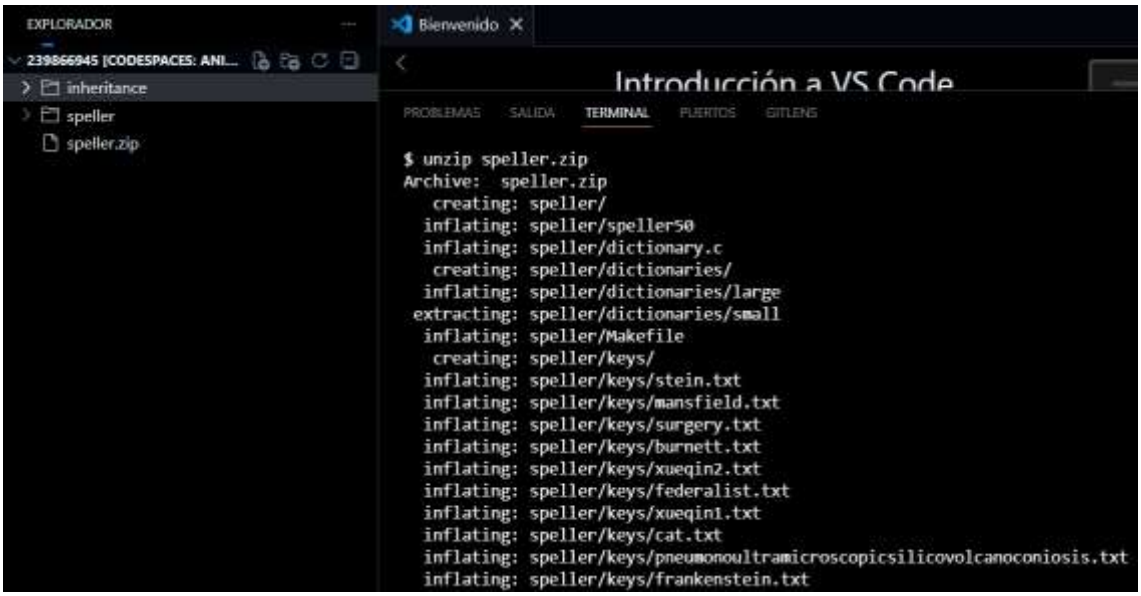
        Next execute

        ```
        wget https://cdn.cs50.net/2026/x/psets/5/speller.zip
        ```

in order to download a ZIP called speller.zip into your codespace.

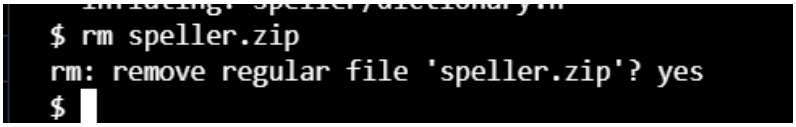
Then execute

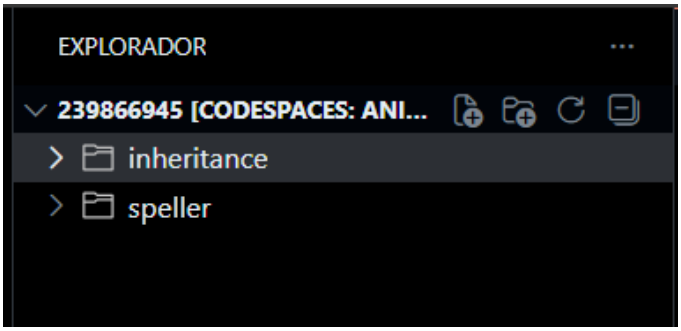
```
unzip speller.zip
```



to create a folder called speller. You no longer need the ZIP file, so you can execute

```
rm speller.zip
```





and respond with “y” followed by Enter at the prompt to remove the ZIP file you downloaded.

Now type

```
cd speller
```

followed by Enter to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
speller/ $
```

Execute ls by itself, and you should see a few files and folders:

