CS22510 – Assignment 2

Alexander Roan

Introduction

This report documents the language choices I have made for the three shipping programs and the reasoning behind them. I compare and contrast my experience of using the three languages in terms of amount of code, clarity and language features that affected my solution. I also discuss any changes in language choices that I would make if I repeated the project.

I implemented the three programs using the languages in the following way:

1. Data Entry – Java
2. Ship Finder (by name or MMSI) – C
3. Proximity Indicator – C++

I will also in some capacity be comparing the features of the three languages when comparing the choices I have made for the three programs. Speed of compilation, execution and memory management were my biggest consideration when deliberating which language to choose for each program, closely followed by how long I thought the code would have to be in each language to achieve the fastest result.

Reasoning behind choices

Breaking program requirements down before the deciding which language to use was vital before any implementation had begun. From my understanding of what the programs would need to be capable of, I deduced that the least memory strenuous program would be the Data Entry program since it would not require any sort of data structure other than perhaps an array of the required ship information that was being entered. Instead, it would create a new ship file every time new information was entered and therefore had no need to store information in memory. The Ship Finder and Proximity Indicator on the other hand would require a 'load' algorithm where all of the ship data was updated into the program so it could be speedily accessed during the program's execution.

Since memory access and data structures are far faster in C and C++ than in Java, I decided that I would use C and C++ for the Ship Finder and Proximity Indicator (both programs would need data structures to hold the current state of the ship data), and create the Data Entry program in Java (at this stage I had not decided which of the other two languages I would use for the Ship Finder and Proximity Indicator). Java is also the language I am strongest in which aided my decision as to what language to choose for Data Entry, the first of the 3 programs that I had planned to create. I knew that in Java I could perfect a bug free file system for the following programs to access and read easily.

Although I had chosen Java for the Data Entry program, I was not sure as to which of the 'C' languages I would chose for the other two programs. After some deliberation I settled on C as my desired language for the Ship Finder program because I felt that I could write a fairly compact program with it and that the memory access would be very efficient for loading in ship information

from the ship files. Initially, this decision was perceived to be the perfect choice and the result was perfectly functional and very fast performance wise. However, on reflection perhaps using C++ for the Ship Finder and C for the Proximity indicator would have been more preferable due to memory considerations that I shall soon explain.

Experience of C, C++ and Java

Java

Already my strongest language, I was confident I could produce a Data Entry program with a small amount of actual code and with very good functionality. The program that I ended up writing was only 198 lines long (including comments and white space) and all of it was in a single Java file making the code quite easy to read and understand. The process of adding a ship is quite simple compared with what the other two programs need to accomplish during their execution. Ship data is entered, it is checked over to make sure that the new ship or the ship that is being moved is in the shipping area and, if that condition proves to be true, the data in the ship file is changed and a message is added to the log file. If the ship is not in the shipping area then the existing file is deleted, or the new one is not created (depending on whether there was one there in the first place), and a message is added to the log file explaining that a ship has been removed due to sailing out of the shipping area. Java provides some very useful file management capabilities which made it very straightforward to create, manipulate and delete files and folders. These include using File objects and methods such as mkdir() and output stream objects.

C

Using C for the Ship Finder program was a fairly straight forward programming procedure. If the user selects to enter an MMSI number instead of a character sequence, that number is used to open the ship file (since the ship files are named after their MMSI numbers) if it exists. If the option to enter a character sequence with the name of the ship (and any name of a ship with that character sequence in it) then the process becomes more complicated and the use of a data structure such as a linked list is needed. Because there are no standard importable data structures in C I created a basic node struct and linked them together into a linked list. This proved to be straight forward. Once the ship linked list was set up, it was a case of reading all of the ship files information into this linked list every time the program was instructed to look for ship names. This way, all of the ship data in the linked list in memory would keep up to date with all the latest ship information in the file system and the program can traverse through the linked list checking the names of the vessels against the data input by the user. The C method ‘strstr()’ was vital in this process as it checks if one string is present in another and returns a pointer to the first occurrence of the second string in the first string. It will return NULL if there is no match found. Passing structs into a C linked list in this fashion is far more memory efficient than passing objects into a Java linked list. This is one of the factors that definitely support my choice of using C for the Ship Finder program and not Java. Without including the header files, the total lines of code in the Ship Finder program amount to 221, in a total of 2 actual C files. This is a fairly lightweight and compact program which performs at a high speed. I think the code I’ve written for this program is easily read and understood with only a few methods in each file which are clearly commented to explain, if there is any doubt, what these methods accomplish when they are called.

C++

I enjoyed using C++ for the Proximity Indicator. Since my first language was Java, an Object Oriented language, I found the culmination of working with both Java and C before this program made the switch to C++ very comfortable. I did try out a few things using the Standard Template Library functionality, and decided to use a vector of strings to store the ship MMSI numbers that would be read into the program from the ship file names every time it looped back on itself to the start point, keeping the ship data in memory up to date. Not only that, but when the ships are loaded in using the load_ships() function, an array of Nodes (class containing variables for the ship MMSI, latitude and longitude) is created (to the same size as the vector of MMSI strings) and each file is read for its MMSI, latitude and longitude. This array of nodes is the structure that is traversed to find any ships nearby when the user enters a latitude and longitude of anywhere in the shipping lane (not to be confused with the vector of strings which hold the MMSI numbers (which are also used as the names of ship data files) of the ships). Overall, the total number of lines of code spanning the three '.CPP' classes is 260 (this is not including the header files) which again is a small amount of code for the program. The code written here is very clear as to what tasks are being performed. I found that working with the standard template library in C++ was very useful in storing the ship information in a vector, a feature not available in C.

Alternate Choices

On reflection of the whole project covering all three programs, I'm fairly pleased with the choices that I have made and the way in which I have implemented the coding for the three programs. I feel that it was the right decision to leave the least memory strenuous task to Java to ensure that the average memory efficiency for the three programs is consistently low.

One factor I have pondered however is whether I should have made different choices regarding the implementation of the Ship Finder and Proximity Indicator. C is known to be the fastest and most memory efficient of the three programs as long as it is coded well and I've used it to implement the Ship Finder program. As explained above, the Ship Finder program only loads the ship files into the linked list if the user selects to search using by the name of a ship or part of the name of a ship(s). If the user choses to search for a ship by MMSI nubmer the program simply attempts to read straight from the file with that MMSI number without any loading into data structures and is done in a tiny amount of time with no large memory requirements. This means that ship data will only be loaded into memory only half of the time. This contrasts to the Proximity Indicator due to the fact that the ship files are always loaded into memory every time the program is run and consequently looped back to the beginning. Perhaps it would have been worth implementing the Ship Finder in C++ where memory is not needed to store the ship data fifty per-cent of the time. Instead, use C to write the Proximity Indicator where data structures are always used when the program runs.

It is not known whether this would make a discernible difference due to the type of the data structures that I have actually used. In C I had to use a rudimentary linked list that I created myself. This may or may not be slower that the vector and the array of node objects in C++, even if they were only loaded half of the time. I do anticipate however, that writing the Proximity Indicator in C would result in a far larger program with more lines of code than is currently present in the programs. Therefore, it is debatable whether making this change would really make any substantial difference in speed or efficiency.