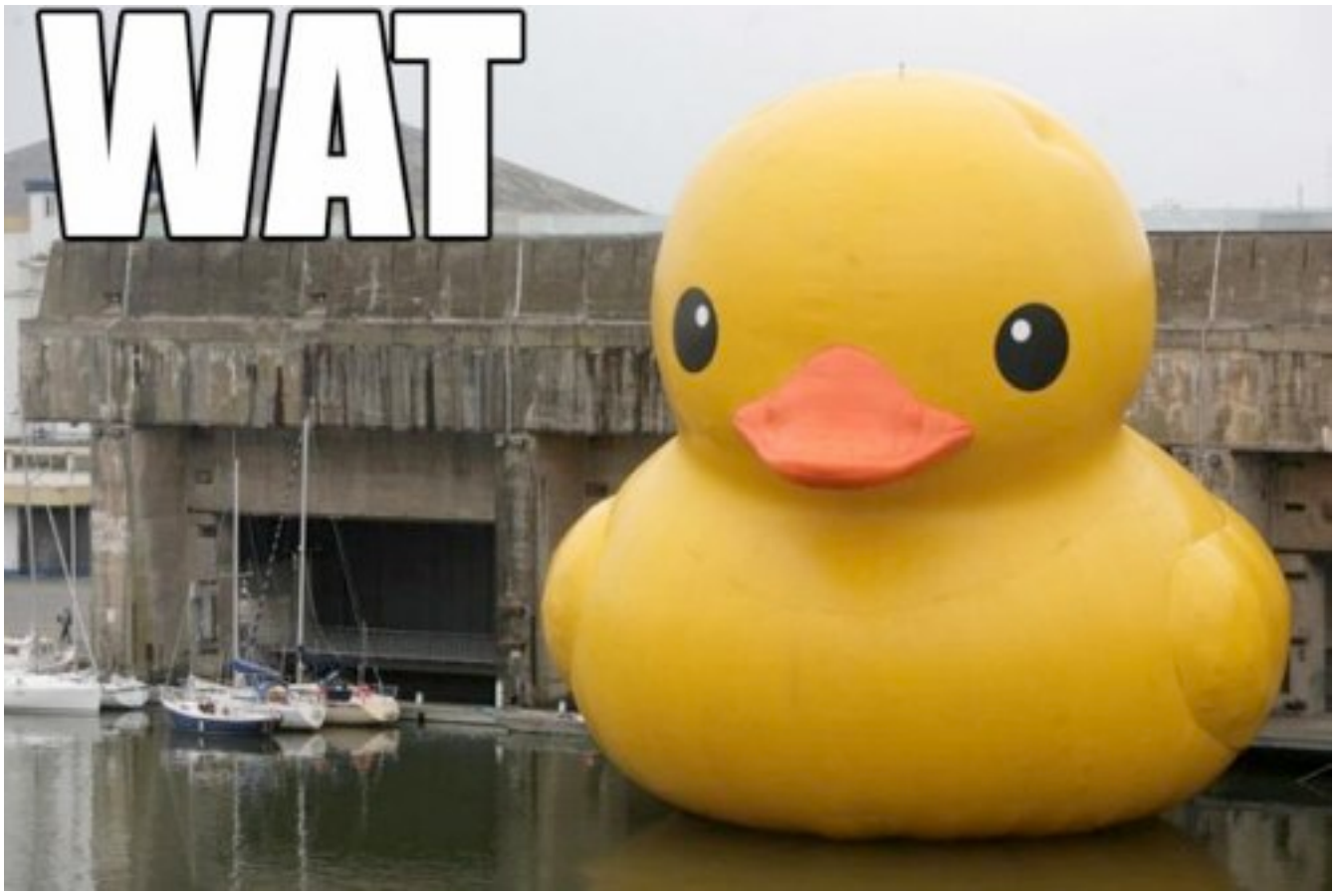
lab_quacks Spiteful Stacks, Questionable Queues

Due: Sunday, October 4 at 11:59 PM

Doxygen for lab_quacks



Assignment Description

In this lab you will learn to apply recursive thinking and use the stack and queue data structures. You might also practice templates.

