COMP 2401 -- Assignment #4

Due: Friday, November 22, 2013 at 12:00 (noon)

Goal

You will write a program in C, in the Ubuntu Linux environment, to manage a movie database.

In this assignment, you will:

- define the data types and structures required to represent a movie collection as a doubly linked list
- write code to interact with the user for the purpose of managing movie data
- implement functions to manipulate the doubly linked list
- create a set of input files to test a program

Learning Objectives

- get familiar with more complex dynamic memory operations by managing a doubly linked list
- practice a wider variety of user I/O interactions using standard library functions
- use redirection of standard input to test a program

Instructions

1. Data types

Define the data types required for the movie database:

- `MovieType` represents one movie; it includes the movie title, the year it was made, and the genre
- `MovieNodeType` corresponds to a node in the **doubly** linked list of movies, implemented as we saw in the "Advanced Linked Lists" section of the course notes
- your main function must define the movie list as a pointer to the head of the doubly linked list
  - there must be only **one** instance of the movie list in the entire program; do not make copies!
  - this is **not** a global variable

Note:

- your program must use the definitions in the header file found here: a4Defs.h

2. Movie management user interface (UI)

Write a main and a main menu function to interact with the user and provide these four options (with 0 to exit):

- add movies
  - `getMovieData` prompts the user for the number of movies to be entered, then prompts for the movie data
- delete a movie
  - the user enters the title of a movie to be removed from the list
- list all movies
  - a list of all movies is printed to the standard output
- list movies by genre
  - the user enters the genre of movies to be listed
  - a list of all movies of that genre is printed to the standard output

Note:

- you can find some helpful sample code for user I/O here: testUtil.c

3. List management functions

Implement the following list management functions:

- `addToMovieList`
  - adds a given movie to the movie list, alphabetically by title, then chronologically by year if titles are identical
- `deleteMovie`
  - removes the given movie from the movie list
- `printMovieData`
  - prints out a list of all movies to the standard output
- `printMoviesByGenre`
  - prints out a list of all movies of the given genre to the standard output
    - you must create a new temporary list containing movies of only that genre – do not copy the movie data!
    - you must reuse the `addToMovieList` and `printMovieData` functions

Notes:

- the print functions must display all the data for each movie
- remember to manage your memory! do not leave any memory leaks

4. Instrumentation function

Define, as a global variable, a pointer to an output file that will contain the contents of the movie list

- the file can be opened at the beginning of the main function and closed at the end of the program

Write a `dumpList` function that prints to the output file the detailed contents of the movie list, including:

- the value of the head, as an address
- for each node:
  - the value of the node, as an address
  - the address of the data, as well as its contents
  - the value of the previous node, as an address
  - the value of the next node, as an address

Notes:

- your program must call `dumpList` at the beginning and end of these functions: `getMovieData`, `addMovieToList`, `deleteMovie`
- your output must be formatted as in Figure 1
- you can reuse the `convertToBytes` and `dumpBytes` functions from previous assignments

5. Test input files

Design a suite of test cases that thoroughly exercise your program's functionality and possible error conditions.

In your readme file, list and number all the cases that must be tested. At minimum, these will include, for each feature of the program, one test case for every normal case and every error case; for the add movies feature, for example, the following must be tested:

- adding a movie to an empty list
- adding a movie to the beginning of the list
- adding a movie to the end of the list

… and many more …

Create a set of test input files; each input file:

- corresponds to one test case
- contains the standard input data to be redirected into your program for that test case

