Reduced ELIZA Implementation

Tuesday 11th June 2013

2-00pm – 5-00pm
THREE HOURS
(including 15 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.

The maximum total is 25.

Credit will be awarded throughout for clarity, conciseness, useful commenting, appropriate use of assertions, and error checking.

Important note: THREE MARKS will be deducted from solutions that do not compile. You should comment out any code which you cannot get to compile.

ELIZA

`ELIZA` was a computer program created between 1964 and 1966 by Joseph Weizenbaum at MIT. Using primitive natural language processing, it attempted to emulate the effect of talking to a *Rogerian* psychiatrist.

In Rogerian psychotherapy (also known as person-centred therapy) the therapist provides a warm and understanding presence whilst allowing the client to decide what is wrong with themselves and how they can change it.

Weizenbaum only intended `ELIZA` to be a parody, yet the reaction to it caused Weizenbaum to publish articles and papers on `ELIZA` demonstrating the simplicity of its inner workings.

Your Task

You have been provided with a partial implementation of an `ELIZA`-like program. You will be asked to complete the program. This will consist of mainly string processing, data structure manipulation, and memory allocation.

Once the parts are complete, you should be able to initiate basic conversations as follows:

```
ELIZA> How do you do.  Please tell me your problem.
USER> I am taking a Lexis test.
ELIZA> Do you enjoy when you are taking a lexis test ?
USER> Why do you ask?
ELIZA> You're not really talking about me -- are you ?
USER> Yes, I am.
ELIZA> Do you believe it is normal to be  ?
USER>
etc.
```

It should be apparent that ELIZA's behaviour consists of a number of rules that decompose statements from the user, and respond with a corresponding message (possibly containing placeholders).

The Supplied Skeleton

In the `eliza` directory in your Lexis home directory you will find a supplied skeleton consisting of a number of files. However, aside from the final question, it will *only be necessary for you to modify the files* `string_utils.c` *and* `map.c`.

A brief description of the rest of the files in the skeleton follows:

Makefile The makefile for your project.

eliza.c The main program.

eliza_state.h, eliza_state.c Header and implementation for the rule database of the ELIZA engine.

error_codes.h Header file containing error codes used in certain places.

fwd.h Header file containing forward declarations of types.

list.h, list.c Header file and implementation for a list ADT.

map.h, map.c Header file and implementation for a map ADT.

parser.h, parser.c Header file and implementation for the ELIZA rule parser.

rule.h, rule.c Header file and implementation for rule searching/application functions.

string_utils.h, string_utils.c Header file and implementation for string utility methods.

script Rules used by the ELIZA engine.

You may wish to look at the file `script` which contains the rule database the executable will use. Many of the data structures built in the `eliza` executable hold representations of the data from this file.

The provided Makefile will build the `eliza` executable by default. A `clean` target is also provided. It should not be necessary to modify the Makefile.

You can build your skeleton by running `make`

```
$ make
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o eliza.o eliza.c
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o list.o list.c
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o parser.o parser.c
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o string_utils.o string_utils.c
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o rule.o rule.c
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o map.o map.c
cc -std=c99 -Wall -pedantic -Werror -g -D_POSIX_SOURCE -c -o eliza_state.o eliza_state.c
cc   eliza.o list.o parser.o string_utils.o rule.o map.o eliza_state.o  -lpcreposix -o eliza
```

Once built, the `eliza` executable can be run as follows:

```
$ ./eliza
```

One of your tasks later will enable the support of a `quit` command. However, regardless of the state of implementation, you should be able to terminate the program by pressing `CTRL+C`.

Part I - String Processing

All code for this part should be written in the file `string_utils.c`. You should pay special attention to ensuring that you do not read or write outside valid memory, and that any strings you create are null-terminated appropriately. Use `assert()` from `assert.h` where appropriate to check for invalid parameters passed to your functions.

If an internal memory allocation routine generates a `NULL` pointer or you encounter another system error, use `perror()` to report the name of the your function and the error, then call `exit()` with a non-successful error code.

1. Write a function:

   ```
   char *empty_string(void);
   ```

   This function should return a new, heap-allocated, null-terminated string of length 0. As before, it should be possible to free the memory of this string using the `free()` function from the C standard library.

   [2 Marks]

2. Write a function:

   ```
   char *clone(const char *str);
   ```

   This function should return a heap-allocated copy of the given string `str`. The original string `str` should not be modified by this function. It should be possible to free the memory of the returned string using the `free()` function from the C standard library.

   [4 Marks]

3. Write a function:

   ```
   char *push_string(char *current, const char *append)
   ```

   This function should return a heap-allocated string that is the concatenation of `current`, followed by `append`.

   This function takes responsibility for the memory management of `current`, i.e. by the end of this function `current` should either have been freed, or used as the return value. Hint: If you choose to use `current` as a return value, you may find the `realloc()` function helpful.

   `append` should not be altered or freed by this function.

