MENU TREES REPORT

ECM2433

660037119

Design Choices

The overall concept of implementing a tree to represent a menu involved multiple structures. I created a ‘DATA’ structure which held reference to all of the physical data needed to identify a menu item. I also create a ‘LINK’ structure with the purpose of being able to act in place of a ‘DATA’ node. This allowed many structures to be able to link together as lists or even trees by using the pointers held within this structure. The main advantage of this segregation was to ensure that duplication of the physical data never occurred, since this would be memory inefficient. Instead if the same menu item occurred twice, only the LINK node with no actual physical data (only pointers) would need to be replicated. I also have made a virtual root node with ID 0 and label NULL, this acts as a super node and all menu items are descendants of it. It is not printed, but helps when searching or printing the tree.

In order to actually process the file, I would first read all the ‘A’ row formats and create a linked list (using the ‘LINK’ structure) with one link referencing each menu item’s ID and label to keep track of them and allow searching for specific menu data. The order of this list is insignificant. Once I reach the ‘B’ row formats I would then use this linked list to search all the data stored to find the relevant data needed. This data can then be used along with the child and parent node in order to add a ‘LINK’ node to the tree. Each tree node would therefore hold a reference to the data it represents, and any directly following siblings or child root LINKS. As such, the children of a parent are in a linked list, in the order of being added to the parent. This puts no limit on the amount of children supported, unlike arrays. The program will look for all instances of the target parent in the tree, and so adding more than one child of the same type is possible with this design.

In terms of how the tree is searched, I wrote an algorithm similar to the depth first tree search to locate parents and insert a node as a child. As such, I search any children recursively before moving onto a sibling of the same tree depth. Printing and freeing the tree is also achieved by using this general search pattern.

Data representation of the Menus

The ‘DATA’ structure holds two attributes. The ID of the menu, and the text label. Since the ID of the menu can never exceed 9999 or hold a negative values, I chose an unsigned short integer to represent this. I believe this is the most viable option in terms of memory preservation. If the program needed to store a larger value in future beyond this limit, it would not take many alterations in the codebase to allow a larger number to be stored. The text label is represented as a string, however the potential size of this string has no strict limit. It is only restricted by the memory available on the system.

I use a function to continually read characters from the file, and dynamically increase the amount of memory allocated to the string if the capacity is met. I implemented this dynamically resizing string in order to conform to the requirements in the project specification. It indicated the length was >1, which meant technically the string could be of infinite length, as such should not be restricted by the software. Also buffer overflows are impossible, which is very important security in a commercial setting.

Program Structure

I have laid out my codebase in a way to conform to good C practices, and to maximise reusability of my production code. I have done this by ensuring that all functions have a matching prototype in their header file, and that structure declarations and includes are also included in the header files.

Most importantly, I have separated some generic functions of which I needed to write but have no direct links tied into the main program. I order to do this I created a new C file and header and wrote the functions in there. This means that these functions are classed as 'extern' functions and are now entirely reusable elsewhere. In a business setting this would be very advantageous, as they could be utilised for future projects. In this case the two functions I have segregated are: 1) To read and store a string of unspecified length until the end of the line of file is met 2)Reading of a specified number of characters from a file, and conversion to an integer.

Memory management

I have done my utmost to make sure the memory usage of the program is clean. I never declare any fixed sized elements that could restrain the functionality of the program with based on input size. I developed a function ('cleanUp') which, with the use of helper functions completely frees all memory that I have ever manually allocated. This is regardless of the situation, for example: If the program finishes successfully or even if the program ends with an error, this function is built to completely free up the memory. By doing this I have enabled my program to perform "graceful shutdown", as well as prevent any memory leaks. I have tested the leakage of the program on a memory monitor "valgrind", which shows 0% leakage or memory errors for the example files I tested this on, even when an error occurs.

The absence of errors reported in "valgrind" proves the program does not ever attempt to free twice or access already freed memory, which is very bad program design and can lead to unexpected results.

Error Reporting

In terms of error reporting, I have integrated a sophisticated feedback system to allow users to quickly identify where a possible error is in an inputted file. I do this by keeping track of the file line number currently being processed, and form this to be part of the error message if one was to occur. By doing this, it could greatly reduce the time needed to fix errors in files and in a business setting, time spent fixing is directly equivalent to economic loss.

Assumptions

- All menu items have unique IDs. (In the case of an ID conflict, my program takes the first declared menu item.) I decided that searching for conflicts would cost too much program efficiency.
- The reader of this report has technical understanding.