CS 2110 - Fall 2025
Project 5: Dynamic Memory

Andy Garcha, Aidan Criscione, Yash, Choudhury, Tanay Kapoor, Bianca Jayaraman, Rohit Rao, Michael Yi, Kepler Boyce, Daniel Huang, Johnny Chen, Suraj Samudrala

Due: 12/1/2025 @ 11:59 PM
Version: 1.1

Contents

1 General

1.1 Introduction

Welcome to project 5 movie enthusiast! You've recently decided that cinema is your passion, and have decided to use your knowledge of dynamic memory allocation to create a platform for critics and philosophical thinkers like yourself to share their hottest takes on movies. In this project, you'll be mimicking the behavior of Letterboxd, an app for people to share their reviews on movies, that allows you to keep track of user's watched movies, reviews, and even pull the highest rated movies each week.

We have provided templates for functions in `letterboxd.h` that you will be implementing in `letterboxd.c`. This document will contain more detailed instructions as well as some additional information that will be useful as you work on the project.

Please read through the entire document before starting. The *Data Structures* section will explain how to implement the data structures used in this project as well as several of the operations that you will need to implement for those data structures. The *Common Mistakes* section will cover several common mistakes that are easy to make when working with dynamic memory allocation in C. Be sure to read both of those sections thoroughly, as they may be very useful when you are implementing your functions.

Start early and if you get stuck, there's always Piazza and office hours.

1.2 What functions can I use?

1. You **are** allowed to make your own helper functions in `letterboxd.c` if you so desire.
2. You can use the following library functions in your project: `printf`, `strlen`, `strcpy/strncpy`, `strcmp/strncmp`, `malloc`, `calloc`, `realloc`.
3. Do **NOT** modify or add your own include statements. Including a new header file is **NOT** allowed and the autograder will get mad at you. This includes **ANY** standard library headers not already provided.

2 Data Structures

For this project, you will need to know how doubly linked lists work, as well as a very basic understanding of heaps, as parts of the project utilize these two data structures. This section will give an overview of how both work as well as how to implement some of their operations that you will need to use in this project.

2.1 Doubly Linked Lists

A doubly linked list is a type of dynamically sized list. In a doubly linked list, we use pointers to chain together nodes that contain our data. Each node contains the data for that element as well as a pointer to the next node and the previous node in the linked list—each node is dynamically allocated individually, so our elements are not necessarily contiguous in memory. The first node of the linked list is the "head," and we will keep a pointer to this node so that we can traverse through the chain from the beginning to reach any node in the list. The final node in the chain, called the "tail," will point to `NULL` to mark this as the end of the list.

We need to define a struct for our linked list nodes—we will call it `ListNode`. Each node will hold some data as well as a pointer to the next node and the previous node, so we will have three fields: `data`, `next`, and `prev`. For demonstration here, we will use an `Element` struct as the data we are storing, so `data` will have type `Element`. The `next` field is a pointer to the next node, so it will have type `ListNode *`. The same goes for `prev`.

```
typedef struct ListNode {
    Element data;
    ListNode *next;
    ListNode *prev;
} ListNode;
```

2.1.1 Initialization

To create a new linked list, we have to initialize our `head` pointer. We don't have any elements in our linked list yet, so the head pointer will be `NULL`. Since we mark the end of our linked list by having the final node point to `NULL` in our implementation, we don't need a variable to keep track of the size. So, the initialization will simply look like this:

```
ListNode *head = NULL;
```

2.1.2 Add to Back

To add an element to the back of our linked list, we have to dynamically allocate a new `ListNode` for that element, follow the chain of pointers to reach the final node in the list, and update the `next` field of that final node to point to our new node. We also have to ensure that the `prev` pointer of our new node points to the old tail of the list. This looks something like this:

```
ListNode *new_node_ptr = (ListNode *) malloc(sizeof(ListNode));
if (new_node_ptr == NULL) {
    return FAILURE;
}
new_node_ptr->data = {
    // Assign struct fields
};
new_node_ptr->next = NULL;

ListNode *curr = head;
while (curr->next != NULL) {
    curr = curr->next;
}
curr->next = new_node_ptr;
new_node_ptr->prev = curr;
```

