UNIX and C Programming [COMP1000]

Assignment Report

STUDENT NUMBER: **18340037**

NAME: **CAMERON PETKOV**

LECTURER: **MARK UPSTON**

DATE OF SUBMISSION: **MONDAY, 28 May 2018**

DATE DUE: **08:00, MONDAY, 28 May 2018**

I hereby declare that the calculation, results, discussion and conclusions submitted in this report is entirely my own work and have not copied from any other student or past student.

Student Signature: _____

Contents

1.0 Introduction 1
2.0 Purpose of Functions 2
2.1 tvguide.c 2
2.1.1 main(): 2
2.1.2 processArgs(): 2
2.1.3 copyToArray(): 2
2.1.4 filter(): 3
2.2 interface.c 3
2.2.1 inputStrings(): 3
2.2.2 outputArray(): 3
2.2.3 isValidDay(): 3
2.2.4 isValidSort(): 3
2.2.5 decideFlush(): 4
2.2.6 flushInput(): 4
2.2.7 makeLower(): 4
2.3 fileIO.c 4
2.3.1 readFile(): 4
2.3.2 writeFile(): 4
2.3.3 processFile(): 5
2.4 linkedList.c 5
2.4.1 createLinkedList(): 5
2.4.2 isEmpty(): 5
2.4.3 insertFirst(): 5
2.4.4 insertLast(): 5
2.4.5 insertBefore(): 5
2.4.6 insertAfter(): 6
2.4.7 removeFirst(): 6
2.4.8 removeLast(): 6
2.4.9 removeValue(): 6
2.4.10 peekFirst(): 6
2.4.11 peekLast(): 7
2.4.12 find(): 7
2.4.13 getLength(): 7
2.4.14 outputList(): 7
2.4.15 clear(): 7

2.5 comparison.c 7
2.5.1 compareByTime(): 7
2.5.2 getDay(): 8
2.5.3 compareByName(): 8
3.0 Filtering 9
3.1 Implementation 9
3.2 Alternative Approach 10
4.0 Output Testing 11

1.0 Introduction

This report documents the code produced for the assignment. This documentation is intended to accompany the source code and the commenting inside that. The purpose of each function is discussed in detail in section two. Section three then goes into detail about the filtering of the tv entries, and any alternatives that would achieve a similar output. Finally, section four gives a step by step breakdown of testing; from unzipping, to compiling, to testing and the expected output.

2.0 Purpose of Functions

Note that this section of the document is designed to be read in conjunction with the function comments in the corresponding .c files. Where significant choices, structures, or concepts are present, I may choose to also discuss the purpose of those choices, structures, and/or concepts. Header files will, in general, not be discussed for the most part, as commenting within those files should be sufficient to explain themselves. Note that I may refer to TRUE or FALSE when explaining functions and may refer to them as Booleans, however these are just integer values defined in the header file "boolean.h". The definition of FALSE is 0, and TRUE is !0, and this definition may be used when referring to return values of functions.

2.1 tvguide.c

2.1.1 main():

main() is the starting point of the program. It takes in the user's command line arguments, either printing to stderr if the number of arguments is incorrect (not 3) and setting the return to 1, or calling a helper function "processArgs()" which calls other functions in this program to display and store a sorted, filtered TV guide output. At the end of main, the variable status is returned to the shell indicating success or failure. As is common convention, if the program was successful it returns 0, and for any other error, the program returns 1.

2.1.2 processArgs():

processArgs() is the helper function to main() calling other functions to complete the objective of reading in a file and outputting a sorted, filtered array to file and screen. processArgs() takes in a char* [] (array of strings), which provides the file to read from and the file to write to. The purpose of this method is in calling functions, passing the relevant passed/created variables (day[], sort[], *list, etc) by reference, then checking the output of those functions for any errors (any returns of not 0 or NULL), and dealing with them as appropriate. If there are no errors, the program proceeds and ends up outputting the array to file and screen, returning a success value of TRUE. If there are errors, the function manages this by bypassing code that cannot handle those errors and deallocating any malloced memory, before returning to main with a success value of FALSE. The function is responsible for making sure any memory leaks are dealt with, regardless of success status, before ending and returning to main.

