CIS 291 Exam #1 - Study Suggestions

* **last modified: 2-22-05**

* The test covers through HW #5, the Week 7 Lab Exercise Exercise, and material through the 2-17-05 lecture/3-01-05 lab.
* Anything that has been covered in **lecture** is fair game;
* Anything covered in a **course handout** or **course posting** is fair game;
* Anything that has been covered in a **lab exercise** or **homework assignment** is ESPECIALLY fair game.

* But, these are some especially-significant topics to help you in your studying for the exam.

* You are responsible for being familiar with, and following, the course **style** guidelines.

* The exam will be closed-book and closed-notes, and you are expected to work individually.
* Test format: will likely be short answer, possibly with a smattering of multiple-choice questions.
    * All you need to provide is a pen or a pencil;

    * EXPECT to have to read and write C++ code, pseudocode, UML notation.

* note that you could definitely be given code and asked questions about it, as in the Week 6 Lab Exercise (answering questions about the different sort implementations).

* the only aspect of namespaces that you are responsible for on this exam is that you need to use **`using namespace std;`** after #include'ing standard libraries in modern, standard C++.

* **data structures**
    * what is a **data structure**? an organized collection of data...

    * what is an **abstract data type** (ADT)? a collection of data PLUS all of the operations for acting on that data;

        * What are some of the benefits of using a well-designed ADT class for a data structure within a program?

        * Be comfortable, too, with such terms as information hiding and abstraction.

    * you should be able to read and use a data structure given a "pseudo-UML" diagram such as that given for stack and queue in-lecture (and available from the course web page). You should be able to answer questions based on reading it, and should be able to write code using such a class.

* **phases of software development and program design recipe handouts**
    * be comfortable with the basic phases of software development as discussed in lecture; be comfortable with the **basic function design recipe** discussed.

        * especially: for a function,
            1. figuring out what data is involved (data analysis),

            2. then writing a CONTRACT,

            3. then writing the HEADER corresponding to that contract (here, remember, we mean the first line of the *implementation*/definition, NOT the prototype/declaration/what goes in the .h file)

            4. then writing the PURPOSE, INCLUDING the parameter names appropriately,

writing PRECONDITIONS and POSTCONDITIONS if called for,

5. then writing the EXAMPLES, actual example calles of the function, including what the function returns or does as a result of that call,

6. and only THEN devising its algorithm, and then translating that algorithm into code.

* what is the course "syntax"/notation for a function **contract**? Given a non-main function or its description, you should be able to write a contract using this syntax/notation.

* in this course, what should be incorporated into the Purpose: statement of a function that has parameters?

* what is a precondition? what is a postcondition? what are the expectations for these?
* you should be able to read and write **assert** statements to verify a function's preconditions (for preconditions for which such tests are reasonable); you should know what happens when an assert's condition is false.

* what goes in the Examples: section of a "regular" function's opening comment block, in this course? How do we write these when the function returns, say, an int?
* when should you come up with specific examples for a function or method? (BEFORE you write it!)

* Given a function and/or its description, you should be able to write examples that adequately test it (cover all major categories of input and boundaries between those categories).

* be comfortable reading and appropriately writing code using EXIT_SUCCESS and EXIT_FAILURE. (remember the course conding standards regarding these.)

* should be able to read, write tester programs (testing main functions) as you have been doing in course assignments.

* **Lab and C++-related details**
  * how can you compile a C++ function on cs-server? how can you compile and link a C++ program on cs-server?

  * what should go in a .h file for a non-main function being written in its own file? How does another function use a non-main function written in its own file?

  * how can you redirect screen output to a file in UNIX?

  * how should you declare a named constant in this course? (be familiar with both the syntax, its meaning, and the course style standards for named constants)
    * Within a class, how many "copies" of a thing declared to be **static** are there?

* **stacks**
  * what is LIFO?

  * you need to be comfortable with the stack ADT (and its pseudo-UML).

  * what is a stack? what are the typical operations defined on stacks? how can a stack be used?

  * in what kinds of situations is a stack appropriate? what are some typical applications of stacks?

  * if I asked you to perform a sequence of pushes and pops on a stack, you should be able to simulate how it would behave and what would result;

- how should you avoid popping from an empty stack?
- how can stacks be implemented? (right now, you should know of at least 2 different ways; we'll be adding at least 2 more later...)
- you should be able to **use** the stack ADT in problem solutions; you also should be able to implement stack operations in the different stack implementations discussed so far.
- how can you implement stacks using static arrays? using dynamic arrays?
  - you should be able to compare/contrast these implementations; discuss their tradeoffs, big(O) complexity for different operations using the different implementations, etc.