As always, we check for `malloc` failure when we allocate our new `ListNode` struct instance. Because the final node in the chain points to `NULL`, we can traverse to the final node using a while loop that will break once we reach a node whose `next` field is `NULL`. However, there is one scenario that this code fails to consider: If our list is currently empty, then `curr` will be initialized to `NULL` (since `head` is `NULL`), and trying to access `curr->next` in our while loop will cause a null pointer dereference—if the list is empty, then there is no final node to traverse to. So, we have to check for this case up front:

```
// malloc new_node_ptr and set its fields as we did before
if (head == NULL) {
    head = new_node_ptr;
} else {
    // iterate to back and add
}
```

2.1.3 Remove from Back

To remove a node from the back of our linked list, we have to traverse through every node of the list until we reach the tail (remember how we checked if a node was the tail in Add to Back). Once we get there, we will free that node and ensure that the node directly before it points to null as it's next. This would look something like this:

```
ListNode *curr = head;

while (curr->next != NULL) {
    curr = curr->next
}

if (curr->prev == NULL) { // Handles case where we only have one element in list
    free(head);
    head = NULL;
} else {
    curr->prev->next = NULL;
    free(curr);
}
```

Additionally, note that if our `ListNode` struct has any dynamically allocated fields, we must free those before we free the pointer to the node instance itself.

2.1.4 Contains

Say we have some `target` value and we would like to check if our linked list contains an element whose `value` field matches `target`. To do this, we need to iterate through every node in the list, checking whether the `value` field of each element equals `target`. We will return `SUCCESS` if we find such an element, otherwise we will return `FAILURE`. For example:

```
ListNode *curr = head;
while (curr != NULL) {
    if (curr->data.value == target) {
        return SUCCESS;
    }
    curr = curr->next;
}
return FAILURE;
```

2.1.5 Destroying

To destroy our linked list, we have to iterate through the chain of list nodes, freeing each one. Because we can't access the `next` field after freeing `node`, we can use a temporary copy of the `next` pointer. We can do this:

```
ListNode *curr = head;

while (curr != NULL) {
    ListNode *next = curr->next;
    free(curr);
    curr = next;
}
```

2.2 Max Heap

A max heap is a binary tree structure that allows quick access to the maximum element by ensuring that the max element is always the root. It does this by ensuring that a parent node is always greater than both of it's children. In implementation, heaps are backed by dynamically sized arrays. The array lists the nodes of the tree in level order (basically top to bottom and left to right). Our heap will also contain information related to it's size and length. In a basic sense, a heap of Elements will look like:

```
struct MaxHeap {
    Element **array; //array is a pointer to a list of pointers to Elements
    int length; // length represents the current MAX number of elements
    int size; // size represents the current number of elements
}
```

We use type `Element **` for our array because it is dynamically sized, so a statically sized array wouldn't work. On top of that, we want it to be an array of pointers so that we can dynamically allocate each individual `Element`, and we don't lose them from the stack when a function returns.

2.2.1 Initialization

To initialize our Heap, we'll have to dynamically allocate a chunk of memory for our backing array. Say that we have some `INITIAL_CAPACITY` to use as the size for our initial backing array. Creating a new heap would look something like this:

```
Element **array = (Element **) malloc(INITIAL_CAPACITY * sizeof(Element *));
if (array == NULL) {
    return FAILURE;
}
int length = INITIAL_CAPACITY;
int size = 0;
```

Notice that when we call `malloc`, the argument we provide is `INITIAL_CAPACITY` multiplied by the size of a pointer to `Element`. This is because the argument to `malloc` is the number of bytes to allocate, so the total number of bytes we need in our backing array will be the number of entries in the array (`INITIAL_CAPACITY`) multiplied by the size of each entry (`sizeof(Element *)`). It is important to think about the amount of space you need in bytes like this when calling `malloc`. The size of some primitives like short, int, and long depends on the system, so you should always use `sizeof` even for primitives.

2.2.2 Add to Heap

To add something to a heap, we must first add to the backing array. Insert the new element at the first empty spot in the backing array, located at index `size`. Then, you have to call `heapify()` which will be covered in another section and **mostly** given to you.

```
Element *newElement = (Element *) malloc(sizeof(Element));
if (newElement == NULL) {
    return FAILURE;
}
// Assign struct fields
array[size] = newElement;
size++;
heapify();
```

Another thing you must consider is if the backing array needs to be resized. Think about when this would be the case.

