AC21008

Multi-paradigm programming & data structures

Image courtesy of freeimages.com
https://www.freeimages.com/photo/digital-dreams-1155914

2019

Discipline of Computing
University of Dundee

Assignment 2 Doubly-Linked List with Iterator and File I/O

About

This document describes your second AC21008 assignment and includes the following: what you are required to submit, the deadline for doing so, the intended learning outcomes of the task, the marking criteria, and other useful information and resources. For your second assignment you are being asked to implement a doubly-linked list to manage a playlist of tracks on an MP3 player. This is outlined below.

Table of contents

Assignment 2 - Doubly-Linked List (15%)

Assignment deadline

The deadline for submitting your assignment is 11:59pm on Monday 4th November, 2019. That's the Monday of Week 8 in Semester 1.

Percentage

The assignment is worth 15% of your total mark for the module.

Late penalties

As per the University policy, the penalty for submitting your assignment late is one grade point per day late (meaning if a submission is one day late and marked as a C2 it will receive a C3 grade). A day is defined as each 24 hour period following the submission deadline including weekends and holidays. Assignments submitted more than 5 days after the agreed deadline will receive a zero mark (AB).

What to do

Implement a doubly-linked list using the C programming language to solve the problem outlined in the 'Task Description' later. This is an individual task, i.e. not team-based.

Learning outcomes

This assignment addresses the following learning outcomes:

- To identify procedural and/or object-oriented programming requirements
- To recognize the roles played by data structures and algorithms in software development.
- To demonstrate an ability to program in C and/or C++
- To identify, select, and/or apply suitable data structures to solve a given computational problem.

- To develop and apply problem-solving, communication, time-management, self-assessment and independent study skills.
- To demonstrate best practices in software development processes.

What you need to submit

Assignments should be submitted to My Dundee - see under ‘Assignments’. You should submit a single file `playlist.c` which provides your implementation of the functions required for the doubly-linked list. See ‘Task Description’ below.

Task Description

A computer program is required which will allow a user to manage a list of tracks in an MP3 playlist, e.g. to create a new playlist, to add or remove tracks from the playlist, to skip to specific tracks in the playlist, to save the playlist, etc.

Your task is to create a program which emulates the list-based application described above. You will make use of a doubly-linked list with an ‘iterator’ to represent and navigate the tracks in the playlist. A doubly-linked list is beneficial in this regard because it allows the list of tracks to be traversed in any direction (the user can skip backwards and forwards) and it is also possible to maintain a reference to the ‘current’ track too.

Starter files

You are provided with some starter code to help you as follows:

- A header file for a doubly-linked list `playlist.h`. This header file declares the data structures that you need to use: an `MP3Track` data structure and a `Playlist` data structure. The header file also declares prototypes of the functions that you are being asked to implement.
- A tester program a (`playlist_tester.c`) which can be used to run some tests on your program and also provide a general example of how to instantiate and use the data structures that you will be implementing. The tester program also contains the `main()` function for the program.

- A text file '`test_playlist.txt`' which contains the contents of a playlist which has been written to a file and can be used to test the '`loadPlaylist()`' function described later.

Your task is to implement the functions which are declared in the `playlist.h` file and also described in more detail further below. You will implement these inside a single file: `playlist.c`.

What to submit

When you submit your assignment you should submit your `playlist.c` file only. This is the ONLY file that we will consult when marking your assignment. You may modify your own copy of the `playlist.h` file if you wish but we do not recommend this. We will use our own copy of this file during the marking of your assignment, not yours, and so we expect your code to adhere to the version that we have provided to you. You may modify the tester programs if you wish, e.g. to comment out tests you aren't ready for yet, to add additional tests or change output messages, etc.

How your code is graded

Your code will be graded automatically by running it through our own tester file. Our tester file contains a suite of tests that are designed to run your program through various scenarios and to check that your program has implemented the doubly-linked list successfully as well as being able to handle invalid and erroneous cases. We will build your `playlist.c` file with our own header file (`playlist.h`) and then run it through our test suite. The output of the tests will be feedback in relation to which tests have passed and which tests have failed along with a grade. NOTE: we expect your code to compile successfully without warnings and errors. If not, it will be subject to mark deductions. The tester program `playlist_tester.c` will provide you with an example of the kinds of tests we will be performing on your code but our own tester program will be more exhaustive.