2.1.3 copyToArray():

copyToArray() takes a pointer to a linked list and an array of pointers to TVEntries (a structure that can be found in "data.h"). This function calls a linked list function to remove the first entry, removeFirst(), assigns it to an array index, and then frees the node. The purpose of this method is to make sure the list's values are put into the array, before freeing the lists' nodes, and the list structure itself at the end. By the end of this method, there should be no TVEntry* mallocs or list malloc that has not been freed by this method. The list should be empty, and the array should contain all the list's values before removing the entries. The success return of TRUE should accompany this function, but upon error, this function will return success as being FALSE.

2.1.4 filter():

filter() takes an array to filter, the size of the array, and a pointer to char (string) used as the filtering criteria. The function goes through every array index, checking to see if the field in the structure is equal to the filtering criteria mentioned previously. If it is equal, that index's value will be assigned to the start of the existing array. This increments a separate index variable that keeps track of the matching filtered entries, which is then returned to the caller function and should be assigned as the new array size. The function's responsibility is to collate filtered array entries to the starting end of the array, and keep a count of how many of these filtered entries exist. With filtered entries being located at the front of the array and knowing how many exist, the array is effectively filtered.

2.2 interface.c

2.2.1 inputStrings():

inputStrings() takes in two empty char[] (strings) to input user's input. The function manages outputting messages to the user, and taking the user's inputs from the terminal. It validates this user input to ensure it is valid, before proceeding and returning the two input strings to the calling function. The purpose of the function is to prompt the user for input, validate that input, and re-prompt for input if it was incorrect or proceed if it was correct. It is responsible for validating that a user has entered a day of the week and a valid sort type, irrespective of input case. It returns a success code that will always be TRUE in the current implementation, but future refactoring could introduce fail conditions to change this behaviour.

2.2.2 outputArray():

outputArray() takes an array of pointers to TVEntry (TVEntry array) and the array size. If the array is empty, an error will be output indicating that there are no TV shows scheduled on the selected day. Otherwise the function will output the entries line by line, in the format specified in the assignment sheet. The function's purpose is to output the array (or warning the lack of) in the correct format. It will return a success code that will always be TRUE in the current implementation, but future refactoring could introduce fail conditions to change this behaviour.

2.2.3 isValidDay():

isValidDay() takes a char[] (string) containing the user's input for the day of the week they want to filter by. This function will compare the user's input (from fgets() in inputStrings()) to the days of the week. By the end of this function, if the string given was a day of the week the function returns TRUE, otherwise it returns false by default. The function's purpose is to validate the user's day input.

2.2.4 isValidSort():

isValidSort() takes a char[] (string) contain the user's input for the sort type they want the entries to be sorted by. The function will compare the user's input (from fgets() in inputStrings()) to the only valid sort types "time" and "name". By the end of this function, if the string given was a correct sort type the function returns TRUE, otherwise it returns false by default. The function's purpose is to validate the user's sort type input.

2.2.5 decideFlush():

decideFlush() takes a char[] (string) containing the user's input. If the user's input string ends with a \n, it is the end of the string and can be designated to have '\0' instead of the newline (which will skip a line if kept). If the string does not end with a newline character, all characters in stdin will be flushed by another function. The function's purpose is to decide if the stdin buffer needs to be flushed, or to simply remove the newline character from the given string. It will return a success code that will always be TRUE in the current implementation, but future refactoring could introduce fail conditions to change this behaviour.

2.2.6 flushInput():

flushInput() takes no input, simply going through the stdin buffer and removing characters from the buffer until it is empty. The purpose of the function is to completely flush the stdin buffer, leaving no characters in it. It will return a success code that will always be TRUE in the current implementation, but future refactoring could introduce fail conditions to change this behaviour.

2.2.7 makeLower():

makeLower() takes a char[] (string) which could be any string. It will then go through each character in the string and call tolower() on it, making the character lowercase if it can. The purpose of the function is to make a string all lowercase, converting any uppercase characters to lowercase equivalents. It will return a success code that will always be TRUE in the current implementation, but future refactoring could introduce fail conditions to change this behaviour.

2.3 fileIO.c

2.3.1 readFile():

readFile() takes a char[] (string) and a pointer to a LinkedList structure. The string is the file the function opens for reading and operates on, and the list is the structure used for storage on the parsed data. The purpose of the function is to open a file, call a function to do processing, check for most sources of file read errors, and close the file. The actual processing of the file is done by a helper function called by readFile(). Any errors found will make the return value of success FALSE, but otherwise the function will return TRUE for success.