2.2.3 Resizing

When resizing we multiply the capacity by some constant factor. In our case, we will multiply capacity by 2. Since we are increasing the size of our dynamically allocated array, we need to **reallocate** it to a larger chunk of memory. For this, we use `realloc` in order to easily increase the size without losing the data in our old memory block.

```
capacity *= 2;
Element **newBackingArray = realloc(array, capacity * sizeof(Element *));
if (newBackingArray == NULL) {
    return FAILURE;
}
array = newBackingArray
```

Note how, with `realloc`, we still have to check if the returned pointer is `NULL`, but in this case we use an intermediate pointer, before directly overwriting our array. This is due to the behavior of `realloc`, which on failure will leave the current block and return `NULL`. Because of this, we don't want to directly overwrite our old array because we could lose access to the pointer and it would remain allocated. On top of that, notice that we don't have to free the old pointer. This is because when `realloc` succeeds, it will free the old pointer for us, after copying over the data.

2.2.4 Contains

If we want to know if a particular piece of data exists in a heap, it is as simple as iterating through the backing array to see if the data exists there. Say we have some `target` which we are looking to match with an elements `value` field.

```
for (int i = 0; i < size; i++) {
    if (array[i]->value == target) {
        return SUCCESS;
    }
}
return FAILURE;
```

2.2.5 Destroying

To destroy the heap, we have to free all the elements in the backing array and then free the backing array itself. This could look something like:

```
for (int i = 0; i < size; i++) {
    free(array[i]);
}
free(array);
```

Note that if `Element` has any dynamically allocated fields, we have to free those, then the pointer to the struct itself.

2.2.6 Heapify

The idea behind heapify is to take a node in a heap and position it correctly as to maintain the heap's primary assurance, that all parent nodes will be greater than their children. It does this by comparing the new node (in the case of insertion) to it's parent, swapping them if needed, and then repeating on the new parent (which was the child) if the swap happened.

3 Structs and Global Variables

For this project, we have defined various structs and global variables in `letterboxd.h` for you to use. It will be useful to understand all of these before you start writing your code.

3.1 Enums and Structs

3.1.1 Movie

```
typedef struct Movie {
    char *name;
    char *director;
    int releaseYear;
} Movie;
```

The `Movie` struct is used for representing information about a single movie. Users will keep a watchlist of `Movie` structs as a Doubly Linked List.

3.1.2 Review

```
typedef struct Review {
    Movie *reviewedMovie;
    int rating;
} Review;
```

The `review` struct represents one review. The `Movie` field represents what movie the review is for, and the `rating` field represents the rating from 1 to 10 stars.

3.1.3 Watchlist Node

```
typedef struct WatchlistNode {
    struct WatchlistNode *next;
    struct WatchlistNode *prev;
    Movie *movie;
    int dayAdded;
} WatchlistNode;
```

The `WatchlistNode` struct represents one movie in a users watchlist. The `next` and `prev` fields point to the next and previous movie in the linked list, respectively. The `movie` field points to the movie in the users watchlist, and the `dayAdded` field tells us the day a movie was added.

3.1.4 Watchlist

```
typedef struct Watchlist {
    WatchlistNode *head;
    char *owner;
} Watchlist;
```

The `Watchlist` struct represents one users watchlist. The `head` field points to the start of a watchlist. The `owner` field gives the name of the owner of the watchlist.

3.1.5 Heap

```
typedef struct Heap {
    Review **backingArray;
    int backingArrayLength;
    int numElements;
} Heap;
```

The `Heap` struct contains the information that we need to define a heap of reviews. The `backingArray` field is a pointer to a list of pointers to reviews. The `backingArrayLength` and `numElements` fields provide information about the backing array, it's current max length and the number of elements currently in it, respectively.

3.2 Global Variables

3.2.1 Review List

```
Heap *reviewList;
```

This will act as the global review list, which contains all posted reviews from all users. Because global variables are zero-allocated, `reviewList` will be instantiated as `NULL`.