   [5 Marks]

An example use of `empty_string()` and `push_string()` follows:

```
char *str = empty_string();
str = push_string(str, "This ");
str = push_string(str, "is ");
str = push_string(str, "a sequence of ");
str = push_string(str, "string concatenations!");
printf("%s\n", str);
free(str);
```

If you have implemented your functions correctly, the above should not leak memory and print the message:

```
This is a sequence of string concatenations!
```

After implementing the string handling functions, you should be able to execute the `ELIZA` program and receive a prompt. You will receive warnings from the functions yet to be implemented, but you should still be able to type statements into the program and receive responses.

Part II - Dynamic Data Structures

All code for this part should be written in the file `map.c`. The `ELIZA` program makes extensive use of maps and lists. You are required to complete the map implementation as follows:

1. The files `map.c` and `map.h` contain a partial map implementation. The `struct map` type represents a map from strings (`char*`) to void pointers (`void*`). The map is represented using a sorted binary tree, as you have seen in the lectures.

   Write the function:

   ```
   struct map_node *map_insert_internal(struct map_node *node,
     const char *key, void *value, int *result);
   ```

   This is a helper function used by primary function `map_insert()`. It is both passed, and returns, pointers to a `struct map_node`.

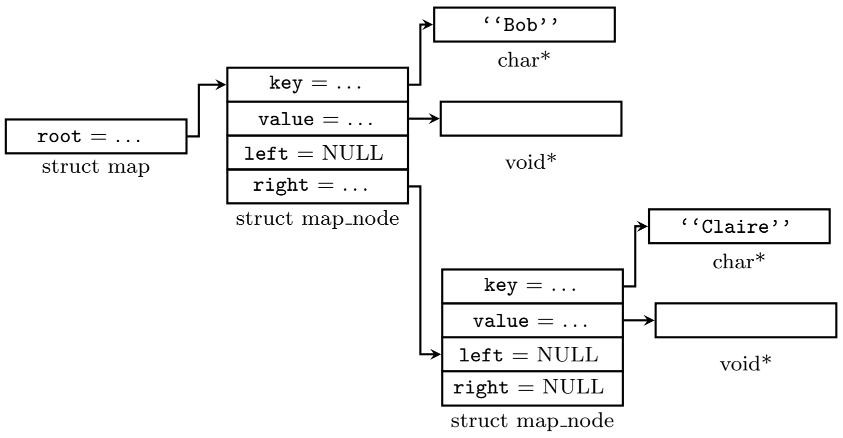
   `struct map` is defined in `map.h` as follows:

   ```
   struct map
   {
     struct map_node *root;
   };
   ```

   `struct map_node` is defined in `map.c` as follows:

   ```
   struct map_node
   {
     char *key;
     void *value;
     struct map_node *left;
     struct map_node *right;
   };
   ```

   Diagrammatically, a map with two elements looks like this:



In a call to `map_insert_internal(node, key, value, result)`:

- `node` will be a pointer to the root node of the current map. If the map is empty, `node` will be equal to `NULL`.
- `key` is the new key to be inserted.
- `value` is the value the new key should be mapped to. If `key` is already found to be in the tree, `value` is left unused and not inserted.
- `*result` holds the result of the insertion. If the key and value were inserted `*result` should be set to 1, otherwise `*result` should be set to 0.
- The return value of the function should be the new root node of the map.

You have been provided the functions `map_alloc_node()` and `map_destroy_node()` for memory allocation of map nodes. Use them if/where necessary.

You *may not* assume that the lifetime of `key` extends beyond the call to `map_insert()`/`map_insert_internal()`. e.g. it may be a stack allocated value, or a heap allocated value that may be subsequently overwritten or freed by the caller. Use the `clone()` function you defined earlier to copy to key where necessary. You may also find the standard library function `strcmp()` useful when implementing this function.

[**7 Marks**]

After you have completed this section, additional `eliza` functionality should become active. In particular, it should now be possible to quit by the session by typing any of the keywords *bye*, *done*, *goodbye*, *exit*, or *quit*. If it is not, your map insertion function is most likely incorrect.

`eliza` should now recognise previously misunderstood syntax. For example, after completing part II, the following exchange can occur:

```
USER> Are you a machine?
ELIZA> Don't you think computers can help people ?
```

This is due to the fact that maps are used by the engine to rewrite words to synonyms, e.g. machine to computer. Without map insertion, `eliza` may attempt a response using the word "machine", but will not mention computers.

Part III - Function Pointers and Memory Leaks

The skeleton code you have been given has a couple of memory leaks. Your task in this part is to fix them.

1. In `map.c`, complete the function:

   `void map_apply_elems(struct map *m, void (*function)(void *))`

   This function is called by existing code with a pointer to the `free()` function as a parameter, which has the effect of freeing the memory used by the values in the map.

Implement this function so that it now takes the supplied `function` pointer and applies it to every value in the map `m`.

You may wish to declare and use a helper method, in which case a suitable prototype has been provided at the top of the file with the name `map_apply_elems_internal()`.

[5 Marks]

2. The skeleton has one more memory leak. Use whatever means you prefer to locate the leak and fix the broken code. We suggest you use Valgrind's memcheck which we looked at in the final lecture. We suggest the following invocation line:

   ```
   $ valgrind --tool=memcheck --leak-check=full ./eliza
   ```

   Place a comment by any change you make containing the keyword `LEAKFIX` so we can easily locate your change.

   [2 Marks]