CSCI 241 Final Exam Study Guide

Linked Lists

- Know the advantages and disadvantages of using a linked list as opposed to an array. Which operations are more efficient? Which are less efficient or impossible?
- Be aware of both singly-linked and doubly-linked lists and of lists that have both front and rear pointers.

Stack and Queue ADTs

- Know the types of errors that can occur when using a stack or queue.
- Be able to add an item to a stack or queue (linked-list implementation).
- Be able to remove an item from a stack or queue (linked-list implementation).
- Be familiar with the other typical operations performed on a stack or queue implemented as a linked list.

Deque ADT

- Know what a deque or "double-ended queue" is. Be able to write code to insert or delete items at either end of a doubly-linked deque.

Templates

- Know how to write a template class or function – and what all of the requirements are (where method implementations must be placed, for example).
- Know how to create an object of a template class.
- Do not be surprised to find template classes in coding questions.

Recursion

- *Recursion* is a technique where the solution to a problem depends on solutions to smaller instances of the problem.
- A *recursive function* or *method* calls itself.
- A recursive call is always conditional – there must be some case (called the *base case*) where recursion does not take place. A recursive call should make progress towards the base case.
- Recursion is never required in C++. A recursive algorithm may always be rewritten with either a loop or a loop and a stack.
- In C++, a recursive algorithm is often less efficient in terms of memory usage and speed than the equivalent non-recursive algorithm. However, the recursive version may be shorter and easier to code.
- You should be able to write a simple recursive function or method to do something (like counting the nodes in a linked list).

Quicksort

- You should know the logic for the quicksort partition function used on Assignment 9 and be prepared to demonstrate how it partitions an unsorted array of integers.

Inheritance

- Inheritance is a way to compartmentalize and reuse code by creating a new class based on a previously created class.
- The previously created class is called a *superclass* or *base class*.
- The new class derived from a base class is called a *subclass* or *derived class*. It represents a smaller, more specialized group of objects than its base class.
- A derived class may add new data members, add new methods, and override methods in the base class.
- Inheritance is used to represent an "is a" relationship between a derived class and a base class. A derived class object is also an instance of all of its base classes (e.g., a `Circle` is a `Shape`).
- Be able to code a derived class using `public` inheritance.
- Be able to code a constructor for a derived class, including one that passes arguments to the base class constructor using constructor initialization list syntax.
- Know the order in which constructor and destructor bodies will execute when you create or destroy an object of a derived class.
- Members of a class with `protected` access can be directly accessed by methods of the class, `friends` of the class, and methods of derived classes of the class.
- Be able to describe the advantages and disadvantages of making data members `protected` versus making them `private` and accessing them using set and get methods.
- Know the difference between overloading a method or function and overriding a method:
    - *Overloading* refers to a new method or function with the same name as an existing one in the same scope, but with a different signature (number of arguments, data types of arguments, order of data types, whether or not a method is const)
    - *Overriding* a method means writing a method in a derived class with the same name, arguments and return data type as a method in the base class. The derived class method effectively replaces the base class method.
- Know how to call a base class version of a method from within a derived class method that overrides it.
- Know how to perform an *upcast* – a conversion of a derived class pointer or reference type to its base class pointer or reference type. In C++ this does not require an explicit type cast.
- A base class pointer or reference can only be used to call methods that are declared in the base class.
- Know how to perform a safe *downcast* – a conversion of a base class pointer or reference to one of its derived class pointer or reference types – using the `dynamic_cast` operator. A `dynamic_cast` requires that *runtime type information* be enabled. Know how to test whether or not a `dynamic_cast` was successful.
- C++ supports *multiple inheritance* – a derived class may have more than one base class.

Polymorphism

- *Polymorphism* is the ability of objects belonging to different types to respond to method calls of the same name, each one according to an appropriate type-specific behavior. In C++, polymorphism is implemented through the use of *virtual methods*.
- You should know how to code a virtual method. A virtual method called using a pointer or reference will use *dynamic binding*. Other method or function calls use *static binding*.
    - With dynamic binding, which version of a method to call is determined at runtime based on the data type of the object a pointer or reference points to (the *dynamic type*), rather than the data type of the pointer or reference (the *static type*).
    - With static binding, the data type of the object name, pointer to an object or reference to an object is used to determine which version of a method or function to call at compile time.
- A *pure virtual method* (also called an *abstract method*) is a virtual method with no implementation, only a prototype (that ends with `= 0`).

- An *abstract class* in C++ is one that contains one or more pure virtual methods. A class that is not abstract is referred to as a *concrete class*.
    - You cannot create an object of an abstract class.
    - You can use an abstract class as a base class for inheritance.
    - You can declare a pointer to an object of an abstract class or a reference to an object of an abstract class. Such a pointer or reference will normally be used to point to an object of one of the abstract class's derived classes.
- A derived class must implement all of the pure virtual methods in an abstract base class or it is also an abstract class.
- An *interface* is an abstract class that contains only pure virtual methods and symbolic constants (`static const` data members).

Sample Exam Questions

A class definition for an abstract C++ class that describes a product is shown below.

```
class Product
   {
   private:

      string productID;
      int numInStock;

   public:

      Product(const string&, int);
      virtual ~Product();

      virtual void get_price() const = 0;
      void get_numInStock() const;
   };
```

1. Write a class definition for a concrete subclass called `Book` that will be derived from the `Product` class given above using `public` inheritance. **You do not need to write the method definitions, just the prototypes that appear in the class definition.**

    - A `Book` contains three additional private data members: a title (a `string`), an author (a `string`), and a price (a `double`).
    - The `Book` constructor takes five arguments – a `string` and an integer to initialize the base class data members, and two `string`s and a `double` to initialize the data members of the `Book` class.
    - A `Book` should also have a destructor and an implementation of the pure `virtual get_price()` method in the superclass.

2. Write the method definition for the `Book` constructor. The constructor should assign the last three arguments to the corresponding `Book` data members. The first two arguments should be passed to the superclass constructor.

3. Assume that the following `vector` of superclass pointers has been declared:

```
vector<Product*> productList;
```

   The pointers in the `vector` point to an unknown number of subclass objects. Some of these subclass objects are objects of the `Book` class, while others are objects of various other subclasses derived from `Product` – `CD`, `DVD`, etc. Write a fragment of C++ code to find the total value (`numInStock * price`) of all of the `Book`s in the `vector`.

4. Rewrite the contents of the following array once the array has been partitioned by the quicksort partition code shown in the separate handout and used on Assignment 9.

   Before partition: `74  38  52  28  46  65  32  54  95  57`

   After partition: ____ ____ ____ ____ ____ ____ ____ ____ ____ ____

The next two questions involve the following data types:

```
template <class T>
struct LNode
   {
   T data;
   LNode<T>* next;
   };


template <class T>
class List
   {
   private:
      LNode<T>* head;
      LNode<T>* tail;

   public:
       .
       .
       .
   };
```

Also assume that `current` is a pointer to a node in the middle of a singly linked list, and that `newNode` is a pointer to a newly allocated node.

5. Write a code fragment (not a complete method) that will insert `newNode` at the front of the list.

6. Write a code fragment that will insert `newNode` at the rear of the list.

7. Write a code fragment that will insert `newNode` *after* the node pointed to by `current`.

8. Write a recursive method that will return the size of the list. The method should take a pointer to a `Node<T>` and return an `int`. When the method is initially called, it will be passed the list's `head` pointer.

9. Write a non-recursive method that will return the size of the list. The method should take no arguments and return an `int`.