CS 61C Fall 2018

Memory Management Review

August 28, 2018

Notes

1 Let's Test Your Memory

1.1 Faulty Memory

Find what is wrong with each function and rewrite the function to return what it specifies in the comments.

```
int * arrayRange(int n){
    //Returns an array where arr[i] = i
    int arr[n];
    int i = 0;
    for (i = 0; i < n; i++){
        arr[i] = i;
    }
    return arr;
}
```

```
int * arrayRange(int n){
    //Returns an array where arr[i] = i
    int *arr = (int *) malloc(sizeof(int)*n);
    int i = 0;
    for (i = 0; i < n; i++){
        arr[i] = i;
    }
    return arr;
}
```

Because we want arr to persist, we need to allocate arr (malloc) on the heap.

```
#include <string.h>
char * reverse(){
    //Returns the string "I LOVE CS61C!" reversed
    char *str = "I LOVE CS61C!";
    int mid = strlen(str)/2;
    int i = 0;
    char tmp;
    for(i=0; i < mid; i++ ){
        tmp = str[i];
        str[i] = str[strlen(str) - i - 1];
        str[strlen(str) - i - 1] = tmp;
    }
```

```
13          return str;
14      }
```

```
1       char * reverse(){
2           char *str = (char *) malloc(sizeof(char) * (strlen("I LOVE CS61C!") + 1));
3           strcpy(str ,"I LOVE CS61C!");
4           int mid = strlen(str)/2;
5           int i = 0;
6           char tmp;
7           for(i=0; i < mid; i++ ){
8               tmp = str[i];
9               str[i] = str[strlen(str) - i - 1];
10              str[strlen(str) - i - 1] = tmp;
11          }
12          return str;
13      }
```

Recall that string literals allocated as a char pointer are allocated in the static segment of memory. Furthermore, string literals such as these are immutable. In order to edit our string, we must allocate it in the heap. We can't allocate the reversed string into our stack because we want this string to persist after this call.

1.2 Finding Waldo

```
1   int main() {
2       char * waldo = "Here I am.";
3       char * vik = (char *) malloc(sizeof(char) * 17);
4       strcpy(vik ,"HI I'M not waldo");
5       char al[] = "HI i'm al";
6
7       char * house[4];
8
9       *(house) = vik;
10      *(house + 1) = waldo;
11      *(house + 2) = "my name is Vin";
12      *(house + 3) = al;
13
14      return 0;
15  }
```

Find where in memory each expression points to.

(a) house[0]

Heap

(b) house

Stack

(c) &house

Stack The address of house (&house) is on the stack; in fact, &house == house. This is another subtle difference between declaring an array vs a pointer. Semantically house and house are different i.e. &house is a pointer to an array of length 4 whereas house is a pointer to a single char pointer (&house + 1 != house + 1).

(d) house[1]

Static

(e) house[2]

Static

(f) house[3]

Stack

2 Assemble the Bear Stack

2.1 Consider the C code below. Assign the result of evaluating each C expression from numbers 1 to 4, based on the C memory model taught in class.

```
#include<stdlib.h>
#include<stdio.h>

typedef struct Bear {
    char* name;
    struct Bear* brothers;
} Bear;

int main(int argc, char const *argv[]) {
    Bear* weBareBears = (Bear*) malloc(sizeof(Bear) * 3);
    for (int i = 0; i < 3; i++) {
        switch(i) {
            case 0:
                weBareBears[i].name = "Grizz"; break;
            case 1:
                weBareBears[i].name = malloc(sizeof(char) * 4);
                weBareBears[i].name[0] = 'I';
                weBareBears[i].name[1] = 'c';
                weBareBears[i].name[2] = 'e';
                weBareBears[i].name[3] = '\0';
                break;
             case 2: weBareBears[i].name = "Panda"; break;
        }
        printf("%s", weBareBears[i].name);
```

```
25          weBareBears[i].brothers = weBareBears;
26      }
27  }
```

(a) `&weBareBears`

4

(b) `&weBareBears[0].name[0]`

1 or 2

(c) `&weBareBears[1].name[0]`

3

(d) `&weBareBears[2].name[0]`

1 or 2

Note: weBareBears[0].name and weBareBears[2].name are both allocated on the static segment of memory, but we can't assume which one is higher/lower, so we accepted either 1 or 2 for b and d. This is because static memory is allocated at compile time and this behavior is dependent on the compiler.

