Project 5

Programming and Algorithms II, CSCI 211

Objectives

- Implement a Priority Queue
- Practice using a struct
- Practice command line arguments
- Practice using file streams
- Practice using assert()
- Instantiate multiple priority queue objects

Tip: You can use the Pqueue and Cust classes you created for Lab 8 as a starting point for Project 5. If you haven't completed Lab 8 yet, you should do that before starting the implementation for this project.

Overview

Write a program to simulate the daily operations of a Kwik-E-Mart.

Background

You will simulate the operation of a convivence store set in the world of The Simpsons television show (you don't really need to know anything about this show to complete this project). During the simulation, customers will enter the store, collect some items, get in line at a check stand, and then either buy the items, or steal them and rob the checker. There are two types of customers, those that buy things (shoppers), and those that steal things and money (robbers).

Your program will:

1. Read in a list of customers (customer name, "shopper"|"robber", arrival time, number of items to buy/steal) from stdin.
2. Run a simulation based on the sequence of customers you read in step 1.

Output will be generated to a specified file (not stdout) while the simulation is running.

The input is not ordered by arrival time, you will use priority queues to manage the ordering of events in the simulation.

Customers will enter the store at a specified arrival time, spend time shopping for their items, get in line to pay/steal (these simulated robbers are very polite, they wait in line to rob the store), spend some time paying or stealing money, and then leave the store. When a customer performs an action (arrives in store, gets in checkout line, starts paying/stealing, finishes paying/stealing) your program will write a message to the specified output file documenting the customer's action and the current time.

<continued on next page…>

Program Requirements

Implementation

Your program must take four command line arguments: **number of checkers** (a positive integer), **length of checker's break after a robbery** (a positive integer), **name of input file** (a C string), and **name of output file (**a C string). Note: You won't read from stdin or write to stdout for Project 5.

Your program must validate that the number of checkers is 1 or greater, the checker's break length is 0 or greater, and that the input file and output files can be opened. If there is an error with a command line argument, write to standard error (cerr) the appropriate message (from the list given below) and exit the program with an exit status of 1. Exit the program after finding a single error (don't look for all possible errors, stop after the very first one):

Check for errors in the following order:

- Error: invalid number of command line arguments.
- Error: could not open input file <filename>.
- Error: could not open output file <filename>.
- Error: invalid number of checkers specified. Error: invalid checker break duration specified.

Time will be represented by a single integer starting at 1. This is just a relative time unit and not meant to represent any particular time or length of time. Think of these as "clock ticks since simulation start". You will use a loop and an integer **clock** variable to keep track of the simulation clock.

The input file will be arbitrarily ordered with respect to time and will have the format:
<name> <"shopper"|"robber"> <arrival time> <number of items to purchase/steal>

Each set of fields will be on a separate line.

For example:

Homer shopper 3 9
Bart shopper 5 23
Lisa shopper 2 12
Maggie robber 27 1

Names will be a single string token without any spaces. Type of customer will always be either "shopper" or "robber". Arrival time will be an integer greater than 0. Number of items will be an integer greater than 0.

Output

A message must be written to the specified output file every time a customer:

1. enters the store
2. finishes shopping
3. starts checkout/stealing
4. finishes checkout/stealing

An example of the expected output is shown below. The order of events must be ordered according to the clock. This is the output for the sample input given above, executed with **2 checkers and a break time of 0**. Note that robbers "stole" "from" and shoppers "paid" "to." Also notice that item is sometimes plural (items) and sometimes singular (item):

```
2: Lisa entered store
3: Homer entered store
5: Bart entered store
21: Homer done shopping
21: Homer started checkout with checker 0
26: Lisa done shopping
26: Lisa started checkout with checker 1
27: Maggie entered store
29: Maggie done shopping
30: Homer paid $27 for 9 items to checker 0
30: Maggie started checkout with checker 0
37: Maggie stole $277 and 1 item from checker 0
38: Lisa paid $36 for 12 items to checker 1
51: Bart done shopping
51: Bart started checkout with checker 0
74: Bart paid $69 for 23 items to checker 0
registers[0] = $69
registers[1] = $286
time = 75
```

The entered/done shopping/started/done checkout messages must be printed in member functions of class Cust. Create a print function for each type of message and pass the **ostream** and **clock** variable (and sometimes other information). For example, the following function prints the "entered store" message:

```
void Cust:: print_entered(ostream &os, int clock) {
   assert(clock == m_arrival_time );
   os << clock << ": " << m_name << " entered store" << endl;
}
```

The above assert is an excellent example of using asserts. The only situation that could result in a customer entering the store at a time different than his arrival time is if there is a logic flaw in the program.

