CSE 303, Spring 2007, Assignment 6
Due: Tuesday 22 May, 9:00AM

Last updated: May 14

Summary: With two other group members, you will build an application that takes files describing parts, products, an inventory, and orders and you will produce files describing a subset of the orders that can be fulfilled and the inventory that remains as a result. You will use your solutions to the previous assignment. You will write new code (about 200–250 lines). You will use `make` and `cvs` to help manage your development.

Requirements:

1. If you exit due to ill-formed input, you should print an appropriate error message to stderr first.

2. Your application (called `parts_and_orders`) should take exactly 4 arguments (else exit), which are all filenames that are used as described below. It should write two files, `new_inventory` and `completed_orders` as described below, replacing them if they already exist. If any of the six files cannot be opened, exit.

3. The four input files should contain only English letters (upper- or lower-case), spaces, and newlines. You must check for this and exit otherwise. Blank lines are possible and should be skipped without giving an error (i.e., proceed to the next line). Additional rules for each file follow; exit if they are not met.

   (a) The first input file is a list of parts. Any non-blank line should contain a sequence of English letters (no spaces) and that word, when converted to all lowercase, is a part you should add to the warehouse. Repeated part names are possible (but not a problem). Example line: `qarts`

   (b) The second input file describes products. Any non-blank line has one or more sequences of English letters separated by one or more spaces. All words should be converted to lowercase. The first word is the name of a product to be added to the warehouse. The remaining words are parts needed to make the product; these facts should be added to the warehouse too. A part name may occur multiple times on a line; i occurrences means i of the part is needed to make the product. Repeated product names are possible; the parts for the product is the collection of the parts on all lines. Example line: `foo pa par pa par  qar  qarts`

   (c) The third input file describes the parts inventory. Any non-blank line has one sequence of English letters then one or more spaces then one positive number (readable by `atoi`, if not a positve number exit). The word, converted to lower-case, should be a part in the warehouse (due to the first file), else exit. The number should be added to the part's quantity. Repeated parts are not a problem; the quantity for the part is the sum of the numbers. Example line: `qarts 17`

   (d) The fourth input file describes the orders. Any non-blank line has one sequence of English letters. The word, converted to lower-case, should be a product in the warehouse (due to the second file). Repeated lines means there is more than one order for the product. Example line: `foo`

4. The output file `completed_orders` should have only one line, which contains numbers separated by spaces. The numbers should be the line-numbers (first line is 1) of the orders that can be filled given the inventory; more on this below. The order does not matter. Example line: `5 4 1`

5. The output file `new_inventory` should have one line for each part, with the part name followed by one space followed by the nonnegative quantity of the part remaining after fulfulling the orders in `completed_orders`. The order does not matter. Example line: `qarts 0`

6. Using the functions in `warehouse.h` to add parts, products, parts to products, quantities, etc. given the input should be straightforward. Do not change the declarations of these functions or their implementations (except to fix bugs).

7. The warehouse functions must use the implementation of identifiers in `identifier.c`. Do not change this file or its interface (except to fix bugs).

8. To determine what orders can be fulfilled, you must use the code in `subset.c`. Do not change this file or its interface (except to fix bugs).

9. *You will need to add new functions to `warehouse.c` and add their prototypes to `warehouse.h`.* Do *not* expose the definitions of any of the structs. Instead, add these functions (making no other changes except to fix bugs):

   (a) `void inventory_to_array(int * arr, struct Warehouse * w);` assumes `arr` points to an array with length equal to the number of parts in the warehouse. It assigns the quantity of the part with id-number `i` to `arr[i-1]`. (Recall part-numbers start at 1.)

   (b) `void inventory_from_array(int * arr, struct Warehouse * w);` assumes `arr` points to an array with length equal to the number of parts in the warehouse. It changes the quantity of the part with id-number `i` to be the number in `arr[i-1]`.

   (c) `void product_to_part_array(int * arr, struct Product * p);` assumes `arr` points to an array with length equal to the number of parts in the warehouse and each element is initialized to 0. After returning `arr[i]` should hold the number of parts with id-number `i+1` needed to make the product.

   (d) `print_inventory(FILE*, struct Warehouse*);` should print the inventory to the file in the format described above.

10. Create a cvs repository for your project's C code, Makefile, and files used for automatic testing. Do not put generated files in the repository.

    (a) Typing `make` should build `parts_and_orders` without unnecessary recompilation as usual.

    (b) Typing `make test` should run your program on sample input and then compare the outputs against precomputed correct answers. (Use multiple commands and the Linux utility `diff`.) The commands should always run (i.e., you are not making a file named `test`), but `parts_and_orders` should be remade first if it is not up-to-date. One interesting test with a few parts and products is sufficient for what you turn in.

    (c) Typing `make clean` should delete any files made by the other targets, including the `test` target.

Advice/Hints:

- While the new functions in `warehouse.c` have been described in detail, how to structure the rest of your new code has been purposely left more open-ended. Style matters. Do *not* put this code in files used for homework 5; use the interfaces defined in the header files.
- Use library functions such as `fopen`, `fclose`, `getline`, `isalpha`, `strtok`, and `tolower`.
- Write some helper functions for the input since the various files have some similarities.
- The sample solution added about 210 lines of C code: 25 in `warehouse.c`, 4 in `warehouse.h`, and the rest in a new file. A lot of the code is for reading the input files.
- You will need to heap-allocate an array of int arrays for the orders (for passing to the code in `subset.c`), but you do not know how many orders there are. One solution is to resize the array as necessary (similar to the set code you wrote in homework 4). An alternate solution is to read the orders file twice.

Extra Credit: The second part builds on the first. You may do only the first (for less credit of course). If you do the extra-credit, do it in a subdirectory `hw6/ec` so we can grade your regular submission separately. (After finishing the regular assignment, copy all the files into this subdirectory then modify them. This is not ideal software-engineering, but it greatly simplifies grading.)

1. Extend your program so that it supports *prices* (in dollars and cents) for parts and products. Change the format of the first two input-files appropriately; in a file in your repository named `README` explain the new formats as precisely as the original formats are described above. Extend the warehouse interface to support getting/setting the prices of parts and products. Have your program print to standard-output the *profit* achieved by fulfilling the orders. (The profit is the cost of the products for the fulfilled orders minus the cost of the parts used.) Note the profit may be negative.

2. Change your code so that you maximize profit rather than maximizing the number of orders fulfilled. Explain in the `README` file how the interfaces and algorithms changed.

Turn-in (different from previous assignments!):

- Email Jimmy a message with subject `cse303: hw 6`. The body of the message should be nothing except an (absolute-path) directory d on `attu` in the project-space assigned to your group.
- The directory d should hold a `cvs` repository. The permissions for the directory should allow access only to members of the operating-system "group" we have given you. The files in the repository should be readable by everyone.
- Jimmy should be able to execute `cvs -d` d `co hw6; cd hw6; make test` and have it build your program and run your tests as described above.