DALHOUSIE UNIVERSITY
DEPARTMENT OF ENGINEERING MATHEMATICS
ENGM2282

ASSIGNMENT # 5, Due date: Tuesday October 16, 2018, 1:00 PM

1. Use the `class stack` discussed in the lectures to make a `stack class template`. Write the methods of the template class in the following program which uses the string class as the template parameter.

   For marking purposes, run program and push `bob`, `joe` and `jean` onto the stack. Print the stack to the output file then pop twice and print again.

```
// File: stacktemplate.cpp
// This program implements a stack template class

#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

template <class T>
class stack{
private:
    int max;           // maximum size of the stack
    int count;         // number of items on the stack
    vector<T> data;    // vector holding the stack items

public:
    stack(int sz);                   // constructor, max = sz

    void push(const T &item);        // push a copy of item onto the stack
    T &pop(void);                    // pop an item off the stack
    void write(ostream &out) const;  // send the data stored to out
    bool empty(void) const;          // check for empty stack
    bool full(void) const;           // check for full stack
};


int main(void)
{
    stack<string> mystack(5);
    ofstream fout ("stackout.txt");
    char ch;
    string x;

    while(1) {
        // print a little menu
        cout << "\n\np = push \n";
        cout << "o = pop\n";
        cout << "s = print to screen\n";
        cout << "f = print to file\n";
        cout << "q = quit\n\n";
        cin >> ch;

        if (ch == 'p') {
            cout <<"\ndata to push :";
            cin >> x;
            if(mystack.full()) {
                cout << "Stack is full" << endl;
            } else {
                mystack.push(x);
            }
        } else if(ch == 'o') {
            if(mystack.empty()) {
                cout << "Stack is empty" << endl;
            } else {
                x = mystack.pop();
                cout << "\n\ndata popped : " << x;
            }

        } else if(ch == 's') {
            mystack.write(cout);
        }else if(ch == 'f') {
            mystack.write(fout);
        }else if(ch == 'q') {
            break;
        }
    }

    fout.close();
    return 0;
}
```

2. There is a disadvantage to the implementation of a queue using a vector object as done in the lectures. In that implementation, when there are `count` elements stored in the queue, they occupy positions 0 through `count-1` in the vector. The start of the queue is position 0 and the last entry in the queue is position `count-1`. The `get` method returns the value at poistion 0 and moves all entries down by 1 position. This shuffling of the vector elements makes the queue less efficient than a stack.

   A faster implementation is to use two positions: `start` and `end`. The entries in the queue are all vector entries between `start` and `end-1`. The term "between" is interpreted in the circular sense. That is, if the vector is of size `max` then we think of the next position after `max-1` to be position 0.

   With this interpretation, the `put` method places a new entry at position `end` and increments `end` while the `get` method returns the entry at position `start` and increments `start`. Increment is understood in the circular sense described above.

   When `start == end` the queue is empty but this also happens when the queue is full so we still need to keep track of the number of elements in the queue with the variable `count`.

   See `http://en.wikipedia.org/wiki/Circular_buffer` for a discussion of circular queue.

   Implement the methods of the class `queue` as described.

   For marking purposes, put the values 1, 2, 3, 4, print the queue, then get 3 times

and print the queue, followed by putting the values 5, 6, 7, print the queue again, get twice and print the queue again.

You only need to code the get, put and write methods.

```
// File: circularqueue.cpp
// This program implements a circular queue of integers

#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

class queue{
private:
    int max;            // maximum size of the queue
    int count;          // number of items in the queue
    int start;          // data[start] is the first entry in the queue
    int end;            // data[end-1] is the last entry in the queue
    vector<int> data;   // vector holding the queue of items

public:
    queue(int sz);                     // constructor, max = sz

    void put(int item);                // put the integer item onto the queue
    int get(void);                     // get an item at the head of the queue
    void write(ostream &out) const;    // send the data stored to out
    bool empty(void) const;            // check for empty queue
    bool full(void) const;             // check for full queue
};

int main(void)
{
    queue myqueue(5);
    ofstream fout ("queueout.txt");
    char ch;
    int x;

    while(1) {
        // print a little menu
        cout << "\n\np = put \n";
        cout << "g = get\n";
        cout << "s = print to screen\n";
        cout << "f = print to file\n";
        cout << "q = quit\n\n";

        cin >> ch;

        if (ch == 'p') {
            cout <<"\ndata to put :";
            cin >> x;
            if(myqueue.full()) {
                cout << "queue is full" << endl;
            } else {
                myqueue.put(x);
            }
        } else if(ch == 'g') {
            if(myqueue.empty()) {
                cout << "queue is empty" << endl;
            } else {
                x = myqueue.get();
                cout << "\n\ndata gotten : " << x;
            }

        } else if(ch == 's') {
            myqueue.write(cout);
        }else if(ch == 'f') {
            myqueue.write(fout);
        }else if(ch == 'q') {
            break;
        }
    }

   fout.close();
   return 0;
}

queue::queue(int sz) : data(sz)
{
    max = sz;
    count = 0;          // the queue has no entries yet
    start = 0;
    end = 0;
}

bool queue::empty(void) const
{
    return (count == 0);
}

bool queue::full(void) const
{
    return (count == max);
}
```

3. In the following program, `growingstack.cpp`, when the stack is full the `push` method enlarges the vector storing the stack by an amount using the `resize` method of `class vector`.

   You only need to write the push, pop and write methods. For debugging purposes, the push method should write the new stack size to `cout`.

   For marking purposes, run the completed program and capture your output.

```
// File: growingstack.cpp
// This program implements a simple stack of integers which grows as needed

#include <iostream>
#include <fstream>
#include <vector>
```

```
using namespace std;

class stack{
private:
    int max;           // maximum size of the stack
    int count;         // number of items on the stack
    int grow;          // data will grow by this amount when needed
    vector<int> data;  // vector holding the stack items

public:
    stack(int sz, int growsz);       // constructor, max = sz, grow = growsz

    void push(int item);             // push the integer item onto the stack
    int pop(void);                   // pop an item off the stack
    void write(ostream &out) const;  // send the data stored to out
    bool empty(void) const;          // check for empty stack
};

int main(void)
{
    stack s(5, 4);
    ofstream fout ("growingstackout.txt");

    for (int i=1; i <= 15; i++) {
        s.push(i);
    }

    cout << "The stack is:\n";
    s.write(cout);

    fout << "The stack is:\n";
    s.write(fout);

    fout.close();
    return 0;
}

stack::stack(int sz, int growsz) : data(sz)
{
    max = sz;
    grow = growsz;
    count = 0;           // the stack has no entries yet
}

bool stack::empty(void) const
{
    return (count == 0);
}
```