

Automated Tailor Shop

A company that manufactures clothing wants to create a virtual tailor shop. In this “shop”, a customer would be able to register (to make orders), and order articles of clothing. The manufacturer also wants this system to allow buyers for other companies to make mass orders of clothing. The system will be used by the customers, stock managers, and sales staff. Ideally, we would implement different interfaces for the different kinds of users; however, since we are only using text input/output we will simulate these differences via the input format (see below).

Before placing the orders, customers would need to register by giving contact and billing information. The system should distinguish between individual and commercial customers, and store different information for both. Specifically, the information for an individual would include a name, an address, and a phone number. The information for a commercial customer would include the company name, a name for a contact, an address and phone number, as well as credit status. Commercial customers may also have a discount that is applied when billing for an order, the sales staff must set the discount. The contact people for the commercial customers who make orders for themselves (as individuals) get the same discount as their company. We assume that no two individual or company names are the same (i.e., you can assume that the system need not store two customers named “Ralph Rogers”).

Customers can also place orders. The system may suggest replacements if the customer’s selections are not available. The system should assign each order a unique serial number, sequentially beginning at 91000.

Stock managers are only concerned with making sure that items are in the stock. They will add new stock items, increase the stock units of existing items, and update the list of suggested replacements for each item. Each item will have a unique alphanumeric item “number” of 4 characters, an item name (which may consist of several words separated by blanks), and quantity (number of units).

Sales staff will need to access customer records, and information about orders. Sales staff may also set discounts for company customers.

Invocation tailor <inputfile> <outputfile>

Input

All input will be from a text script file. The input will simulate having different interfaces for each of the different kinds of user. The following commands will switch between “user interfaces”, each of which only allows specific commands. The “interface” commands are:

```
customer
    Interface should only accept customer commands.
stock
    The interface should only accept stock commands
sales
    The interface should only accept sales commands
```

All command input will be syntactically correct, but it is possible that some input will be invalid. Most commands will be on a single line, with the fields separated by tabs. Unless specified all fields may contain any characters other than tabs.

Customer commands are:

```
registerI   <name>           <address>   <phone>       <billing info>
```

Register the named individual. All fields may include spaces and other punctuation.

```
registerC   <company>   <contact>   <address>   <phone>       <credit>
```

Register the named company. All fields may include spaces and other punctuation.

```
order <name>
item <item number>    <quantity>
. . .
endorder
```

This is a multi-line command. The name is the name of the individual customer or company making the order. Each item of the order is listed on a separate line. If the item is not in the stock database, it should be flagged as unknown. If an item is available in the specified quantity, it should be flagged as filled, and the stock should be reduced by the given quantity. Otherwise, if the item is not in stock, then flag it as backordered, and log any suggested alternatives. Backordered items are not charged when the amount due for the order is calculated. Once each item is dealt with, the order should be assigned a number, and the number and total amount due for the order should be logged.

Stock manager commands are:

```
add <item number>    <item name> <units>    <price>
```

Either add a new item to the inventory, or update an existing item. The item number should be used to determine if the item is already in the inventory. In this case, the number of units in stock should be increased by the units given in the command, and the other fields changed to the new values. The item number is a four character alphanumeric string. The item name may consist of several words with blanks. The price is a nonnegative decimal value with precision 2.

If possible, backorders of the specified item should be filled with preference given to orders with lower order numbers. Unknown items should not be updated.

```
view stock
```

Display (to the output file) the list of stock items, including for each item the item number, item name, current number of units in stock, and price.

```
suggest <item number>    <substitute item number>
```

Specify suggested substitutions. (This is not transitive.) Both items should be in the stock database.

Sales staff commands are:

```
view customer <name>
```

Display all information pertinent to the customer (different for individual and company) and a list of order numbers and corresponding amounts due.

```
view orderlist
```

Display the list of orders, including the order number and customer name.

```
view invoice <order number>
```

Display the customer name, list of ordered items and suggested substitutions (properly labeled), each including item number, description, quantity and status (unknown, filled or backordered), and amount due.

```
discount <company> <percentage>
```

Set the discount for a particular company. The percentage should be a whole number between 0 and 100. It is a logical error to specify a discount for an individual customer.

