CS 61C Memory Management Review Fall 2018

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.

```
1
int * arrayRange(int n){
2
//Returns an array where arr[i] = i
3
int arr[n];
4
int i = 0;
5
for (i = 0; i < n; i++){
6
arr[i] = i;
7
}
8
return arr;
9
}
10
```

```
1
#include <string.h>
2
char * reverse(){
3
//Returns the string I LOVE CS61C! reversed
4
char *str = I LOVE CS61C! ;
```

5

```
int mid = strlen(str)/2;
```

6

```
int i = 0;
```

7

```
char tmp;
```

8

```
for(i=0; i < mid; i++ ){
```

9

```
tmp = str[i];
```

10

```
str[i] = str[strlen(str) - i - 1];
```

11

```
str[strlen(str) - i - 1] = tmp;
```

12

```
}
```

13

return str;

14

```
}
```

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]

(b) house

(c) &house

(d) house[1]

(e) house[2]

(f) house[3]

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.

```
1
#include<stdlib.h>
2
#include<stdio.h>
3
4
typedef struct Bear {
5
char* name;
6
```

```
struct Bear* brothers;
7
} Bear;
8
9
int main(int argc, char const *argv[]) {
10
Bear* weBareBears = (Bear*) malloc(sizeof(Bear) * 3);
11
for (int i = 0; i < 3; i++) {
12
switch(i) {
13
case 0:
14
weBareBears[i].name = Grizz ; break;
15
case 1:
16
weBareBears[i].name = malloc(sizeof(char) * 4);
17
weBareBears[i].name[0] = I ;
18
weBareBears[i].name[1] = c ;
19
weBareBears[i].name[2] = e ;
20
weBareBears[i].name[3] = \0 ;
21
break;
22
case 2: weBareBears[i].name = Panda ; break;
23
}
24
printf( %s , weBareBears[i].name);
```

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

(a) &weBareBears

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

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

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

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
20
21
}
22
void addEntry(int key, int value, Table *table) {
23
int index = hashCode(key) % table->size;
24
25
26
27
28
29
```

```
30
31
32
33
34
35
36
37
38
39
40
41
}
42
43
void deleteEntry(int key, Table *table) {
44
int index = hashCode(key) % table->size;
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
}
60
61
```

```
62
63
64
65
66
67
68
void freeTable(Table *table) {
69
70
71
72
73
74
75
```

```
76
77
78
79
80
81
82
83
84
85
86
}
87
88
void resizeTable(Table *table, size_t newSize) {
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
}
```