

CSC 209 Review 8 Solution

August 28, 2020

1. I need to create a wrapper function `my_malloc` that does the following:

- ask `my_malloc` it to allocate `n` bytes
- call `malloc`
- test `malloc` doesn't have a null pointer
- return pointer from `malloc`

The solution to this problem is:

```
1  void *my_malloc(int n) {  
2      void *p;  
3  
4      p = malloc(n);  
5  
6      if (!p) {  
7          printf("ERROR: Malloc allocation failed");  
8      }  
9  
10     return p;  
11 }
```

Notes

- Learned that void function can return value
- **Dynamic Storage Allocation**
 - Allows to allocate storage during program execution
 - Allows to create data structures and shrink and grow array as needed
 - e.g. `malloc`, `calloc`, `realloc`
- **Memory Allocation Functions**
 - `malloc` - Allocates a block of memory but doesn't initialize it
 - * doesn't initialize the allocated memory

- * more efficient than `calloc`
 - * accessing the content → segmentation fault (accessing value at invalid mem. location) or garbage values
 - `calloc` - Allocates a block of memory and clears it
 - * allocates memory and initializes the memory block to zero
 - * accessing the content of blocks would return 0
 - `realloc` - Resizes a previously allocated block of memory
- **Null Pointer**
 - is returned when it fails to allocate a block of memory large enough to satisfy the request

Example

```
p = malloc(10000);
if (p == NULL) {
    /* allocation failed; take appropriate action */
}
```

2. I need to write a function named `duplicate` that uses dynamic storage allocation to create a copy of a string.

The requirements of the function are

- `duplicate` allocates space for a string of the same length as `str`
- `duplicate` copies the contents of `str` into the new string
- `duplicate` returns a pointer to it
- `duplicate` returns a null pointer if the memory allocation fails

The solution to this problem is:

```
1  #include <stdio.h>
2  #include <stdlib.h> // malloc
3  #include <string.h> // strlen
4
5  char *duplicate(const char *str);
6
7  int main(void) {
8      char s[] = "hello world", *p;
9
10     p = duplicate (s);
```

```
11     printf("Duplicate: %s\n", p);
12
13     free(p);
14     return 0;
15 }
16
17
18
19 char *duplicate(const char *str) {
20     char *p, *q;
21     const char *r;
22
23     int n = strlen(str);
24
25     p = (char *)malloc(n + 1);
26
27     if (!p) {
28         return p;
29     }
30
31     r = str;
32     q = p;
33     while (r < str + n) {
34         *q = *r;
35         q++;
36         r++;
37     }
38
39     *q = '\0';
40
41     return p;
42 }
```

Correct Solution:

```
1  #include <stdio.h>
2  #include <stdlib.h> // malloc
3  #include <string.h> // strlen
4
5  char *duplicate(const char *str);
6
7  int main(void) {
8      char s[] = "hello world", *p, *q;
9      ;
10     p = duplicate (s);
11
12     printf("Duplicate: %s\n", p);
13
14     free(p);
15     return 0;
16 }
17
```

```

18
19 char *duplicate(const char *str) {
20     char *p, *q;
21     const char *r;
22
23     int n = strlen(str);
24
25     p = (char *)malloc(n + 1);
26
27     if (!p) {
28         p = ((void*)0);
29         return p;
30     }
31
32     r = str;
33     q = p;
34     while (r < str + n) {
35         *q = *r;
36         q++;
37         r++;
38     }
39
40     *q = '\0';
41
42     return p;
43 }

```

Note

- Null pointer has value `((void*)0)`
- `const` tag in parameter prevents the function from modifying what its pointer variable is pointing to.
 - value is modifiable
 - changes the parameter to pass by value

```

31 int *create_array(int n, int initial_value) {
2     int *array;
3
4     array = malloc(n * sizeof(int));
5
6     if (array == NULL) {
7         return array;
8     }
9
10    for(int i = 0; i < n; i++){
11        array[i] = initial_value;
12    }
13
14    return array
15 }

```

Notes

- Dynamically Allocated Arrays

- Syntax:

```
int *a;  
a = malloc(n * sizeof(int));
```

- returns null pointer if allocation fails

```
41 #include <stdio.h>  
2  #include <stdlib.h>  
3  #include <string.h>  
4  
5  struct point {int x, y};  
6  struct rectangle {struct point upper_left, lower_right};  
7  
8  int main(void) {  
9  
10     struct rectangle *p;  
11  
12     p = malloc(sizeof(struct rectangle));  
13  
14     p->upper_left.x = 10;  
15     p->upper_left.y = 25;  
16     p->lower_right.x = 20;  
17     p->lower_right.y = 15;  
18  
19     printf("%d %d %d %d",  
20           p->upper_left.x,  
21           p->upper_left.y,  
22           p->lower_right.x,  
23           p->lower_right.y  
24     );  
25  
26     return 0;  
27 }
```

Notes

- -> doesn't carry over to accessing nested members. Only works when struct is a pointer

Example

```
p->upper_left.x
```

- Linked Lists

- Declaring Node Type

* **Syntax (Node structure):**

```
struct node {
    int value;           /* data stored in the node */
    struct node *next;   /* pointer to the next node */
};
```

```
struct node *first = NULL;
```

Pointer to first node
in the linked list



– **Creating a Node**

* **Syntax (Allocating using malloc):**

```
struct node *new_node;
new_node = malloc(sizeof(struct node));
```

* **Assigning value**

```
(*new_node).value = 10;
```

– **-> Operator**

* is a short form of (*STRUCT_NAME).MEMBER_NAME

Example

```
(*new_node).value = 10;
```

Is the same as

```
new_node->value = 10;
```

5. b) and c) are legal

```
61 struct node *delete_from_list(struct node *list, int n)
62 {
63     struct node *curr, *to_be_freed;
64
65     for (curr = list; curr != NULL && curr->value != n; curr = curr
66     ->next) {
67         if (curr->next != NULL && curr->next->value == n) {
68             to_be_freed = curr->next;
69             curr->next = curr->next->next;
70             free(to_be_freed);
71
72             return list;
73         }
74     }
75 }
```

```
14
15
16     return list;
17
18 }
```

Notes

- Searching a Linked List

- Syntax: for (p = first; p != NULL; p = p ->next)

Example:

```
struct node *search_list(struct node *list, int n)
{
    struct node *p;

    for (p = list; p != NULL; p = p->next)
        if (p->value == n)
            return p;
    return NULL;
}
```

- Deleting Node from a List

- Steps

1. Locate the node to be deleted

* Syntax (Searching for the node of value n to be deleted):

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
    ;
```

2. Alter the previous node so that it "bypasses" the deleted node

```
if (cur == NULL)
    return list;
if (prev == NULL)
    list = list->next;
else
    prev->next = cur->next;
```

3. Call **free** to reclaim the space occupied by the deleted code

```
free(cur);
```

Putting together, we have

```
struct node *delete_from_list(struct node *list, int n)
{
    struct node *cur, *prev;

    for (cur = list, prev = NULL;
         cur != NULL && cur->value != n;
         prev = cur, cur = cur->next)
        ;

    if (cur == NULL)
        return list;
    if (prev == NULL)
        list = list->next;
    else
        prev->next = cur->next;
    free(cur);
    return list;
}
```

7. The statement is incorrect because it removes the current node before its pointer moves to the next node.

As a result, the remaining nodes cannot be removed, and this is not good.

To fix the problem, the pointer p must move to the next before removing the current node, as shown below:

```
1  struct node *to_be_freed;
2
3  for (p = first; p != NULL;) {
4      to_be_freed = p;
5      p = p->next;
6      free(p);
7  }
```