THE UNIVERSITY OF ARIZONA.
DEPARTMENT OF COMPUTER SCIENCE

CSc 352: C Coding Standards

The C code you write will be *required* to adhere to the following standards (which are actually not very tight). Failure to do so will be penalized.

The standards given in this document will evolve through the semester as we cover more of the C language and supporting tools. For each assignment, please check to see what the relevant standards are.

Programming Style

1. The beginning of each file should contain a comment giving the name of the file, its author, and its purpose. For example:

   ```
   /*
   * File: random.c
   * Author: J. Doe Jr.
   * Purpose: Defines a function to compute a random integer
   * between 1 and 100.
   */
   ```

2. Indentation must be appropriate and consistent. Each time you enter a new block, the code must be indented by a reasonable amount; when you leave the block, you should revert to the indentation level of the enclosing block.
3. Each function in the file other than main should be preceded by a brief comment describing:
   i. what the function does;
   ii. what arguments it takes and what value it returns; and
   iii. any assumptions the function makes (e.g., about the argument values); and
   iv. anything else of interest about the function.

   For example:

   ```
   /*
   * mindist(node1, node2) -- returns the shortest distance between
   * nodes node1 and node2 in an undirected graph. It assumes that
   * edge distances are non-negative. It uses Dijkstra's algorithm.
   */
   ```

4. The code should be appropriately modularized: functions should not be "too big". The way you determine whether a function is appropriately modularized is to see whether its behavior ("what it does") can be accurately and adequately described in a single brief comment (see above). If it cannot, the function is too complex and should be broken into two or more smaller functions.

   Note that this notion of "too big" is *semantic* rather than *syntactic* — we're interested in the *conceptual effort* involved in understanding the code for that function rather than relying on some arbitrary count of the number of lines it spans.

5. Error messages should always be sent to **stderr**. Normal (non-error) output should never be sent to **stderr**.
6. Any library call that may encounter an error should be checked to see whether it was able to execute without any errors. In case of an error, an appropriate error message must be given.
7. (Recommended, not required) In case of errors in system library routines that can fail in different ways, and which indicate the reason for failure by setting the variable **`errno`**, error messages should use **`perror()`** to give specific and informative error messages.

   Whether or not a library routine sets **`errno`** is indicated in its man pages. For example, **`fopen()`** sets **`errno`** but **`malloc()`** does not.

8. Header files should be protected against multiple inclusions.

Compiler and Runtime Issues

1. The code should compile with *no* compiler errors or warnings when compiled using **gcc -Wall**.
2. **Exit status:**
   - A program that executes without encountering any errors should indicate this with a return value (exit status) of `0`.
   - If an error is encountered during execution, this should be indicated by a non-zero return value (exit status). The precise exit status in this case should be determined from the assignment specification; if no error exit status is explicitly specified in the assignment spec, use the value `1`.