```
dictionaries/  dictionary.c  dictionary.h  keys/  Makefile  speller.c  speller50  texts/
```

If you run into any trouble, follow these same steps again and see if you can determine where you went wrong!

3. Background

Theoretically, on input of size n, an algorithm with a running time of n is “asymptotically equivalent,” in terms of O, to an algorithm with a running time of 2n. Indeed, when describing the running time of an algorithm, we typically focus on the dominant (i.e., most impactful) term (i.e., n in this case, since n could be much larger than 2). In the real world, though, the fact of the matter is that 2n feels twice as slow as n.

The challenge ahead of you is to implement the fastest spell checker you can! By “fastest,” though, we’re talking actual “wall-clock,” not asymptotic, time.

In , we’ve put together a program that’s designed to spell-check a file after loading a dictionary of words from disk into memory. That dictionary, meanwhile, is implemented in a file called . (It could just be implemented in , but as programs get more complex, it’s often convenient to break them into multiple files.) The prototypes for the functions therein, meanwhile, are defined not in itself but in instead. That way, both and can the file. Unfortunately, we didn’t quite get around to implementing the loading part. Or the checking part. Both (and a bit more) we leave to you! But first, a tour. speller.c dictionary.c speller.c dictionary.c dictionary.h speller.c dictionary.c #include

Understanding

- **dictionary.h**

  Open up , and you’ll see some new syntax, including a few lines that mention . No need to worry about those, but, if curious, those lines just ensure that, even though and (which you’ll see in a moment) this file, will only compile it once. dictionary.h DICTIONARY_H dictionary.c speller.c #include clang

  Next notice how we a file called . That’s the file in which itself is defined. You’ve not needed it before, since the CS50 Library used to that for you. #include stdbool.h bool #include

  Also notice our use of , a “preprocessor directive” that defines a “constant” called that has a value of . It’s a constant in the sense that you can’t (accidentally) change it in your own code. In fact, will replace any mentions of in your own code with, literally, . In other words, it’s not a variable, just a find-and-replace trick. #define LENGTH 45 clang LENGTH 45

Finally, notice the prototypes for five functions: , , , , and . Notice how three of those take a pointer as an argument, per the : check hash load size unload *

```
bool check(const char *word);
unsigned int hash(const char *word);
bool load(const char *dictionary);
```

```
12 // Prototypes
13 bool check(const char *word);
14 unsigned int hash(const char *word);
15 bool load(const char *dictionary);
```

Recall that is what we used to call . So those three prototypes are essentially just: char * string

```
bool check(const string word);
unsigned int hash(const string word);
bool load(const string dictionary);
```

And , meanwhile, just says that those strings, when passed in as arguments, must remain constant; you won't be able to change them, accidentally or otherwise! Const

- **dictionary.c**

  Now open up . Notice how, atop the file, we've defined a called that represents a node in a hash table. And we've declared a global pointer array, , which will (soon) represent the hash table you will use to keep track of words in the dictionary. The array contains node pointers, and we've set equal to for now, to match with the default function as described below. You will likely want to increase this depending on your own implementation of . dictionary.c struct node table N N 26 hash hash

  Next, notice that we've implemented , , , and , but only barely, just enough for the code to compile. Notice too that we've implemented with a sample algorithm based on the first letter of the word. Your job, ultimately, is to re-implement those functions as cleverly as possible so that this spell checker works as advertised. And fast! load check size unload hash

- **speller.c**

  Okay, next open up and spend some time looking over the code and comments therein. You won't need to change anything in this file, and you don't need to understand its entirety, but do try to get a sense of its functionality nonetheless. Notice how, by way of a function called , we'll be "benchmarking" (i.e., timing the execution of) your implementations of , , , and . Also notice how we go about passing , word by word, the contents of some file to be spell-checked. Ultimately, we report each misspelling in that file along with a bunch of statistics. speller.c getrusage check load size unload check

  Notice, incidentally, that we have defined the usage of to be speller

  ```
  Usage: speller [dictionary] text
  ```

  where is assumed to be a file containing a list of lowercase words, one per line, and is a file to be spell-checked. As the brackets suggest, provision of is optional; if this argument is omitted, will use by default. In other words, running dictionary text dictionary speller dictionaries/large

```
./speller text
```

will be equivalent to running

```
./speller dictionaries/large text
```

where is the file you wish to spell-check. Suffice it to say, the former is easier to type! (Of course, will not be able to load any dictionaries until you implement in ! Until then, you'll see .) text speller load dictionary.c Could not load

Within the default dictionary, mind you, are 143,091 words, all of which must be loaded into memory! In fact, take a peek at that file to get a sense of its structure and size. Notice that every word in that file appears in lowercase (even, for simplicity, proper nouns and acronyms). From top to bottom, the file is sorted lexicographically, with only one word per line (each of which ends with). No word is longer than 45 characters, and no word appears more than once. During development, you may find it helpful to provide with a of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory. In is one such dictionary. To use it, execute\n speller dictionary dictionaries/small

```
./speller dictionaries/small text
```

where is the file you wish to spell-check. Don't move on until you're sure you understand how itself works! text speller

Odds are, you didn't spend enough time looking over . Go back one square and walk yourself through it again! speller.c

- **texts/**

  So that you can test your implementation of , we've also provided you with a whole bunch of texts, among them the script from La La Land, the text of the Affordable Care Act, three million bytes from Tolstoy, some excerpts from The Federalist Papers and Shakespeare, and more. So that you know what to expect, open and skim each of those files, all of which are in a directory called within your directory. speller texts pset5

  Now, as you should know from having read over carefully, the output of , if executed with, say, speller.c speller