Resources

You will recall from the lectures that a doubly-linked list allows both a forward and backwards traversal of the list (using 'next' and 'prev' fields). It is also possible to store the current position in the list and to move the current position backwards and forwards (using a 'curr' pointer). Your list will provide functionality for inserting items into the list both before and after the current position as well as being able to remove items from the current position. To help you get started, please review the following:

1. Lecture slides. In the lectures, we walked through the design of how a doubly-linked list works. From this you should be able to infer the pseudocode that you need to implement the steps required in your doubly-linked list functions (and there are some hints to the actual code required too).
2. The Stack sample code and the code from your first assignment. The code that you require for the doubly-linked list will have great similarities to the code of these other data structures and so they will be an effective starting point for you to build upon.

Error codes

The header file `playlist.h` contains a number of `#define` statements for error codes which you should use in the code that you implement. When your functions are called they are expected to return a value indicating the success or otherwise of the function. You should use the values defined in the header file for this. Our automated testing system relies on you doing this otherwise the tests could fail. Please see the sample code for the Stack data structure for an example of using error codes.

Defensive programming

We expect you to practice defensive C programming skills. In particular, you will not get far in our tester program without verifying parameters that are passed to your function. The first tests we run for each of your functions will look for illegal or invalid input parameters, so remember to check for these. Remember to check for special cases in your list. Remember to make sure that you always initialize your pointer variables before using them. Remember to free up the memory you have used.

What you need to implement

You will implement the functions described below. We suggest implementing '`createPlaylist()`' first and moving down from there. We also recommend reviewing the `playlist.h` file to understand the `#defines` and `structs` provided while you read through the specification below.

NOTE: when you allocate memory within your program, you must use a function that we have provided to you: `myMalloc()`. This function works exactly the same way as `malloc()` but it allows us to run memory allocation tests on your code. If you don't use this function, then your code will fail these tests and you will lose marks. See more details about `myMalloc()` later.

NOTE: all functions should return a value of SUCCESS if they are completed successfully. If any errors are detected within a function (e.g. invalid inputs, failed checks) then the appropriate error code should be returned.

NOTE: some assumptions to be aware of when implementing your code: the track name for any MP3Track is expected to be up to a maximum of 49 characters in length. The track length for an MP3Track is any non-zero integer value.

NOTE: When handling strings in your code (for track names), you may find the following functions useful which are in C's `<string.h>` header file: `strlen()` which can be used to determine the length of a string; `strncpy()` which can be used to copy the contents of one string into another.

Here are the functions that you are being asked to implement with their weightings in terms of marks:

Function	Marks
`int createPlaylist(Playlist **listPtr);` This function should create and initialize a new, empty doubly-linked list and store a pointer to it within the variable provided (listPtr). **Remember to use `myMalloc()`** to allocate memory instead of `malloc()`.	7

`int insertBeforeCurr(Playlist *listPtr,`
`                     char trackName[],`
`                     int trackLength);`
 	
This function should use the trackName and trackLength provided to add a new MP3 Track into the given playlist 'listPtr'. The new track should be added immediately **BEFORE** the current position in the list. **Remember to use `myMalloc()`** to allocate memory instead of `malloc()`.
	16

`int insertAfterCurr(Playlist *listPtr,`
`                    char trackName[],`
`                    int trackLength);`
 	
This function should use the trackName and trackLength provided to add a new MP3 Track into the given playlist 'listPtr'. The new track should be added immediately **AFTER** the current position in the list. **Remember to use `myMalloc()`** to allocate memory instead of `malloc()`.
	16

`int skipNext(Playlist *listPtr);` This function should allow the user to skip to the next track in the playlist, e.g. shift the 'curr' pointer one position forwards in the list – if valid to do so.	5
`int skipPrev(Playlist *listPtr);` This function should allow the user to skip to the previous track in the playlist, e.g. shift the 'curr' pointer one position backwards – if valid to do so.	5

`int getCurrentTrack(Playlist *listPtr,`
`                      MP3Track *pTrack);`
 	
This function should get the data for the current track in the playlist, storing the data within the fields of the variable provided (*pTrack).
	7

`int removeAtCurr(Playlist *listPtr,`
`                 MP3Track *pTrack,`
`                 int moveForward);`
 	
This function should remove the MP3 track from the current position in the list. The track's data should also be copied into the fields of the variable provided (*pTrack). The variable 'moveForward' is used to indicate what needs to happen to the 'curr' pointer in the list after the track is removed (because the track that 'curr' currently refers to is being removed). If 'moveForward' is set to 1, then the 'curr' pointer should be set to point to the next track in the list which occurs immediately after the track which has just been removed. Otherwise, 'curr' should be set to point to the previous track in the list, immediately prior to the one which has just been removed.
	20

