CS237 Assignment 2012
Monitoring Runners and Riders

Connor Luke Goddard (clg11)

November 2012

Contents

1 Introduction

1.1 Purpose of this Document

The purpose of this document is to describe and justify the implementation of my solution written in the C programming language to the CS23710 assessed assignment 2012, as well as displaying examples of output from the system and providing the build/run logs of the system and its source code.

1.2 Scope

This document describes my implementation of the C & Unix assignment 2012. It should be read in conjunction with the highlighted appendices.

1.3 Objectives

The objectives of this document are:

- To describe my choice of application for the specified data structures, including a justification for each.
- To describe and justify any key design decisions that I chose to make.
- To provide references to further documentation that are required as part of the assignment specification.

2 Justification of the Design

2.1 Data Structures

As part of the assignment it was specified that both linked-list and array data structures must be used in the program implementation.

2.1.1 Linked-List

Within my solution I have created four linked-list data structures that are used to store course nodes, course tracks, courses and entrants.

By using linked data structures, new items (such as entrants/courses etc..) can be quickly and efficiently stored at the end of the linked list by using a *tail* pointer defined in the linked-list structure. This pointer is always set to the last item in the list, and is updated everytime a new item is added to the end of the list. Similarly, items in a linked-list can be quickly searched using a *head* pointer in the linked-list structure and a *next item* pointer contained within every data item that points to the next item in the list.

After much research, I decided to utilise a generic pointer in the linked-list item structure. A generic pointer, defined using the **void *** ***pointer name*** syntax, creates a reference to a value of any type using **type casting** to ensure values can be accessed correctly. By using a generic pointer as a reference to the data stored in a linked-list item structure, I was able to define a single, universal structure for a linked-list, that could then be used (via type casting) multiple times throughout the program for use with the different data types.

I was initially unsure about using generic pointers, as research I conducted indicated that it can be dangerous if types are not casted correctly, and can also make debugging difficult. However, I decided that as a large number of different data types were utilising linked-lists in the program, it was a viable choice.

2.1.2 Arrays

I decided to implement an array of pointers to the various nodes that make up a course, which would be stored in a particular course structure. This array would be initially be created when a new course structure was created by searching the existing linked-list of nodes and for matching node ID numbers, and creating a pointer to that node element. This resulted in an array of node pointers that exactly matched the node sequence for the course specified in the text files read in by the program.

This array is vital for updating entrant locations/statuses in the application. This is because the program uses the array index of the current node being accessed to update an entrant, as a way of indicating the current progress of

the entrant in relation to the total number of nodes that make up the course. Therefore the system can predict where on a course an entrant currently is using this *"current progress"* value obtained via the array index. This also allows the system to accuratly obtain the correct node an entrant is currently at if nodes are repeated throughout a course.

2.2 Application Structure

Throughout the design and implementation of the application, I attempted to structure the application in such a way as to enforce modularisation as this would allow particular functions to have multiple uses and become re-usable, and to ease debugging.

Source File	Header File	Description
main.c	main.h	Bootstrap loader for application and allows user to interact with the application.
event.c	event.h	Defines structure for an event and loads event data from file.
node.c	node.h	Defines structure for a course node and loads node specified data from file.
track.c	track.h	Defines structure for a course track and loads specified track data from file.
course.c	course.h	Defines structure for an event course and loads specified course data from file.
entrant.c	entrant.h	Defines structure for an event course and loads specified course data from file.
process.c	process.h	Provides functionality used to process and update entrant locations/statuses.
display.c	display.h	Provides functionality used to display information to the user.
N/A	linked_list.h	Defines generic structure of linked-list.

2.3 Key Design Decisions

2.3.1 Modularised Functions

As described in section 2.2, I attempted to create modular functions where applicable to help enforce modularisation within the program and to allow code to become reusable. A key example of this is the '**openFile()**' function. This takes a "user prompt" as a parameter and allows the user to enter a file name. It then attempts to open the file and pass it into a **FILE** type variable before returning this to the required function elsewhere in the program. This function therefore can be called multiple times by the program and used to load various data types including nodes, courses and entrants.

