CSC172 LAB 8

SELF-REFERENCE IN C

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Linked Lists in C

The use of pointers to recursive data structures in the C programming language can seem complex at first glance. However, this area constitutes one of the most beautiful aspects of the C language syntax for those who learn to appreciate it.

1. Begin this lab by implementing a simple C program with a “main” function. Include the following conventional definitions to make your code more elegant and readable.

```
#define TRUE 1
#define FALSE 0
typedef int BOOLEAN;
```

2. Before your main() function, you should declare the Node structure as follows.

```
struct Node {
    int value;
    struct Node *next;
};
```

   Once this structure is declared, you can make a list in your main method simply by declaring a pointer to a Node.

3. It is important to be able to add elements to a list. The insert() function needs to be able to modify the list (i.e. the pointer to the Node). In order to do this we need to send a pointer to the list (i.e. a pointer to a pointer to a Node) as a parameter to the insert() function. Doing so would require us to prototype the insert function as:

   ```
   void insert(int x, struct Node **pL);
   ```

   Then, in the main() method, we could add some code to fill the list.

   ```
   for (i = 1; i < 20; i += 2)
       insert(i,&L);
   ```

   Implement the insert function as follows:

   1. If the pointer to the list `(*pL)` is NULL, then
      1. Allocate memory the size of a Node with `malloc()`.
      2. Set the node to refer to this memory location
      3. Set the *value* variable in the Node to the parameter *x*.
      4. Set the *next* pointer in the Node to NULL.

   2. Otherwise, if the list is not empty (if (*pL) is not NULL), then simply recursively call the insert function. `insert(x,&((*pL)->next));`

4. Test your insert() function by implementing a simple recursive printList() function.

   ```
   void printList(struct Node * L);
   ```

   Add a call to your main method that demonstrates the workings of insert and printList.

5. With a minor modification to the printList method you can make a lookup() function.

   ```
   BOOLEAN lookup(int x, struct Node * L);
   ```

   The lookup method is just like the printList method, except that at every Node we compare x to value and return TRUE if they are the same. If they are not the same, then we recursively call lookup on the next Node. If we come to the end of the list (a NULL list), we return FALSE.

   Add a call to your main method that tests lookup:

```
for (i = 0; i < 20; i++)
  printf("%d %s FOUND\n",i,((lookup(i,L) == TRUE) ? "": "NOT"));
```

6. The last function we need is the delete function.

   ```
   void delete(int x, struct Node **pL);
   ```

   Delete is a lot like lookup. We traverse the list recursively until we find the matching data element. Then, it is a simple matter to splice out the current node.

   ```
   (*pL) = (*pL)->next;
   ```

   Add some calls to your main function to test your delete function

   ```
   for (i = 0; i < 20; i += 3)
       delete(i,&L);
   ```

   and print the before and after version of the list with `printList()`.

3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. "zipped" file that contains the following.)

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.). And a one sentence explaining the contents of all the other files you hand in.
2. The source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4 Grading

Each section (1-6) accounts for 15% of the lab grade (total 90%)

(README file counts for 10%)