Programming Project One: Path Finding based on a Queue

Out: October 23, 2012; Due: November 6, 2012.

I. Motivation

1. To give you experience in implementing a queue abstract data type (ADT) using a circular array.
2. To solve a real problem using a queue.

II. Programming Assignment

You will first implement a templated `Queue` ADT using a circular array. Then, you will use the `Queue` to build an application of finding a shortest path in a three-dimensional grid.

1. The `Queue` ADT

The `Queue` is a templated class. It is built based on a circular array, which is dynamically allocated initially and reallocated with a large array if it cannot hold more elements. It supports the following operational methods:

```
isEmpty: Return true if the queue is empty; return false otherwise.

enqueue: Put an element at the rear of the queue.

dequeue: If the queue is not empty, remove the front of the queue and
return the removed element. Throw an exception if the queue is empty.

front: If the queue is not empty, return the front element of the
queue. Throw an exception if the queue is empty.

rear: If the queue is not empty, return the rear element of the queue.
Throw an exception if the queue is empty.
```

Note that while this queue is templated across the type `T`, it stores only **pointers-to-`T`**, not instances of `T`. This ensures that the `Queue` implementation knows that: it owns inserted objects,

it is responsible for copying them if the queue is copied, and it must destroy them if the queue is destroyed.

The complete interface of the `Queue` class is provided in the `queue.h`, which is available in the Assignments/Programming-Project-One folder on Sakai. The code is replicated here for your convenience.

```
#ifndef __QUEUE_H__
#define __QUEUE_H__

class emptyQueue
{
    // OVERVIEW: an exception class
};

template <class T>
class Queue
{
    // OVERVIEW: queue based on a circular array.
    // The queue stores pointers-to-T.

public:

    // Operational methods

    bool isEmpty() const;
    // EFFECTS: return true if the queue is empty, false otherwise

    void enqueue(T *op);
    // MODIFIES: this
    // EFFECTS: put op at the rear of the queue

    T *dequeue();
    // MODIFIES: this
    // EFFECTS: remove the front of the queue and return the removed
    //          element. Throw an instance of emptyQueue if empty.

    T *front();
    // EFFECTS: return the front element of the queue.
    //          Throw an instance of emptyQueue if empty.

    T *rear();
    // EFFECTS: return the rear element of the queue.
    //          Throw an instance of emptyQueue if empty.

    // Maintenance methods

    Queue(int cap = 1);                         // constructor
    Queue(const Queue &q);                      // copy constructor
    Queue &operator=(const Queue &q);           // assignment operator
```

```
    ~Queue();                                   // destructor

private:
    T **array;      // A dynamic array storing T*
    int capacity;   // The capacity of the dynamic array
    int count;      // The number of elements in the queue
    int ifront;     // The index of the first element in the queue
    int irear;      // The index of the last element in the queue

    // Utility methods

    void removeAll();
    // MODIFIES: this
    // EFFECTS: called by destructor/operator= to remove and destroy
    //          all the elements in the queue.

    void copyAll(const Queue &q);
    // MODIFIES: this
    // EFFECTS: called by copy constructor/operator= to copy elements
    //          from a source instance s to this instance.

    void grow();
    // MODIFIES: this
    // EFFECTS: called by the enqueue function to double the size of
    //          the array when the array is full. The pointers stored
    //          in the old array are copied to the new array.
};

#endif /* __QUEUE_H__ */
```

In addition to the five operational methods, there are the usual four maintenance methods: the default constructor, the copy constructor, the assignment operator, and the destructor. Be sure that your copy constructor and assignment operator do **full deep copies**, including making copies of `T`'s owned by the list.

Finally, the class defines three private utility methods `removeAll`, `copyAll`, and `grow`, which are called by the maintenance methods and the operational methods.

You must implement each `Queue` method in a file called `queue.C`. We will test your `Queue` implementation separately from the other components of this project, so it must work independently of the application described below.

To instantiate a `Queue` of pointers-to-`int`, for example, you would declare:

```
Queue<int> iqueue;
```

This instructs the compiler to instantiate a version of `Queue` that contains pointers-to-`int`.

To compile a program that uses `Queue`, you need to include both `queue.h` and `queue.C`. **To prevent multiple definitions of the class methods, you should use `#ifndef`, `#define`, `#endif` preprocessor directives in the `queue.C`. You do not need to compile the source file `queue.C`**, because you already include it in other codes where the `Queue` ADT is used.

