Combined Homework 9 & 10

CS 173: Intermediate Computer Science — Spring 2014
Instructor: Thomas Bressoud — Due: 2013-05-02

- In many ways, this is a capstone assignment. In this assignment you will
  - continue to practice dynamic memory classes and use of pointers by implementing a doubly linked list in C++,
  - use C++ templates to make your DLList class more abstract and usable for lists of differing underlying types,
  - integrate the template DLList class for use by the OrderedSet of Events and the Queue of Customers, and
  - design your own, more complex, discrete event simulation, modeling and analyzing a system with multiple queues, multiple generators, and multiple servers.
- The earlier guidelines for documenting and testing your code hold. Our attention to pre- and post- conditions waned somewhat in HW7 and HW8, and we want to keep good habits on this last assignment.
- You may choose to work individually on this assignment, or you may work in a team of two of your own selection.
- You should use Makefiles and your project should, even over the various parts, build the appropriate executables for all four of the phases of the project described below.

First phase: Class DLListInt

Implement a doubly linked list of integers in C++ (non-template). Use the following class declaration as the basis for your DLListInt.h file. The below example header file, also provided for you on submitbox, defines the supporting `IntNode` class as well as some other supporting structures. There are notes below, but the operation of most functions have already been discussed in class and will not be repeated here.

```
// Forward declarations and prototypes used in the classes below
class DLListInt;

std::ostream& operator<<(std::ostream& os, const DLListInt& l);

class IntNode {
  friend class DLListInt;
public:
  IntNode(int item, IntNode * next = NULL, IntNode * prev = NULL)
    : item_(item), next_(next), prev_(prev) {}
private:
  int item_;
  IntNode * next_;
  IntNode * prev_;
};

class DLListInt {
public:
  // Constructors

  DLListInt()
    : head(NULL), tail(NULL), nextiter(NULL), size(0) {}
  DLListInt(const DLListInt& from);

  // Destructor

  virtual ~DLListInt();

  // Member functions

  DLListInt& operator=(const DLListInt& from);
  int length() const { return size; }
  void append(const int& item);
  void insert(int index, const int& item);
  int pop(int index=-1);
  int& operator[](int index);
  std::string str();
  void resetForward() { nextiter = NULL; }
  int next();

private:
  IntNode * head;
  IntNode * tail;
  IntNode * nextiter;
  int size;

  void allocCopy(const DLListInt& from);
  void dealloc();
  IntNode * _find(int index);
  int _delete(int index);


  // Non-member friends
  friend std::ostream& operator<<(std::ostream& os, const DLListInt& l);
};

// Auxiliary classes for throwing "exceptions" on violated preconditions
class IndexError { };
class StopIteration { };
```

Notes

- The non-member-function overload of `operator<<` allows clients of the DLListInt class to use statements like:

  ```
  DLListInt L;
  ...
  cout << L << endl;
  ```

  Recall that, for a member function, C++ translates `cout << L` into `cout.operator<<(L)`, but this means the function for outputting a `DLListInt` would have to be a member of the `ostream` class (which we cannot modify). On the other hand, if a non-member function exists, the C++ translation becomes `operator<<(cout, L)`, where both the `ostream` object and the `DLListInt` object are explicit parameters, but it still allows us to overload and make printing of our class easier for clients.

- In the `.cpp` file, the `operator<<` function definition, because it is *not* a member function, will not have the `DLListInt::` scope resolution.

- The `IntNode` class is not providing a default constructor. In my implementation, it did not need one, based on Node creation in the DLListInt class. This allows a cleaner translation to a template class in the second phase of this assignment.

- As a one-liner, I give a definition for `resetForward`, so it will not need a corresponding definition in the `.cpp` file.

- As discussed in class, the `next()` operation should return the next *item* in the list, and advance the `nextter` pointer. After a `resetForward()`, it should move to the head of the list. If there is no next IntNode, it should throw an object constructed from the (empty) StopIteration class.

- Your implementation should check for violations of preconditions, particularly on a specified index not being valid, and should throw an object constructed from the (empty) IndexError class if the precondition is violated.

- All other methods have standard definitions as discussed in class and in the book. The public interface must have *at least* the functions/interface described here, but your private data and structures could be different than that proposed here.