2.3.2 writeFile():

writeFile() takes a char[] string, a pointer to TVEntry (array of TVEntry), and the array size. The string is the file the function opens for writing and operates on, and the array is the source of data for the write operation. The function will output the array entries line by line, in the format specified in the assignment sheet. The function's purpose is to output the array to the user specified file in the correct format. The function will indicate success or failure through returning success with TRUE or FALSE.

2.3.3 processFile():

processFile() is a helper function to readFile() which does all of the parsing of the input file, taking a FILE* and the pointer to the LinkedList structure. The function is responsible for ensuring it does not read past the end of file, memory allocation occurred correctly, and that data is validated as it is read in to structure, and added to the list. This data validation involves checking if lines are empty (besides a '\n'), the correct number of variables are scanned in for, as well as day, hour, and minute being valid values. The function's purpose is to ensure only valid files are processed and stored, and that the relevant success code is returned.

2.4 linkedList.c

2.4.1 createLinkedList():

createLinkedList() takes no arguments, and allocates memory for a pointer to a LinkedList structure. The structure is defined in the related "list.h" file. The function's purpose is to allocate memory for the structure and set the structure's fields to be default list values. The function then returns a pointer to the structure to the calling program.

2.4.2 isEmpty():

isEmpty() takes a pointer to a LinkedList structure that was previously created, and checks the length to see if the list is empty or not. The purpose of the function is to return an integer representing if the list is empty (TRUE) or not (FALSE).

2.4.3 insertFirst():

insertFirst() takes a pointer to a LinkedList structure that was previously created, and a pointer to a void datatype. The void* acts as a generic datatype, so that any data could be entered into the list. The purpose of this function is to insert a value at the front of the list, calling various other functions to determine if there are any special circumstances to consider. For example, the list must be checked to see if it is empty or if there is any duplicate data already stored in the list. The function returns a success code which is dependent on the success of other functions and if the input value is a duplicate list value.

2.4.4 insertLast():

insertLast() takes a pointer to a LinkedList structure that was previously created, and a pointer to a void datatype. The purpose of this function is to insert a value at the back of the list, calling various other functions to determine if there are any special circumstances to consider. For example, the list must be checked to see if it is empty or if there is any duplicate data already stored in the list. The function returns a success code which is dependent on the success of other functions and if the input value is a duplicate list value.

2.4.5 insertBefore():

insertBefore() takes a pointer to a LinkedList structure that was previously created, and two pointers to void datatype. One void pointer is a value to find, so that the other void pointer is inserted before

that found value. The purpose of this function is to insert a value before some other value, calling various other functions to determine if there are any special circumstances to consider. For example, the list must be checked to see if it is empty or if there is any duplicate data already stored in the list. The function returns a success code which is dependent on if the input value is a duplicate list value and the success of the malloc.

2.4.6 insertAfter():

insertAfter() takes a pointer to a LinkedList structure that was previously created, and two pointers to void datatype. One void pointer is a value to find, so that the other void pointer is inserted after that found value. The purpose of this function is to insert a value after some other value, calling various other functions to determine if there are any special circumstances to consider. For example, the list must be checked to see if it is empty or if there is any duplicate data already stored in the list. The function returns a success code which is dependent on if the input value is a duplicate list value and the success of the malloc.

2.4.7 removeFirst():

removeFirst() takes a pointer to a LinkedList structure that was previously created. The purpose of this function is to remove a value at the front of the list. The function must check the list to see if it is empty and deal with it. The function returns the first node's value if successful, and NULL if unsuccessful.

2.4.8 removeLast():

removeLast() takes a pointer to a LinkedList structure that was previously created. The purpose of this function is to remove a value at the back of the list. The function must check the list to see if it is empty and deal with it. The function returns the last node's value if successful, and NULL if unsuccessful.

An important note to make is that removeLast() does not use removeValue() because as stated in commenting, it would be inefficient to access the last node with searching from the front. Each removal would require traversing through the entire list if implemented with removeValue(), so this function manages its own removal. removeFirst() can use removeValue() as it will search from the front, therefore it is efficient to use this method.

2.4.9 removeValue():

removeValue() takes a pointer to a LinkedList structure that was previously created, and a pointer to a void datatype. The purpose of this function is to remove a node from the list that matches the pointer to the void datatype. In other words, this function removes a specific value from the linked list if it can find it, and outputs an error if it was unable to.