Recursion

What is recursion? Recursion is a way of thinking about problems that allows the computer to do more of the heavy lifting for us. It is analogous to the mathematical definition of functions, where you can define a function call in terms of other function calls and basic arithmetic operations, but not in terms of loops.

Why recursion? While being able to think recursively is one of the harder parts of computer science, it is also one of the most powerful. In fact, there are whole languages that entirely use recursion instead of loops, which, even though it may seem inefficient, leads to some very useful optimizations a compiler can make when dealing with such code. There are probably more problems in computer science that are simpler recursively than they are iteratively (using loops). Also, once you have a recursive algorithm, it is always possible to transform it into an iterative algorithm using a stack and a while loop. In this way, computer scientists can think about

problems recursively, then use that recursive solution to make a fast iterative algorithm (and in the grand scheme of big-O notation, using recursion has little overhead compared to the rest of the running time). Here we'll only ask you to do the first part.

How do I write recursively? Recursion just means calling a function within that function's body. This may sound crazy, but in fact it is not. Let's take an iterative function to calculate the factorial of a number n, $n!$:

```C++
int factorial(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
    return result;
}
```

Okay, so four lines of code. Pretty short and understandable. Now let's look at a recursive version:

```C++
int factorial(int n)
{
    if (n == 0) return 1;
    return (factorial(n-1) * n);
}
```

Only two lines of code! (Depending on whether you like putting your return statement on the same line.) Even on such a small problem, recursion helps us express ourselves more concisely. This definition also fits better with the mathematical definition:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

A typical recursive function call consists of three parts. Let's examine the function more closely to see them. Here's the same code again, with more discussion.

```C++
int factorial(int n)
{
    if (n == 0)     // Here is our base case.
        return 1;   // The base case is the smallest problem we can thi
                    // one we know the answer to. This is the "n = 0" c
                    // the mathematical definition.

    // optional 'else' here

    return (factorial(n-1) // This is our recursive step. Here we are
                           //   smaller version of the same problem. W
                           //   make a leap of faith here - trust that
```

```
                        //   solution to the (n-1) case is correct.
                        //   the same as the mathematical definitio
                        //   to figure out n!, we need to first fig
                        //   (n-1)!
            * n         // Here is our incremental step. We are tra
                        //   our solution to the smaller problem in
                        //   solution to our larger problem. This i
                        //   same * n from the mathematical definit
        );
}
```

Checking out the code

To check out your files for this lab, run:

```
svn up
```
TERMINAL

from your `cs225` directory.

This will create a new folder in your working directory called `lab_quacks` and it will contain the 6 files detailed below. To test your function, compile and run `./quackfun`.

STL Stack and Queue

These activities use the standard template library's `stack` and `queue` structures. The interfaces of these abstract data types are slightly different than in lecture, so it will be helpful for you to look up "STL Stack" and "STL Queue" on Google (C++ reference has good information). In particular, note that the `pop()` operations do not return the element removed, and that you must look that up before calling `pop()`.

As usual, to see all the required functions, check out the Doxygen.

Recursive Exercises

No loops!

You may not use any loops for this section! Try to think about the problem recursively: in terms of a base case, a smaller problem, and an incremental step to transform the smaller problem to the current problem.

Sum of Digits

Given a non-negative `int n`, return the sum of its digits recursively (no loops). Note that modulo (`%`) by 10 yields the rightmost digit (`126 % 10 == 6`), while divide (`/`) by 10 removes the

rightmost digit (`126 / 10 == 12`).

```C++
int sumDigits(int n);
```

```
sumDigits(126) -> 1 + 2 + 6 -> 9
sumDigits(49)  -> 4 + 9     -> 13
sumDigits(12)  -> 1 + 2     -> 3
```

Triangle

We have triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on:

Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

```C++
int triangle(int rows);
```

```
triangle(0) -> 0
triangle(1) -> 1
triangle(2) -> 3
```

> **Note**
>
> These examples were stolen from http://codingbat.com/java/Recursion-1. All credit goes to CodingBat. If you are having a hard time with `sum` (below), we encourage you to go to CodingBat and try more recursive exercises. These are in Java, but there are links at the bottom of the page describing the differences of strings and arrays in Java from C++, which are minor.