- Your implementation should pass all 26 Google Test unit test functions in the provided *testDLListInt.cpp* program. This can be found as a submitbox attachment.

Second phase: Template Class DLList

Your next task is to convert the integer-item-specific non-template class to a template class. This is a case where I want you to work through the exercise of "converting" a specific class (that works) into its corresponding template class. Thus, I am not going to provide you with the interface or test files for this phase of the assignment. Some of the steps to take are outlined below:

- Create a new header file (`DLList.h`) from `DLListInt.h`, but keep the original. We discussed in class how to templatize each of the elements of both template functions and template class definitions. You will also add a `#include "DLList.hpp"`.
- Create your templatized `DLList.hpp` from your current `DLListInt.cpp` file, again following the methodology discussed in class.
- For testing, you can start with a copied version of `testDLListInt.cpp`, called `testDLList.cpp` and changing it to use your new template class, beginning with specialization of all DLLists of integers.
- Once the integer version of the template class is compiling and successfully runs all 26 tests, create an additional suite of tests *in the same test .cpp file* but with DLLists of doubles. This will help ensure that your template version is not making some assumption about the "integerness" of the item type.

Third phase: Template Queue, OrderedSet

With a template version of DLList, you can now use a DLList of arbitrary type `T` in a templated version of Queue, and a DLList of type `T` in a templated version of OrderedSet. Coding this can be all-at-once, or you can separate this phase into two steps. In the first step, you create a templated versions of Queue and OrderedSet that use the current implementation (array storage allocated from the heap), and then in the second step, you change the implementation to use a DLList of the same type as the templated Queue.

Note also that we are not employing inheritance here. The Queue "has-a" DLList; it is not a derived class inheriting from DLList. The end result class definitions for these two classes are given below. But I strongly encourage you to follow the two-step process so that, if problems arise, you are not in the situation of having changed multiple aspects of the implementation, as it is then much more difficult to isolate the problem. Note also that these are not complete .h files, nor should they be combined into a single .h file.

After the class definitions, I will give some guidance to help you in your development process.

```
template <typename T>
class Queue{
friend std::ostream& operator<<(std::ostream& os, const Queue& q)
       { os << q.str(); return os; }
public:
    Queue(int maxsize=12);
    ~Queue();
    int enqueue(T & x);
    T dequeue();
    int len();
    std::string str();
private:
    DLList<T> L;
    int maxsize_;
};

template <typename T>
class OrderedSet {
friend std::ostream& operator<<(std::ostream& os, const OrderedSet& s)
       { os << s.str(); return os; }
public:
    OrderedSet(int maxsize = 8);
    ~OrderedSet();
    int insert(T x);
    T removeFirst();
    int remove(int id);
    int len() const;
    std::string str();

private:
    DLList<T> L;
    int maxsize_;
};
```

Guidance

- In the above, both the `Queue` and `OrderedSet` class have overloaded stream insertion. By including the definition inline, we need not create a templated definition separately. In preparation for use by DLList with items of type Event pointer and type Customer, those classes should also support a non-member function overloading stream insertion.
- Note that the DLList is specialized in the private data, and is a simple data member, not a pointer. There is no need for dynamic memory allocation, as it is the Nodes that allow the structure to grow, and these *are* dynamically allocated.
- In both class definitions, to maintain interface compatibility with our earlier versions

and code written to that interface, there is still a `maxsize` constructor parameter and a corresponding data member. You have the choice of either continuing to respect the maxsize, or to ignore it, and to make the corresponding change to the test and not EXPECT_THROW when adding to a "full" structure.

- Note that the new `OrderedSet`, while using generic facilities of a DLList, must nonetheless make assumptions about the abilities of the elements contained in the OrderedSet, and thus the underlying DLList. In particular:
  - The elements of the OrderedSet must be pointers, as operations like `insert` use dereferencing as they compare an element to be inserted with the elements already in the OrderedSet.
  - We assume the objects pointed to allow relational comparisons.
  - We assume the objects pointed to support an `id()` method, and these id's can be used to determine whether or not an element is already in the OrderedSet.
- Be careful on the Makefile. There will *not* be targets associated with .hpp files, as the object code for template functions gets generated when the .cpp that specializes the class/function gets compiled. Also, the dependencies for the executable need to change, and drop the .o files that used to be the result of compiling the now-templatized class.
- I realize that folks will speed the process by using cut-and-paste and copying files to go from a non-template to a template version of a class. But don't forget to make sure the comments *are appropriate* and updated/accurate in every file you turn in.

