ECE 2574 ♦ Supplementary Notes for HW4

Contents:

1. STL

The Standard Template Library (STL) is a C++ software library which heavily influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functional, and iterators.
The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.
The STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

2. STL Container `list`

One way to implement the ADT for the *event list* of your simulator is to use the STL's `list` container. The STL's list container is implemented as a doubly linked list. You might wonder why there are both list and vector containers in the STL—the reason is that the underlying representations are different, and each representation has its own costs and benefits. The vector has relatively costly insertions into the middle of the vector, but fast random access, whereas the list allows cheap insertions, but slow access (because the list has to be traversed to reach any item). Some algorithms, such as merge sort, even have different requirements when applied to lists instead of vectors. For instance, merge sort on vectors requires a scratch vector, whereas merge sort on lists does not. Using the STL list class is about as simple as the STL vector container. In fact, to declare a list, all you need is to give the type of data to be stored in the list.

For instance, the following code declares a list storing integers:

```
std::list<int> integer_list;
```

Like the vector class, the list class includes the push_back and push_front functions, which add new elements to the front or back of the list respectively.

For instance,

```
std::list<int> integer_list;

integer_list.push_front(1);
integer_list.push_front(2);
```

creates a list with the element 2 followed by the element 1.

Correspondingly, it is possible to remove the front or back element using the pop_front() or pop_back() functions. Using these functions alone, it would be possible to create a queue or stack data structure based on the list class.

What about adding elements into the middle of the list—that is, after all, one of the advantages of a list. The insert function can be used to do so: insert requires an iterator pointing to the position into which the element should be inserted (the new element will be inserted right before the element currently being pointed to will).

The insert function looks like this:
```
iterator insert(iterator position, const T& element_to_insert);
```

Fortunately, the list container supports both the begin (returning an iterator to the beginning of the list) and end (returning an iterator past the last element of the list) iterator functions, and you can declare iterators as with any other container, in the following manner:
```
list<type>::iterator iterator_name;
```

Note that the STL list container supports iterating both forward and in reverse (because it is implemented as a doubly linked list).

Using insert and the function end, the functionality of push_back, which adds an element to the end of the list, could also be implemented as
```
std::list<int> integer_list;
integer_list.insert(integer_list.end(), item);
```

The list class also includes the standard functions size and empty. There is one caveat to be aware of: the size function may take O(n) time, so if you want to do something simple such as test for an empty list, use the empty function instead of checking for a size of zero. If you want to guarantee that the list is empty, you can always use the clear function.

```
std::list<int> integer_list;
\\ bad idea
if(integer_list.size() == 0)
    ...
\\ good idea
if(integer_list.empty())
    ...

integer_list.clear();
\\ guaranteed to be empty now
```

Lists can also be sorted using the sort function, which is guaranteed to take time on the order of O(n*log(n)). Note that the sort algorithm provided by the STL works only for random access iterators, which are not provided by the list container, which necessitates the sort member function:

```
instance_name.sort();
```

Lists can be reversed using
```
instance_name.reverse();
```

One feature of using the reverse member function instead of the STL algorithm reverse (to be discussed later) is that it will not affect the values that any other iterators in use are pointing to.

Another potentially useful list function is the member function called unique; unique converts a string of equal elements into a single element by removing all but the first element in the sequence. For instance, if you had a list consisting of
```
1 1 8 9 7 8 2 3 3
```

the calling unique would result in the following output:
```
1 8 9 7 8 2 3
```

Notice that there are still two 8's in the above example: only sequential equal elements are removed! Unique will take time proportional to the number of elements in the list.

If you want each element to show up once, and only once, you need to sort the list first! Try the following code to see how this works and see many of the previous functions in action!

```
std::list<int> int_list;
int_list.push_back(1);
int_list.push_back(1);
int_list.push_back(8);
int_list.push_back(9);
int_list.push_back(7);
int_list.push_back(8);
int_list.push_back(2);
int_list.push_back(3);
int_list.push_back(3);
int_list.sort();
int_list.unique();

for(std::list<int>::iterator list_iter = int_list.begin();
    list_iter != int_list.end(); list_iter++)
{
    std::cout<<*list_iter<<endl;
}
```

3. STL Container queue

The `queue` is a container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. Queues are implemented as containers adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements. Elements are pushed into the "back" of the specific container and popped from its "front".

