CSC190: Computer Algorithms and Data Structures
Lab 3
Assigned: Feb 2, 2015; Due: Feb 9, 2015 @ 10:00 a.m.

1 Objectives

In this lab, you will create an array of structures that store information about local variables in recursive function calls. We have provided files that you must use as a starting point to complete this lab. You will download the contents in the folder `Lab3` which contains two folders (`code` and `expOutput`) into your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `lab3.c` and include necessary libraries in `lab3.h` as required for implementation. `main.c` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `main.c` file to test all your implementations. Folder `expOutput` contains outputs expected for the supplied `main.c` file. Note that we will **NOT** use the same `main.c` file for grading your lab. Do **NOT** change the names of these files or functions.

2 Grading

It is **IMPORTANT** that you follow all instructions provided in this lab very closely. Otherwise, you will lose a *significant* amount of marks as this lab is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this lab (total of 20 points):

- Successful compilation of all program files with no memory leaks i.e. the following command results in no errors (2 points):
  ```
  gcc main.c lab3.c -o run
  valgrind --quiet --leak-check=full --track-origins=yes ./run 1
  valgrind --quiet --leak-check=full --track-origins=yes ./run 2
  ```
- Output from Part 1 exactly matches expected output (5 points)
- Output from Part 2 exactly matches expected output (5 points)
- Code content (8 points)

Sample expected outputs are provided in folder `expOutput`. We will test your program with a set of completely different data files. Late submissions will **NOT** be accepted.

3 Introduction

Consider the following recursive function implementation that computes the result of raising a number (`base`) to an exponent (`exp`). For example, the expression 2^7 can be can be computed via the function call `power(base, exp)` where $base = 2$ and $exp = 7$:

```
int power(int base, int exp)
{
    int res=0;
    if (exp==0) {
        res=1;
    }
    else{
        res=power(base, exp-1)*base;
    }
    return res;
}
```

When the function call `power(2, 7)` is made, there are 8 calls to the function `power` (including the first call) as follows:

$$power(2,7)->power(2,6)->power(2,5)->power(2,4)->power(2,3)->power(2,2)->power(2,1)->power(2,0)$$

At each recursive call, two local variables `exp` and `res` are created. `exp` denotes the current exponent being evaluated and `res` waits for the result from the local recursive call. The value returned by the local recursive call updates `res`. For instance, the function `power(2,4)` is the fourth recursive call. Its local variable `exp` will take the value 4. The other local variable `res` waits for the result from `power(2,3)` which is 8. After the function returns, `res` is updated to 16. 16 is then returned to the function `power(2,3)` which had made the original recursive call to `power(2,4)` Hence, every recursive function call has three distinguishing attributes and these are the values of `res` and `exp` and the recursive call number.

In this lab, you will trace recursive function calls that are made when `power(base, exp)` is called for a specific instances of `base` and `exp` in an array of structures. Each structure represents a recursive call and will contain the members `currentValueRes`, `recursiveCallNumber` and `exp` which are all of type integer. The member `recursiveCallNumber` will hold the recursive call number of the function call. For instance, in the example above, `power(2,4)` is the fourth recursive call. The member `exp` will hold the current exponent being evaluated (i.e. `exp`=4 for `power(2,4)`). Before the function terminates, the final value stored in `res` is recorded in `currentValueRes` (i.e. `currentValueRes` =16 for `power(2,4)`).

Part 1: Initializing, Printing and Freeing the Structure Array

In this part, first you will define a general structure prototype with the aforementioned integer members `currentValueRes`, `recursiveCallNumber` and `exp`. This structure is to be named `functionCall` . Define this structure in the `lab3.h` file. Then, you will implement:

- `struct functionCall * createStructArray(int n);`
- `void assignValues(struct functionCall * array, int index, int currentValueRes, int recursiveCallNumber, int exp);`
- `void initArray(struct functionCall * array, int n);`
- `void printArray(struct functionCall * array, int n);`
- `void freeArray(struct functionCall * array);`

In the `createStructArray` function, an array of structure elements of size `n` will be dynamically created and the pointer to this array will be returned by the function. Elements of this array will be of type `struct functionCall`. In the `assignValues` function, values `currentValueRes`, `recursiveCallNumber` and `exp` are passed as arguments. This function then assigns these values to appropriate members of the structure element located at position `index` of the structure array. The pointer to the array and index of the structure element to be assigned (`array` and `index`) are passed as arguments to the function. In the `initArray` function, you will initialize all members of each structure element in an `array` containing `n` elements to 0. Into this function, a single pointer `array` and the size of the array `n` are passed as arguments. `printArray` function will print out the data stored in members of every structure element in the array to the console. Contents of various structure elements are separated into different lines. Contents of members in a structure element are printed in a single line and each member of the same structure element is separated by a whitespace. Members are printed in the following order: `currentValueRes`, `recursiveCallNumber` and `exp`. Please refer to the files in the `expOutput` folder for output formatting examples. The `freeArray` function will free the dynamically allocated array pointed to by the single pointer `array` passed as an argument to the function. Test your implementation via:

- `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`

Ensure that your output matches the sample output file `P1.txt` located in the `expOutput` folder.

Part 2: Recording Recursive Call Parameters

In this part, you will implement the function:

- `void recordRecursiveCalls(struct functionCall * array, int exp, int base);`

The function `recordRecursiveCalls` will compute the value that will be stored in the variable `res` which is local to the i^{th} recursive call when the exponent is `exp`. The corresponding members of the structure element representing the i^{th} recursive call are then updated. For instance, for the structure element representing the function call `power(2,4)`, values stored in this structure element are `exp`=4, `currentValueRes` =16 and `recursiveCallNumber` =4. This is repeated for all elements in the array. Test your implementation via:

- `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`

Ensure that your output matches the sample output file `P2.txt` located in the `expOutput` folder.

4 Code Submission

For this lab, if your submit via `git`, you will receive **bonus** points. Otherwise you can submit through the `submitcsc190s` command on your ECF machine (no bonus points). Ensure that you submit through only one venue.

4.1 Submission through `Git`

Once you have completed this lab, you will submit your work by:

- Log onto your ECF account
- Browse into the directory you had cloned in Lab 0 (i.e. `cd ~/UTORID/`)
- Create a folder named `Lab3` (i.e. `mkdir Lab3`) in that cloned directory
- Ensure that your code compiles in the ECF environment
- Copy all your completed code (`lab3.h` and `lab3.c`) into the Lab3 folder
- Browse into the `~/UTORID/` directory
- Add all files in the Lab3 folder (i.e. `git add *`)
- Commit all files that have modified in the Lab 3 folder (i.e. `git commit -m "adding lab files"`)
- Push all changes committed to the `git` server (i.e. `git push origin master`)

4.2 Submission through `submitcsc190s`

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory containing your completed code (`lab3.h` and `lab3.c`)
- Submit by issuing the command:
  `submitcsc190s 4 lab3.h lab3.c`

4.3 Checklist

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names
- Your code compiles without error in the ECF environment (if it does not compile then your maximum grade will be 3/20)
- Do not resubmit any files in Lab 3 after the deadline (otherwise we will consider your work to be a late submission)