2. Finding Path in a Three-Dimensional (3D) Grid

In class, we have shown you an algorithm which can find a path between two squares in a two-dimensional grid – the Lee's algorithm. It applies a queue to solve the problem. In this project, we will solve an extend version of this problem by applying a similar algorithm.

Suppose that we have a 3D grid. It has l levels, with each level as an $N \times N$ grid of squares. The level number increases from the bottom to the top, with the bottom level numbered as 0. Each square in the 3D grid falls into one of the following six types:

1. an open area (indicated by a blank space " ");
2. a wall (indicated by an "X");
3. a starting point (indicated by an "S");
4. a destination point (indicated by a "D");
5. a stairway up (indicated by a "^");
6. a stairway down (indicated by a "V").

You can travel north, south, east or west. If you are on a stairway up ("^"), you may also go up, besides travelling north, south, east or west. If you are on a stairway down ("V"), you may also go down, besides travelling north, south, east or west. Note that taking a stairway up leads directly to the same relative (x,y) location on the level above. Similarly, taking a stairway down leads directly to the same relative (x,y) location on the level below. This does not necessarily mean that a corresponding stairway of the opposite type exists at the same relative (x,y) location (although one could). That is, taking an up stairway from location (3,5) on level one always leads directly to location (3,5) on level two. However, there is not necessarily a corresponding down stairway at location (3,5) on level two even though there is an up stairway in location (3,5) on level one.

You may not travel diagonally in the 3D grid. You may not travel through walls. You may not travel outside of the 3D grid. **There is exactly one starting point in the grid.** However, there

may be multiple destination points in the grid. We assume that the distance between any two adjacent squares in the 3D grid is one. Your task is to indicate a shortest path from the starting point to a destination point using directional indicators: “n” for north, “e” for east, “s” for south, “w” for west, “u” for upstairs, and “d” for downstairs.

III. Input Format

The program will take a text file describing the 3D grid as input. The size of each level is specified on the first line as a single integer N representing an N by N grid. On the second line, the total number of levels in the 3D grid is specified as a single integer. After the first two lines, each level in the grid is described, starting at the top and **descending level by level** to the bottom. The description of each level consists of N lines. We refer to these lines as **map lines**. Any line beginning with a ‘#’ should be ignored; this is a mechanism for commenting 3D grid files. You should discard extra characters at the end of each map line (i.e., the characters which are at positions larger than N), extra lines beyond the last map line, and blank lines. The following is an example of a legal grid file:

```
5
2
#level 1
XXXXX
X   D
X XX
X   X
XX  X
#level 0
XXXXX
X
X XXS
X   X
XX ^X

# A comment
Also a comment
```

Note that all the characters in the grid file are **case sensitive**. Each grid line has at least N characters, including blank spaces.

IV. Output Format

All the outputs are printed through `cout`.

If there exists no path from the starting point to any of the destination points, print

```
No path found!
```

Otherwise, there exists at least one path from the starting point to a destination point. Then you should indicate the **shortest** path from the starting point to one of the destination points. You should first print the following two lines:

```
Path found!
The minimal distance is <min-distance>.
```

where `<min-distance>` should be replaced with the minimal distance your program obtains. Then you should print the 3D grid in a similar format as the input file but with path indicated. You first print the size N of each level and then the number of levels in the 3D grid. You should add the letters “n”, “e”, “s”, “w”, ‘u”, and “d” as needed to specify the path you have taken. This begins at the starting location. You should overwrite the squares in the 3D grid to indicate which square to move to next along the shortest path. You simply reprint the original 3D grid square if it is not part of the path. When reprinting the 3D grid, print comments in the form of

```
#level <i>
```

to indicate breaks between levels. You should not print any other comments.

An example to the above example input file is

```
Path found!
The minimal distance is 18.
5
2
#level 1
XXXXX
XeeeD
XnXX
XnwwX
XX nX
#level 0
XXXXX
Xswww
XsXXn
```

```
Xes X
XXeuX
```

Note that for a 3D grid, there may exist multiple optimal paths. Your program only needs to print one of them.

V. Program Arguments

Your program takes a single command-line argument, which is the name of the file describing the 3D grid. Let your compiled program be `p1`. It should be invoked as

```
./p1 <grid-file>
```

where `<grid-file>` gives the name of the file describing the 3D grid.

VI. Error Checking and Error Message

You only need to check for the following two errors:

1. Whether the user provides a grid file. If not, print the following error message and then exit:

```
Error: Missing arguments!
Usage: ./p1 <grid-file>
```

2. Whether file open is successful. If not, print the following error message and then exit:

```
Error: Cannot open file <filename>!
```

where `<filename>` should be replaced with the name of the file that fails to be opened.