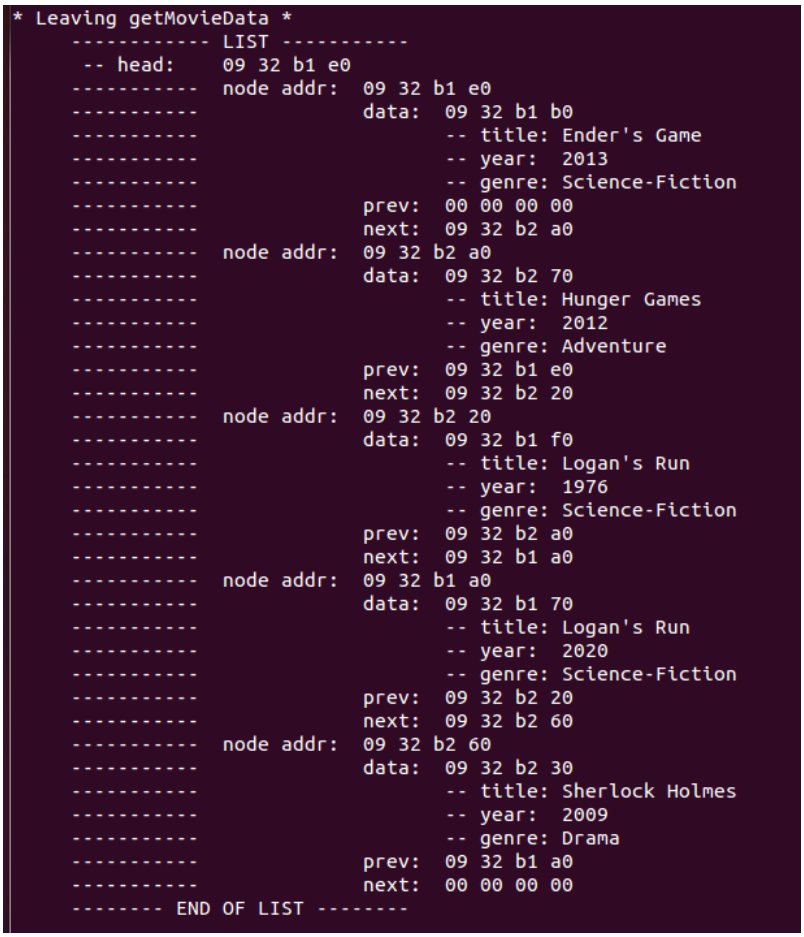

Figure 1 – Sample output of instrumentation function

Constraints

- **Design**:
  - you must separate your code into modular, reusable functions
  - never use global variables, unless otherwise instructed
  - compound data types **must** be passed by reference, not by value
  - you must manage your memory! use valgrind to find memory leaks
- **Reuse**:
  - you must include the given header file in your program and use its function prototypes exactly as defined
- **Implementation**:
  - your program must perform all basic error checking
  - it must be thoroughly commented
- **Execution**
  - programs that do not compile, do not execute, or violate any constraint are subject to severe deductions
- **Testing**
  - submissions that do not include a list of test cases and a suite of test input files are subject to severe deductions

Submission

You will submit in *cuLearn*, before the due date and time, **one** tar file that includes all the following:

- all source and header files
- a Makefile
- a readme file, which must include:
  - a preamble (program author(s), purpose, list of source/header/data files)
  - exact compilation command(s)
  - launching and operating instructions
  - a list of test cases that test your program
- a set of test input files

Grading

- **Marking breakdown:**

Component	Marks
data types	10
movie management UI	16
list management functions	62
instrumentation function	12

- **Assignment grade:**
  - Your grade will be computed based on the completeness of your implementation, plus bonus marks, minus deductions.
- **Deductions:**
  - **100 marks** if:
    - any files are missing from your submission, or if they are corrupt or in the wrong format
  - **50 marks** if:
    - the Makefile is missing
    - the code does not compile using gcc in the Ubuntu Linux shell environment
    - unauthorized changes have been made to the header and/or source files provided for you
    - code cannot be tested because it doesn't run
  - **25 marks** if:
    - your submission consists of anything other than exactly one tar file
    - your program is not broken down into multiple reusable, modular functions
    - your code is not correctly separated into header and source files
    - your program uses global variables (unless otherwise explicitly permitted)
    - the readme file is missing or incomplete
  - **10 marks** for missing comments or other bad style (non-standard indentation, improper identifier names, etc)
- **Bonus marks:**
  - Up to 5 extra marks are available for fun and creative additional features