The `sum` Function

Write a function called `sum` that takes one stack by reference, and returns the sum of all the elements in the stack, leaving the original stack in the same state (unchanged). You may modify the stack, as long as you restore it to its original values. You may use only two local variables of type `T` in your function. Note that this function is templatized on the stack's type, so stacks of objects overloading the addition operator (`operator+`) can be summed. Hint: think recursively!

STL Stack

We are using the Standard Template Library (STL) stack in this problem. Its `pop` function works a bit differently from the stack we built. Try searching for "STL stack" to learn how to use it.

```C++
template <typename T>
T QuackFun::sum(stack<T> & s);
```

Non Recursive Exercise

The `scramble` Function

Your task is to write a function called `scramble` that takes one argument: a reference to a `std::queue`.

```C++
template <typename T>
void QuackFun::scramble(queue<T> & q);
```

You may use whatever local variables you need. The function should reverse the order of SOME of the elements in the queue, and maintain the order of others, according to the following pattern:

- The first element stays on the front of the queue.
- Then the next two elements are reversed.
- Then the next three elements are placed on the queue in their original order.
- Then the next four elements are reversed.
- Then the next five elements are place on the queue in their original order.
- etc.

Hint: You'll want to make a local `stack` variable.

For example, given the following queue,

```
front                                                  back
0   1 2   3 4 5   6 7 8 9   10 11 12 13 14   15 16
```

we get the following result:

```
front                                                  back
0   2 1   3 4 5   9 8 7 6   10 11 12 13 14   16 15
```

Any "leftover" numbers should be handled as if their block was complete. (See the way 15 and 16 were treated in our example above.)

STL Queue

We are using the Standard Template Library (STL) `queue` in this problem. Its `pop` function works a bit differently from the queue we built. Try searching for "STL queue" to learn how to use it.

Good luck!

(Extra-Credit) The `verifySame` function

Complier errors

Submitting code that doesn't compile **will result in a zero** on the entire lab. This includes code in the extra credit portion and should be common sense. Be sure to test your code before you submit.

Extra Credit

This function is NOT part of the standard lab grade, but is **extra credit**. It was also a previous exam question, and something similar could show up again.

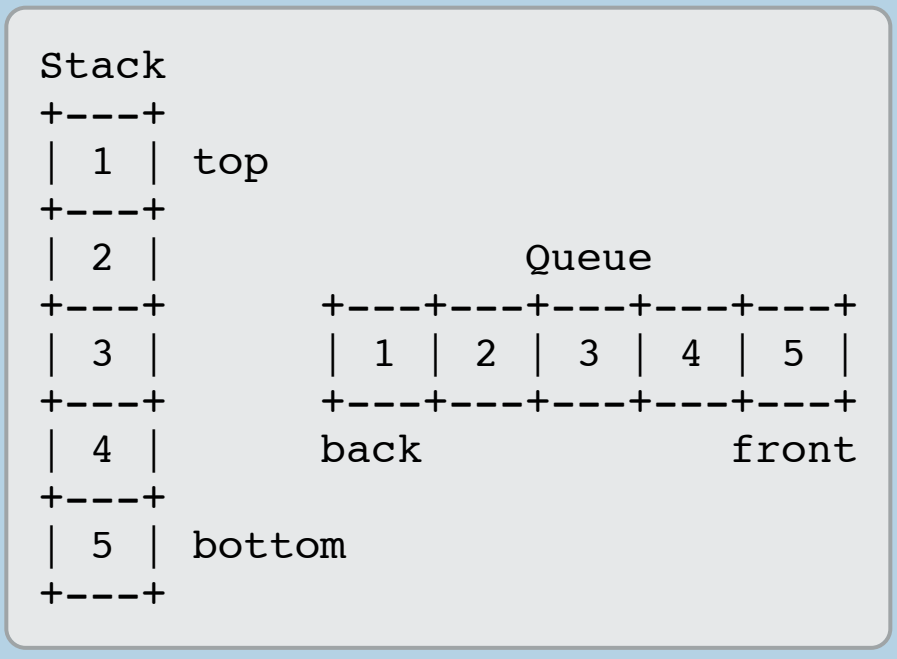
Write the recursive function `verifySame` whose function prototype is below. The function should return `true` if the parameter `stack` and `queue` contain only elements of exactly the same values in exactly the same order, and `false` otherwise (see example below). You may assume the stack and queue contain the same number of items!

