Homework 7

CS 173: Intermediate Computer Science — Spring 2014
Instructor: Thomas Bressoud — Due: 2014-04-02

- In this assignment you will practice classes in C++.
- There will be no "main" application program for this assignment. Instead, we will use GoogleTest to build unit tests in C++ analogous to what we did in Python.
- This assignment will also require students to work in teams. Each team of two will be randomly selected (from the same section) and, although both team members are responsible for the entirety of the code submitted and its correct operation, team members will get to practice specializing in the different roles of "Test Engineer," and "Head Programmer" for each the various classes developed.
- In this assignment, even the test functions for the test suite should have good function documentation explaining the purpose and expected outcome of the test. Class functions should be documented per ongoing standards of pre- and post-conditions and inline comments.

The goal of this assignment is to build many of the pieces required for a C++ version of the discrete event simulation (DES). The ultimate goal will go beyond our Python implementation, but for now the foundation will consist of four classes: a Customer class/ADT to represent customer entities in the DES; an Event class/ADT that will be directly comparable to our Event superclass in Python; and two container classes – an OrderedSet (of Events) and a Queue (of Customers).
Both container classes will be implemented using a dynamically allocated array of the appropriate object type, created/allocated at class construction and dynamically deallocated at class destruction. The constructor will include an optional parameter specifying the maximum number of elements the container can hold.
Working in a team: Say that a team consists of Alice and Bob. The responsibility of the team members would be as follows: Alice would be Head Programmer for the Customer Class and the Queue of Customer class. Bob would be Head Programmer for the Event class and the OrderedSet of Event class. Alice would be Test Engineer for the Event class and the OrderedSet of Event class, and Bob would be Test Engineer for the Customer class and the Queue of Customer class. The Test Engineer should design and implement a rigorous battery of tests for each class they are responsible for. The set of tests should use knowledge of the class definition as well as knowledge of the implementation strategy in order to design tests that cover many possible points of failure. Part of the grade will be based on the number and quality of the test suite and its ability to have individual tests that target *distinct outcomes*. The team members should start by working together to agree that each header file (`Event.h, Customer.h, Queue.h, OrderedSet.h`) satisfies the specification given below.

Event Class

The Event class, implemented in file `Event.cpp` and class definition in `Event.h` should maintain a `double` for the *time* of the event and an integer `id` for the Event, and should support the following member functions:

- `Event(int id=0, double time=-1.0)`: constructs an `Event` object
- `std::string str() const`: return a C++ `string` object of the form `<Event n: value>`, where n is the id, and *value* is the time.
- `double time() const`: returns the time value from the object
- `int id() const`: returns the id from the object
- Operator overloads for all 6 relational operators (`<, >, <=, >=, ==, !=`): functions that return a `bool`, are `const` functions, and whose parameter is `const Event & other`
- `setTime(double time)`: update the data member of the time value of the event
- `void execute()`: function simply outputs the string version of itself to `std::cout`

Note that ids are only unique if the creator of Events makes them unique.

OrderedSet Class

The OrderedSet class, implemented in file `OrderedSet.cpp` and class definition in `OrderedSet.h` should maintain a collection of Events maintained in time order. The collection should be implemented by, at construction time, dynamically allocating an array of Event objects whose maximum size is given at construction time and defaults to 8. The "front" of the OrderedSet should be at index 0. The data members will include a pointer to the dynamically allocated array as well as an integer size, maintaining the number of Event elements valid in the OrderedSet, and an integer for recording the maximum size of the collection. Following our earlier specification, Events should be unique based on their given id at Event construction time.