4 Instructions

4.1 General instructions

You have been given 2 C files: `letterboxd.h` and `letterboxd.c`. All of the functions you need to implement are in `letterboxd.c` and will be explained briefly below. However, you should look at the `letterboxd.h` file first—it provides definitions for the structs that you will use throughout the project as well as declarations of the functions that you are required to implement.

You can also find a list of a few common mistakes with dynamic memory allocation in C in the *Common Mistakes* section. Be sure to read this section before starting on your function implementations—it may save you from some difficult debugging.

You can use any function included in `stdlib.h`, `string.h`, and `stdio.h` as they are included in `letterboxd.h`. **Including any other library is forbidden.**

You should not be leaking memory in any of the functions. For any functions using malloc, if malloc returns null, return `FAILURE` and ensure that no memory is leaked. The autograder catches memory leaks with Valgrind. See the *Valgrind* section for details.

4.2 Functions to Implement

For each of these functions, handle `NULL` or invalid arguments by returning `FAILURE` or `NULL`, as per the instructions.

4.2.1 Initializers

- `Movie *create_movie(char *name, char *director, int releaseYear);`
  Creates a new `Movie` with name `name`, director `director`, and release year `releaseYear`. Returns `NULL` if `name` is empty, `director` is empty, `releaseYear` is negative, or `malloc` fails. Otherwise, returns a pointer to the new movie.
- `Review *create_review(Movie *movie, int rating);`
  Creates a new `Review` about the movie referenced by `movie`, with rating `rating`. Returns `NULL` if `movie` is `NULL`, `rating` is zero, negative, or greater than 10, or `malloc` fails. Otherwise, returns a pointer to the new review.
- `Watchlist *create_watchlist(char *owner);`
  Creates a new instance of the `Watchlist` struct as a linked list with a `NULL` head and owner `owner`. If `owner` is empty or `NULL`, return `NULL`. Returns `NULL` upon `malloc` failure and a pointer to the new doubly linked list otherwise.
- `int initialize_heap(int initialCapacity);`
  Initializes the `reviewList` global variable as a `Heap` with initial capacity `initialCapacity`. Returns `FAILURE` upon `malloc` failure or invalid inputs, and `SUCCESS` otherwise.

4.2.2 Watchlist Operations

- `int add_movie(Watchlist *watchlist, Movie *movie, int dayWatched);`
  Adds a new movie pointed to by `movie` to the end of the linked list of movies in `watchlist`. If a movie with the same name, director, **and** release year already exists in `watchlist`, do not add it again and return `FAILURE`. Returns `FAILURE` upon `malloc` failure or invalid inputs and `SUCCESS` otherwise.
- `int remove_movie(Watchlist *watchlist, Movie *movie);`
  If a movie `movie` exactly matches an existing movie (all fields) in `watchlist`, remove that movie from `watchlist`. Return `SUCCESS` if a movie is successfully removed, and `FAILURE` otherwise.

- `int watchlist_contains_movie(Watchlist *watchlist, Movie *movie);`
  If `watchlist` contains a movie `movie`, return `SUCCESS`, otherwise returns `FAILURE` if `movie` is not found or the inputs are invalid.

4.2.3 Heap Operations

- `int post_review(Review *review);`
  Adds `review` to the `reviewList` heap. Because global variables are zero-allocated, `reviewList` will be instantiated as `NULL`, so check for that case. Also, be sure to check for size constraints and resize if needed. Returns `FAILURE` upon `malloc` failure or invalid inputs and `SUCCESS` otherwise. Be sure to call `heapify` after you add to the backing array to maintain heap property.

- `int resize_review_list(void);`
  Resizes the review list to be 2 times it's initial size. Make sure you don't lose the reviews already in the review list, or try to resize a `NULL` review list! Returns `FAILURE` upon failure and `SUCCESS` otherwise.

- `int review_posted(Review *review);`
  Checks if a review that matches in fields to `review` exists in the `reviewList`. Make sure to check that the movie matches in fields as well. Returns `FAILURE` if `review` is not found or the input is invalid, and `SUCCESS` otherwise.

- `int heapify(int index);`
  Most of this is implemented for you, **but you must update it**. Heapify takes the index where the new element is, and moves it to the correct place in the array to maintain the heap property. It does this by comparing it against its parent and swapping them if needed. The only part we ask you to implement is the `swap(int i, int j)` subroutine, which is described in the following section. Heapify will return `FAILURE` when it fails (which should never happen in the expected cases), and `SUCCESS` otherwise.