We're going to constrain your solution so as to make you think hard about solving it elegantly:

- Your function may not use any loops;
- In your function you may only declare ONE local boolean variable to use in your return statement, and you may only declare TWO local variables of parametrized type `T` to use however you wish. No other local variables can be used; and
- After execution of `verifySame`, the stack and queue must be unchanged. Be sure to comment your code VERY well.

Example

This stack and queue are considered to be the same. Note that we match the bottom of the stack with the front of the queue. No other queue matches this stack.



Click here to see the answer

Committing Your Code

Commit in the usual way.

Grading Information:

The following files are used in grading:

- `exercises.cpp`
- `exercises.h`
- `quackfun.cpp`
- `quackfun.h`

All other files including any testing files you have added will not be used for grading.

(Exam-Prep) Practice Stack/Queue Questions

> **Note**
>
> These questions will also NOT be graded, but are very similar to the types of questions we ask about stacks and queues on exams.

1. Consider a stack implemented as an array where the size is increased by a constant factor every time the array fills up.
    - What is the average running time of the `push` operation?
    - What about if the array size is tripled when the array fills up?

2. What is a potential downside of implementing a queue as a static array?

3. Consider a stack implemented using a singly linked list with a head and tail pointer (and no sentinel nodes) with the top of the stack at the tail of the list.

    - What is the average running time for peek (return the value of the top element without removing it)?
    - What is the average running time for `pop`?
    - What is the average running time for `push`?

4. Consider the following situations:

    - Discuss any advantages and disadvantages of efficiently implementing a queue using an array.
    - Consider a queue implemented using a singly linked list with a head and tail pointer (with no sentinel nodes).
        - Discuss any advantages and disadvantages of implementing the queue in this fashion if the front of the queue is at the front of the list and the end of the queue is at the back of the list.
        - Discuss any advantages and disadvantages of implementing the queue in this fashion if the front of the queue is at the back of the list and the end of the queue is at the front of the list.

5. Consider the following situations:

    - Consider a stack on which an intermixed sequence of `push` and `pop` operations are performed. The `push` operations put the numbers zero through nine in order on the stack and the `pop` operations print out the result of removing a number from the top of the stack. Which of the following sequence(s) could not occur?
        - `2 5 6 7 4 8 9 3 1 0`
        - `4 6 8 7 5 3 2 9 0 1`
        - `1 2 3 4 5 6 9 8 7 0`
        - `0 4 6 5 3 8 1 7 2 9`
        - `4 3 2 1 0 9 8 7 6 5`
        - `2 1 4 3 6 5 8 7 9 0`
        - `4 3 2 1 0 5 6 7 8 9`
        - `1 4 7 9 8 6 5 3 0 2`
    - Consider a queue on which an intermixed sequence of enqueue and dequeue operations are performed. The enqueue operations put the numbers zero through nine in order on the queue and the dequeue operations print out the result of removing a number from the back of the queue. Which of the following sequence(s) could not occur?
        - `0 1 2 4 5 3 6 7 9 8`
        - `9 8 7 6 5 4 3 2 1 0`

- 0 1 2 3 4 6 5 7 9 8
- 0 1 2 3 4 5 6 7 8 9