Sample input files will be posted on the course website by Friday, November 12.

Output

All output should go to a text file. Output should be attractively formatted for readability and compactness. Labels and other descriptive text may be useful.

Design

You should spend as much time as possible on your design, and initially focus on objects that relate directly to the domain in which the program will work. Database objects should be implemented with STL maps encapsulated inside the database objects. For each kind of database, you should keep all customers in one data structure (specifically, you may not use different structures for the different kinds of customers, or different structures for the different kinds of orders).

Unnecessary duplication of information must be avoided.

There are good opportunities to use all of aggregation, association and inheritance. You should take advantage of those opportunities.

Programming Standards:

You'll be expected to observe good programming/documentation standards, as described in the *Elements of Programming Style*. Some specifics:

- You must include a header comment, preceding `main()`, specifying the compiler and operating system used and the date completed.
- Each `.cpp` and `.h` file must begin with a header comment block containing the name and e-mail address of programmer and the date of last modification of file
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.
- The organization of your implementation into `.h` and `.cpp` files should follow your design. In particular, classes should be declared and implemented in `.h/ .cpp` pairs, named with the corresponding class name.

Neither the GTAs nor the instructors will help any student debug an implementation, unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook, the notes, and the CS 2704 website. Note that code examples from these sources are not designed for the specific purpose of this assignment, and are therefore likely to require modification. Such code may, however, provide a useful starting point. You may use template classes from the STL, but may not use code from the MFC, or a similar library.

Evaluation

Your score on this project will be based on several factors:

- Runtime testing of your program, and correctness and completeness of output.
- Quality of external documentation (class diagram, class/operation forms).
- Quality of design, and adherence to good software engineering practice.
- Quality of internal documentation.

The relative weighting of these factors will be decided later. We may require you to demonstrate your program. To receive partial credit for programs that are non-working, or are not fully functional, a brief one or two paragraph description of the problem(s) must be included in the assignment submission, in an ASCII text file named `problems.txt`. The location of the problem, minimally identified to a specific function, must also be specified along with possible corrections that need to be made.

Deliverables

You must submit (electronically) an interim design document, containing a class diagram and class/operation forms, no later than the midnight on Friday, November 12. Your submission must be either readable by MS Word or a PDF file. **Do not zip the file and do not submit multi-file designs.**

Your final project submission must include:

- All source code (`*.cpp` and `*.h` files) comprising your project.
- MS Visual C++ project files (`dsp` and `dsw`) or Unix `makefile`, as appropriate. (VC++ users: do not submit unnecessary files, such as `ncb`, `opt`, `ilk`, `obj`, `pch`, or `pdb` files.)
- One set of input/output files; this should include redirected output.
- Revised design documentation reflecting the final design of your project.
- A brief ASCII text readme file, named `readme.txt`, containing:
 - Your name and e-mail address.
 - The section of the course you're enrolled in (example : MWF 11:00-11:50).
 - Your instructor's name.
 - The project name.
 - The platform and compiler you're using (example: MSVC++ 6.0 under NT).
 - Any special execution instructions. (If your program is not fully functional, be sure to mention that here and to follow the directions in the Evaluation section above.)
- An ASCII text file, named `pledge.txt`, containing the Honor Code Pledge listed below:

On my honor:

- I have not discussed the C++ language code in my program with anyone other than my instructor or the teaching assistants assigned to this course.
- I have not used C++ language code obtained from another student, or any other unauthorized source, either modified or unmodified.
- If any C++ language code or documentation used in my program was obtained from another source, such as a text book or course notes, that has been clearly noted with a proper citation in the comments of my program.

student's name

Your project submission will consist of a zipped archive file containing all of the items specified above.

You are allowed to submit your solution up to five times, in case you detect (or solve) problems after your first submission. Your last submission will be the only one tested and graded. Note that, due to late penalties, it is possible that a fixed but late submission may receive a lower score than a faulty but on-time submission.