- `int swap(int i, int j);`
  Swaps elements in `reviewList` at index i and index j. Return `FAILURE` in the case of any invalid inputs, and `SUCCESS` otherwise.

- `Movie *get_highest_rated_movie(void);`
  Returns the highest rated movie among posted ratings. Returns `NULL` if anything is invalid.

- `Movie *get_best_from_watchlist(Watchlist *watchlist);`
  Returns the highest rated movie among movies on watchlist `watchlist`. Returns `NULL` if anything is invalid.

4.2.4 Destroying

- `int destroy_watchlist(Watchlist *watchlist);`
  Destroys the watchlist pointed to by `watchlist`, making sure to free only what can safely be freed, without leaking any memory. Return `FAILURE` if `watchlist` is invalid, and `SUCCESS` otherwise.

- `int destroy_reviewlist(void);`
  Destroys the `reviewList` itself. Note that reviews can exist without the review list, so you should not free all of the reviews. Returns `FAILURE` if `reviewList` is uninitialized, and `SUCCESS` otherwise.

4.3 Common Mistakes

There are several traps that are easy to fall into while using dynamic memory allocation in C. Make sure to keep these in mind while you implement the functions in this project.

4.3.1 Forgetting to check for malloc failures

If `malloc` fails to find a block of our desired size, it will return `NULL`. This means that every time we use `malloc`, we must check that the pointer it returns is not `NULL`. If we don't check the return value of `malloc` and continue as normal, then we will encounter a null pointer dereference later when we try to access the block that we just tried to allocate. In this project, we handle `malloc` failures by returning `FAILURE`, which is a macro that has been defined for you. For example:

```
MyStruct *my_ptr = malloc(size);
if (my_ptr == NULL) {
    return FAILURE;
}
```

4.3.2 Incorrectly handling realloc failures

If `realloc` fails to find a new block of our desired size, it will return `NULL` and leave the original block allocated in its current location. For example, consider this code that attempts to reallocate `my_ptr` to a new block:

```
my_ptr = realloc(my_ptr, new_size);
if (my_ptr == NULL) {
    return FAILURE;
}
```

It looks like this code is checking for `realloc` failure, but it is actually doing so incorrectly. With this code, we will lose our original `my_ptr` pointer in the event of `realloc` failure, as `my_ptr` will get reassigned to `NULL`. This will create a memory leak, as our original block will remain allocated when `realloc` fails, but we no longer have any way to access that block. To avoid losing our original pointer and properly check for `realloc` failure, we have to use a temporary pointer to hold the return value of `realloc`:

```
MyStruct *temp_ptr = realloc(my_ptr, new_size);
if (temp_ptr == NULL) {
    return FAILURE;
}
my_ptr = temp_ptr;
```

4.3.3 Incorrect size when dynamically allocating strings

Remember that every string must end with a null terminator (`'\0'` character) to indicate that this is the end of the string. The `strlen` function from `string.h` returns the length of the string ignoring the null terminator, so you have to remember to add 1 for the total size when allocating space for a string. For example, if we want to allocate a block for a string `word`, we would do the following:

```
char *word_ptr = malloc(strlen(word) + 1); // CORRECT
```

rather than this:

```
char *word_ptr = malloc(strlen(word)); // INCORRECT
```

4.3.4 Incorrectly handling malloc failures in chains of mallocs

If we have a chain of mallocs (for example, when allocating a struct instance and its fields), we need to remember to free everything that we allocated earlier in the chain in reverse order if we get a `malloc` failure. For example, say we have the following struct:

```
typedef struct Person {
    char *first_name;
    char *last_name;
} Person;
```

If we want to dynamically allocate a new instance of `Person` given strings `my_first_name` and `my_last_name` for the two fields, we have to do the following:

```
Person *person = malloc(sizeof(Person));
if (person == NULL) {
    return FAILURE;
}
person->first_name = malloc(strlen(my_first_name) + 1);
if (person->first_name == NULL) {
    free(person);             // IMPORTANT PART
    return FAILURE;
}
person->last_name = malloc(strlen(my_last_name) + 1);
if (person->last_name == NULL) {
    free(person->first_name); // IMPORTANT PART
    free(person);             // IMPORTANT PART
    return FAILURE;
}
```

If we do not free everything that we allocated earlier in the chain upon `malloc` failure, we will create a memory leak. Furthermore, if we free things in the wrong order, we will create a use-after-free error—in this example, this is why we must free `person->first_name` before `person` within the last if statement. An easy way to avoid this is by simply freeing everything in the opposite order as how they were allocated.