2.4.10 peekFirst():

peekFirst() takes a pointer to a LinkedList structure that was previously created. The purpose of this function is to return a value from the front of the list without deleting it. The function must check

the list to see if it is empty and deal with it. The function returns the first node's value if successful, and NULL if unsuccessful.

2.4.11 peekLast():

peekLast() takes a pointer to a LinkedList structure that was previously created. The purpose of this function is to return a value from the back of the list without deleting it. The function must check the list to see if it is empty and deal with it. The function returns the last node's value if successful, and NULL if unsuccessful.

2.4.12 find():

find() takes a pointer to a LinkedList structure that was previously created, and a pointer to a void datatype. The purpose of this function is to search the list for a match with the pointer to the void datatype. In other words, this function returns TRUE if the value given exists in the linked list, or FALSE if it does not.

2.4.13 getLength():

getLength() takes a pointer to a LinkedList structure that was previously created. The function's purpose is to return the length of a list, equivalent to typing "list->length". This could be further modified in the future to output an error if the length is negative, or warn if the length is 0 or similar.

2.4.14 outputList():

outputList() takes a pointer to a LinkedList structure that was previously created. The function was designed for debugging the linked list. In debugging, the data that was stored in the list was pointers to integers, and as such these could be dereferenced and output as integers. The purpose of the function is purely for debugging int* linked lists, and should not be used for any production code. It returns a success code of TRUE if it can output its data, and FALSE if the list is empty.

2.4.15 clear():

clear() takes a pointer to a LinkedList structure that was previously created. The function's purpose is to remove all nodes from the list, freeing the nodes as it goes. The function then returns a success code of TRUE if the list is empty, and FALSE if it is not.

2.5 comparison.c

2.5.1 compareByTime():

compareByTime() takes two pointers to the void datatype. These two pointers represent two "things" that are to be compared with each other. The purpose of this function is to compare these two things (which will be pointers to TVEntry structures), by comparison of their day, hour, and minute structure fields. The first thing being: "less than" returns -1, "greater than" returns 1, and "equal" returns 0.

2.5.2 getDay():

getDay() takes a char* (string) which represents the TVEntry structure's day field. It then matches that day field's contents with the corresponding day of the week, giving it a code/ID that will be used for comparison. The purpose of the function is to give each day string a corresponding ID for comparison between two days to see which comes first. It returns the ID to the caller for comparison.

Note that Monday is considered the start of the week, and is therefore the lowest value.

2.5.3 compareByName():

compareByName() takes two pointers to the void datatype. These two pointers represent two "things" that are to be compared with each other. The purpose of this function is to compare these two things (which will be pointers to TVEntry structures), by comparison of their title structure field. The titles will be compared character by character until a case-insensitive non-match occurs. From there, if the first thing is: "less than" returns -1, "greater than" returns 1, and "equal" returns 0.

3.0 Filtering

3.1 Implementation

Filtering was done in the function filter(). filter() is passed a pointer to the TVEntry struct (array of TVEntry struct), a char* (string) representing the user's filter criteria, and the size of the array. There are two counters/indexes that are used, ii and jj. ii is used to loop through the entire array, and is the node that is to be compared with the filter criteria. jj is used to reference from the start of the array, overwriting the values at the front of the array with later values, incrementing itself to not overwrite values it has modified itself.

filter() goes through each array index, checking to see it the structure that it holds, TVEntry, has a day field value that matches with the day the user inputted as the filter criteria. If it does, this overwrites the beginning array values, starting at jj = 0, with the current array index's struct pointer. jj is then incremented so that the newly modified index will not be modified again. If there is no match, it will go through the next indexes searching for matches. Finally, the jj count (number of matching elements) is returned as the new array size.

In a big picture view, the filtering only occurs after the linked list is copied to an array, and the array is sorted. This filtering is done on the array, and directly modified itself, rather than creating a new array and copying values to the new array (such as with memcpy or using realloc() similarly).