  ```
  ./speller texts/lalaland.txt
  ```

  Below's some of the output you'll see. For information's sake, we've excerpted some examples of "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

  ```
  MISSPELLED WORDS

  [...]
  AHHHHHHHHHHHHHHHHHHHHHHHHHHHT
  [...]
  Shangri
  [...]
  fianc
  [...]
  Sebastian's
  ```

```
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

TIME IN load represents the number of seconds that spends executing your implementation of . represents the number of seconds that spends, in total, executing your implementation of . represents the number of seconds that spends executing your implementation of . represents the number of seconds that spends executing your implementation of . is the sum of those four measurements. speller load TIME IN check speller check TIME IN size speller size TIME IN unload speller unload TIME IN TOTAL

Note that these times may vary somewhat across executions of speller, depending on what else your codespace is doing, even if you don't change your code.

Incidentally, to be clear, by "misspelled" we simply mean that some word is not in the provided. Dictionary

- **Makefile**

  And, lastly, recall that automates compilation of your code so that you don't have to execute manually along with a whole bunch of switches. However, as your programs grow in size, won't be able to infer from context anymore how to compile your code; you'll need to start telling how to compile your program, particularly when they involve multiple source (i.e.,) files, as in the case of this problem. And so we'll utilize a , a configuration file that tells exactly what to do. Open up , and you should see four lines: make clang make make .c Makefile make Makefile

  1. The first line tells to execute the subsequent lines whenever you yourself execute (or just). make make speller make
  2. The second line tells how to compile into machine code (i.e.,). make speller.c speller.o
  3. The third line tells how to compile into machine code (i.e.,). make dictionary.c dictionary.o
  4. The fourth line tells to link and in a file called . make speller.o dictionary.o speller

  Be sure to compile speller by executing make speller (or just make). Executing make dictionary won't work!

4. Specification

Alright, the challenge now before you is to implement, in order, , , , , and as efficiently as possible using a hash table in such a way that , , , and are all minimized. To be sure, it's not obvious what it even means to be minimized, inasmuch as these benchmarks will certainly vary as you feed different values for and for . But therein lies the challenge, if not the fun, of this problem. This problem is your chance to design. Although we invite you to minimize space, your ultimate enemy is time. But before you dive in, some specifications from us. load hash size check unload TIME IN load TIME IN check TIME IN size TIME IN unload spelle rdictionary text

- You may not alter or . speller.c Makefile
- You may alter (and, in fact, must in order to complete the implementations of , , , , and), but you may not alter the declarations (i.e., prototypes) of , , , , or . You may, though, add new functions and (local or global) variables to . dictionary.c load hash size check unload load hash size check unload dictionary.c
- You may change the value of in , so that your hash table can have more buckets. N dictionary.c
- You may alter , but you may not alter the declarations of , , , , or . dictionary.h load hash size check unload
- Your implementation of must be case-insensitive. In other words, if is in dictionary, then should return true given any capitalization thereof; none of , , , , , , , , and should be considered misspelled. check foo check foo foO fOo fOO fOO Foo FoO FOo FOO
- Capitalization aside, your implementation of should only return for words actually in . Beware hard-coding common words (e.g.,), lest we pass your implementation a without those same words. Moreover, the only possessives allowed are those actually in . In other words, even if is in , should return given if is not also in . check true dictionary the dictionary dictionary foo dictionary check false foo's foo's dictionary
- You may assume that any passed to your program will be structured exactly like ours, alphabetically sorted from top to bottom with one word per line, each of which ends with . You may also assume that will contain at least one word, that no word will be longer than (a constant defined in) characters, that no word will appear more than once, that each word will contain only lowercase alphabetical characters and possibly apostrophes, and that no word will start with an apostrophe. dictionary \n dictionary LENGTH dictionary.h
- You may assume that will only be passed words that contain (uppercase or lowercase) alphabetical characters and possibly apostrophes. check
- Your spell checker may only take and, optionally, as input. Although you might be inclined (particularly if among those more comfortable) to “pre-process” our default dictionary in order to derive an “ideal hash function” for it, you may not save the output of any such pre-processing to disk in order to load it back into memory on subsequent runs of your spell checker in order to gain an advantage. text dictionary
- Your spell checker must not leak any memory. Be sure to check for leaks with. valgrind
- **The hash function you write should ultimately be your own, not one you search for online.**

Alright, ready to go?

