CS 61C Management Spring 2019

More Advanced C, Memory

Discussion 2: July 1, 2019

1 Advanced C

Suppose we've defined a linked list **struct** as follows. Assume `*lst` points to the first element of the list, or is `NULL` if the list is empty.

```
struct ll_node {
    int first;
    struct ll_node* rest;
}
```

1.1 Implement `prepend`, which adds one new `value` to the front of the linked list. Hint: why use ll_node $**lst$ instead of ll_node$*lst$?

```
1 void prepend(struct ll_node** lst, int value) {
2     struct ll_node* item = (struct ll_node*) malloc(sizeof(struct ll_node));
3     item->first = value;
4     item->rest = *lst;
5     *lst = item;
6 }
```

1.2 Implement `free_ll`, which frees all the memory consumed by the linked list.

```
1 void free_ll(struct ll_node** lst) {
2     if (*lst) {
3         free_ll(&((*lst)->rest));
4         free(*lst);
5     }
6     *lst = NULL; // Make writes to **lst fail instead of writing to unusable memory.
7 }
```

2 Memory Management

2.1 For each part, choose one or more of the following memory segments where the data could be located: **code, static, heap, stack**.

(a) Static variables

Static

(b) Local variables

Stack

(c) Global variables

Static

(d) Constants

Code, static, or stack

Constants can be compiled directly into the code. `x = x + 1` can compile with the number 1 stored directly in the machine instruction in the code. That instruction will always increment the value of the variable `x` by 1, so it can be stored directly in the machine instruction without reference to other memory. This can also occur with pre-processor macros.

```
#define y 5

int plus_y(int x) {
    x = x + y;
    return x;
}
```

Constants can also be found in the stack or static storage depending on if it's declared in a function or not.

```
const int x = 1;

int sum(int* arr) {
    int total = 0;
    ...
}
```

In this example, `x` is a variable whose value will be stored in the static storage, while `total` is a local variable whose value will be stored on the stack. Variables declared **`const`** are not allowed to change, but the usage of **`const`** can get more tricky when combined with pointers.

(e) Machine Instructions

Code

(f) Result of `malloc`

Heap

(g) String Literals

Static or stack.

When declared in a function, string literals can be stored in different places. **`char`**`* s = "string"` is stored in the static memory segment while **`char`**`[7] s = "string"` will be stored in the stack.

2.2 Write the code necessary to allocate memory on the heap in the following scenarios

(a) An array `arr` of k integers

```
arr = (int *) malloc(sizeof(int) * k);
```

(b) A string `str` containing p characters

`str = (char *) malloc(sizeof(char) * (p + 1));` Don't forget the null terminator!

(c) An $n \times m$ matrix `mat` of integers initialized to zero.

```
mat = (int *) calloc(n * m, sizeof(int));
```

Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

```
mat = (int **) calloc(n, sizeof(int *));
for (int i = 0; i < n; i++)
    mat[i] = (int *) calloc(m, sizeof(int));
```

2.3 What is wrong with the C code below?

```
int* pi = malloc(314 * sizeof(int));
if (!raspberry) {
    pi = malloc(1 * sizeof(int));
}
return pi;
```

There's a memory leak if `raspberry` is false as the original value of `pi` will be unreachable.