Fourth phase: Extended Discrete Event Simulation

The ultimate goal of this capstone assignment is to incorporate what we have learned this semester into a non-trivial application, and to have that application be one of your own design. With the foundation from phases one, two, and three, it should be a straight forward exercise to modify your existing MM1 Queueing simulation from HW8 to use the new templated OrderedSet and Queue. This should be your first step in this phase, validating the results against what you obtained in HW8.
How you proceed from there depends on your target for an extended DES, and involves a more creative design process. You will need to determine what information is needed by the top level simulator. The top level simulator, in its setup (or equivalent) will have to create all of the elements involved in the simulation. When there are more components, different pieces may have to "know" about more than one other piece. This will potentially involve changing the parameters of constructors and/or adding other interfaces so that each piece can appropriately "communicate" with any "neighbor" elements. Design changes may also be needed to maintain and gather statistics appropriate to the new simulation.

As you are designing the new simulation, try to avoid the "I can fix this quickly by making a change *here*" development cycle. Think about what class is most appropriate for maintaining a piece of information. Try and keep things both simple and general, so that the same classes could be used in different models without requiring lots of customization.

Possible Extensions

For many of the models described below, in addition to a generator of entities, a queue, and a service, we need two additional concepts (at least logically – implementation may differ from student to student). First, we need a *splitter.* A splitter is an object with one "incoming" channel and two "outgoing" channels. It is a pass-through element (i.e. no waiting) and splits a fraction of the entities from the incoming channel to go out the first outgoing channel, and the remainder to go out the second outgoing channel. The second concept is that of a *sink.* The sink gives us a place to gather statistics and marks the exit of an entity from the system.

Feedback: In this extension, we have a model with a generator of entities going to a queue, the output of the queue going to a service, and the output of the service going to a splitter. One of the output channels of the splitter goes back to the queue, while the other output channel goes to the sink. This feedback scenario might model a visit to the DMV:

- A customer arrives and takes a number (i.e. goes into the queue).
- At the head of the line, service takes place, but it is not always successful. Sometimes, the customer finds they need something else to be able to complete their goal at the DMV.
- If unsuccessful, the customer resolves the problem, but has to take another number and go back in at the end of the queue. (We are making the simplifying assumption of instant resolution of problems.)
- If successful, the customer leaves the DMV.

Assembly Line: In this extension we have a single generator of entities going into a first queue, and from the first queue into a service. The output of the service then feeds another queue and another service. So in a four-stage assembly line, there would be one generator, four queues, four services, and one sink, all arranged in series. This model is sometimes called a "network of queues" and is an important problem in assembly line productions. If you pursue this model, be sure and both predict and experiment with the effects of a single slow server at different points in the line, and with ascending versus descending rates of service.

Multilevel Feedback: Consider the life of a programmer. We can see two clear stages – the coding stage, and the testing stage. Code specs are generated and go into a queue for the programmer. At the head of the queue, the programmer does their thing and we have one of two outcomes: the code has problems and gets resubmitted to this first queue for re-coding, or the programmer thinks it is good and passes it along to the queue for the testing stage. When code in the testing stage reaches the QA engineer, it is tested and again has two outcomes: the code passes the tests and goes to the sink, or the code does not pass the tests and goes back to the first queue for re-coding.

Quality Control: One could consider an extension to the Assembly Line model where, at the end of each stage, the service output is not guaranteed to go to the next queue in line, but rather goes to a splitter, and a fraction of the output of the service goes to a sink. Here we introduce multiple sinks to model these items rejected from the assembly line. In practice, this models the processes of quality control, where the outcomes of a stage of an assembly line may be determined to be faulty, and not proceed on.

Variations of the Checkout Problem: One could also envision multiple variations of our original Checkout Problem motivating our MM1 Queue:

- A system with two clerks (services), both operating off the same queue.
- A system with two clerks and two queues. So a generator would feed a splitter, and the splitter would feed the two queues associated with the two clerks. This splitter might do its job by fraction, or it could do its job by using some other metric (like queue length) for which output channel to select.
- Once we generalize to two clerks and two queues, it should be a reasonable extension to go to n queues and n clerks.