- Implement. load
- Implement. hash
- Implement. size
- Implement. check
- Implement. unload

5. Hints

- **Implement load**

  Complete the function. should load the dictionary into memory (in particular, into a hash table!). should return if successful and otherwise. load load load true false

  Consider that this problem is just composed of smaller problems:

  1. Open the dictionary file
  2. Read each word in the file
     1. Add each word to the hash table
  3. Close the dictionary file

  Write some pseudocode to remind yourself to do just that:

```
bool load(const char *dictionary)
{
  // Open the dictionary file

  // Read each word in the file

    // Add each word to the hash table

  // Close the dictionary file
}
```

```
35 // Loads dictionary into memory, returning true if successful, else false
36 bool load(const char *dictionary)
37 {
38     // TODO
39     return false;
40 }
```

Consider first how to open the dictionary file. fopen is a natural choice. You can use mode r, given that you need only *read* words from the dictionary file (not *write* or *append* them).

```
bool load(const char *dictionary)
{
  // Open the dictionary file
  FILE *source = fopen(dictionary, "r");

  // Read each word in the file

    // Add each word to the hash table

  // Close the dictionary file
}
```

```
35 // Loads dictionary into memory, returning
36 bool load(const char *dictionary)
37 {
38     // TODO
39     FILE *source = fopen(dictionary, "r");
40
41     return false;
42 }
```

Before moving on, you should write code to check whether the file opened correctly. That's up to you! It's also best to ensure you close every file you open, so now's a good time to write the code to close the dictionary file:

```
bool load(const char *dictionary)
{
  // Open the dictionary file
  FILE *source = fopen(dictionary, "r");

  // Read each word in the file

    // Add each word to the hash table
```

```
    // Close the dictionary file
    fclose(source);
}
```

```
35 // Loads dictionary into memory, returning
36 bool load(const char *dictionary)
37 {
38     // TODO
39     FILE *source = fopen(dictionary, "r");
40
41     fclose(source);
42           You, ahora • Uncommitted changes
43     return false;
44 }
```

What remains is to read each word in the file and to add each word to the hash table. Return true when the entire operation is successful and false if it ever fails. Consider following this problem's walkthrough and continue to break sub-problems into even smaller problems. For example, adding each word to the hash table might only be a matter of implementing a few, even smaller, steps:

1. Create space for a new hash table node
2. Copy the word into the new node
3. Hash the word to obtain its hash value
4. Insert the new node into the hash table (using the index specified by its hash value)

Of course, there's more one way to approach this problem, each with their own design trade-offs. For that reason, the rest of the code is up to you!

```
// Loads dictionary into memory, returning true if successful, else false
bool load(const char *dictionary)
{
    // Open the dictionary file
    FILE *source = fopen(dictionary, "r");
    if (source == NULL)
    {
        return false;
    }

    // Initialize hash table to NULL
    for (int i = 0; i < N; i++)
    {
        table[i] = NULL;
    }

    // Buffer for reading words
    char word[LENGTH + 1];

    // Read each word in the file
    while (fscanf(source, "%s", word) != EOF)
    {
        // Create a new node
        node *n = malloc(sizeof(node));
        if (n == NULL)
```

```
    {
        fclose(source);
        return false;
    }

    // Copy word into node
    strcpy(n->word, word);
    n->next = NULL;

    // Hash word to get index
    unsigned int index = hash(word);

    // Insert node into hash table at index
    n->next = table[index];
    table[index] = n;

    // Increment word count
    word_count++;
  }

  // Close dictionary file
  fclose(source);
  return true;
}
```