Tip: This assert will point out a logic flaw which otherwise might be very difficult to discover.

After the main loop is over (at the end of the simulation), print the amount of money in each register (see above example for format, the register is part of the Checker struct) and print the current time –– the value of the clock (see above example for the format). This time should be one greater than the time the last customer left the store. If the input file is empty, and there are no errors, the clock should be 1.

Other Requirements

Use one queue for all the checkers. When taking a customer off of the checker queue, assign the customer to the empty available checker with the lowest number (when a checker is on break he is not available).

All items in the Kwik-E-Mart cost **$3**. All checkers start the simulation with **$250** in their register. When a customer buys items from a checker, increment the checker's balance. When a checker is robbed, she gives all her money to the robber.

Customers must shop for exactly 2 clock ticks for each item they buy/steal. For example, if a customer arrives at time 10, and buys 15 items, the customer does not start the checkout / steal process until time = 40, that is, (10 + 15*2). **Shopping time is dependent on the number of items (even for robbers).**

During the checkout process, **shoppers** must spend **1 clock tick for each item they are buying**. For example, if a shopper is removed from the checker queue and assigned a checker at clock tick 28, and they are buying 7 items, they won't be done until clock tick = 35, that is (28 + (1 * 7)). **Checkout time is dependent on the number of items.**

Robbers spend 7 clock ticks for the checkout process. For example, if a robber is removed from the checker queue and assigned a checker at time = 40 and they are stealing 17 items, they won't be done until time = 47, that is, (40 + 7). **Robbing time is not dependent on the number of items**.

If two (or more) customers arrive at the same time, put them in the arrival queue **in the same order they appear in the input file**. For example, if Homer arrives at clock tick 10 and Lisa arrives at clock tick 10, if Homer is before Lisa in the input file, Homer should be in front of Lisa in the arrival queue.

Getting robbed is very stressful. Thus the checkers at the Kwick-E-Mart take a break after each time they are robbed. The length of the break (break duration) is a command line argument.

The Pqueue class is a priority queue of Cust pointers. It must only contain functions related to a priority queue. While you may include a function like

```
int Pqueue::getPriorityOfFirstElement()
```

do not include any code in Pqueue that has to do with the simulation. Class Pqueue must be implemented using a linked-list.

Required classes:

Class Cust: Stores information about a customer. Start with your Cust class from Lab 8.

Class Pqueue: Priority queue of Cust * (clock tick / time will be used as the priority). Start with your Cust class from Lab 8.

struct Checker:

- (int) money in the register (integer)
- (int) done_time (clock-tick) a checker is done serving the current customer OR time checker's break ends
- (Cust*) a pointer to a Cust (if currently checking out a customer, the Cust pointer will point to that customer, if not serving a customer it will point to NULL)

  <continued on next page…>

Plan of Attack

You will use three priority queues (an **arrival** queue, a **shopping** queue and a **checkout** queue), and an array of Checker **structs** to run the simulation.

First you will read the input file contents, create Cust objects, and enqueue them onto the arrival queue, using the arrival time as the priority.

Each customer in the input contains the string “shopper” or the string “robber”. Use a boolean value (bool) to store this state. Consider the following call to the Cust constructor. It converts the string role string into a boolean that is true if role string == "robber".

Perform all the input in a helper function called from main(). Read the customers one at a time, create a Cust object for each new customer, and then insert each Cust onto a priority queue ordered by their arrival time. Use the arrival_time as the priority for each customer in the arrival queue.

