ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 11 **due: 10 p.m. on Monday 15 April**

Depths III

This week you enable saving and restoring the Depths game to a file on disk (on the Nexus, to the SD card). The objective for this week is for you to gain some experience with file I/O. Since you have an exam this week, the assignment is fairly light, requiring fewer than 70 lines of code. Implementing all of the challenges takes about 170 lines.

You should copy your `prog9.c` and `prog10.c` solutions from last week into this week's code and commit them before you start working. There is a short new function implemented in the `prog10.c` provided to you that you will need to either copy into last week's version of the file or rewrite.

Your tasks are to write one function that stores the state of the game from a data structure to a file and a second function that reads the state back from the file into a data structure.

A new burst screen has been added to the game. When a saved game exists, the user can elect to resume playing it or to start a new game. To see the complete game, and to obtain examples of the save file format necessary for some of the challenge problems, either install the Android demo version on your Nexus, or run the `prog11demo` in the distribution on an EWS lab machine.

As always, routine details such as how to obtain the code and how to hand in your program can be found in the specification for Program 1. Instructions on targeting and making the Linux and Android versions can be found in the specification for Program 4.

The Pieces

We are using the same package to allow cross-compilation on different platforms as well as support for the graphical user interface. This week's distribution is 225 files. As usual, you need only look at a few of those files—the rest serve to incorporate your code into the game on Android, Linux, Windows, or WebOS.

The files that you should examine include the header file and the source file that you must complete. You may also need to review the source and header files that you have used and developed in the last couple of weeks. Let's discuss each of these in a little more detail:

`jni/prog9.h`	This header file is identical to last week's.
`jni/prog9.c`	Write over with your `prog9.c` from last week, then commit.
`jni/prog10.h`	This header file has one new function declaration added, and the comments tell you how to implement it (in one line).
`jni/prog10.c`	Copy the new function from the bottom of the file into last week's `prog10.c` file, then replace the distribution version and commit. (You can also simply re-write the function, if you'd prefer.).
`jni/prog11.h`	The header file for this week. Defines and explains the data structures passed to your functions, and declares and explains the functions that you must write. Read the comments for some tips on things to think about before you start coding.
`jni/prog11.c`	The main source file for your code. Function headers for the two functions as well as some arrays of strings for challenge functions are provided to help you get started.

Testing is again left to you, but see the Testing section of this document for some tips. A gold version, `prog11gold`, is provided as part of the distribution. In programming studio, we will work on related file I/O examples, but not directly on this assignment. **Thus all parts of this assignment must be completed on your own.**

Details

You should read the descriptions of the functions in the header file and peruse the function headers in the source file before you begin coding. Refer to the specification for Program 9 for any information about the functions in `prog9.c`, and to the specification for Program 10 for information about the functions in `prog10.c`. The new `pointer_to_inventory` function simply returns the address of the variable `inventory` so that it can be accessed outside of `prog10.c`. Other than ensuring that `pointer_to_inventory` is present, all required work for this week is in `prog11.c`.

Here are function signatures for the two functions that you must write:

```
int32_t write_save_file (FILE* f, const save_data_t* save);
int32_t read_save_file (FILE* f, save_data_t* save);
```

You should start by writing `write_save_file`, which is called to write the game data from the `save` data structure to the stream `f`. The data structure includes information about the player, the world, the mixer, and the inventory. The structure is organized so that it can be written to a file with a few calls to the standard library. Use binary I/O for your first implementation.

Once you can save a game file, you may want to inspect the results, which will appear in the file `depthsSaveGame` in the `debug` directory on Linux, or under "Internal storage/Android/data/debug" on Android. (Normally, we would hide these files from user access, but these locations make them easier for you to debug.)

The second function, `read_save_file`, reads the data back from the stream `f` into the `save` structure. In this case, you must dynamically allocate space for the world contents as well as the inventory elements. Read the header file for details.

The challenge section again requires a different implementation, but you are encouraged to develop a binary strategy that works before tackling any challenges.

NOTE: The same binary format works on both my desktop and my Nexus, so you can probably move saved game files between the lab and your Android platform, but there is no guarantee that a binary file format is portable, as we discussed in class.

Challenges for Program 11

Here are some challenges for this week.

(8 points) Use an ASCII file format instead of a binary format. Print all of the values using `fprintf` and read them back in with `fscanf` or a combination of `fgets` and `sscanf`. Such a format is much more portable across platforms.

(12 points) (IMPLIES PREVIOUS CHALLENGE) Use a file format compatible with the gold version of the program. World contents are printed as strings, as are item types. Lines are prefixed with explanations of their meaning. The gold version ignores the exact content of these strings, but they must be present (and not contain spaces). The content/item names must match for `read_save_file` to succeed, of course. Arrays of item and content name strings have been provided for your use in `prog11.c`. Your code must work in both directions: that is, you must be able to save from your code and load into the gold version, and be able to save from the gold version and load into your code. You will need to look at files produced by the gold version to see the exact order of items used in the format.

Specifics

- Your code must be written in C and and must be contained in the `prog9.c`, `prog10.c`, and `prog11.c` files provided to you — we will NOT grade files with any other names.
- You must implement `write_save_file` and `read_save_file` correctly (as well as `pointer_to_inventory`).
- Your program must be able to save and restore arbitrary game state correctly for full credit.
- Your code must be well-commented. You may use either C-style (`/* can span multiple lines */`) or C++-style (`// comment to end of line`) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.

Testing

When you want to test your routines, we encourage you to use two copies of the program. Such methods might lead to loss of saved games, but you presumably don't care much about losing them in this case. Run both copies under `gdb`. Examine the exact contents of the data structure being written by your code. You can make sure that all of the data written are in the file by flushing it: type "`call fflush (f)`" while `gdb` is stopped in your `write_save_file` routine. Don't let the routine finish—that way you can continue to inspect the save structure after another copy of the program reads in the file. Examine the contents of the file with the `xxd` tool to make sure that you produced what you intended. Next, in another copy of the program, again running under `gdb`, walk through your `read_game_file` routine. Once it's done, compare the two `save_data_t` structures to see that you got the right answers.

Grading Rubric

Functionality (60%)

15% - `write_save_file` function works correctly
10% - `read_save_file` function works correctly for player and mixer
15% - `read_save_file` function works correctly for world
20% - `read_save_file` function works correctly for inventory

Style (20%)

10% - compilation generates no warnings (note: any warning means 0 points here)
5% - does not use global variables (file-scoped exist already)
5% - indentation and variable names are appropriate and reasonably meaningful (index variables can be single-letter)

Comments, clarity, and write-up (20%)

5% - introductory paragraph explaining what you did (even if it's just the required work)
15% - code is clear and well-commented

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.