ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 10 **due: 10 p.m. on Monday 8 April**

Depths II

This week you will extend the Depths game to a much larger world. In order to survive, our hero now needs to collect a wider variety of materials, and must combine them to create useful components that prolong the fuel and air supplies. Your task is to build functions utilizing a cyclic, doubly-linked list for the hero's inventory, and to supply the logic behind mixing items to make new ones.

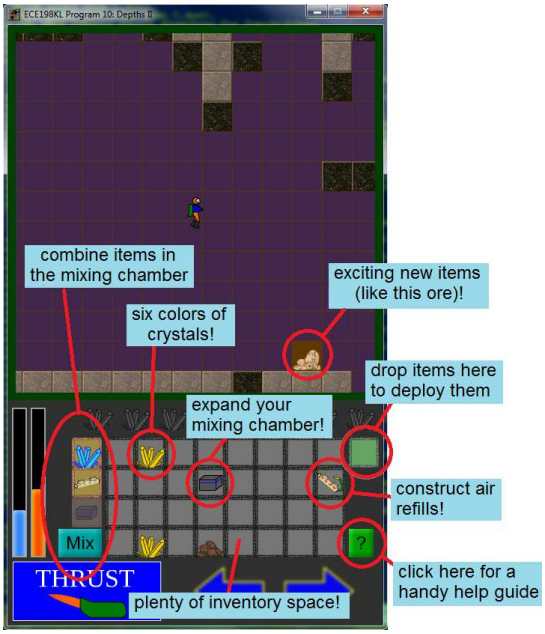
Please note that you should copy your `prog9.c` solution into this week's code and commit it before you start working. You may need to tweak your solution slightly (or significantly if you used numbers instead of enumerated symbolic names) to adjust to the new versions of the headers.

The game world is twelve times as large, but is still built using your functions from last week. Only six green crystals are included—others must be constructed from other materials for the player to complete the game.

Your tasks are to write functions that add and remove objects from the inventory, which is maintained using a cyclic, doubly-linked list, and to write the function that specifies the result of mixing up to three types of items together. For a challenge, you can instead read in and parse a file containing all of the valid mixtures, creating a data structure that you can then use to implement the mixing function. To see the complete game, including all challenge problems, either install the Android demo version on your Nexus, or run the `prog10demo` in the distribution on an EWS lab machine.

The objective for this week is for you to gain experience with linked lists. Next week, we will extend the game to save and restore the state of the world, inventory, and so forth. This week's challenge problems offer you some experience with I/O before that point. Including all challenges (and both implementations of the mixing function), this assignment takes less than 200 lines of code.

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. With a number of image files and more sample tests, this week's distribution is 220 files. As usual, you need only look at a few of those files—the rest serve to incorporate your code into the graphical game and to allow you to build the game as a text version on the lab machines or as a graphical version on Android, Linux, Windows, or WebOS.

The three files that you should examine include two header files and the source file that you must complete.

Let's discuss each of these in a little more detail:

`jni/prog9.h` This header file has been extended to include new content types for the game world. If you used symbolic names (as you should have), your solution to last week's program should continue to work. However, notice that I have added several boundary elements to separate types of contents (empty space, items, and types of stone). Some of the challenges problems from

last week, such as filling unreachable locations with stone, will require a little tweaking to replace other types of unreachable items.

`jni/prog9.c` Write over with your `prog9.c` from last week, then commit.

`jni/prog10.h` This header file provides enumerated names for items, defines the dimensions of the inventory as well as the structure for one list element (an `item_t`), and includes declarations and descriptions of the functions that you must write for this assignment.

`jni/prog10.c` The main source file for your code. Function headers for all regular and challenge functions are provided to help you get started. The sentinel structure for the player's inventory, called `inventory`, is also already declared and initialized (be sure to look at how it is done).

Testing is again left to you. A gold version, `prog10gold`, is provided as part of the distribution. In programming studio, we will do a mock exam for next week's midterm. **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`. All required work for this week is in `prog10.c`.

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

```
int32_t     add_object (item_type_t type, int32_t* xptr,
                        int32_t* yptr);
item_type_t get_object (int32_t x, int32_t y);
int32_t     place_object (item_type_t type, int32_t x,
                          int32_t y);
item_type_t perform_mix (item_type_t one, item_type_t two,
                         item_type_t three);
```

You should start by writing `add_object`, which is called to add an object to any free location in the inventory. The organization of the linked list (sorted or not sorted) and use of additional data structures (for example, creation of an array that maps the 8×4 inventory to `item_t` structures) is left to your discretion. You can also simply search the list for an unused space. See the header and source files for details on the exact behavior of the function. Once you have implemented the function, the player will be able to pick up items from the world by touching them.

Next, complete `get_object`, which removes an object from a specific location in the inventory. Again, see the files for further detail. The function is called when the player tries to drag an item from the inventory, so once you have the function working, you will be able to move items around (but not control where they go in the inventory).

To complete your linked list implementation, write `place_object`, which attempts to place a new item into the inventory at a specific location. Once all three of these functions are complete, you will be able to move items around freely.

Finally, implement the `perform_mix` function, which controls the logic of creating new items from sets of items. This function is called when the player presses the "Mix" button. Be sure to read the challenge section on the next page, as a different implementation is required with the challenge. Note that the items passed to the function **are not sorted**. You may wish to sort them before you make any comparisons. Also, the function can be called with all mixing slots empty, in which case all three input values are `ITEM_NONE`, and `ITEM_NONE` should be returned. Otherwise, a complete list of allowed mixtures and results can be found in the game (the "Mixing Guide" page) and in the `mixtures` file provided for the challenge. Other non-empty mixtures should result in `ITEM_DUST`.

Challenges for Program 10

Here are some challenges for this week. You can find the function signatures in the header or the source file. You can test correct behavior by comparing with the lab demo.

(5 points) Implement `find_item_number`, which translates a string into an item number. An array of strings has been provided to you already.

(15 points) (REQUIRES PREVIOUS CHALLENGE) Implement `read_mixtures`, which reads and parses the mixtures file. You will need to copy `mixtures` into the `debug` directory on Linux, or (via USB) into "Internal storage/Android/data/debug" on Android. You must define file-scope variables to hold the results of your function in such a way that they can be used by `perform_mix`, and must rewrite `perform_mix` to make use of them instead of hardcoded mixture rules.

Specifics

- Your code must be written in C and and must be contained in the `prog9.c` and `prog10.c` files provided to you — we will NOT grade files with any other names.
- You must implement `add_object`, `get_object`, `place_object`, and `perform_mix` correctly.
- Your routines' changes to the game world and outputs must match the gold version's exactly 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.

Grading Rubric

Functionality (65%)
- 15% - `add_object` function works correctly
- 15% - `get_object` function works correctly
- 10% - `place_object` function works correctly
- 25% - `perform_mix` function works correctly

Style (15%)
- 5% - 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.