You can assume that other than the above two errors, there are no errors. The format of the input grid file will be correct.

VII. Implementation Requirements and Restrictions

- You must use your `Queue` ADT to implement the path finding application. You may use any type you see fit as the templated type. However, remember that you only insert and remove **pointers-to-objects**. You must use the at-most-once invariant plus the Existence, Ownership, and Conservation rules when using your `Queue`.
- You must make sure that your code compiles successfully on a Linux operating system. We provide you with a sample `Makefile`. You should change it according to your source code files and submit it together with your source code files.
- **Your compiled program for finding the shortest path must be named as `p1` exactly.**
- You should name C++ source code files with the extension ".C" (**CAPITAL C**) instead of ".cpp".
- You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call `valgrind`, which can tell whether you have any memory leaks. (You need to install `valgrind` first if your system does not have this program.) The `Makefile` we supply contains a rule to call `valgrind` to check for memory leak. The rule looks like:

  ```
  memcheck: $(TARGETS)
     valgrind --leak-check=full ./$(TARGETS)
  ```

  You can check memory leak by typing "`make memcheck`" in your Linux terminal. However, you should change the command in the rule properly to reflect the actual program you are testing. For example, if you want to check whether running program

  ```
  ./p1 grid.txt
  ```

  causes memory leak, then you should change the command to

  ```
  valgrind --leak-check=full ./$(TARGETS) grid.txt
  ```

- You must fully implement the `Queue` ADT. Note that the implementation of the path finding application may not exercise all of the `Queue's` functionality, but this is fine.
- You may `#include <iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library, including standard template library (STL).
- Output should only be done where it is specified.
- You may assume that there is no error in the input file. Therefore, you don't need to check the error. This means you need not perform error checking. However, when testing your code in concert, you may use the `assert()` macro to program defensively.

VIII. Source Code Files and Compiling

There are one header file `queue.h` and one sample `Makefile` located in the programming-project-one-related-files.zip from our Sakai Resources.

You should copy these files into your working directory. **DO NOT modify `queue.h`!** However, you should modify `Makefile` to compile the program `p1` and to check memory leaks for different test cases.

You need to implement all the methods of `Queue` ADT in a file called `queue.C`. When implementing the path finding application, you can define your own source files.

In order to guarantee that the TAs compile and test your program successfully, you should make sure that the program compiled using your `Makefile` for finding the shortest path is named as `p1`. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason. We will test your implementation of the `Queue` methods by building a program from our `queue.h` (in order to make sure that you do not modify it), **your** `queue.C`, and our test file `test.C`. We will test your path finding program by building a program from our `queue.h`, **your** `queue.C`, and the other of **your** source code files using your `Makefile`.

IX. Testing

We provide you with a file called `test.C` in the programming-project-one-related-files.zip to help you test a few very basic behaviors of the `Queue` ADT. You should compile `test.C` together with our `Queue.h` and your `Queue.C`. If it compiles successfully, run the program. After running the program, you can look at the return value of the program to see if your implementation of the `Queue` ADT passes this test or not. If the return value is 0, it passes the test; otherwise it fails the test.

In Linux you can check the return value of a program by typing

```
echo $?
```

immediately after running the program.

We have also supplied an input file called `grid.txt` for you to test your path finding program. To do this test, type the following into the Linux terminal once your program has been compiled:

```
./p1 grid.txt
```

The length of the shortest path should be 18. A sample output is shown in Section IV. Note that for a 3D grid, there may exist multiple optimal paths. Your program only needs to print one of them. To program defensively, we also suggest you to write a routine to verify that the path indicated by your output is a valid path.

These are the minimal amount of tests you should run to check your program. Those programs that do not pass these tests are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak using `valgrind` as we discussed above. For those programs that behave correctly but have memory leaks, they only get half of the grade.

X. Submitting and Due Date

You should submit a `queue.C`, a `Makefile`, and the other related source files via the Assignment tool on Sakai. The `Makefile` compiles a program named `p1` for solving the path finding problem. The due date is 11:59 pm on Nov 6th, 2012.

XI. Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style
4. Performance

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. **For those programs that behave correctly but have memory leaks, they will only get half of the implementation points.** General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will

lead to General Style deductions. Part of your grade will also be determined by the performance of your algorithm. Algorithms that run slowly may receive only a portion of the performance points.