The underlying container may be one of the standard container class template or some other specifically designed container class. The only requirement is that it supports the following operations:

- **`front()`**
- **`back()`**
- **`push_back()`**
- **`pop_front()`**

Therefore, the standard container class templates deque and list can be used. By default, if no container class is specified for a particular queue class, the standard container class template deque is used.

In their implementation in the C++ Standard Template Library, queues take two template parameters:

```
template < class T, class Container = deque<T> > class queue;
```

Where the template parameters have the following meanings:

- **`T`:** Type of the elements.
- **`Container`:** Type of the underlying container object used to store and access the elements.

In the reference for the queue member functions, these same names are assumed for the template parameters.

4. Event-Driven Simulation

Let us consider a queue at a bank. The queue, along with the teller, is referred to as a "system". The "variables" or “state variables” of a system are the quantities that represent the current state of a system. For this example, the queue length is a state variable of the system. Quantitatively, the "state" of the system at time t is represented by the specific values of the state variables of the system at time t. An "event" is any incident that would cause a change in the system's state. That is, an “event” is any incident that would change the state variables of the system. For example, the arrival of a customer is an "event". This event results in a change to the queue length, the number of customers in the system, etc. The departure of a customer is an event - its evaluation results in a reduction in queue length, etc.

A simulator maintains a set of variables in computer memory which represent the state variables of a system. For every event that occurs, the simulator changes the value of the variable representing "state", and thus allows us to examine how the "state" of the system changes over time.

Event-driven simulators update the current time to the time of occurrence of the next event, skipping the time-steps when no events occurred. Such simulators keep a list of events, sorted by time of occurrence, and then evaluate events in sorted order. An event is evaluated only once all events with a lower time of occurrence are evaluated. The evaluation of an event will result in an update to the “state” of the system, and could generate new events. An event driven simulation of our example bank queue system will update the variables representing the queue length, the number of people in the system, etc. every time an event is evaluated.

We don't generate all the events in the list at the beginning (this would be analogous to knowing the entire sequence of states of the simulation at the outset). Instead we initialize the simulation with certain events, and their times of occurrence. Certain events may be handled by scheduling later events, which are inserted at the appropriate place in the event list.

If events are not guaranteed to occur at regular intervals, and we don't have a good bound on the time step (it shouldn't be so small as to make the simulation run too long, nor so large as to make the number of events unmanageable), then it's more appropriate to use an event-driven simulation. A typical example might be simulating a lineup at a bank, where customers don't arrive at regular time intervals, and may be deterred by a long lineup.

In this project, you need to design your own class for simulating a line at a bank. The following code snippet of the header file, **`simulate.h`**, is provided as an example. Note that your code may look quite *different* from this example; this code is given only as an example.

```
#include <cmath>
#include <iomanip>
#include <fstream>
#include <iostream>
#include <queue> //STL queue header file
#include <list> //STL list header file

using namespace std;

//customer node in the queue that represents a line at the bank

struct customer
{
     int arrival_time;                //arrival time of a customer
     int trans_time;                  //transaction time required
};

//eventnode in the list that represents future events
struct eventnode
{
     int arrival_time;                //arrival time of customer
     int trans_time;                  // transaction time for customer
     int depart_time; //departure time, used only for departure events
     bool arrival;              // false if departure, true if arrival
};

//simulate class - used to represent the bank simulation
```

```
class simulate
{

public:

//queue used to represent the line at the bank - STL queue container
     queue <customer> line;

     //the list used to represent future events - STL list
     list <eventnode> event_list;

     //this list is used to store the final list of events - STL list
     list <eventnode> final_list;

     //a variable used to represent the current time during simulation
     double curtime;

     //a counter variable used to count the total number of customers
     double total_customers;

     //a variable used to record the max line length
     double max_line_length;

     //a variable used to record the total time spent by all customers
     double all_time;

//variable used to record total time spent waiting by all customers
     double all_time_wait;

     //variable used to record the total time elapsed
     double event_time;

     //variable used to record total number of customers who passed
     double line_amt_total
     //default constructor
     simulate();

     //destructor
     ~simulate();

     //this function adds an arrival event into the eventlist
     void add_arrival(int arrival_time, int trans_time);

     //function which updates the line and the event_list – places
//departure events into the event list, as well as many other
//functions
     void update();

//function used to calculate the time elapsed after eventlist is empty
     void count_events();

};
```