4.3.5 Forgetting to free dynamically allocated fields when freeing struct instances

If we have a dynamically allocated struct instance that we would like to free, we have to remember to free any dynamically allocated fields on that struct instance before freeing the struct instance itself. For example, using the same `Person` struct from above, we would free a `Person` struct instance called `person` by doing the following:

```
free(person->first_name);
free(person->last_name);
free(person);
```

If we only did `free(person)`, we would cause a memory leak, as the `first_name` and `last_name` fields would never get freed.

4.3.6 Incorrectly freeing linked list nodes

When freeing all nodes of a linked list, we might naively try to iterate through the list and free each node like this:

```
// BAD CODE
ListNode *curr = head;
while (curr != NULL) {
    free(curr);
    curr = curr->next;
}
```

However, this will result in undefined behavior—after we free `curr`, we no longer have access to the fields of that struct instance, so trying to access `curr->next` on the following line is a use-after-free error. To fix this, we have to save a pointer to the next node before freeing the current node, which we can do like this:

```
// FIXED CODE
ListNode *curr = head;
ListNode *next;

while (curr != NULL) {
    next = curr->next;
    free(curr);
    curr = next;
}
```

5 Building & Testing

5.1 Helpful Info

5.1.1 Man Pages

The `man` command in Linux provides "an interface to the on-line reference manuals." This is a great utility for any C and Linux developer for finding out more information about the available functions and libraries. In order to use this, you just need to pass in the function name to this command within a Linux (in our case Docker) terminal.

For instance, entering the following command will print the corresponding man page for the strlen function:

```
    $ man strlen
```

Additionally, the man pages are accessible online at: http://man.he.net

NOTE: You can ignore the subsections after the "RETURN VALUE" (such as ATTRIBUTES, etc) for this homework, however, pay close attention to function descriptions.

5.1.2 Debugging with GDB and printf

We highly recommend getting used to "`printf` debugging" in C. By sprinkling some `printf` statements throughout your code, you can track variable values, program flow, and state changes at various points, making it easier to identify and fix bugs.

Moreover, If you run into a problem when working on your homework, you can use the debugging tool, GDB, to debug your code! Former TA Adam Suskin made a series of tutorial videos which you can find here. GDB is an essential tool for any C program, so we definitely recommend learning it now.

When running GDB, if you get to a point where user input is needed, you can supply it just like you normally would. When an error happens, you can get a Java-esque stacktrace using the `backtrace` (`bt`) command which allows you to pinpoint where the error is coming from. For more info on basic GDB commands, search up "GDB Cheat Sheet".

5.2 Checking Your Solution

5.2.1 Makefiles

Make is a common build tool for abstracting the complexity of working with compilers directly. Makefiles let you define a set of desired targets (files you want to compile), their prerequisites (files which are needed to compile the target), and sets of directives (commands such as gcc, gdb, etc.) to build those targets. In all of our C assignments (and also in production level C projects), a Makefile is used to compile C programs with a long list of compiler flags that control things like how much to optimize the code, whether to create debugging information for gdb, and what errors we want to show. We have already provided you a Makefile for this homework, but we highly recommend that you take a look at this file and understand the `gcc` commands and flags used to understand how to compile C programs. If you're interested, you can also find more information regarding Makefiles here.

Since your program is connected to an autograder with multiple files that need to be compiled using particular settings, it's a little difficult to compile it by hand. The Makefile allows us to simply type the command `make` followed by a target such as `tests`, `run-tests`, or `run-gdb` to compile and run your code.

Please **DO NOT** modify your makefile as it may break your local autograder.

5.2.2 Valgrind

Valgrind is an open-source memory debugging tool. You can use it to check your code for various memory management errors, such as memory leaks, use-after-free errors, and so on. To test for memory management errors in your code, you can use `make run-valgrind` from within the Docker container as shown in the next section.