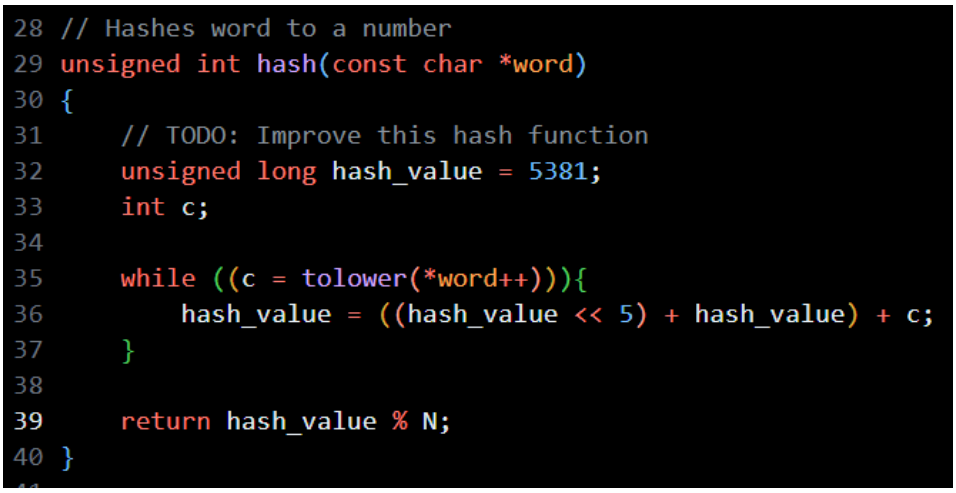
```
35  // Loads dictionary into memory, returning true if successful, else false
36  bool load(const char *dictionary)
37  {
38      // TODO
39      FILE *source = fopen(dictionary, "r");
40
41      if (source == NULL){
42          return false;
43      }
44
45      for (int i = 0; i < N; i++){
46          table[i] = NULL;
47      }
48
49      char word[LENGHT +1];
50
51      while (fscanf(source, "%s", word) != EOF){
52          node *n = malloc(sizeof(node));
53          if(n == NULL){
54              fclose(source);
55              return false;
56          }
57
58          strcpy(n->word, word);
59          n->next = NULL;
60
61          unsigned int index = hash(word);
62
63          n->next = table[index];
64          table[index] = n;
65
66          word_count++;
67      }
68
69      fclose(source);
70
71      return true;
72  }
```

- **Implement hash**

  Complete the hash function. hash should take a string, word, as input and return a positive (“unsigned”) int.

  The hash function given to you returns an int between 0 and 25, inclusive, based on the first character of word. However, there are many ways to implement a hash function beyond using the first character (or characters) of a word. Consider a hash function that uses a sum of ASCII values or the length of a word. A good hash function reduces “collisions” and has a (mostly!) even distribution across hash table “buckets”.

```
// Hashes word to a number
unsigned int hash(const char *word)
{
   unsigned long hash_value = 5381;
   int c;

   while ((c = tolower(*word++)))
   {
      hash_value = ((hash_value << 5) + hash_value) + c; // hash * 33 + c
   }

   return hash_value % N;
}
```



```
// Hashes word to a number
unsigned int hash(const char *word)
{
    // TODO: Improve this hash function
    unsigned long hash_value = 5381;
    int c;

    while ((c = tolower(*word++))){
        hash_value = ((hash_value << 5) + hash_value) + c;
    }

    return hash_value % N;
}
```

- **Implement size**

  Complete the size function. size should return the number of words loaded in the dictionary. Consider two approaches to this problem:

  - Count each word as you load it into the dictionary. Return that count when size is called.
  - Each time size is called, iterate through the words in the hash table to count them up. Return that count.

  Which seems most efficient to you? Whichever you choose, we’ll leave the code up to you.

```
// Returns number of words in dictionary if loaded, else 0 if not yet loaded
unsigned int size(void)
{
   return word_count;
}
```

```
81 // Returns number of words in dictionary if loaded, else 0 if not yet loaded
82 unsigned int size(void)
83 {
84     // TODO
85     return word_count;       You, ahora • Uncommitted changes
86 }
```

- **Implement check**

  Complete the check function. check should return true if a word is located in the dictionary, otherwise false.

  Consider that this problem is also composed of smaller problems. If you've implemented a hash table, finding a word takes only a few steps:

  1. Hash the word to obtain its hash value
  2. Search the hash table at the location specified by the word's hash value
     1. Return true if the word is found
  3. Return false if no word is found

  To compare two strings case-insensitively, you may find strcasecmp (declared in strings.h) useful! You'll likely also want to ensure that your hash function is case-insensitive, such that foo and FOO have the same hash value.

```
// Returns true if word is in dictionary, else false
bool check(const char *word)
{
   // Hash the word to obtain its hash value
   unsigned int index = hash(word);

   // Search the hash table at that index
   node *cursor = table[index];

   while (cursor != NULL)
   {
      // Use strcasecmp for case-insensitive comparison
      if (strcasecmp(cursor->word, word) == 0)
      {
         return true;
      }
      cursor = cursor->next;
   }

   return false;
}
```

```c
// Returns true if word is in dictionary, else false
bool check(const char *word)
{
    // TODO
    unsigned int index = hash(word);

    node *cursor = table[index];

    while (cursor != NULL){
        if (strcasecmp(cursor->word, word) == 0){
            return true;
        }
        cursor = cursor->next;
    }

    return false;
}
```

- **Implement unload**

  Complete the unload function. Be sure to free in unload any memory that you allocated in load!

