This is CS50

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Inheritance

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 `inheritance.c` in a folder called `inheritance`, simulate the inheritance of blood types for each member of a family.

Demo

```
$ make inheritance
```

Recorded with **asciinema**

Distribution Code

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

▼ **Download the distribution code**

Log into cs50.dev (https://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/2023/fall/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.

```
inheritance/ $
```

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!

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.

Hints

Click the below toggles to read some advice!

▼ **Understand the code in `inheritance.c`**

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

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

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

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

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

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

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

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

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

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

▼ Complete the `create_family` function

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

- 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 `malloc` to allocate memory, and `sizeof(person)` to get the number of bytes to allocate.

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

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

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

- Remember, to access a variable via a pointer, you can use arrow notation. For example, if `p` is a pointer to a person, then a pointer to this person's first parent can be accessed by `p->parents[0]`.
- You might find the `rand()` function useful for randomly assigning alleles. This function returns an integer between `0` and `RAND_MAX`, or `32767`. In particular, to generate a pseudorandom number that is either `0` or `1`, you can use the expression `rand() % 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[rand() % 2];
new_person->alleles[1] = parent1->alleles[rand() % 2];
```

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

```
// 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();
```

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

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

▼ **Complete the `free_family` function**

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

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

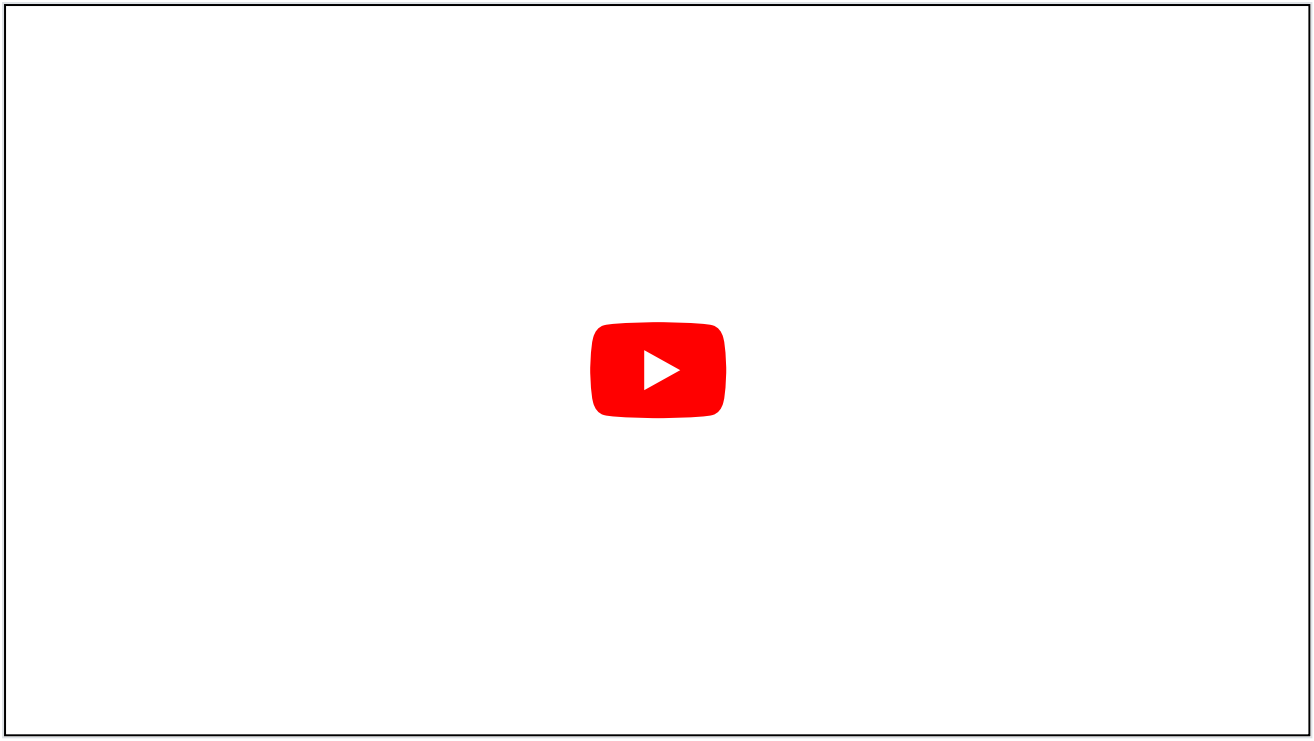
The below is quite the hint, but here's how to do just that!

```
// 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);
}
```

Walkthrough



▼ **Not sure how to solve?**

How to Test

Upon running `./inheritance`, 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.

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
```

Correctness

```
check50 cs50/problems/2024/x/inheritance
```

Style

```
style50 inheritance.c
```

How to Submit

```
submit50 cs50/problems/2024/x/inheritance
```