# Stacks and Queues

*Handout written by Jerry Cain.*

## Stack Primer

The stack container is a data structure that's like a **Vector**, save that insertions and deletions occur at index 0. Here's the reduced interface for the **Stack**:

```
template <typename T>
class Stack {
public:
    Stack();
    ~Stack();

    bool isEmpty();
    int size();
    void push(T elem);
    T pop();
    T peek();
};
```

Begin reading Chapter 5, and try to finish by the end of this coming Wednesday's lecture. Chapter 5 is dedicated to a client-side treatment of the CS106 container classes like **Vector**, **Grid**, **Stack**, **Queue**, and so forth.

In principle, we could just use a **Vector** and limit all **insert** and **remove** calls to involve index 0, but a dedicated **Stack** class is better because:

o   its method names are more descriptive, invoke the illusion of a true stack of plates at a cafeteria, and often contribute to a better narrative as to how an algorithm is working, and…
o   it can be optimized to make insertion and deletion much faster than the **Vector** can. The **Stack** has the advantage of knowing that all operations impact one end of the container, so it can optimize for that. The **Vector** is more generic and can't easily optimize that at the expense of other equally important insertion and deletion points.

## Simple Example

```
static void printLinesInReverse(ifstream& infile) {
    Stack<string> lines;
    while (true) {
        string line;
        getline(infile, line);
        if (infile.fail()) break;
        lines.push(line);
    }

    while (!lines.isEmpty()) {
        cout << lines.pop() << endl;
    }
}
```

## Calculator Example

```
/**
 * Returns true iff the provided character is one of
 * the binary arithmetic operators we've elected to support.
 */
static bool isSupportedOperator(char op) {
    switch (op) {
        case '+': case '-': case '*': case '/': case '%': return true;
        default: return false;
    }
}

/**
 * Computes and returns the integer that results when the integer
 * values identified via first and second are combined using
 * the supplied (presumably legitimate) operator.
 */
static int compute(int first, char op, int second) {
    switch (op) {
        case '+': return first + second;
        case '-': return first - second;
        case '*': return first * second;
        case '/': return first / second;
        case '%': return first % second;
    }
    error("Operator isn't supported.. Aborting...");
    return 0; // some compilers don't know that error kills the program
}

/**
 * Accepts the instruction (e.g. a string of the form "48+9-13**") and
 * the provided instruction as an RPN expression, using a stack for storage.
 * If all goes well, then the stack contains just one integer after
 * it's all over, and that integer is the result of the computation.
 * We return true if and only if the computation was executed without drama.
 */
static int evaluate(const string& instructions) {
    Stack<int> memory;
    for (int i = 0; i < instructions.size(); i++) {
        char inst = instructions[i];
        if (isdigit(inst)) {
            int value = inst - '0';
            memory.push(value);
        } else if (isSupportedOperator(inst)) {
            if (memory.size() < 2) error("Too few digits for operator");
            int second = memory.pop();
            int first = memory.pop();
            int result = compute(first, inst, second);
            memory.push(result);
        } else {
            error("Invalid character in instruction string");
        }
    }

    if (memory.size() > 1) error("Too many digits in result");
    return memory.pop();
}
```

```
/**
 * Interactive read-eval-print loop in place to exercise
 * the above functionality.
 */
int main() {
    cout << "Welcome to our RPN calculator.  You'll love this." << endl;
    while (true) {
        string response = getLine("Data sequence? ");
        if (response == "") break;
        try {
            int result = evaluate(response);
            cout << response << " evaluates to " << result << "." << endl;
        } catch (ErrorException ex) {
            cout << "There were problems with your instruction sequence." << endl;
            cout << "Error: " << ex.getMessage() << endl;
            cout << "Please try again." << endl;
        }
    }

    return 0;
}
```