  Recall that valgrind is your newest best friend. Know that valgrind watches for leaks while your program is actually running, so be sure to provide command-line arguments if you want valgrind to analyze speller while you use a particular dictionary and/or text, as in the below. Best to use a small text, though, else valgrind could take quite a while to run.

  ```
  valgrind ./speller texts/cat.txt
  ```

  If you run valgrind without specifying a text for speller, your implementations of load and unload won't actually get called (and thus analyzed).

  If unsure how to interpret the output of valgrind, do just ask help50 for help:

  ```
  help50 valgrind ./speller texts/cat.txt
  ```

  If unsure how to interpret the output of valgrind, do just ask help50 for help:

  ```
  // Unloads dictionary from memory, returning true if successful, else false
  bool unload(void)
  {
    // Iterate through all buckets
    for (int i = 0; i < N; i++)
    {
      node *cursor = table[i];

      // Free all nodes in this bucket
      while (cursor != NULL)
      {
        node *temp = cursor;
        cursor = cursor->next;
        free(temp);
      }

      table[i] = NULL;
    }
  ```

```
    // Reset word count
    word_count = 0;

    return true;
}
```

```
 99 // Unloads dictionary from memory, returning true if successful, else false
100 bool unload(void)
101 {
102     // TODO
103     for (int i = 0; i < N; i++){
104         node *cursor = table[i];
105
106         while (cursor != NULL){
107             node *temp = cursor;
108             cursor = cursor->next;
109             free(temp);
110         }
111         table[i] = NULL;
112     }
113
114     word_count = 0;
115
116     return true;       You, hace 19 minutos • Uncommitted changes
117 }
```

6. How to Test

How to check whether your program is outting the right misspelled words? Well, you're welcome to consult the "answer keys" that are inside of the keys directory that's inside of your speller directory. For instance, inside of keys/lalaland.txt are all of the words that your program should think are misspelled.

You could therefore run your program on some text in one window, as with the below.

```
./speller texts/lalaland.txt
```

```
speller/ $ ./speller texts/lalaland.txt

MISSPELLED WORDS

Chazelle
L
TECHNO
L
Thelonious
```

```
Mia
Sebastian's
L

WORDS MISSPELLED:     955
WORDS IN DICTIONARY:  143091
WORDS IN TEXT:        17756
TIME IN load:         0.03
TIME IN check:        1.03
TIME IN size:         0.00
TIME IN unload:       0.00
TIME IN TOTAL:        1.06

speller/ $
```

And you could then run the staff's solution on the same text in another window, as with the below.

```
./speller50 texts/lalaland.txt
```

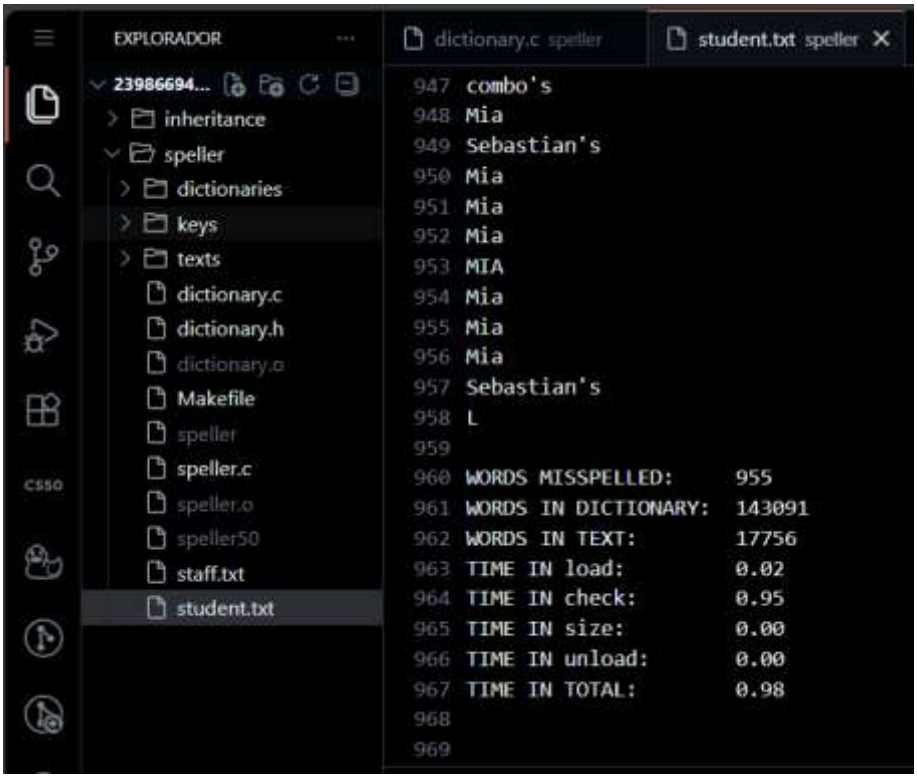
```
speller/ $ ./speller50 texts/lalaland.txt

MISSPELLED WORDS

Chazelle
L
TECHNO
L
Thelonious
```