5.2.3 Autograder

The autograder must be run with the provided Docker container. To enter the Docker container, `cd` into the project directory and enter:

```
# macOS or Linux
./cs2110docker-c.sh

# Windows
.\cs2110docker-c.bat
```

If you are on macOS or Linux, you may get an error such as

```
zsh: permission denied: ./cs2110docker-c.sh
```

If so, make sure to enable execute permissions on the script: `chmod +x ./cs2110docker-c.sh`. You should only need to do this once.

Once you are in the Docker container, you can run the autograder:

```
# Clean your directory (remove any made files)
$ make clean

# Build and run all tests
$ make

# Build and run one test
```

```
$ make TEST=test_add_movie::null_watchlist

# Build and run all tests in a single file
$ make TEST=test_add_movie::*

# Build and run all tests in GDB
# run-gdb takes the same parameters above
$ make run-gdb

# Build and run with valgrind to check for memory leaks
$ make run-valgrind
```

5.3 Important Notes

1. Only the `letterboxd.c` file will be graded on Gradescope.
2. All non-compiling homework will receive a zero (with all the flags specified in the Makefile/Syllabus).
3. **NOTE: DO NOT MODIFY THE HEADER FILES.**

   Since you are not turning them in, any changes to the `.h` files will not be reflected when running the Gradescope autograder.

We reserve the right to update the autograder and the test case weights on Gradescope or the local checker as we see fit when grading your solution.

6 Deliverables

Turn in `letterboxd.c` to Gradescope by the due date.

Please do not wait until the last minute to run/test your project, history has proved that last minute turn-ins will result in long queue times for grading on Gradescope. You have been warned!!!

7 Rules and Regulations

7.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit.
2. Please read the assignment in its entirety before asking questions.
3. Please start assignments early, and ask for help early. Do not email us a few hours before the assignment is due with questions.
4. If you find any problems with the assignment, it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

7.2 Submission Conventions

1. In order to submit your assignment, submit the files individually to the Gradescope assignment.
2. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

7.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Gradescope.
3. Projects turned in late receive partial credit within the first 48 hours, as defined in the syllabus. Between 0 and 24 hours late, you can receive a maximum score of 70%. Between 24 and 48 hours late, you can receive a maximum score of 50%. We will not accept projects turned in over 48 hours late.
4. You alone are responsible for submitting your project before the assignment is due; neither Canvas/Gradescope, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until the deadline.

7.4 Coding Guidelines C/C++ code

1. You must turn in ALL files specified in the "Deliverables" section of the assignment instructions. We reserve the right to impose a penalty on submissions that do not follow the given submission directions.
2. You must provide a Makefile that compiles and links your code by default. If you are given a Makefile with the project, we expect your code to compile with `make`.
3. Your code must compile with gcc on Ubuntu 22.04 LTS (aka compile using the CS 2110 Docker container). If your code does not compile, you will receive a 0 for the assignment.
4. You will be penalized if your code produces warnings when compiled with the given Makefile, or the following flags if no Makefile is provided: gcc -Wall -Wextra -Wstrict-prototypes -pedantic -O2
5. Code should be well commented and use a clean, consistent (readable) style (i.e., proper indenting, etc.). We reserve the right to impose style requirements, and deduct for non-conforming solutions. This is not the obfuscated C code competition (`http://ioccc.org`).

7.5 Syllabus Excerpt on Academic Misconduct

The goal of all assignments in CS 2110 is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. **As a Georgia Tech student, you have read and agreed to the Georgia Tech Academic Honor Code.** The Academic Honor Code defines Academic Misconduct as "***any act that does or could improperly distort Student grades or other Student academic records.***"
2. You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct**.
3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person or persons*").

4. Publishing your assignments on public repositories, github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.
5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.
6. Students suspected of Academic Misconduct are informed **at the end of the semester**. Suspects receive an *Incomplete* final grade until the issue is resolved.
7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.
8. Submitting an assignment with code or text from an AI assistant (e.g., ChatGPT) is academic misconduct.
9. **If you are not sure about any aspect of this policy, please ask your lecturer**.

Using AI Assistants

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: Never hit "Copy" within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.

Heuristic 2: Do not have your assignment and the AI agent open at the same time. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.