ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 9 **due: 10 p.m. on Monday 1 April**

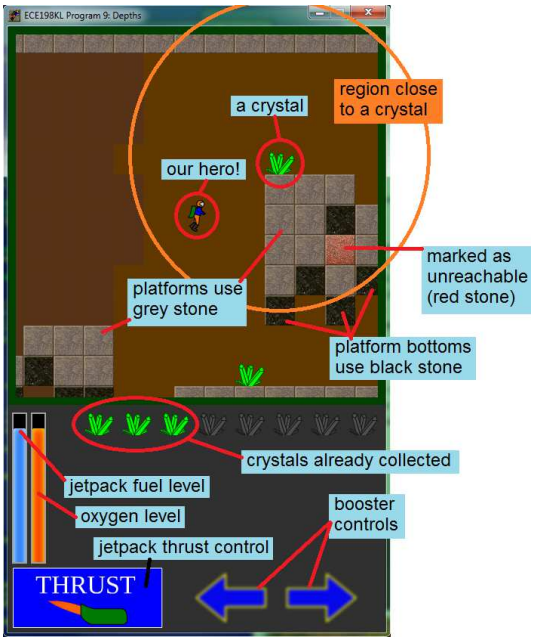
Depths

Your ship has crash landed on an alien world, and you must collect enough energy crystals before your oxygen runs out! Well...in the game, anyway. Your task this week is to implement two recursive functions used as part of the game just mentioned.

The game world is built randomly using one of your functions, then eight crystals are placed into it. Your second function determines how many of these crystals can be reached by the player. Extra credit functions that you can choose to write mark unreachable regions with a different type of stone and create halos around crystals so as to make them easier to find. To see the complete game, including all extra credit, either install the Android demo version on your Nexus, or run the prog9demo in the distribution on an EWS lab machine.

The objective for this week is for you to gain some experience with recursion. The recursive functions operate on a two-dimensional world defined as a one-dimensional array of enumerated values, which will also help to reinforce your familiarity with array layouts and usage in C. Including all extra credit, this assignment takes less than 100 lines of code, so it should be fairly straightforward.

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 over 170 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 two files that you should examine are a header file 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 provides function declarations and descriptions of the functions that you must write for this assignment, along with an enumeration of world space types and a couple of other functions already provided to you.

`jni/prog9.c` The main source file for your code. Function headers for all regular and challenge functions are provided to help you get started. Wrappers for your recursive functions and some utility functions are already in the file. Note that the recursive functions are defined locally to this file, so only the wrapper function signatures appear in the header file.

Testing is left to you, but we may provide more help during the week. A gold version, `prog9gold`, is provided as part of the distribution—if you use the same random number seed and follow the specification exactly, you should produce the same world. In programming studio, we will do related exercises but not work 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. The `set_seed` routine is a close variant of the one that we used with Program 5, and has been provided for you. Note that the use of pseudo-random number generators is easier with known sequences, since you will be able to reproduce bugs exposed by specific world structures (for example). The same seed entry window is used in this program as was used for Program 5, and the mechanism is more or less identical. Once you have debugged your code (but **not in the version that you submit for grading**), feel free to change the random seed to be, for example, time-based (`srand (time (NULL));`), so as to provide a more realistic feeling of randomness.

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

```
static void platform (int32_t x, int32_t y, int32_t chance);
static int32_t reachable (int32_t x, int32_t y);
```

The `platform` routine recursively creates a platform of stone in the game world. The `reachable` routine returns the number of crystals reachable from the given point in the world without passing through either stone nor through previously visited spaces. Only vertical and horizontal moves are allowed when calculating reachability. The details of the approach taken to count reachable crystals are left to you, provided that your function obtains the correct answer.

Wrapper functions receive and record the world contents, width, and height in file scope variables. A wrapper for the `reachable` function also creates an array of integers (initialized to 0) that you can use to record which locations have already been seen in `reachable`.

The world is cylindrical: the rightmost edge connects to the leftmost edge. The top and bottom of the world do not connect. All of your code must handle this aspect accurately. You may want to review modulo arithmetic in C to ensure that you are able to easily implement the functions correctly.

You must use the algorithm described below for generating platforms. When recursing, you should only consider a direction—**and should only check the percentage chance!**—if another space exists in that direction and that space is currently empty.

Step 1: You need to do something to prevent infinite recusion. Be aware that the file-scoped `worldSeen` array is not defined in this function.
Step 2: Recurse downward with percentage chance `(chance - 25)`, passing `(chance - 15)` as the new `chance` argument.
Step 3: Recurse to the left with percentage chance `(chance - 25)`, passing `chance` as the new `chance` argument.
Step 4: Recurse to the right with percentage chance `(chance - 25)`, passing `chance` as the new `chance` argument.
Step 5: If the square below this one exists and is empty, fill this space with WORLD_STONE_2 (black stone). Otherwise, fill this space with WORLD_STONE_1 (grey stone).

All random decisions in the `platform` function must be made by calling the `check_percentage` provided to you in `prog9.c`. Be sure not to call `srand` nor `rand` on your own. Calling either of these disrupts the sequence of random numbers and will cause your game worlds to differ from ours. Please also be aware that **the pseudo-random number generators on Linux and Android are different**, so the same random seed will produce different solution codes on the lab machine and your Android device.

Challenges for Program 9

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.

(6 points) Implement `replace_unreachable`, which replaces every location in the world that neither contains stone of any kind nor is reachable with WORLD_STONE_3.

(10 points) Implement `mark_as_cold`, which marks spaces in the world that do not contain either stone nor crystal as empty, cold, colder, or coldest, depending on how far they are from the nearest crystal. Spaces that are within distance 5 of a crystal should be marked as WORLD_EMPTY. Spaces that are not within 5 but are within 10 should be marked as WORLD_COLD. Spaces that are not within 10 but are within 15 should be marked as WORLD_COLDER. Remaining spaces should be marked as WORLD_COLDEST.

(4 points) (REQUIRES PREVIOUS CHALLENGE) Be sure that your `mark_as_cold` function can be called again—the function is called every time a crystal is collected, so you cannot assume that all spaces containing neither stone nor crystal are marked as WORLD_EMPTY.

Specifics

- Your code must be written in C and and must be contained in the `prog9.c` file provided to you — we will NOT grade files with any other name.
- You must implement `platform` and `reachable` correctly.
- You may NOT change the wrapper functions that call these functions.
- 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.

Building and Testing

A gold version has been provided for visual comparisons. Run the two versions side by side with the same random seed and make sure that the world appears identical.

Grading Rubric

Functionality (60%)
- 30% - `platform` function works correctly
- 30% - `reachable` function works correctly

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.