As an important note, I decided not to use realloc() (or similar) to shrink the existing array. This is because the realloc function does not always free the memory to the operating system for other processes to use. Rather, realloc() usually keeps the memory that is now unused for the array as allocation for future process mallocs. Considering the program does not malloc() after this point, the unused memory will remain unused in the process until the array itself is freed (which occurs shortly after anyway). If the program is extended to be longer running and/or use mallocs after this point, there may be a benefit to refactor the code to use realloc(). As it is, the function returns a new size for the array and this is updated in the calling function. Essentially, this means that anything after the filtered array is considered outside bounds, and is only freed when the array structure is freed after outputting the array to screen and file. The difference in approach is realloc() is an expensive call time-wise, while just changing the "apparent" size of realloc() could be expensive memory-wise. I decided the trade-off for allocating to a new array, checking for NULL (failure), and then making the old array point to the new one, for a chance that some memory is freed earlier, is not worth the potential speed difference.

3.2 Alternative Approach

There are many alternative ways to approach the filtering step. An obvious one from the implementation I have given above, would be using realloc() to shrink the array instead of updating the array size. As mentioned, this would result in a time-cost, but come at a possible memory saving. This would be implemented by creating a new array, with size of jj (number of matching elements), and then matching elements could be copied across with the realloc().

As demonstration of this alternative approach, see the code on the following page for how I would implement realloc(), adding the code below the for loop.

Code adapted from M.M's answer at https://stackoverflow.com/a/26226613 (2014):

*Note: arrayCopy would be declared at the top of type TVEntry**
Using realloc():

```
arrayCopy = realloc( array, jj * sizeof(TVEntry* ) );
if ( arrayCopy == NULL ) /* unsuccessful realloc */
{
        /* array will be the original size */
        fprintf( stderr, "Error: Could not shrink array!\n" );
}
else /* successful realloc */
{
        array = arrayCopy;
}
return jj; /* new size */
```

Another possible alternative approach would be to filter earlier in the program, before sorting and before transferring from a linked list into an array. This would mean the filter would be performed on the linked list itself, rather than on an array. The filtering of a linked list could be helped by extending the linked list to have more methods, such as allowing peeking further into the list. The way this could work is peekFirst() could be called to check if the TVEntry has the correct day of the week, if it did it would be kept on the list, otherwise it would be removed (and freed!). However, if it was to remain on the list, unmoved, it would help to have a function to peek past that first (or second, or third, etc) node, looking deeper into the list for invalid entries. This could be programmed to work in the filter method though, having a currentNode that is checked to see if it abides by the filter (removing it if it does not), and then moving onto the next node by making currentNode = currentNode->next.

The above solution would mean there is no need for realloc() or similar, and additionally the transfer into an array would be quicker as there are less entries. This would result in the sort being faster as well, all while the memory consumed by the program should be reduced. The reason I did not implement the solution above is because the assignment specification has a step-by-step implementation of functions, asking for the list to be copied to the array, then the array sorted, then the array filtered. This may mean the above approach may not be appropriate, if this specification cannot be deviated from.

4.0 Output Testing

The finished code can be tested quite easily, as shown below.

After obtaining the tarball file, type:
tar -xvzf 18340037_Assignment.tar.gz

The tarball contains all the relevant .c and .h files. In addition to this and the report, the makefile and some input test files are present in the package. The input test files are various types of tests I have done on the program. These can be generated through the included generateInput executable.

Firstly, compile the program using the command *make*.
From here, the executable can be called using the following format:
./ProductionBuild *inputfile outputfile*
Where inputfile and outputfile represent files in the current directory. For example, typing:
./ProductionBuild test1.txt output.txt, will use test1.txt as the source of TV entries, and output.txt will be generated by the program, storing the sorted, filtered list of TV entries.

In fact, the above command can be run simply with the shorthand makefile commands:
make run OR *make runm*
These commands will pass test1.txt and output.txt as the default parameters, with the "runm" variant calling Valgrind. If any errors occur in Valgrind, the verbose version of this command: *make runmext* can be used as well.

In addition to the test files included, I include a program that generates files that can be used as input for the assignment program. This can be called as below:
./generateInput *<EntriesRequired> <filename>*
Where EntriesRequired is the amount of entries to generate, and filename is the name of the generated file. For example, typing:
./generateInput 100 test6.txt, will generate 100 TV entries into the file test6.txt.
This could then be tested by the assignment program by using ./ProductionBuild test6.txt output.txt

For the included test files, the following are a description of their contents and what they test.
test1.txt: 10 000 entries, tests large data source for any memory leaks or any snowballing affects.
test2.txt: 113 entries, tests the "name" sort by having many different variants of the same title. Includes other titles to effectively show the name sort in action. Test by inputting "Monday" and then "name" for the input prompts.
test3.txt: 100 entries, tests a more standard amount of input data. More representative of real world input.
test4.txt: Contains various errors in the file. Empty lines, incorrect days, hours, minutes, and just general non-sense data. Tests the ability of the program to handle input file errors.

