CSc352 (Summer03) Homework 6

Start: 07/21/03

Due: 08/01/03 (9pm)

Turnin ID: `cs352_assg6`

0. Background Reading
   - Linked list data structure (http://cslibrary.stanford.edu/103/LinkedListBasics.pdf)
   - Function pointers (Class notes and textbook)
   - Library building and using (Class materials)
   - [Optional] X Windows Motif Programming (http://docs.linux.cz/motif/motif.html)
1. Simple Ordered Linked List Library

   You are to build a simple Ordered Linked List Library. The structure of the list node is defined in the provided header file as the following.

```
struct __node {
    void *data;
    struct __node *next;
};
typedef struct __node NODE;
```

   "data" is a void pointer pointing to the data.

   "next" is a pointer pointing to the next node on the linked list or NULL if there isn't the next node.

   You are to implement the following functions in the library:

   - int list_init(int (*compare_arg)(const void *, const void *),
     void (*print_arg)(const void *));

     Initialize the linked list and set it to be empty. **compare_arg** is the function pointer pointing to the user specified compare function that compares two data when ordering the data in the linked list in function list_add(). The compare function returns 3 types of values:

     1. Negative return value: means the data of the 1st parameter is less than that of the 2nd parameter of the compare function.
     2. Zero: means the two data are equal.
     3. Positive return value: means the data of the 1st parameter is greater than that of the 2nd parameter of the compare function.

     print_arg is the function pointer pointing to the user specified printing function that print out the data of a node when list_print() is called.

     By calling list_init(), the linked list is initialized. But if after that, list_int() is called a second time, it will first clean up the originally initialized linked list and then initialize a new empty linked list. By "clean up", I mean it has to traverse the old linked list and free all the nodes.

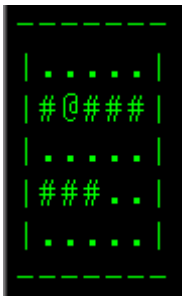
     If the initialization is successful, it returns **ERR_OK**. It returns **ERR_MEM** when there are any memory related errors (e.g. malloc, etc.). It returns **ERR_FUNC** when any of the function pointers passed to list_init() is NULL.

- int list_add(void *data);
  Add the data pointed by the pointer "**data**" into the linked list and always keep the ascending order specified by the compare function provided by the user when list_init() is called. If there are already some nodes having data that are equal to the new data from the compare function's point of view, insert the new data before all of them.
  If the function list_add() is called before the function list_init(), it returns ERR_INIT. It also returns ERR_MEM if some memory related errors occure (e.g. malloc failure). Otherwise, it returns ERR_OK.
- int list_del(void *data);
  Traverse the whole linked list. For each node **n**, compare the data pointed by **n->data** and the data pointed by the parameter **data** with the compare function specified by the user when initializing the linked list. If equal, then delete the node from the linked list. If there are multiple nodes that satisfy the equality comparison, delete them all.
  If the function list_del() is called before the function list_init(), it returns ERR_INIT. When there is no node that can be deleted, return ERR_NOTFOUND. Otherwise return ERR_OK.
- int list_print();
  Traverse the whole linked list and print the data of each node with the print function specified when calling list_init().
  If the function list_print() is called before the function list_init(), it returns ERR_INIT. Otherwise it returns ERR_OK.
- void list_err(int ret, char *msg);
  Print the error message according to the error code and user specified message. This function is provided in list_err.c.

A user sample program **list_test.c** is provided. It is to give you an idea of how other applications may make use of your linked list library. After building your linked list library, you may compile **list_test.c** and link your library with it to see if **list_test.c** is working correctly.

2. Eating Beans Game

On the map there are stones ('#'), beans ('.') and a kid ('@'). The kid can only go upward, downward, leftward and rightward. The kid can't go across or step onto stones. The kid can't go beyond the boundary of the map. The goal of the game is to move the kid so that he/she can eat all the beans.



The program gets, from **stdin**, the map and a sequence of commands ('u' means move upward; 'd' means move downward; 'l' means move leftward; 'r' means move rightward) of the kid's movement. Then according to the movement commands, the program simulates the whole process of the kid's movement. Either when all the beans are eaten up by the kid or all

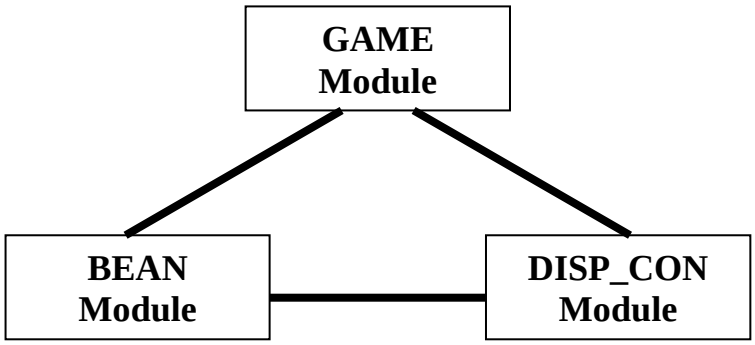
the movement commands are finished, the program exits. If the beans are all eaten up, the return value of the program is 0, otherwise the program returns 1.

The input map is the same format as the matrix we did in assignment 3. You can assume the input data are always valid: (1) size is always between 1 and 20; (2) map is given in square shape; (3) all characters on the map are legal ones. Etc.

For any movement command, if the target is a stone or out of the map boundary, this command is ignored and the kid does nothing for this command. Then the program continues with the next command if any.