3 Heaps of Fun

3.1 HashTables are very useful data structures, and we want to implement one in C. Fill in the `newTable`, `addEntry`, `deleteEntry`, `resizeTable` and `freeTable` functions. Assume a function `int hashCode(int key)` is defined for you. You may define any helper functions you would like. For allocation failures, assume all you must do is print an error statement.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   typedef struct Entry {
5       int key;
6       int value;
7       Entry* nextEntry;
8       } Entry
9
10  typedef struct Table {
11      size_t size;
12      Entry* buckets;
13  }
14
15  Table *newTable(size_t initialSize) {
16
17
18
19
```

```

}
void addEntry(int key, int value, Table *table) {
    int index = hashCode(key) %  table->size;









}

void deleteEntry(int key, Table *table) {
    int index = hashCode(key) % table->size;









}


```

```

void freeTable(Table *table) {









    }
void resizeTable(Table *table, size_t newSize) {









}
```

```
#include <stdlib.h>
#include <stdio.h>

typedef struct Entry {
    int key;
    int value;
```

```
    Entry* nextEntry;
    } Entry

typedef struct Table {
    size_t size;
    Entry **buckets;
} Table

Table *newTable(size_t initialSize) {
    Entry **buckets = calloc(initialSize, sizeof(Entry *));
    if (!buckets) {
        printf("Creating table failed!");
        return NULL;
    }
    Table* table = malloc(sizeof(Table));
    if (!table) {
        printf("Creating table failed!);
        return NULL;
        }
    table->size = initalSize;
    table->buckets = buckets;
    return table;
}

void addEntry(int key, int value, Table *table) {
    int index = hashCode(key) %  table->size;
    Entry *new = malloc(sizeof(Entry));
    if (new == NULL) {
        printf("Creating new entry failed!")
        return;
        }
    new->key = key;
    new->value = value;
    new->nextEntry = NULL;
    Entry *current = table->buckets[index];
    if (current == NULL) {
        table->buckets[index] = new;
        }
    else {
        while(current->nextEntry != NULL) current = current->nextEntry;
        current->nextEntry = new;
        }
}

void deleteEntry(int key, Table *table) {
    int index = hashCode(key) % table->size;
```

```
    Entry *current = table->buckets[index];
    Entry *prev = NULL;
    while (current != NULL) {
        if (current->key == key) {
            if (prev != NULL) prev->nextEntry = current->nextEntry;
            free(current);
            }
        else {
            prev = current;
            current = current->next;
            }
        }
    printf("Failed to delete, entry not found!");
}

void freeBucket(Entry* entry) {
    if (entry->nextEntry == NULL) {
        free(entry);
    }
    else {
        freeBucket(entry->nextEntry);
        free(entry);
        }
}
void freeTable(Table *table) {
    for (int i = 0; i < table->size; i++) {
        if (table->buckets[i] != NULL) {
            freeBucket(table->buckets[i]);
            }
        }
    }
    free(table->buckets);
    free(table);

void resizeTable(Table *table, size_t newSize) {
    Table *resizedTable = newTable(newSize);
    for (int i = 0; i < table->size; i++) {
        if (table->buckets[i] != NULL) {
            Entry *current = table->buckets[i];
            while (current != NULL) {
                addEntry(current->key, current->value, resizedTable);
                current = current->nextEntry;
            }
        }
    }
    freeTable(table);
```

```
99  }
```