CSCI 104L Lecture 8: Stacks and Queues

```
class QueueADT {
  public:
    void enqueue(const T& data);
    const T& peekfront() const; //look at the oldest element
    void dequeue(); //remove the oldest element
};
```

Notice the following:

- Enqueue cannot change the input parameter.
- Whoever called peekfront cannot change the return parameter they received.
- Peekfront cannot change anything data members in the Queue (**this** cannot change).
- Passing a parameter by const reference allows you to avoid copying the input parameter, while promising the user you won't change their data.

Question 1. What data structure should you use to implement a queue? How would you implement it?

Question 2. Is there anyway to do this efficiently with an array?

```
class StackADT {
  void push(const T& data);
  const T& top() const; //look at the newest element
  void pop(); //remove the newest element
};
```

Question 3. What data structure should you use to implement a stack? How would you implement it?

We can define a stack recursively; a stack is either:

1. The empty stack, or
2. S.push(data), where S is a stack, and data is a data item.

The following "stack axioms" describe stack behavior:

1. For all stacks s, `s.push(data).top()` = data
2. For all stacks s, `s.push(data).pop()` = s

Question 4. Use the above 4 points to determine the result of the following operations:

- s.push(5).push(4).pop().top()
- s.push(5).top().pop()

An algorithm you would normally solve recursively can instead be solved with a stack:

```
stringstream word;
stack s;
cout << "Enter a word: ";
cin >> word;
for(int i=0; i < word.str().size(); i++) {
  char c;
  word >> c;
  s.push(c);
}
while (!s.empty()) {
  cout << s.top();
  s.pop();
}
```

Question 5. Which strings are properly parenthesized?

1. ([ab]]
2. ab}{
3. ([ab{c}]de())

Figure 1: XKCD # 859 "(": Brains aside, I wonder how many poorly-written xkcd.com-parsing scripts will break on this title (or ;;""{<<[' this mouseover text.

You can solve the above problem using a stack; here is an outline:

1. If there are no more characters in the string, and the stack is empty, accept.
2. If there are no more characters in the string, and the stack is not empty, reject.
3. If the next character is an open parentheses, open brace, or open bracket, push it on the stack.
4. If the next character is a close parentheses, close brace, or close bracket, then pop the top item from the stack. If the top item from the stack is of a different type (or the stack was empty), reject.
5. Loop back to step 1

To actually calculate the value of the equation, push all characters (not just parentheses), and on step 4, pop back to the matching parentheses and evaluate the popped expression. Then push the result on the stack.

A queue can also be defined recursively; one is either:

1. The empty queue, or
2. Q.enqueue(data), where Q is a queue, and data is a data item.

A **Deque** allows you to add and remove elements from both ends.

Question 6. How should a deque be implemented?

Question 7. Why stacks and queues, when Deques are more powerful?

Question 8. Why Deques, when a vector is more powerful?

Operator Overloading

Can you define both of these functions simultaneously?

```
int minimum(int x, int y);
double minimum(double x, double y);
```

How about these?

```
bool foo(int x);
void foo(int y);
```

Operators such as `==`, `<=`, etc, are functions too. They can be overloaded as well.

```
class IntArray {
  public:
    bool operator==(const IntArray& otherArray) {
      if (this->size != otherArray.size) return false;
      for(int i=0; i<size; i++)
        if (this->data[i] != otherArray.data[i]) return false;
      return true;
    }
    int& operator[] (int index) {
      return data[index];
    }
    IntArray& operator++ () {
      for (int i = 0; i < size; i++) data[i]++;
      return *this;
    }
  private:
    int size;
    int *data;
};
```

We can now write code such as the following:

```
if (firstArray == secondArray) ++firstArray;
secondArray[0] = 0;
```

Question 9. Why do you suppose `operator++` has a return type, instead of return void?

Question 10. Why do you suppose operator[] returns by reference?

Let's look at the implementation of `operator++`.

```
IntArray IntArray::operator++(int dummy) {
  IntArray copy = *this;
  ++(*this);
  return copy;
}
```

Let's also look at the implementation of `operator*`.

```
IntArray IntArray::operator* (int multiplier) {
  IntArray newArray;
  newArray.size = size;
  for (int i = 0; i < size; i++) newArray.data[i] = data[i]*multiplier;
  return newArray;
}
```

Question 11. Why did we not return by reference?