CMPS 260
Fall 2005
Programming Project #5
2005.04.09

Date Assigned: **Monday, April 11, 2005**
Due Date: **10:00 PM, Saturday, April 30, 2005**

Folder due in your instructor's TA's mail box or office by close of business on Monday, May 2, 2005.

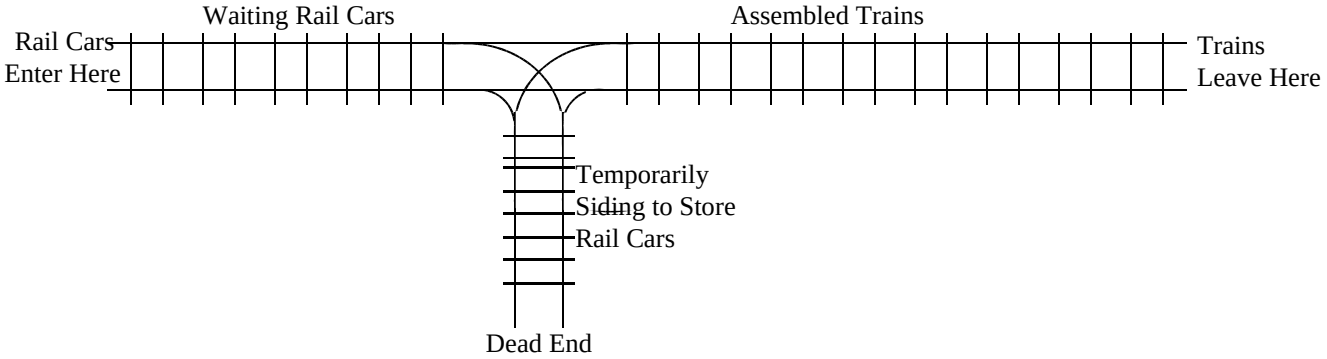
The coded solution to the following problem is to be done by you and only you. You may ask help from the class teaching assistants and the instructors, but you may not ask for help on this project from anyone else. You may use your notes, C++ texts, on-line tutorials, etc., but the code must be your own.

Assignment Description:

Railroad enthusiasts are numerous and active. They ride trains, visit stations, create models of rail systems and use software to simulate trains and rail systems. One of the favorite areas of a rail system for many enthusiasts is the "switching yard" or "roundhouse" where engines and rail cars are assembled into trains and sent out on various errands. Your assignment will be to create a program that allows the user to assemble rail cars into trains of cars of the correct order using a simple switching yard.

Description of the Switching Station:

The Switching station will consist one track that originates off the board and terminates off the board (i. e. has no visible beginning or end) and one dead end track that is connected to the middle of the first track. Rail cars are placed on the side reserved for waiting rail cars in the order that they arrive. Rail cars are assembled into trains in the correct order by moving them to the area reserved for assembled trains.



Rail cars have the following properties:

- rail cars are identified by a unique car number
- rail cars also have a destination number, but more than one car in a train may have the same destination number

Cars may be moved as follows:

- the waiting car closest to the siding junction may be moved to the top of the siding
- the waiting car closest to the siding junction may be moved to the rear of the assembled area
- the top car on the siding may be moved to the right most position in the waiting area, i. e. it will become the waiting car closest to the siding junction
- the top car on the siding may be moved to the rear of the assembled area
- no other moves are possible, i. e. when a car is placed in the assembled area it cannot be moved back to the siding or the waiting area

The user's goal is to assemble a train from the given waiting cars so that all cars with the same destination number are adjacent to one another and the cars of the assembled train are in order of destination number. The smallest destination number belongs at the front of the train, the largest destination number belongs at the rear of the train. At the start of the program, waiting cars are read from a user specified file. At the end of the program, assembled trains are saved to a user specified file.

