CSC 209 Review 8 Solution

August 31, 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:

```c
void *my_malloc(int n) {
    void *p;

    p = malloc(n);

    if (!p) {
        printf("ERROR: Malloc allocation failed");
    }

    return p;
}
```

   Notes

   - Learned that void function can return value
   - **Dynamic Storage Allocation**
     - Allows to allocate storage during program execution
     - Allows to create data structures and shink 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:

```c
#include <stdio.h>
#include <stdlib.h>  // malloc
#include <string.h>  // strlen

char *duplicate(const char *str);

int main(void) {
    char s[] = "hello world", *p;

    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              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 prevetns the function from modifying what it's pointer variable is pointing to.
  - value is modifiable
  - changes the parameter to pass by value

3.

```
1   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

4.

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

struct point {int x, y;};
struct rectangle {struct point upper_left, lower_right;};

int main(void) {

    struct rectangle *p;

    p = malloc(sizeof(struct rectangle));

    p->upper_left.x = 10;
    p->upper_left.y = 25;
    p->lower_right.x = 20;
    p->lower_right.y = 15;

    printf("%d %d %d %d",
        p->upper_left.x,
        p->upper_left.y,
        p->lower_right.x,
        p->lower_right.y
    );

    return 0;
}
```

Notes

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

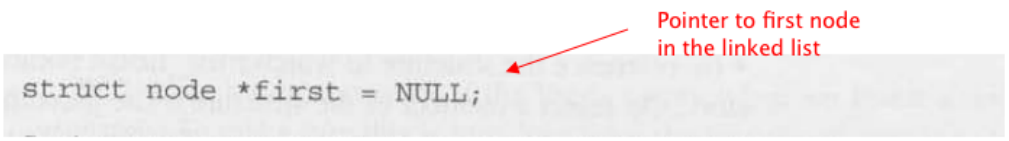
  Example

  `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 */
};
```



– **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

6.
```
1  struct node *delete_from_list(struct node *list, int n)
2  {
3      struct node *curr, *to_be_freed;
4
5      for (curr = list; curr != NULL && curr->value != n; curr = curr
   ->next) {
6          if (curr->next != NULL && curr->next->value == n) {
7              to_be_freed = curr->next;
8              curr->next = curr->next->next;
9              free(to_be_freed);
10
11             return list;
12         }
13     }
```

```
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;                /* n was not found */
  if (prev == NULL)
    list = list->next;          /* n is in the first node */
  else
    prev->next = cur->next;     /* n is in some other node */
  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    }
```

8.

```
1    #include <stdio.h>
2    #include <stdbool.h>
3    #include <stdlib.h>
4    #include <stddef.h> // NULL
5
6    struct node {
```

```
7         int value;
8         struct node *next;
9     };
10
11    struct node *top = NULL;
12
13    void make_empty(struct node *top) {
14        struct node *temp;
15
16        while (top != NULL) {
17            temp = top;
18            top = top->next;
19            free(temp);
20        }
21    }
22
23    bool is_empty(void) {
24        if (top == NULL) {
25            return true;
26        }
27
28        return false;
29    }
30
31    bool push (int n, struct node *top) {
32        struct node *new_node;
33
34        new_node = malloc(sizeof(struct node));
35
36        if (new_node == NULL) {
37            return false;
38        }
39
40        new_node->value = n;
41
42        if (top == NULL) {
43            top = new_node;
44        } else {
45            new_node->next = top->next;
46            top->next = new_node;
47        }
48
49        return true;
50    }
51
52    int pop(void) {
53        struct node *temp;
54        int return_val;
55
56        temp = top;
57        return_val = temp->value;
58        top = top->next;
59        free(temp);
60
```

```
61          return return_val;
62      }
```

9. True. With & sign, the `struct node` becomes `type pointer`.

   With pointer, -> can be used.

   Thus, `x.a` is the same as `(&x)->a`.