2.3.2 Updating an Entrant Time/Location

As part of the assignment brief, it is specified that the checkpoint times for a particular entrant should be allowed to either be read in from a file, or manually entered.

I therefore chose to create a series of functions that update the current entrant being updated, before updating the location and status of all other entrants relative to the newly entered time value. These functions use data including node ID numbers, track ID numbers, the array of nodes in a particular course, and the new time value to update an entrant's location and status, as well progressing their progress along the course. Once that entrant has been updated, all other entrants are updated also using the recently entered time value to predict their new location/status now that time has progressed.

Two separate functions are used to pass the relevent information into these modular functions either through file parsing, or prompting for input from a user.

2.3.3 Estimation of Entrant's Location

Due to the nature of the events the application is monitoring, there are times where the system has to "predict" where a particular entrant is located using data such as their last logged checkpoint/time and average time taken to complete tracks. The application therefore makes particular use of the node, track and course information, as well as new times entered via a file or manually, to predict where an entrant is located along their assigned course.

This prediction works on the basic principle that if an entrant has been logged at Checkpoint A, but has not arrived at the next specified checkpoint (Checkpoint B), then they must currently be at a location between the two. This location can either be a track, or a junction.

Using the average time taken to complete a track (specified in tracks.txt), the last recorded time of an entrant, and the current time (i.e. the most recent time value to be processed by the system), it can be predicted whether or not an entrant is still currently on a track, or at the next junction. An entrant location cannot be automatically set to a checkpoint.

2.3.4 Function Pointers as Parameters

When loading data files into the application, the user is prompted to enter a file name. There will therefore inevitably be occasions where a user will enter an incorrect file name, and so error checking and validation needs to take place. This error checking will be very similar for all files entered by the user.

I therefore decided, that instead of producing separate error checking code for

each file prompt, by using function pointers as a parameters, and specifying each load function to return an *int* if loading was successful or not, one function could provide error checking for all the separate load functions.

This generic function calls the function specified by the function pointer, and if the particular load function cannot open the file, it informs the user before allowing them to enter another file name.

2.3.5 Processing Medical Checkpoints (Extended Mission)

It is specified in the assignment brief that any delays that occur at medical checkpoints should not count towards an entrant's finish time.

I have therefore written a specific function that calculates the total time an entrant spends at a medical checkpoint, and displays the total delay occurred at all medical checkpoints in the entrant results table.

The delay at a medical checkpoint is calculated by converting the time an entrant entered and left the checkpoint into minutes. **Converting to minutes is accurate for this application as it is specified that no event will take place over more than a 24 hour period**. The leaving time is then subtracted from the entry time (both in minutes) to calculate the total time spent in the medical checkpoint.

3 Highlighted Issues & Assumptions

I have identified some assumptions that I have made for this assignment, and any issues that could occur when using the application if these assumptions are not shared:

- Checkpoint times supplied via file parsing or manual input will be chronological. - If time values that are previous to existing time values (i.e back in time) are entered, the system may not produce accurate predictions of entrant locations/statuses.
- No event will take place over longer than a 24 hour period. - The system converts time values to minutes to allow calculations that aid in entrant location prediction and medical checkpoint delay totals to take place. If an event was to take longer than 24 hours, predictions based on this conversion would be inaccurate.

References

[1] David Price, Fred Long *CS23710 Assessed Assignment 2012-2013. Monitoring the Runners and Riders* 2012: Aberystwyth University, Aberystwyth.

[2] Anders Ahlstrom *Void pointers in C* 2012: antoarts.com, http://www.antoarts.com/void-pointers-in-c/.

Appendix

1. **Appendix 1** - Main Mission Source Code
2. **Appendix 2** - Main Mission Build Log
3. **Appendix 3** - Main Mission File Load Log
4. **Appendix 4** - Main Mission Example Usage
5. **Appendix 5** - Extended Mission Source Code
6. **Appendix 6** - Extended Mission Build Log
7. **Appendix 7** - Extended Mission File Load Log
8. **Appendix 8** - Extended Mission Example Usage