The code below gives you psuedocode for the `update()`method, which updates the current time, the `event_list`, the queue and the `final_list`:

`update()`

```
cur_event = event_list.front()

if cur_event is an arrival:
     Set current_time to cur_event.arrival_time

     Place a customer into line with the following assignments
          arrival_time = cur_event.arrival_time
          trans_time = cur_event.trans_time

     //Remove cur_event from event_list
     Pop the front element of the event_list

     Add cur_event to final_list

     //if line has only 1 person – the person that just arrived
     If the size of line is 1:
          cur_customer = line.front()

     Add a departure event at the correct position with the following
     assignments
          arrival_time = cur_customer.arrival_time
          trans_time = cur_customer.trans_time
          departure_time = current_time+trans_time


if cur_event is a departure:
     Set current_time to cur_event.departure_time
     Pop a customer from line

     Pop the front element of the event_listAdd cur_event to
     final_list

     If line size is greater than 0:  //line has at least 1 person
          cur_customer = line.front()

     Add a departure event at the correct position with the following
     assignments
          arrival_time = cur_customer.arrival_time
          trans_time = cur_customer.trans_time
          departure_time = current_time+trans_time
```

We add events such that `event_list` remains sorted. We add an arrival or departure event into `event_list` at a position such that all events after it have a greater arrival time (in case of arrival events) or departure time (in case of departure events). We observe that since the arrivals are in sorted order in the input file, and only 1 departure event is present in the event_list at a time, update() may be called till the number of pending events comes to 1. We use this fact when placing a new event correctly in the event_list - we perform a simple comparison of times of the new event and the event present on the event_list, and place the new event accordingly.

The code below gives you psuedocode for the `add_arrival()`method. This is one possible implementation of this method; your code may look very different.

```
add_arrival(int arrival_time, int trans_time)

while event_list.size > 1:
     update();

Create a new event_list node with parameters
     eventnode e = new eventnode;
     e.arrival_time = arrival_time;
     e.trans_time = trans_time;

Increment total_customers

if event_list is empty:
          event_list.push_front(e);

else if event_list is not empty:
          eventnode temp;
          temp = event_list.front();

          if (e.arrival_time < temp.depart_time):
               event_list.push_front(e);
          else if (e.arrival_time > temp.depart_time):
               event_list.push_back(e);
          else if (e.arrival_time == temp.depart_time):
               update();
               event_list.push_front(e);
```

The pseudo code for the main function in the client file (bank.cpp) is given below. This is one possible implementation; your code may look very different.

```
int main(int argc, char* argv[])

simulate bank;                //declaration of a simulate object
ifstream in( argv[1] );       //stream the declared input file

if (!in.is_open()) :
```

```
          cout << "The input file could not be opened." <<endl;
     else :
          ofstream out;
          out.open( argv[2] );
          while(in >> a_time >> t_time):

               //streams numbers in from the input file
               bank.add_arrival(a_time, t_time);
               //update the list after placing an arrival event
               bank.update();

          while (!bank.line.empty() || !bank.eventlist.empty()):
               bank.update();

          //the number of unique time instances are counted
          bank.count_events();

          //the average length of the line is calculatd
          avgline = bank.line_amt_total / bank.event_time;

          Print all events from final_list into the out ofstream
          Print all statistics in bank into the out ofstream
```

Reference links:

Material herein was obtained from the following sources:

1. http://en.wikipedia.org/wiki/Standard_Template_Library
2. http://www.cplusplus.com/reference/stl/
3. http://stackoverflow.com/questions/3599197/event-driven-simulation-with-objects
4. http://stdcxx.apache.org/doc/stdlibug/11-3.html