`int clearPlaylist(Playlist *listPtr);` This function should empty the entire contents of the playlist, freeing up any memory that it currently uses. After being emptied, the playlist should be in a state that it could be used again if required.	4

`int savePlayList(Playlist *listPtr,`
`                 char filename[]);`
 	
NOTE: this is an additional task for those who are willing to do more to achieve extra marks. This function should save all of the tracks in the current playlist into the given file. The data should be written to the file in a specific format in order to enable automated grading of your function. Each track in the playlist should be written to a separate line in the file with the following format:
	9

```
<track-name>#<running-time>#<newline>
```

So, assume you have the following tracks in your playlist:

Track name	Running time (seconds)
“I am a Giant”	192
“Creep”	204
“Let it go”	176

When you output it to a file, it should be as follows:

```
I am a Giant#192#
Creep#204#
Let it go#176#
```

PLEASE NOTE that the file you create should be written into the same folder as your C program rather than any other path. This should be the default if you specify a filename only (such as ‘playlist.txt’) rather than a full file path.

NOTE: If you decide not to attempt this function then please just provide an empty implementation for it which simply returns a value of `NOT_IMPLEMENTED`.

```
int loadPlayList(Playlist **listPtr,
                 char filename[]);
```

11

NOTE: this is an additional task for those who are willing to do more to achieve extra marks. This function should load details of a playlist from a given file and add these as MP3 tracks into a playlist. NOTE: this function is an alternative to the ‘`createPlaylist()`’ function. Whereas `createPlaylist()` simply creates and initializes an empty playlist ready to be used, `loadPlaylist()` will do the same (i.e. create an empty playlist ready to be used) BUT then also fill up the playlist with MP3 tracks on the basis of the track information found in the given file. What the function should do is: (i) create a new, empty playlist ready to be used (see your `createPlaylist()` function for the steps required here); (ii) open the given file, which contains a list of the names and lengths of a series of MP3 tracks; (iii) for each track in turn, add a new MP3Track into your playlist to contain the relevant track information– you can do this by calling one of the 'insert' methods that the playlist already has and which you have implemented yourself previously; (iv) finally, you should store a pointer to your newly created playlist in the variable provided as an input to the function (listPtr) – this is similar to what you have to do in your `createPlaylist()` too.

NOTE: when inserting nodes into your playlist (as read from the file), please use the 'insertAfterCurr' function – our automated tester relies on this when checking the validity of what you have read in from the file.

In the assignment information that you are provided with you will find a text file (`test_playlist.txt`) which contains a ready-made playlist for you to use during testing if you wish. This is the file that our automated tester will be using too. If you look at the contents of this file you will see that it stores data in the same format that is expected for the `savePlaylist()` function above. It will be in the format below:

```
<track-name>#<running-time>#<newline>
```

Example:

```
Agadoo#23#
Macarena#44#
My Way#37#
```

NOTE please ensure that the `test_playlist.txt` file is placed in the same folder as your C program and when you are reading from the file you should assume it is in the current folder rather than any other file path. If not, the tester could fail.

NOTE: If you decide not to attempt this function then please just provide an empty implementation for it which simply returns a value of `NOT_IMPLEMENTED`.

IMPORTANT! – MEMORY ALLOCATION – Use ‘myMalloc()’

You will notice the following code at the top of the tester program that has been provided to you (`playlist_tester.c`):

```
int mallocFail = 0;
void *myMalloc(size_t size) {
   if (mallocFail) { return NULL; }
   else { return malloc(size); }
}
```

This is a memory allocation function that we want you to use in your code. It works exactly the same way as `malloc()` but is used by us during the tests that we run on your code. Any time you allocate memory in your program use `myMalloc()` instead of `malloc()`. `myMalloc()` has exactly the same function prototype as the regular `malloc()`. It takes a `size_t` which represents the number of bytes of memory to allocate and then returns a pointer to the memory allocated as a ‘`void *`’ or ‘NULL’ if the memory allocation failed.

Commenting your code

For identification purposes, please make sure that you place a comment at the top of your `playlist.c` file before you submit it which includes your full name, matriculation number and module code. So, it may appear thus for example:

```
/*
    Name: Craig Ramsay
    Matric number: 234545656
    Module code: AC21008
*/
```

You can place this right at the top of the file, before any `<include>` statements.

