CIS 291 Final Exam - Study Suggestions

* **last modified: 5-04-05**

* SOMETHING NEW: for the final, you may bring **a single 8.5" by 11" sheet with your name prominently on it** on which you have **handwritten** the course material of your choice, along with your name --- this sheet **must be turned in with your final exam**, and will not be returned. (You may write on both the front and the back, if you wish; and you must **personally** create/hand-write the sheet.)

* final is CUMULATIVE!
  * if it was fair game for exams #1 or #2, it is fair game for the final;
  * so, studying the review suggestions for exams #1, #2 would be wise; (they're still available from the course web page, under "Homeworks and Handouts", if you've misplaced your earlier copies);
  * there may indeed be similar styles of questions as on those exams;

* general style of questions will be similar to those on exams #1 and #2;

* Anything that has been covered in **lecture** is fair game;
* Anything covered in a **course handout** is fair game;
* Anything covered in a **lab exercise** or **homework assignment** is **ESPECIALLY** fair game.

* The exam will be closed-book and closed-notes (except for the one 8.5" sheet mentioned above), 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 (and the 8.5" sheet mentioned above, if desired).

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

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

  * there **could** be question(s) on the Final Exam like those on the earlier exams where you had to give Contract, Purpose, Preconditions, Postconditions, Examples, etc.

  * that is, you should still be comfortable with the design recipe we've been using for functions, and should be able to fill in the opening comment block "templates" we've been using appropriately.

* You should still be comfortable with the design recipe we've been using for functions, and should be able to fill in the opening comment block "templates" we've been using appropriately.

* 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++.

* there could certainly be questions involving big-O notation; for example, I might ask what the big-O notation would be for the {average, best, worst} case run-time complexity for an instance of a particular ADT containing **n** items implemented in a particular way.

* note that all of the **sorting algorithms** discussed --- including **treesort** and **heapsort** --- are most certainly fair game on this final! I want you to be able to compare/contrast/reason about/recognize all of them;

* You will likely be given pseudo-UML's for different typical container classes and/or template classes; you could be asked to write code **using** instances of such classes, and you could be asked to write **implementations** of member functions (or possibly even the entire class) given within such pseudo-UML's.

* high points from material SINCE Exam #2:
  * (which, admittedly, is not a lot of new material, because of the particular scheduling of Exam #2...!)

* **graphs**
  * what is a graph? What is its mathematical definition? What is the relationship between graphs and trees?

  * You should be comfortable with the basic graph terminology: vertex, node, edge, graph, subgraph, adjacent, path, simple path, cycle, simple cycle, connected, disconnected, complete, undirected graph, directed graph, undirected edge, directed edge, weighted edge, successor, predecessor.
    * for the above terms for which it is appropriate, be comfortable with their (sometimes different) meanings for directed graphs and indirected graphs.

  * what are some typical operations defined on graphs? how can graphs be used?

  * in what kinds of situations is a graph appropriate?
  * what are some typical applications of graphs?

  * traversals: depth-first-search (DFS)-based, breadth-first-search (BFS)-based
    * given a graph and a starting vertex, you should be able to show an order in which a graph's vertices could be visited using each of these;

    * for which is a recursive approach easier? For that traversal, if you do not use recursion, what abstract data type is particularly appropriate for use in that traversal?

    * in implementing a DFS-based traversal, which abstract data type is particularly useful/appropriate, if recursion is not used? Can recursion be used easily for this?

    * in implementing a BFS-based traversal, which abstract data type is particularly useful/appropriate, if recursion is not used? Can recursion be used easily for this?

  * need to be comfortable with both adjacency-list and adjacency-matrix implementations of graphs
    * you should be able to compare/contrast these implementations; discuss their tradeoffs, big(O) complexity for different operations using the different implementations, etc.

    * you should be able to sketch a conceptual depiction of a graph within an adjacency matrix implementation or within an adjacency list implementation, whether it is directed or undirected, whether it has weighted edges or not. You should be able to perform graph manipulations to a graph expressed in such form.
      * (you should be able to read/manipulate/express a graph as a drawing or in G = (V, E) form as well)

  * need to be able to **use** a given graph ADT or UML or .h file to solve problems; you also need to be able to implement (and reason about the big O complexity of) graph operations in the different graph implementations.

  * if you are given a connected, undirected graph --- what is an easy way to tell if it has any cycles?

  * some graph-related facts you should know:
    * if a connected, undirected graph has n vertices, at least how many edges should it have?
    * if it has exactly that many edges, what do you know that it cannot obtain?
    * if it has more than that many edges, what do you know it must contain?

  * if you have a connected undirected graph with cycles and you remove edges until there are no cycles, what do you get?

  * What is a circuit? What is an Euler circuit (a circuit that passes through every **edge** exactly once)?

What is a Hamilton(ian) circuit (a circuit that passes through every **vertex** exactly once, except it begins and ends at the same vertex v)?

* **make** and **makefiles**
  * when/why might one choose to use a **makefile** for a C++ program? [they are more general than that, of course, but in this course we're most interested in using them in that particular area.]

  * what is the basic syntax of a **makefile**? what is a makefile **entry**?
    * what is a **target**? what are **dependencies**?

    * what character MUST precede the command(s) to be run subject to the target and dependencies?

  * in a makefile for a C++ program, what is generally the **first** target?

  * what command do you use to make use of a **makefile**?
    * how does that command change depending on the actual name of the makefile?

    * what happens when you type this command?

  * Given a description of a scenario, you should be able to write a basic makefile for that scenario; give a makefile, you could read it and answer questions about it.
    * this could include deducing what would be re-done if a file was changed, and then **make** was called;

  * for a typical C++ program: what will be the dependencies for the corresponding executable target? what will be the command for compiling it to create that executable target?

  * for a typical C++ class: what will be the dependencies for its corresponding **.o** target? what will be the command for compiling it to create that **.o** target?

* **collection implementation options**
  * at this point in the semester, you should be comfortable with a wide variety of possibilities for implementing a class for a specialized collection of items; you should be able to reason about this, discuss this, answer questions about the possibilities, tradeoffs, performance considerations, etc.

  * (you should now be prepared to make reasoned use of the C++ STL - Standard Template Library, although the STL itself will not be on the final, of course)