CMSC 212 Discussion exercise, Wednesday, September 16 Fall 2009

Note that, as described in the class syllabus, 5% of your course grade will be based on submitting in–class exercises such as this one. Here's how these exercises will be graded:

- You must submit the exercise by the end of your own discussion section, even if you don't finish it. We will only be grading submissions made by the end of your own discussion section (the last one, if you submit more than once during your discussion).
- **Other than today's exercise**, the submit server will not run any discussion section exercises.
- We won't be grading the exercises based on how well they work or whether you finish them, only on whether you made a reasonable attempt at doing the exercise.
- Obviously you'll get more out of the exercises if you do get them to work. You're encouraged to continue working on them on your own even if you don't finish an exercise during class. If you get stuck, you can come to office hours. But there's no sense in resubmitting the exercise once your discussion time is over, since that submission wouldn't be graded.
- Each exercise will be graded out of 1 point (0 if not done, or 1 if done), and the results will be visible on the grade server at some point after the discussion in which the exercise was done.

1. Log into `linux.grace.umd.edu` and, from your TerpConnect home directory, copy the compressed tarfile `discussion05.tgz` from the directory `~/212public/discussions/week03` to your directory, using the `cp` command (this is assuming you created the symbolic link `~/212public` during the earlier dissusion section), for example:

   For all programming work you must use your extra disk space, which you created a link to in the earlier discussion section, which you also should have created a symbolic link named `~/212` to during the earlier discussion, so the command could be:

   ```
   cp ~/212public/discussions/week03/discussion05.tgz ~/212
   ```

2. `cd` to your extra disk space where you just copied the tarfile ("`cd 212`") and list the contents of the tarfile using the command "`gtar -ztvf discussion05.tgz`".

   Note: `tar` is the UNIX/Linux tape archiver program (`gtar` is the Free Software Foundation's version of `tar`), which can be used to save data on backup tapes, and also to combine and store separate files into a single tarfile (like zipfiles in the Windows world), which is what we're using it for here. The convention is that files with an extension of `.tar` are tarfiles created by `tar` (or `gtar`). The filename extension `.tgz` is by convention used for tarfiles which, to save disk space, are also compressed using the `gzip` utility; tar has the ability to create tarfiles and compress them at the same time.

   `tar` can be used to create a tarfile consisting of separate files, to list the contents of a tarfile as you just did, to extract the files from a tarfile, along with various other capabilities which we won't be needing here.

3. Now extract the files in the tarfile using the command `gtar -zxvf discussion05.tgz`. This will create a subdirectory named `discussion05` which contains a file `functions.h`, a file `functions.c`, one or more files with names like `public01.c`, and a (hidden) file `.submit`.

   `functions.h` contains prototypes of the functions which you're supposed to complete.

   `functions.c` contains skeletons of the functions you're supposed to write. The comments before the prototypes explain what the functions are supposed to do.

   `public01.c` (and similar files) each contain a `main()` function that calls the functions you wrote in `functions.c`.

   `.submit` is what allows you to submit this exercise to the submit server.

4. Now write the functions!

   It doesn't matter how you write them, since you're not being graded on your programming style.

   The functions already have `return` statements, so the code will compile as is, before you even write them. This is just to allow you to write one or two functions and submit if you want to see your results on some of them before finishing all of them.

5. To keep things simple for this exercise, we won't use separate compilation or the make utility (both to be covered in class later). You can use the following command to compile the program:

   ```
   gcc -ansi -pedantic-errors -Wall -Werror -o public01.x functions.c public01.c
   ```

   The meaning of the options in this command are:

   - "`-ansi`" will cause the compiler to recognize only programs which follow the ANSI C standard.
   - "`-pedantic-errors`" will cause the compiler to strictly follow the ANSI standard, and reject programs which don't follow it with errors rather than warnings.
   - "`-Wall`" turns on all optional compiler warnings.
   - "`-Werror`" will cause the compiler to treat all warnings as errors.
   - "`-o public01.x`" will cause the compiler to use the filename "`public01.x`" for the executable version of the program (assuming there are no syntax errors in your code), rather than the name "`a.out`".

   Recall that in the tcsh command shell which your TerpConnect account is set up by default to use, the up–arrow key will cycle through earlier commands, which can then be re–entered, so you only need to type the above command once.

6. When you finish the first function (it should be very easy), run the first public test and check its results. To run a program in UNIX, just type the filename of the executable file ("`public01.x`" as produced by the compilation command above) as a command. If your function has any problems, re–edit `functions.c` and correct them.

7. When you want to compile the second test program (`public02.c`), just use the up–arrow key to retrieve the compilation command and use the left–arrow key to move back and change it to read (just the two '1's have been changed to '2's):

   ```
   gcc -ansi -pedantic-errors -Wall -Werror -o public02.x functions.c public02.c
   ```

8. Continue writing the remaining functions and compile and run all the tests. You can also add your own tests, since you may notice that the public tests don't test the functions in all of the situations that they should work for.

9. Submit your code using the "submit" command. Just run the command `submit` while you're located in the directory `discussion05`, and it'll send the necessary files in the directory to the submit server.

   (As long as you completed the steps in the earlier discussion section correctly the submit command will work for you, otherwise you will be running the OIT submit program, not the CMSC 212 submit program, so your functions won't be sent to the submit server.)

10. Log into the submit server linked to from the class webpage and verify that your functions produced the same on the public test programs as you got by running them by hand. If not, fix the functions and resubmit.

11. Even if your functions worked fine, submit them again at least once with an intentional syntax error, and at least once with an intentional semantic or logic error, so you can see what the results on the submit server would look like in all three cases.

12. This exercise has several secret tests. Their results will be visible tomorrow, 24 hours after the end of the last discussion section today, so log back in to the submit server then to see your results on them.

13. Even if you don't finish, be sure to submit at least once **before the end** of your discussion section in order to get credit for doing the exercise!