Compiling your code

When compiling your program, please use the following compiler options:

```
-Wall -Wextra -pedantic -Werror
```

This will help to ensure that you identify any errors and warnings in your code. When we grade your assessment, we expect your program to compile successfully without errors or warnings. Otherwise, marks can be deducted. Consider using a `makefile` to streamline the process of compiling your program during development and testing.

Using the tester program

You are provided with a header file (`playlist.h`) and a tester file `playlist_tester.c`. You need to implement code for the functions that are declared in the `playlist.h` header file and you should implement these in the source file `playlist.c`. The tester file contains a `main()` function which will automatically run a series of tests on your code and report the results. Therefore, you don't need to add a `main()` function to your own .c file (but you can add one temporarily during testing of your code if you wish). The tester program runs a few basic tests on your code to help you get started and also reveals some of the checks you may need to make in your code as well as examples of how to use the doubly-linked list data structure that you create. Our own tester program will run a set of similar tests but it will also display marks for the tests that pass and a final grade. **PLEASE NOTE THAT OUR MAIN TESTER CONTAINS A MORE EXHAUSTIVE SET OF TESTS ON YOUR LOGIC AND LIST VALIDITY.** The tester file that we have provided you with is intended to help you get started and to provide some clues to things that you need to think about. If you ensure that you have implemented the linked list functionality correctly and that you also check for invalid inputs in your code or other relevant error scenarios then you should be able to achieve a good mark

Once you have started writing the code in your `playlist.c` file, you just need to compile it together with the tester program in order to run it. The example below shows using the `playlist_tester.c` program:

```
    gcc playlist_tester.c playlist.c
```

This will compile the code in both the tester file and your own source file and link them together to produce a single output file of `a.out`. You can then run `./a.out` to run your program and see the test results.

Getting started

When you first get started, perhaps it would be worth adding a basic, empty 'stub' for each of the functions that you need to implement in your .C file. You can then start fleshing them out as you go along. For example, one of the functions declared in the `playlist.h` header file is the following:

```
    int skipPrev(Playlist *listPtr);
```

In your `playlist.c` file, a starting point for implementing this function would be to add a basic, empty implementation as below. You can use the return value of 'NOT_IMPLEMENTED' to signify that this function is empty and still needs to be implemented.

```
    int skipPrev(Playlist *listPtr)
    {
        return NOT_IMPLEMENTED;
    }
```

You can then add the rest of the content to this function at a time that is convenient. If there are any functions that you don't manage to implement (e.g. if you run out of time) then you can leave them in this empty state. This will ensure that your program will still compile successfully, even if it doesn't pass all of the relevant tests yet.

Unused variable warning

As noted in your previous assignment; one possible side effect of creating an ‘empty’ function implementation such as that shown above is that you could receive warnings from the compiler about ‘unused’ variables in your code. In the case above, the compiler will likely complain that the input parameter into the function ‘listPtr’ is unused because we haven’t done anything with it yet. You may recall that there is a simple tip or trick that you can use to prevent the unused variable warning from occurring during this temporary situation in your code which is to use a `void` cast as shown in **bold** in the code below:

```
int skipPrev(Playlist *listPtr)
{
    (void)listPtr;
    return NOT_IMPLEMENTED;
}
```

To remind you, the `void` cast here effectively fools the compiler into thinking that you are now doing something meaningful with your variable. We aren’t really of course; we are attempting to cast / convert the value of the variable into nothing / void which doesn’t really have any effect at all. When it comes time to implement the code for this function later you should remove this line of code because it isn’t needed anymore.

NOTE: remember, if you have several input parameters into your function then you will need to add a separate `void` cast for each of them.

NOTE: remember, if you receive ‘unused’ variable warnings in your code because you really do have unused variables that you don’t need or have forgotten to use then you shouldn’t be using the `void` cast tip shown above to try and ignore or cover up these warnings; you should investigate what the warnings are and remove the unused variables from your code if necessary.

Reminder of files to use

Here is a reminder of the files that you are being provided with and/or will create yourself (see Table below):

File	Purpose / how to use
playlist.h	A header file that provides you with the data structures, error codes, and function prototypes for the doubly-linked list that you are being asked to implement.
playlist.c	A file that **you** will create to contain the implementations of the functions of your doubly-linked list. See the file `stack.c` from the Stack sample code as a starting point for this.
playlist_tester.c	A program which is provided to run some tests on your code and which shows you with an example of using the doubly-linked list data structure and some error checks to think about.
test_playlist.txt	A formatted text file which contains a saved playlist that may be helpful for you to use when implementing your `loadPlaylist()` function. Our main tester program makes use of this file too.