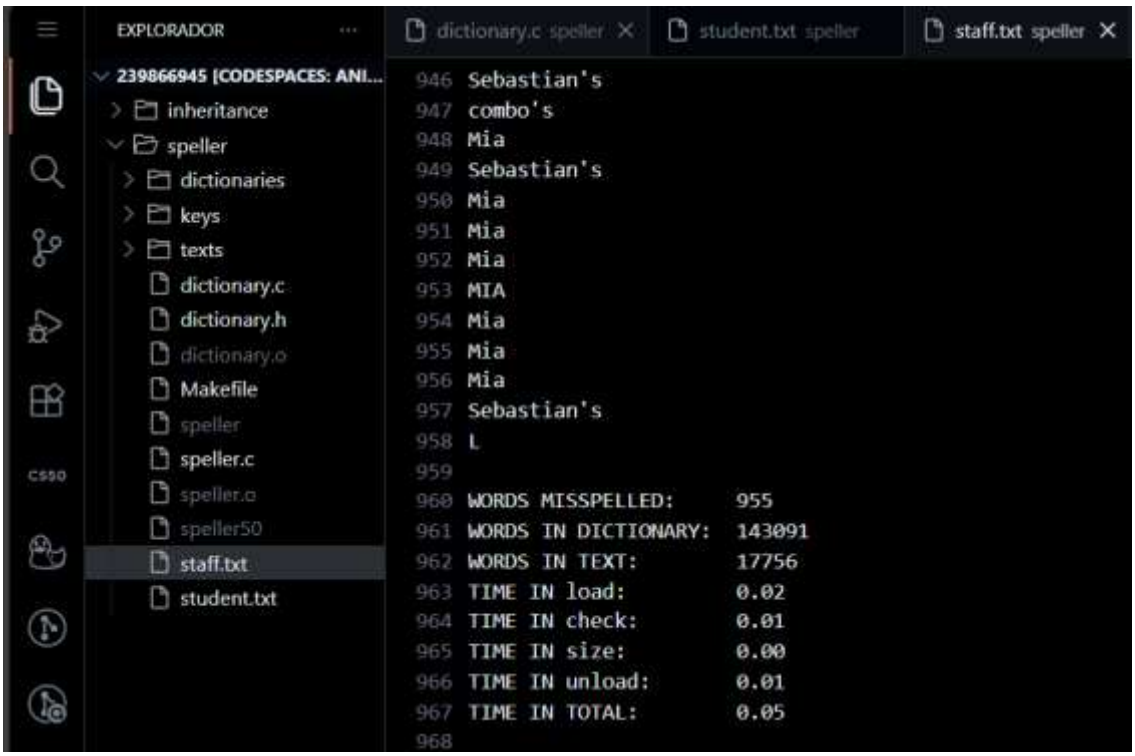
```
Mia
Sebastian's
L

WORDS MISSPELLED:     955
WORDS IN DICTIONARY:  143091
WORDS IN TEXT:        17756
TIME IN load:         0.02
TIME IN check:        0.02
TIME IN size:         0.00
TIME IN unload:       0.01
TIME IN TOTAL:        0.06

speller/ $
```

And you could then compare the windows visually side by side. That could get tedious quickly, though. So you might instead want to "redirect" your program's output to a file, as with the below.

```
./speller texts/lalaland.txt > student.txt
./speller50 texts/lalaland.txt > staff.txt
```

```
speller/ $ ./speller texts/lalaland.txt > student.txt
speller/ $ ./speller50 texts/lalaland.txt > staff.txt
speller/ $
```





You can then compare both files side by side in the same window with a program like diff, as with the below.

```
diff -y student.txt staff.txt
```

```
speller/ $ diff -y student.txt staff.txt

MISSPELLED WORDS                                        MISSPELLED WORDS

Chazelle                                                Chazelle
L                                                       L
TECHNO                                                  TECHNO
L                                                       L
Thelonious                                              Thelonious
Prius                                                   Prius
MIA                                                     MIA
L                                                       L
```

```
Mia                                                     Mia
Sebastian's                                             Sebastian's
L                                                       L

WORDS MISSPELLED:     955                               WORDS MISSPELLED:     955
WORDS IN DICTIONARY:  143091                            WORDS IN DICTIONARY:  143091
WORDS IN TEXT:        17756                             WORDS IN TEXT:        17756
TIME IN load:         0.02                              TIME IN load:         0.02
TIME IN check:        0.95                            | TIME IN check:        0.01
TIME IN size:         0.00                              TIME IN size:         0.00
TIME IN unload:       0.00                            | TIME IN unload:       0.01
TIME IN TOTAL:        0.98                            | TIME IN TOTAL:        0.05

speller/ $
```