* **queues**
  - what is FIFO?
  - you need to be comfortable with the queue ADT (and its pseudo-UML).
  - what is a queue? what are the typical operations defined on queues? how can a queue be used?
  - in what kinds of situations is a queue appropriate? what are some typical applications of queues?
  - if I asked you to perform a sequence of enqueues and dequeues on a queue, you should be able to simulate how it would behave and what would result;
  - how should you avoid dequeuing from an empty queue? what is queue underflow?
  - how can queues be implemented? (you should know of at least 2 different ways; we'll be adding at least 2 more later...)
  - you should be able to **use** the queue ADT in problem solutions; you also should be able to implement queue operations in the different queue implementations discussed so far.
  - how can you implement queues using static arrays? using dynamic arrays?
    - you should be able to compare/contrast these implementations; discuss their tradeoffs, big(O) complexity for different operations using the different implementations, etc.
    - in an array-based implementation of a queue, what is rightward/downward drift? How can it be avoided? (Or: what do we mean by a **circular array**? Why is it a useful approach in implementing a queue?)
      - hint: it involves modulo arithmetic, the % operator in C++...
    - in each implementation, how can you distinguish between a full queue and an empty one?

* **template classes**
  - you should be able to declare and instantiate an instance of a template class;
  - given an instance of a template class header file and implementation file, you should be able to:
    - read it and answer questions about it;
    - modify it (including adding methods);
    - write another template class using the provided one as a reference;

- why would one want to use a template class? How does a template class differ from a regular class?

- what has been our class practice for the suffixes for the two files involved in creating a template class? What should be #include'd where?

- When is a template class compiled?

- if a program is to use a template class...
  - ...what should it #include? (And what must that #include'd file #include?)
  - ...how does it declare an instance of that class?
  - ...how does it call a public member function for an instance of that class?
  - ...how do you compile and link that program on cs-server?

- **searching (sequential search, binary search, and hashing)**
  - **sequential search and binary search**

    - you are responsible for knowing sequential search and binary search.

    - you should be able to describe the basic algorithm for each; you should know their run-time complexities.

    - if code was given,you should be able to recognize which of the above is being implemented within that code.

    - (frankly, you should be able to code some version of sequential search at the drop of a hat...)

    - you should be able to reason about variations on these basic algorithms

  - **hashes, hash tables, and hashing**
    - what is a hash function? What is a hash table? What is hashing?

    - what does a hash function do? How is it used?
    - what are some of the desired properties for a hash function? you should be able to implement a simple hash function, and be able to assess its quality;
      - what are some examples of hash functions/some examples of typical hash function techniques?

    - what are some typical operations defined on hash tables? how can a hash table be used?
    - how are items inserted into a hash table? how are items retrieved from a hash table? What is the average case time complexity for such insertion and deletion? what is the worst-case time complexity for these operations (and when does it occur)?
    - what particular typical "collection of things" operation is particularly inefficient when hashing is used to implement that collection?

    - Be comfortable with open-addressing (plain array-based hash table) collision-resolution techniques such as linear probing, quadratic probing,double hashing, rehashing.
      - what is meant by clustering? (primary and secondary clustering)
      - why is clustering a problem?

    - What is the significance of hash table size in hashing's performance, in general?
      - what kind of characteristics are considered desirable for a hash table's size?

    - in what kinds of situations is a hash table appropriate?
    - what are some typical applications of hash tables?
    - what operation is particular NOT so well-behaved when implementing a collection of items using a

hash table?

* need to be comfortable with array-based hash table implementations; need to be comfortable with the concept of separate chaining/buckets-and-chaining hash table implementations
  * I could ask you to implement array-based hashing; I will not ask you to implement buckets-and-chaining at this point.
  * you should be able to compare/contrast these implementations; discuss their tradeoffs, big(O) complexity for different operations using the different implementations, etc.
  * Given a hash function and a hash table implemented as an array of pointers to linked lists (that is, implemented using **separate chaining/buckets-and-chaining**), could you show what hashing would do for a collection of actions (insertions and deletions of specified values)? Could you do so for a hash table implemented as an array (open addressing) using a given collision strategy?
* need to be able to **use** a given hash table ADT or UML or interface to solve problems; you also need to be able to implement (and reason about the big O complexity of) hash table operations in the different hash table implementations.

* **running time analysis**
  * what is big-O notation? What does it mean? How can it be useful?
  * given a formula representing the number of steps that some algorithm requires for a problem of size n, you should be able to give the big-O notation for such an algorithm.
  * what is average-case run-time complexity? worst-case? best-case? What are the differences between these?
  * you should know (or be able to figure out) the run-time complexities for "simple" operations, and express them using big-O notation;
  * you should know (or be able to figure out) the average-, worst-, and best-case time complexities for:
    * sequential search and binary search
    * selection sort, insertion sort, bubblesort, merge sort, quicksort, and radix sort
  * what phrase is equivalent to O(1)? to O(n)? to O(log n)? to $O(n^2)$? to $O(2^n)$?
    * except for $O(2^n)$, you should be able to give an example of an algorithm that takes that average-case running-time; you should be able to give an example of an algorithm that average-case running time for O(n log n), also.
  * (remember: in computer science, when log n is written, base 2 is assumed.)

* **sorting**
  * you are responsible for knowing selection sort, insertion sort, bubblesort, merge sort, quicksort, and radix sort
  * you should be able to describe the basic algorithm for each; you should know their run-time complexities (best-case, average-case, and worst-case)
  * if code was given, you should be able to recognize which of the above is being implemented within that code.
  * you should be able to reason about variations on these basic algorithms (using bubblesort within quicksort when the list size is sufficiently small, for example)