- `OrderedSet(int maxsize=8)`: constructs an `OrderedSet` object
- `int insert(Event & x)`: inserts an Event element `x` into the ordered set. The position of the insert is based on using relational operators on the elements of the ordered set to find the position such that any subsequent elements are $>$ the inserted element. Thus, if there are elements that compare equal to the inserted element, the inserted element will be the last of the equal elements. If the object referred to by `x` is already in the collection (as determined by the Event id), it should be repositioned to its proper ordered position, and should not be present in the collection more than once. Return 1 if insert is successful and 0 if insert failed

because of no additional room in the OrderedSet. Note that, because of the array implementation, an insert is going to involve copying all later events to the next slot (starting from the end) in order to allow room for the inserted element.

- `Event removeFirst()` removes and returns the first element from the ordered set `s`. An integer value is "thrown" if `removeFirst()` is invoked on an empty `OrderedSet`. For now the particular integer value is not important. On a successful remove, all subsequent elements in the array implementation will move down to the next earlier index slot.
- `int remove(int x)` removes Event element with id `x` from the ordered set. A zero should be returned if `x` is not present in the ordered set, and 1 if the remove was successful.
- `int len()` returns the number of elements in the ordered set.
- `std::string str()` returns a string representation of the ordered set `s`. The string should begin with '[', the first element e, if it exists, should be the str(e) immediately followed by '*', and remaining elements should be comma-space separated string versions of each element. The string should terminate with ']'.

Customer Class

The Customer class, implemented in file `Customer.cpp` and class definition in `Customer.h` should maintain a `double` for the *time* of the customer's arrival and a C++ `std::string` for the Customer, and should support the following member functions:

- `Customer(double time=0.0, std::string label="")`: constructs an `Customer` object
- `std::string str() const`: return a C++ `string` object of the form `<Customer` l`:` *value*`>`, where l is the label, and *value* is the time.
- `double time() const`: returns the time value from the object
- `std::string label() const`: returns the label from the object
- `void setTime(double time)`: updates the arrival time of the customer

Queue Class

The Queue class, implemented in file `Queue.cpp` and class definition in `Queue.h` should maintain a collection of Customers. By the property of queues, Customer objects should be added at one end of the sequence by an enqueue operation, and removed from the other end of the sequence by a dequeue operation. The collection should be implemented by, at construction time, dynamically allocating an array of Customer objects whose

maximum size is given at construction time and defaults to 12. To make implementation of the enqueue and dequeue operations efficient, you will use a technique known as a "circular array", wherein both ends of the queue are maintained as integer indices into the array, and these integer indices progress from 0 up through the last index and then back to 0 as enqueue and dequeue operations are performed. This technique is detailed in your book in section 5.4

- `Queue(int maxsize=12)`: constructs an `Queue` object
- `int enqueue(Customer & x)`: inserts an Customer element `x` into the queue at the tail of the queue and advances the tail. Return 1 if enqueue is successful and 0 if enqueue failed because of no additional room in the Queue.
- `Customer dequeue()` removes and returns the customer element from the head of the queue. An integer value is "thrown" if `dequeue()` is invoked on an empty `Queue`. For now the particular integer value is not important.
- `int len()` returns the number of elements in the queue.
- `std::string str()` returns a string representation of the queue. The string should begin with '[', the front/head element c, if it exists, should be the c.str() immediately followed by '*', and remaining elements should be comma-space separated string versions of each element. The string should terminate with ']'.

In testing, when a GoogleTest test function expects an exception to be thrown, the `EXPECT_THROW()` macro checks for correct operation. In particular, say that a Queue named `Q` was empty, and we attempt a `dequeue` operation. This could be accomplished in a test function with the following:

```
Q = Queue();
EXPECT_THROW(Q.dequeue(), int);
```

Your submission should contain

- `Event.h`
- `Event.cpp`
- `testEvent.cpp`
- `Customer.h`
- `Customer.cpp`
- `testCustomer.cpp`
- `Queue.h`
- `Queue.cpp`
- `testQueue.cpp`
- `OrderedSet.h`
- `OrderedSet.cpp`

- `testOrderedSet.cpp`
- `Makefile`, a makefile that separately compiles all source files and creates four executebles: `testEvent, testCustomer, testQueue, testOrderedSet`.