As you read in each customer from the specified input file, create a corresponding Cust object using code like this:

```
new Cust(name, (role_string == "robber" ? 1 : 0), arrival_time, num_items);
```

Tip: The conditional operator ?: is also known as the ternary operator, because it is the only operator in C++ that takes three operands. It makes it possible to write an if / else conditional as an expression.

(conditional expression) ? (if true do this) : (if false do that)

The file sim.cpp will contain the functions main() and run_simulation(). You may add any other helper functions as needed.

Once all the customers have been read and inserted into the arrival queue, call run_simulation(). You will need to pass to run_simulation() the arrival queue, the number of checkers, the checker’s break time, and an ofstream (to be used for output: all non-error output is written to the file given on the command line).

Implement the checkers as an array of Checker structs. Allocate the checker array dynamically, using malloc:

```
void run_simulation(Pqueue &arrival_queue , int num_checkers , int break_duration , ostream
&os ) {
    // create an array of Checker structs
    Checker *checker_array = (Checker*) malloc(...);
    // now use a loop to initialize all elements of the Checker structures ...
}
```

The body of the simulation should proceed as follows:

initialize num_customers to equal the number of customers in the arrival queue.

Use a for loop like this one to control the simulation clock and take the appropriate actions during each clock tick:

```
for (clock = 1; num_customers > 0; clock++) {
```

For each customer on the arrival queue with a priority equal to the current clock tick:

- dequeue them from the arrival queue
- write the appropriate message to the specified output file
- calculate what time this customer will be done shopping
- place this customer on the shopping queue using **the time they will be done shopping** as the priority

For each customer on the shopping queue that is done shopping (has a priority equal to the current clock tick):

- dequeue them from the shopping queue
- write the appropriate message to the specified output file
- place this customer on the checker queue (use 0 as the priority for all customers)

For each customer in the checker array with a done_time equal to the current clock tick:

- increment/decrement that checker's total cash
- write the appropriate message ("paid" or "stole" message) to the specified output file
- decrement the number of customers
- if the customer was a robber, update this checker so they will end their break at the correct time
- delete this customer
- set the checker's customer pointer to NULL (indicates it has no customer)

While there is an available checker and there is a customer on the checker queue:

- remove the customer from the checker queue and assign them to an available checker. Always pick the available checker w/the lowest checker array index).
- calculate the time the customer will be done checking-out/stealing (store this as done_time for this checker).
- write the "start checkout" message to the specified output file

Tip: If you don't follow the order as given above, your output may not be in the correct order!

<continued on next page…>

General Requirements

1. Free all dynamically allocated (heap) memory prior to program exit.
2. No function or method contains more than 30 to 40 lines, including white-space and comments.
3. Avoid code duplication.
4. Use a **single** return statement for all functions & methods that return a value.
5. Follow these variable naming rules: Local variables are all lowercase, with underbars between words, e.g. my_array. Class member variables begin with m_ and are all lowercase, with underbars between words, e.g. m_count. Class names and Struct names are "camel case", starting with an uppercase letter, e.g. IntStack. Constants are ALL CAPS.
6. Use descriptive names for all classes, structs, functions, methods, parameters, variables and types.
7. Do not declare any non-constant **global** variables.
8. Do not submit commented-out code.
9. Do not submit unused / un-called code.
10. Code blocks are properly aligned.
11. No more than three submissions to TurnIn (use run_tests to validate locally). Ask for help if you are having trouble with your TurnIn submissions (e.g. all local tests are passing but not all TurnIn tests are not passing).
12. Use exception handling (try/catch/throw) for all error handling specified above (those cases where you are to write to stderr and exit the program). This should actually make your life easier.

The first lines of all your files (both .h and .cpp) must contain the following comments:

```
// filename
// last name, first name
// ecst_username
```

Comment your program. You should have block comments that explain each function & method. You should also have in-line comments for each line or short code block.

Submission

Once all local tests are passing, submit the required source files for Project 5 to https://turnin.ecst.csuchico.edu

sim.cpp pqueue.h pqueue.cpp cust.h cust.cpp

Each student will complete and submit this assignment individually. BE VERY CAREFUL ABOUT DOING YOUR OWN WORK.

Watch the calendar: Know the due date and cut-off date.