Exam 2 Review Topics

1. Templates
- What is a template function, and how is it useful?
- What advantages does a templated function have over a function that uses a typedef?
- How are template functions declared?
- When are template functions instantiated / unified? What does that even mean, anyway?
- Know what restrictions a template parameter must meet... For example, if the function uses the copy constructor of the template parameter, it should probably have a copy constructor available to use!
- How do you declare a template class?
- How do you implement methods (non-inline) for a template class?
- Why must the implementations of template class methods be included directly in the class header file?
- How do you declare objects that belong to a template class (e.g., Node<int> for a Node that stores an int)?
- Can you use multiple different data types with a templated class in a single program?

2. The Standard Template Library and Iterators
- Know your STL containers... Be able to name most of them, ideally.
- Know that all the STL containers are templated
- Make sure you know how to declare STL container objects (variables) in your code
- Know how to <u>use</u> vector, list, stack, and queue (push_back and pop_back methods for vector and list; push / pop and either front/top for stack and queue)
- Be comfortable with the standard interface provided by the STL containers
- What are iterators?
- What is the standard pattern used with the STL iterators?

  -
    ```
    for (it = container.begin(); it !+ container.end(); ++it) {
       cout << *it << endl;
    }
    ```
  - why is `++it` preferred over `it++`?
- What is meant by a "left-inclusive" pattern?
- What element does container.begin() point to?
- What element does container.end() point to?
- How do you dereference an iterator (access the value it references)?
- Are the following statements valid? Why or why not?
  - *(container.begin())
  - *(container.end())
- Knowing how to use the STL algorithm library's swap function would be useful to you

3. Stacks
- What is a stack data structure? What real-life concept does it represent?
- What does LIFO mean? How does this describe a stack?
- What "end" of a stack can we access? What is this end called?
- What are some common applications in which stacks are useful?
- Do stacks have STL iterators available to them?

- What three main methods allow you to manipulate a stack? What do each of these methods do?
- What is stack overflow / underflow? What actions cause each of these phenomena?
- Why should you always test whether the stack is empty or not before popping or accessing the top item?
- What is the most common case where you see stack overflow?
- Know how to use the STL stack class and the interface it provides.
- <u>Understand</u> the array-based implementation of a stack (member variables, methods, implementations
- <u>Understand</u> the linked-list implementation of a stack (member variables, methods, and implementations)

4. Queues
- What is a queue data structure? What real-life concept does it represent?
- What does FIFO mean? How does this describe a queue?
- What ends of a queue can we access? What are these ends called?
- What are common applications in which queues are useful?
- Do queues have STL iterators available to them?
- What three main methods (four, if you include back) allow you to manipulate a queue? What do each of these methods do?
- Is the following statement valid? Why or why not?
  - my_queue.front() = 42;
- <u>Understand</u> the circular array implementation of a queue (member variables, methods, implementations
- What is a circular array? What do the index variables first and last represent?
- How do you calculate the next index in a circular array?
- <u>Understand</u> the linked-list implementation of a queue (member variables, methods, and implementations)
- What is a priority queue? How is it different from a regular queue?
- How does the STL priority class determine the "priority" of objects by default?
- How does a priority queue treat items with the same priority (what order are they served)?

5. Recursion
- What is recursion to a computer scientist?
- What does "divide and conquer" have to do with recursion?
- What is a "base case" or "stopping case"? Why is it important?
- What is a "recursive step"?
- Be able to write and evaluate simple recursive functions.
- Understand how the data structures we have seen so far can be treated like recursive structures
- How does your computer keep track of function calls?
- What is an activation record? What information does it store?
- What is infinite recursion? Is it truly infinite?
- What are common pitfalls in using recursion? Be able to spot errors in recursive functions.
- Understand why the recursive fibonacci function is so inefficient
- Be able to trace output produced by simple recursive functions.