## Queue Primer

The **Queue** is a **Vector**-like data structure that constrains all insertions to be made at the back and all extractions be drawn from the front.  The name **Queue** is used for what I'm assuming are fairly obvious reasons: queues prompt images of lines at a bank, a supermarket, or an amusement park.  Here's the reduced interface:

```
template <typename T>
class Queue {
public:
    Queue();
    ~Queue();

    bool isEmpty();
    int size();
    void enqueue(T elem);
    T dequeue();
};
```

Again, we could just go with the **Vector** and ignore the **Queue**, but you're making a statement when you use the **Queue**—that some first-come-first-served policy is relevant to the description of the algorithm you're implementing, and that you'd prefer to go with a data structure making use of stronger verbs (**enqueue**, **dequeue**—very strong!).  The **Queue** is also able to optimize for insertion at the end and deletion from the front in a way that the general **Vector** cannot.

## Josephus Survivor Problem

The most interesting introductory example that I can think of involving a **Queue** is framed in terms of the famous, cartoonishly morbid Josephus problem.  The setting: n people are trapped in a cave with no food and presumably no chance for escape, so they collectively opt for group suicide over starvation.  All n people sit in a circle and adopt the tactic of executing every other person until everyone's expired.  The problem: given the number of people, where would you like the sit so that you're among the last two executed, just in case

you and that one other people decide that killing yourself is nonsense and that starvation is much, much more attractive?

The program below generalizes the problem a bit and allows you to specify not only the size of the circle, but the number of people skipped over with every execution. It helps to answer questions like:

- o If there are 32 people and every other person is executed, where should I sit?
- o If there are 26 people and every seventh person is executed, where should I sit?
- o If there are 12 people and every 23$^{rd}$ person is executed, where should I sit?

The program identifies the indices of the last two people to be executed, given the circle size and the number of people spared with each execution.

```
/**
 * Given the number of people in the circle and the
 * number of people that should be overlooked with
 * every execution, determineSurvivors simulates the
 * execution of all numPeople people, keeping track
 * of the two most recently executed, ultimately returning
 * the ids of the last two killed by placing
 * them in the space referenced by lastKilled and
 * nextToLastKilled.
 */
void determineSurvivors(int numPeople, int numToSpare,
                        int& lastKilled, int& nextToLastKilled) {
   Queue<int> circle;
   for (int n = 1; n <= numPeople; n++) circle.enqueue(n);
   while (!circle.isEmpty()) {
      for (int numSpared = 0; numSpared < numToSpare; numSpared++) {
         int next = circle.dequeue();
         circle.enqueue(next);
      }
      nextToLastKilled = lastKilled;
      lastKilled = circle.dequeue();
   }
}

/**
 * Simple utility function prompting the user to
 * enter an integer from a specific range.
 */
int promptForInteger(const string& prompt, int low, int high) {
   while (true) {
      int response = getInteger(prompt);
      if (response >= low && response <= high) return response;
      cout << "We need a number between " << low << " and "
           << high << ", inclusive.  Try again." << endl;
   }
}
```

```
/**
 * Manages the Josephus problem simulation, which is
 * charged with the task of identifying which two
 * people in a circle would be executed last if executions
 * are dished out in a round robin fashion: executing one,
 * skipping k, executing one, skipping k, etc, until only
 * two people remain.
 */
int main() {
    bool keepPlaying = true;
    cout << "Welcome to the Josephus survivors problem" << endl;
    while (keepPlaying) {
        int numPeople =
            promptForInteger("Enter the number of people in the circle: ", 10, 35);
        int numToSpare =
            promptForInteger("Enter the number of people spared per iteration: ",
                             1, 2 * numPeople);
        int lastKilled = 0, nextToLastKilled = 0;
        determineSurvivors(numPeople, numToSpare, lastKilled, nextToLastKilled);
        cout << "Last two people executed: " << lastKilled
             << " and " << nextToLastKilled << endl;
        string response = getLine("Again? ");
        keepPlaying = (response != "no");
    }
}
```