

Parallel Arrays:**Vehicle Database**

This project requires implementing a simple database application for storing information about a collection of vehicles and responding to searches on that database. You will be given four pieces of information about each vehicle: license number, manufacturer, model and year. This information must be read and stored in four "parallel" arrays (the use of `struct` variables is explicitly forbidden).

Your program will then read various search commands from a script file and log the results of those searches to an output file.

Input files:

This project will involve two input files, one containing the vehicle data and another containing the search commands that are to be processed. The vehicle data file, named "VehicleData.txt", will contain two header lines and then an variable number of lines of data. The first header line will start with a label, terminated by a ":", and then specify the number of vehicle data lines to be read. Your program should read exactly that many data lines. There may be extra data lines; if so, your program should ignore them. There will never be fewer data lines than the file header indicates. The second header line is just a separator and should be ignored.

Each data line will contain four logical values for a vehicle, separated by tab characters and terminated by a newline:

```
<license><tab><manufacturer><tab><model><tab><year><newline>
```

The first three data values are just strings, and may contain spaces. The last data value will be an integer. There will not be more than 100 vehicle data lines. Here is a sample vehicle data file:

```
Number of vehicles: 18
-----
ZRX 7220 Honda CRV 1983
RHG 7801 Mercedes 190E 1986
QWG 5065 Dodge Neon 1994
IRQ 9347 Chrysler 300M 1998
KBD 3886 Dodge Stratus 1983
EHP 4860 GMC Sonoma 1985
QAN 1669 Ford Pinto 1987
GCE 7789 Oldsmobile Cutlass 1984
RST 2662 Mazda 626 1998
EHF 9150 Chevrolet Camaro 1994
MYJ 2745 BMW 323is 2000
ZIP 7275 Volvo 740 1981
MLF 7568 Pontiac Grand Am 1994
NAI 8335 BMW 318i 1994
VXQ 8535 Subaru Outback 1988
YPJ 1292 Subaru Forester 1991
DID 2151 Dodge Laser 1986
MIL 0584 Pontiac Grand Am 1983
NZM 0799 Dodge Laser 1988
HSX 0195 Ford F100 1975
LRA 9157 Lincoln Continental 1987
ZBK 4803 Mercury Mystique 1986
FUW 5654 Buick LeSabre 2002
GXY 8242 Chrysler PT Cruiser 2000
NIH 9530 Subaru Outback 1997
OOE 0474 Dodge Intrepid 2000
```

Note: you must store the vehicle data in the order in which it is read in order to produce some of the search output in the expected order.

The second input file, named "Searches.txt", contains a list of search commands, one per line. Each search command is to be processed as soon as it is read, so you will not need to read and store all of the search commands. Each search command will target a particular vehicle data value:

license<tab><license string><newline>

Search the vehicle data for a vehicle whose license matches the given license string exactly. If a matching vehicle is found, log its license, manufacturer, model and year. If no match is found, log an error message. (If there should be two or more vehicles with that license, just report the first one.)

make<tab><manufacturer name><newline>

Search the vehicle data for all vehicles made by the given manufacturer. For each matching vehicle, log its license number. If no matching vehicles are found, log an error message.

There is no limit on the number of search commands that may be given. Your program should be designed to stop when an input failure is reached. Here is a sample search file:

make	Honda
license	ABC 7908
license	OOE 0474
make	Hyundai
license	EHF 9150
license	MIL 0584

Both input files are guaranteed to conform to the specified syntax. Additional sample input files will be posted on the course website.

Log file:

Your program will write all of its output to a log file, named "Log.txt". As usual, the log file will begin with some identification lines. Then, for each search command, the log file will echo the complete command and then the resulting output. As always, be careful to match the fixed text shown below (e.g., "not found").

The output for each command must be delimited for easy reading, as shown. Each echoed command must have a descriptive label, as shown.

For a license search:

- For a successful search, echo the vehicle data as shown, separated by whitespace of your choice.
- For a failed search, the error message must begin with the given license string.

For a manufacturer search:

- For a successful search, echo the matching license numbers on a single line, separated by whitespace.
- For a failed search, the error message must begin with the manufacturer name.

Here is a sample log file:

```
Programmer:  Bill McQuain
CS 1044 Fall 2001 Project 7

-----

Searching for manufacturer: Honda
      ZRX 7220

-----

Searching for license: ABC 7908
ABC 7908 not found

-----

Searching for license: OOE 0474
OOE 0474 not found

-----

Searching for manufacturer: Hyundai
Hyundai not found

-----

Searching for license: EHF 9150
EHF 9150      Chevrolet          Camaro          1994

-----

Searching for license: MIL 0584
MIL 0584      Pontiac            Grand Am        1983

-----
```

Evaluation:

Everything that was said in the earlier project specifications about testing still applies here. Do not waste submissions to the Curator in testing your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that are provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 100, based upon the runtime testing performed by the Curator System. We will also be evaluating your submission of this program for documentation style and a few good coding practices. This will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

As before, your implementation must meet the following requirements:

- Choose descriptive identifiers when you declare a variable or constant. Avoid choosing identifiers that are entirely lower-case.
- Use named constants instead of literal constants when the constant has logical significance.
- Use C++ streams for input and output, not C-style constructs.
- Use C++ `string` variables to hold character data, not C-style character pointers or arrays.
- You must make appropriate use of functions. You should have a function that reads the vehicle data file and stores its contents into the data arrays. You should have a function for each of the search commands. You should have a function that initializes the data arrays to sensible starting values. There are certainly other good candidates for functions. Aside from some testing functions, my solution uses seven functions besides `main()`.
- None of your functions can include more than 30 executable statements. An executable statement is one that causes something to happen. Comments, constant and variable declarations (even if they initialize) do not count as executable statements.
- When you pass an array to one of your functions, use pass by constant reference unless pass by reference is logically necessary. (Remember that for array parameters, pass by reference is the default.)

Reflecting the primary facet of this project, you must use four arrays to store the vehicle data. (The use of `struct` variables is explicitly banned for this project.) You must initialize each of the arrays to some sensible default values. As always when arrays are involved, be careful about array indices and about the difference between the dimension and the usage of an array.

Read the *Programming Standards* page on the CS 1044 website for general guidelines. You should comment your code in the same manner as the code given for the first two programming assignments. In particular:

- You should have a header comment identifying yourself, and describing what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every major block of code should have a comment describing its purpose.
- Every function implementation must have a header comment. The format of the header comment is described in the course notes, as well as on the *Programming Standards* page on the course website.
- Adopt a consistent indentation style and stick to it.

Understand that the list of requirements here is not a complete repetition of the *Programming Standards* page on the course website. It is possible that requirements listed there will be applied, even if they are not listed here.

Submitting your program:

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* can be found at: <http://ei.cs.vt.edu/~eags/Curator.html>

The submission client can be found at: <http://eags.cs.vt.edu:8080/curator/>

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement that was provided with the earlier project specifications in the header comment for your program.

Failure to include the pledge statement will result in a final project score of zero.