Alternatively, to save time, you could just compare your program's output (assuming you redirected it to, e.g.,) against one of the answer keys without running the staff's solution, as with the below. student.txt

```
diff -y student.txt keys/lalaland.txt
```

```
Mia                                                     Mia
Mia                                                     Mia
Sebastian's                                             Sebastian's
L                                                       L

WORDS MISSPELLED:     955                               WORDS MISSPELLED:     955
WORDS IN DICTIONARY:  143091                            WORDS IN DICTIONARY:  143091
WORDS IN TEXT:        17756                             WORDS IN TEXT:        17756
TIME IN load:         0.02                            <
TIME IN check:        0.95                            <
TIME IN size:         0.00                            <
TIME IN unload:       0.00                            <
TIME IN TOTAL:        0.98                            <
                                                      <
speller/ $
```

If your program's output matches the staff's, diff will output two columns that should be identical except for, perhaps, the running times at the bottom. If the columns differ, though, you'll see a > or | where they differ. For instance, if you see

```
MISSPELLED WORDS                                        MISSPELLED WORDS

TECHNO                                                  TECHNO
L                                                       L
                                                      > Thelonious
Prius                                                   Prius
                                                      > MIA
L                                                       L
```

that means your program (whose output is on the left) does not think that Thelonious or MIA is misspelled, even though the staff's output (on the right) does, as is implied by the absence of, say, Thelonious in the lefthand column and the presence of Thelonious in the righthand column.

Finally, be sure to test with both the default large and small dictionaries. Be careful not to assume that if your solution runs successfully with the large dictionary it will also run successfully with the small one. Here's how to try the small dictionary:

```
./speller dictionaries/small texts/cat.txt
```

Correctness

```
check50 cs50/problems/2026/x/speller
```

Style

```
style50 dictionary.c
```

7. Staff's Solution

How to assess just how fast (and correct) your code is? Well, as always, feel free to play with the staff's solution, as with the below, and compare its numbers against yours.

```
./speller50 texts/lalaland.txt
```

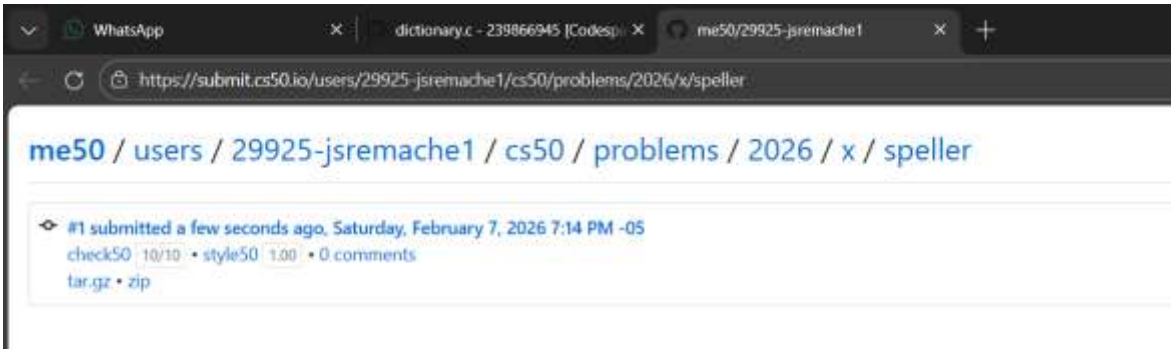
```
speller/ $ ./speller50 texts/lalaland.txt

MISSPELLED WORDS

Chazelle
L
TECHNO
L
Thelonious
Prius
```

```
Mia
Sebastian's
L

WORDS MISSPELLED:     955
WORDS IN DICTIONARY:  143091
WORDS IN TEXT:        17756
TIME IN load:         0.03
TIME IN check:        0.02
TIME IN size:         0.00
TIME IN unload:       0.01
TIME IN TOTAL:        0.06

speller/ $
```

8. **How to Submit**

   In your terminal, execute the below to submit your work, answering the prompts that come up as well.

   ```
   submit50 cs50/problems/2026/x/speller
   ```



When to Do It

By Tuesday, June 30, 2026 at 6:59 PM GMT-5.

Advice

- Try out any of David's programs from Week 5.

- If you see any errors when compiling your code with , focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking for help. For instance, if trying to compile, and make help50 speller
- make speller

is yielding errors, try running

help50 make speller

instead!