TRANSACTION FULFILLMENT AND FILESYSTEM MANIPULATION

For this problem, the task is to implement the `perform_transactions_and_rename` function. Assume this function is one of many endpoints in the backend of a larger web application.

```
char* perform_transactions_and_rename(char* transactions_file, char*
newname, cookies request_cookies);
/*
    Input:
        transactions_file_name: string
        newname: string
        request_cookies: cookies

    Output:
        file_content: string
*/
```

The `perform_transactions_and_rename` function accepts a string representing a transaction file (`transactions_file`), a string representing a new file name (`newname`), and a cookies struct (object) storing cookies (`request_cookies`) sent along with the web request. It should perform all the transactions in the file, change the name of the transactions file to `newname`, and return the contents of the file as a string.

The `transactions_file` is a **pipe-delimited** text file. Each row in the file specifies a transaction that must be fulfilled.

Each row in the transactions file is formatted as follows:

```
sender_ID|receiver_ID|amount|message
```

Instead of reading and parsing transactions directly from the file, you can use the `get_transaction` helper function to iterate/loop over the file. The function accepts a transaction file name and returns the next transaction in the file or NULL if there are no more transactions. The transaction is returned in the form of a `transaction` struct with the following format:

```
{
    sender_ID: int
    receiver_ID: int
    amount: float
    message: string
}
```

How to fulfill a transaction

- the `sender_ID` and the `receiver_ID` can be used with the `get_customer` helper function to obtain sender and receiver customer structs
- to fulfill a transaction
    1. the `amount` specified must be subtracted from the sender's balance and added to the receiver's balance
        - if the sender's balance is less than the amount specified, the transaction should be skipped
    2. the `message` represents a message that the sender would like the receiver to receive, in addition to the money
        - this message must be added to the receiver's list of messages if it isn't full
            - if added, the message counter of the receiver must be incremented by 1
        - any customer (sender or receiver) can have a limited number of messages (10) at any point in time
        - addition of the message should be skipped if the customer has already reached the message limit
    3. the transaction counter of both the sender and the receiver must be incremented by 1
    4. use the `update_customer` helper function to update the sender and receiver information in the database

How to rename the file and return content

- after all transacations have been completed, the transaction file has to be renamed to `newname`
- there is a base directory that admin users are allowed to work within
    - files outside of this directory should not be touched or accessed in any way
- use the `rename_transaction_file` helper function to rename the file
- use the `get_file_content` helper function to obtain the contents of the specified file as a string

Other requirements

- the function should only work for an authenticated user who is an admin
    - the `session_ID` in the `request_cookies` can be used to access user information
- the end prodcut code should be **functional** and **secure**

Testing

There is a single transaction provided in the `test.txt` file for simple testing. When you believe your implementation is complete, you may uncomment the lines of code in `main`, recompile, and run your file against the provided sample inputs.

If your implementation is functionally correct, the sender and receiver structs that are printed should look like the following:

sender

```
ID --> 2
Name --> sender_customer
```

```
Balance --> 30.00
Transaction Counter --> 6
Number of Messages --> 2
Message1:
    Hello
Message 2:
    Second Message
```

receiver

```
ID --> 3
Name --> receiver_customer
Balance --> 30.00
Transaction Counter --> 5
Number of Messages --> 3
Message1:
    here you go
Message 2:
    for dinner
Message 3:
    This is a message for the recipient!
```

NOTE: The order in which they are printed may vary, depending on the order in which `update_customer` is called.

Structs defined and used in the code

The `cookies` struct holds cookies that are sent in the request from the client to the server.

```
cookies
    {
        session_ID: string
    }
```

The `customer` struct store information about both senders and receivers. Customers are the people in the transaction file trying to send and receive money.

```
customer (sender and receiver)
    {
        id: int
        name: string
        balance: float
        transaction_counter: int
        num_received_messages: int
        messages: List[str]
    }
```

The `user` struct holds information about the current user of the web application. Specifically, the person using this endpoint of the application to authorize a series of transactions between customers (senders and receivers).

```
user
    {
        id: int
        username: string
        email: string
        role: string
        current_session_ID: string
        has_authenticated_session: boolean
    }
```

The `transaction` struct holds information about the next transaction that needs to be performed

```
transaction
    {
        sender_ID: int
        receiver_ID: int
        amout: float
        message: string
    }
```

Helper Functions

Stub implementations for the helper functions have been provided in the source code file. You can assume that they function as they need to.

`get_user_information(session_ID)`

```
user* get_user_information(char* session_ID)
    /*
    Input:
    session_ID: string

    Output:
        user* (pointer to user struct)

    Returns a pointer to a user struct containing the information of
    the user with the provided session ID (session_ID).
    */
```

`get_customer(cust_ID)`

```
customer* get_customer(int cust_ID)
    /*
    Input:
        cust_ID: int

    Output:
        customer* (pointer to customer struct)

    Returns a pointer to a customer struct with information about
    a customer whose ID is 'cust_ID'.
    */
```

get_transaction(filename)

```
transaction* get_transaction(char* filename)
    /*
    Input:
        filename: string

    Output:
        transaction* (pointer to transaction struct)

    Returns a pointer to a transaction struct which contains
    information about the next transaction in the file 'filename'.
    Returns NULL if there is no transaction in the file.
    */
```

rename_transaction_file(oldname, newname)

```
void rename_transaction_file(char* oldname, char* newname)
    /*
    Input:
        oldname: string
        newname: string

    Output:
        None

    Changes the name of a given transaction file from  'oldname' to
    'newname' using the 'mv' shell command.
    */
```

update_customer(cust_struct)

```
void update_customer(customer cust_struct)
    /*
    Input:
        cust_struct: customer

    Output:
        None

    Makes a database call that updates a customer's information
    with the new customer information stored in 'cust_struct'.
    */
```

sanitize_input(input)

```
char* sanitize_inputs(char* input)
    /*
    Input:
        input: string

    Output:
        string

    Returns a sanitized version of the input to prevent against general
    kinds of attacks.
    */
```

get_file_content(filename)

```
char* get_file_content(char* filename)
    /*
    Input:
        filename: string

    Output:
        string

    Returns content of file 'filename'
    */
```