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 write code to add an item to a stack or queue (linked-list implementation).
- Be able to write code 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.

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 be familiar with the logic for the quicksort partition function used on Assignment 8 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

True / False

1. An "is-a" relationship between two classes (e.g., "a Car is a Vehicle") can be represented using `public` inheritance.

2. C++ supports multiple inheritance.

3. You cannot declare an object of an abstract class.

4. An *abstract class* in C++ is a class that contains at least one pure `virtual` method.

5. In C++, you can convert a pointer to a derived class object into a pointer to a base class object (i.e., "upcast" the pointer) without an explicit type cast.

6. An "interface" can be represented in C++ as an abstract class that contains only symbolic constants and pure `virtual` methods.

Multiple Choice

1. In C++, a call to a standalone function will always use

   A. dynamic binding.
   B. automatic binding.
   C. static binding.
   D. external binding.

2. Suppose you have a template class named `stack`. What is the correct syntax for declaring a `stack` of `string` objects?

   A. `stack<string> s;`
   B. `stack s[string];`
   C. `stack s;`
   D. `string stack s;`

3. Which of the following is an advantage of arrays over linked lists?

   A. Arrays always take up less memory than linked lists.
   B. You can run a binary search on a sorted array, but not on a linked list.
   C. You can insert anywhere in an array without having to move the existing data in the array/
   D. Arrays can easily grow in capacity, while linked lists can't.

4. Suppose that `p` is a pointer variable that contains `nullptr`. What happens if your program tries to access `*p`?

   A. A syntax error occurs at compilation time.
   B. A "segmentation fault" run-time error occurs when `*p` is evaluated.
   C. The program runs to completion and then a "segmentation fault" run-time error occurs.
   D. The results are unpredictable.

5. With `public` inheritance, the `private` members of a base class can be accessed by

   A. methods of that base class.
   B. code anywhere in your program.
   C. methods of any derived classes derived from that base class.
   D. both A and C.

6. A base class pointer that points to a derived class object can be used to call

   A. only methods declared in the derived class.
   B. only methods declared in the base class.
   C. any method declared in either the base class or the derived class.
   D. any method in any class included in your program.

7. When a derived class object is destroyed, in what order are the destructors executed?

   A. First the derived class destructor is executed, then the base class destructor is executed.
   B. First the base class destructor is executed, then the derived class destructor is executed.
   C. Only the base class destructor is executed.
   D. Only the derived class destructor is executed.

8. Assume that `Vehicle` is a base class, and `Car` is a class derived from `Vehicle` using `public` inheritance. What is the correct way to call the `print()` method of the `Vehicle` class from the `print()` method of the `Car` class?

   A. `Car::print();`
   B. `Vehicle::print();`
   C. `super.print();`
   D. `base.print();`

9. When a derived class object is created, in what order are the constructor bodies executed?

   A. First the derived class constructor body is executed, then the base class constructor body is executed.
   B. First the base class constructor is executed, then the derived class constructor is executed.
   C. Only the base class constructor is executed.
   D. Only the derived class constructor is executed.

10. Recursion may always be replaced by

    A. a loop.
    B. either just a loop or a loop and a stack.
    C. either just a loop or a loop and a queue.
    D. Nothing can take the place of recursion. IT'S JUST THAT AWESOME!!!

Coding and Short Answer

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 double getPrice() const = 0;
      int getNumInStock() const;
   };
```

1. Write a class definition for a concrete derived class 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 reference to a constant `string` and an integer to initialize the base class data members, and two references to constant `string`s and a `double` to initialize the data members of the `Book` class.
   - A `Book` should also have an implementation of the pure `virtual` `getPrice()` method in the base class.

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 base class constructor.

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

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

   The pointers in the `vector` point to an unknown number of derived class objects. Some of these derived class objects are objects of the `Book` class, while others are objects of various other classes derived from `Product` – `CD`, `DVD`, etc. Write a fragment of C++ code to find the total number of the `Book`s in stock. The total equal to sum of the number in stock for all of the `Book` objects 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 8.

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

   After partition: _____ _____ _____ _____ _____ _____ _____ _____ _____ _____

5. Write a method definition for the `pop()` method of the `Stack` class shown on the separate handout. You may assume the method will not be called if the stack is empty.

6. Write a method definition for the `push()` method of the `Queue` class shown on the separate handout.

7. Write a method definition for the `clear()` method of the `Stack` class shown on the separate handout.

8. (4 points) Consider the following recursive C++ function:

```
int compute(int n)
   {
   if (n <= 1)
      return 1;
   else
      return n + compute(n - 1);
   }
```

Assume we initially call this routine from `main()` with the statement `total = compute(6);`

What value (if any) will eventually be returned and placed in the variable `total`?

_____

Quicksort partition algorithm:

```
int partition(int set[], int start, int end)
   {
   int pivotIndex, mid;
   int pivotValue, temp;

   mid = (start + end) / 2;

   temp = set[start];
   set[start] = set[mid];
   set[mid] = temp;

   pivotIndex = start;
   pivotValue = set[start];

   for (int scan = start + 1; scan <= end; scan++)
      {
      if (set[scan] < pivotValue)
         {
         pivotIndex++;
         temp = set[pivotIndex];
         set[pivotIndex] = set[scan];
         set[scan] = temp;
         }
      }

   temp = set[start];
   set[start] = set[pivotIndex];
   set[pivotIndex] = temp;

   return pivotIndex;
   }
```

Class declaration for a linked list-based `Stack` class:

```
template <class T>
struct Node
   {
   T data;
   Node<T>* next;
   Node(const T& = T(), Node<T>* = nullptr)
   };

template <class T>
Node<T>::Node(const T& newData, Node<T>* newNext)
   {
   data = newData;
   next = newNext;
   }

template <class T>
class Stack
   {
   public:

      Stack();                                  // Default constructor
      Stack(const Stack<T>&);                   // Copy constructor
      ~Stack();                                 // Destructor
      Stack<T>& operator=(const Stack<T>&);  // Copy assignment operator

      void clear();                 // Sets the stack back to empty
      size_t size() const;          // Returns number of items in stack
      bool empty() const;           // Returns true if stack is empty

      void push(const T&);          // Insert item at top of stack
      void pop();                   // Remove top item from stack
      const T& top() const;         // Return top item

   private:

      Node<T>* sTop;         // Pointer to top (first) node in linked list
      size_t sSize;          // Number of items stored in the stack

      void copyList(const Stack<T>*);
   };
```

Class declaration for a linked list-based `Queue` class:

```
template <class T>
struct Node
   {
   T data;
   Node<T>* next;
   Node(const T& = T(), Node<T>* = nullptr)
   };

template <class T>
Node<T>::Node(const T& newData, Node<T>* newNext)
   {
   data = newData;
   next = newNext;
   }

template <class T>
class Queue
   {
   public:

      Queue();                                  // Default constructor
      Queue(const Queue<T>&);                   // Copy constructor
      ~Queue();                                 // Destructor
      Queue<T>& operator=(const Queue<T>&);  // Copy assignment operator

      void clear();                 // Sets the queue back to empty
      size_t size() const;          // Returns number of items in queue
      bool empty() const;           // Returns true if queue is empty

      void push(const T&);          // Insert item at back of queue
      void pop();                   // Remove front item from queue
      const T& front() const;       // Return front item
      const T& back() const;        // Return back item

   private:

      Node<T>* qFront;      // Pointer to front (first) node in
                            // linked list
      Node<T>* qBack;       // Pointer to back (last) node in
                            // linked list
      size_t qSize;         // Number of items stored in the queue

      void copyList(const Queue<T>*);
   };
```