Design Requirements:

- Rail cars are to be modeled by a class that includes the car number and destination number as data members, plus constructors and methods to set and retrieve these values. The class must include overloading methods on all relational operators and the assignment operator. Additionally, friend functions to overload the input stream operator (>>) and the output stream operator (<<) must be included. Each rail car is to be represented by an object of a that class.
- Waiting cars are to be stored in an object of class *linkedListType*. Cars are inserted at the end and removed from the front. Cars returned to this list from the siding are to be inserted at the front
- Cars moved to the siding are to be stored in an object of class *linkedStackType*. Cars are pushed onto the top and popped from the top of the siding stack.
- Cars added to an assembled train are stored in an object of class *linkedQueueType*. Cars are added to the train queue at the rear. Once a car is added to the train queue, it cannot be returned to the siding or the waiting area.

Example Run of Program:

An Example typescript of a run is available at the following URL:

http://fidelio.cacs.louisiana.edu/260/projects/pa5_script.txt

Note: Your program does not have to look like this when run. You are free to come up with your own interface and representation of the switching yard components.

Hint: In order to display the list, queue and stack, your code may have to copy each of these objects into duplicate objects in order to avoid damaging or losing the data contained.

Files of Waiting Cars:

At the start of the program, cars are loaded from a file and placed in the waiting area. There are three example input files of waiting cars for testing purposes. The data in each of these files is in the following order. The data on the first and second lines belongs to the first car in the file, the data on the third and fourth lines belongs to the second car in the file, etc.

integer-car-number
integer-destination-number
integer-car-number
integer-destination-number
integer-car-number
integer-destination-number
...
integer-car-number
integer-destination-number

The test files are *train1.txt*, *train2.txt* and *train3.txt* and are located on the UCS network at */w1/cs260*x*/trains/*, where *x* is your section number.

Abstract Data Type Implementation Files

The following files are located at */w1/cs260*x*/linked/*, where *x* is your section number.

linkedNode.h	// template struct nodeType
linkedList.h	// template class linkedListType
linkedList.cpp	// template class linkedListType
linkedStack.h	// template class linkedStackType
linkedStack.cpp	// template class linkedStackType
linkedQueue.h	// template class linkedQueueType
linkedQueue.cpp	// template class linkedQueueType

Additional Requirements:

- You are responsible for following the requirements as given in documents reachable from the class web site via the **Minimum Documentation** and **Naming Conventions** links. Methods in your class definitions must has accompanying pre-condition and post-condition documentation. (See the link **Additional Documentation** on the class web site.)

- A makefile is to be created for and submitted with this assignment.

- Each class design is to be placed in a file named for the class and ending in ".h" (a header file). Each class implementation is to be placed in a file named the same as the file of the design, but ending in ".cpp". Each header file is to include wrappers to prevent unnecessary recompilation

- Your program may have global constants but may not have global variables.

- Your program solution must use good programing style. For example, call a function load the value in the data file into the array of account objects at the start of the program, then call a function to at the end of the program to write the values in the array elements back to the file. In between, call another function to handle the logic necessary to maintain communicate with the user and maintain the accounts. To rephrase, try to make your main function an outline of the logic of the program solution, one that uses calls to functions and object methods to direct the order of the operations rather than trying to write all operations in the main.

Submitting:

You are responsible for submitting your work both electronically and via a hard copy.

- Instructions for submitting an electronic copy of your project can be found on the class web site, via the **Submitting Your Work** link. When prompted, name your project "proj5".

- A hard copy of the code in your project and a hard copy of a typescript of a run of your program is due in your instructor's mail box (in Conference Center room 400) or office by close of business on Monday, May 2, 2005. Place the print out of your code in a manila folder, write your name, clid and section on the folder.

  "script" is a utility on Unix and Linux system that creates a text file of all commands, programming output and user input while "script" is running. A typical script session is as follows:

  1. run the script command: script
  2. compile your code
  3. run your code, demonstrating all features
  4. exit the script: Ctrl-d
  5. a file named "typescript" has been created, print it