Right after the program reads in the map, it prints out the map to the **stdout**. After each movement command, the program prints out the resulting map due to the command.

You need to implement the program with a modularized design as below:



The GAME Module is just a top level calling functions in the other 2 modules. This module is given in the file **game.c**. You have to use it in your program.

The BEAN Module is responsible for maintaining the data structure representation, read the map from the **stdin**, modify the map data structure according to the movement command, judge if the game wins (all beans are eaten up), etc. All the functions (and the explanation) you may need to implement in this module are declared in **bean.h**.

The DISP_CON Module is responsible for the UI on the console. It read the movement command from the **stdin**, call the correspondent movement function in BEAN Module according to the command it reads (for example, it calls bean_move_up() when reads in a command 'u'), print the resulting map after executing the commands, etc. All the functions (and the explanation) you may need to implement in this module are declared in **disp.h**.

In this program, sharing global variables across files are discouraged. Modules only talk to each other by calling certain functions. For example, in DISP_CON Module, when you need to print the map, you get the map data by calling BEAN Module's function bean_get_map() instead of reading a global variable shared by both modules. In order to protect the variables, you need to declare global variables that are only used within a .c file to be static.

The following is a typical flow of the program. The GAME Module calls BEAN Module's bean_map_init() to read the map from **stdin** and put in a char array. Then GAME Module calls the DISP_CON Module's disp_main_loop(). In the main loop, DISP_CON Module keeps getting the commands from the **stdin** and according to the command call a certain bean_move_xxx() function to make the movement and update the array storing the map. After that DISP_CON get the newly updated map by calling bean_get_map() and print out the new map to the **stdout**. At the end, DISP_CON start over the next iteration of the main

loop waiting for the next command from the **stdin**.

This console version of the game should be compiled into an executable called **game_con**.To grade this program, the grading script will feed a file containing the input map and movement commands into your program through **stdin**. Your program will print out the initial map and the intermediate maps after each movement command. The grading script will check your output and the return value.

There are two sample input files provided: **sample_input_bean** and **sample_input_bean_lose**. You may have a try with them to get an idea of how the program works.

The following are all **possible** errors you may experience. The method to handle them is simply to print out an error message to the **stderr** and **exit(1)**.

Situation	Message to be printed out
Malloc failure	"Out of memory\n"
Map pointer is NULL	"NULL map\n"

3. Extra Credit (20 points beyond the original total score of 100 points)

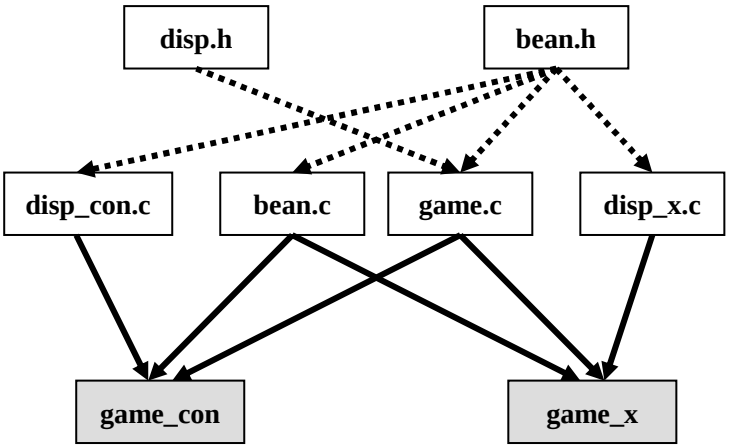
You implemented the "Eating Beans" game above in a modularized way. So now you can easily replace one of the modules without affecting other modules.

The original game is a console game. The goal of this extra credit problem is to provide X-Window GUI for this game. A premise requirement is that you can ONLY replace the DISP_CON Module with a new module called DISP_X and you should NOT touch any other modules.

The new DISP_X Module is responsible for the X-Window GUI. It creates a frame window and draws the map in the window. Each cell of the map is represented by a "push button" widget. The movement commands in the input file are ignored here. Instead, you click the cell immediately next to the kid to control the kid to move. If the cell you click is not a stone, then the kid would move to the cell you clicked. If you click on the stone or some cell that is not immediately next to the kid, nothing will be done. Right after the last bean is eaten by the kid, the program should terminate. We won't check the program return value here.

This X version of the game should be compiled into an executable called **game_x**.This program will be graded manually. We will try your program and play the game you implement to see if it works correctly.

File structures:



You may make use of the class samples and the samples on the reference webpage:
http://docs.linux.cz/motif/motif.html
There are the following ways of developing and debugging your X application:

1. Use the Debian Linux machines in GS-228
2. Use the X-win32 X server on the Windows machines in GS-228 (You need to set the SSH tunneling for X)
   (http://www.cs.arizona.edu/computer.help/policy/windows_tunneling_with_ssh.html)
3. Download an X Server for Windows from the university site-license website and install on your own machine at home. Then use it the similar way as X-win32.
   (https://sitelicense.arizona.edu/excursion/excursn.shtml)

4. Packaging

You are to put all your programs and Makefile in a directory called **hw6**. Then use the **tar** and **gzip** program we introduced in the previous assignment to package the directory hw6 into **hw6.tar.gz**.

When running command “make” in your **hw6** directory, executable **list_test** and **game_con** and library **liblist.a** should be produced (if you did the extra credit, the executable **game_x** should also be produced).

When running command “make clean” in your **hw6** directory. All the executables and the intermediate files (e.g. .o files, etc.) generated by the “make” command should be removed so that the directory is clean as before.