I have tested all the above files and verified that the program behaves as expected, and has no memory leaks, segmentation faults, or any other Valgrind errors.
However, for the report I will discuss a valid file and its expected vs actual output. See the next page.

Testing test3.txt, 100 entries of data that is fairly expected.
Input data is included and can be analysed in test3.txt
Firstly, run the program with: valgrind ./ProductionBuild test3.txt output.txt
When the program prompts to enter a day of the week, you can enter invalid input to validate that the program will re-prompt and deal with it. Being case-insensitive, enter “Monday” or any case variation of it. Following this, when asked the sort type, invalid input can once again be input to verify that the program deals with it. Enter “name” or any case variation of it, and the output will be printed to the screen and file.

By visual inspection of the screen output below-left, these entries are alphabetically organised:

```
[18340037@saeshell04p Assignment]$ valgrind ./ProductionBuild test3.txt output.txt
==16921== Memcheck, a memory error detector
==16921== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==16921== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==16921== Command: ./ProductionBuild test3.txt output.txt
==16921==
Enter a day of the week:
Monday
Sort by "time" or "name"?
name


21:59 - Baby Driver
20:40 - black-ish
14:35 - Fear the Walking Dead
 6:53 - Fringe
19:51 - Ghost in the Shell
 3:07 - Guardians of the Galaxy Vol. 2
 9:03 - Infinite Challenge
 2:17 - Marvel's Agents of S.H.I.E.L.D.
13:30 - Maze Runner: The Death Cure
20:28 - Monsters, Inc.
 1:29 - Mysterious Personal Shopper
23:19 - Narcos
 6:06 - Snowpiercer
 6:38 - Super Sentai
14:38 - The Originals
==16921==
==16921== HEAP SUMMARY:
==16921==     in use at exit: 0 bytes in 0 blocks
==16921==   total heap usage: 206 allocs, 206 frees, 29,848 bytes allocated
==16921==
==16921== All heap blocks were freed -- no leaks are possible
==16921==
==16921== For counts of detected and suppressed errors, rerun with: -v
==16921== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 8 from 6)
[18340037@saeshell04p Assignment]$ vi test3.txt
```

Terminal Program Output

Additionally, Valgrind does not have any errors or memory leaks present.

While, we’ve validated that the name sort appears to work, to ensure these entries are in fact on Monday, we can run the bash command:
grep "Monday" test3.txt

This will output any lines with Monday in it. As can be seen below, the times for the day match up with the terminal output of the program (unsorted however).

```
[18340037@saeshell04p Assignment]$ grep "Monday" test3.txt
Monday 6:38
Monday 2:17
Monday 21:59
Monday 1:29
Monday 20:40
Monday 6:06
Monday 9:03
Monday 20:28
Monday 14:38
Monday 13:30
Monday 19:51
Monday 23:19
Monday 3:07
Monday 14:35
Monday 6:53
[18340037@saeshell04p Assignment]$ grep "Monday" test3.txt  | wc -l
```

Grep Output

File Output

Finally, the output file can be seen by opening it: *vi output.txt*

As seen on the left; this corresponds to exactly what the terminal output, and thus further indicates the program is correct.

Therefore, the program appears to be able to handle correct input data.

Testing test4.txt, the file has many errors that would prevent it from being parsed correctly.
When reading this file, upon error the program should safely exit early, output an error message, free all relevant memory, and not change the output file.
Run the program with: valgrind ./ProductionBuild test4.txt output.txt
For day of the week, input Monday.
For sort type, input time or name.

The program terminates, outputting the error “Incorrect time in file!”.
Valgrind indicates no errors.
output.txt has not been modified from the previous test.

Therefore, the program appears to be able to handle incorrect input data.

Clearly there are many different methods of testing this program. A different day and different sort method could be input. The input file itself could be different. The system could be using a majority of its RAM and not be able to malloc for large input files.

As a result, I have provided four different variations of input files that have been and are able to be tested. Additionally, I have created the generateInput program which will randomly generate N entries, where N is specified by the user.

This program should also work with test files that are external to me.