10.
```
1       struct part {
2           int number;
3           char name[NAME_LEN+1];
4           int on_hand;
5       };
6
7       ...
8
9       void print_part(struct part *p)
10      {
11          printf("Part number: %d\n", p->number);
12          printf("Part name: %s\n", p->name);
13          printf("quantity on hand: %d\n", p->on_hand);
14      }
```

11.
```
1       #include <stddef.h>
2
3       struct node {
4           int value;
5           struct node *next;
6       };
7
8       int count_occurences (struct node *list, int n)
9       {
10          int count;
11          struct node *top;
12
13          for (top=list; top != NULL; top = top->next) {
14              if (top->value == n) {
15                  count++;
16              }
17          }
18
19          return count;
20      }
```

12.
```
1       #include <stddef.h>
2
3       struct node {
4           int value;
5           struct node *next;
6       };
7
```

```c
struct node *find_last (struct node *list, int n)
{
    struct node *last = NULL, *top;

    for (top=list; top != NULL; top = top->next) {
        if (top->value == n) {
            last = top;
        }
    }

    return last;
}
```

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stddef.h>

struct node {
    int value;
    struct node *next;
};
struct node *insert_into_ordered_list (struct node *list, struct node *new_node);

struct node *insert_into_ordered_list (struct node *list,
                                       struct node *new_node)
{
    struct node *cur, *prev;

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

    prev->next = new_node;
    new_node->next = cur;
    return list;
}
```

Notes

- Passing `NULL` results in segmentation fault

```
insert_ordered_list(NULL, 4);
```

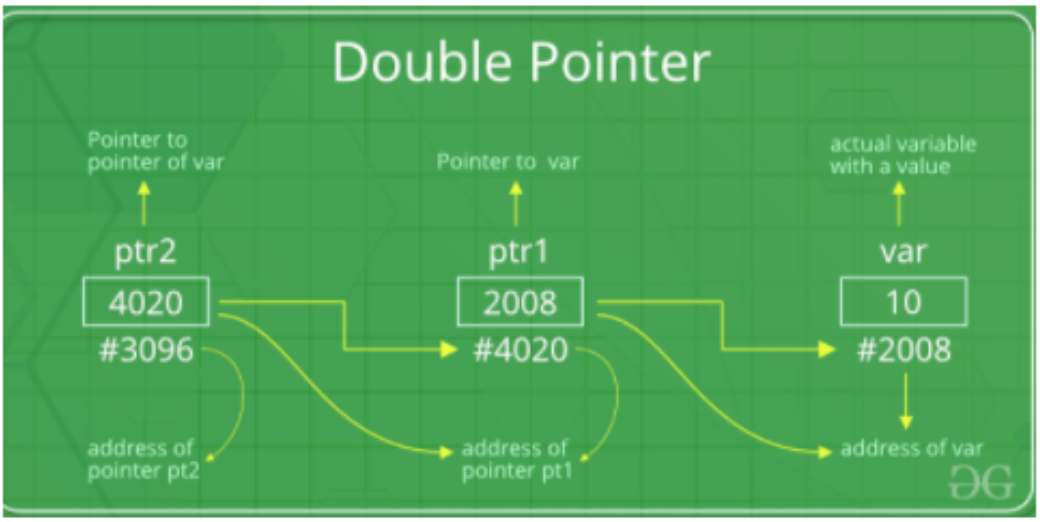
```c
void delete_from_list(struct node **list, int n)
{
    struct node *cur, *prev = NULL;
    cur = *list;

    while (cur != NULL) {
```

```
7             if (cur->value == n) {
8                 break;
9             }
10
11            prev = cur;
12            cur = cur->next;
13
14        }
15
16
17        if (prev == NULL) {
18            *list = (*list)->next;
19        } else {
20            prev->next = cur->next;
21        }
22
23        free(cur);
24    }
```

Notes

- **Pointers to Pointers**
  - **Syntax:** `TYPE **ptr`
  - pops up frequently in data structures
  - is used to store the address of the first pointer



Example

Passing a pointer `*first` that points to `malloc(struct node)` (a pointer pointing to malloc) to function `*add_to_list(struct node *list, int n)` (nono).

* at point of call, `first` is copied to `list`
* `*list` is a pointer that must point to `*first`. So, this is invalid form. For above to be valid, double pointer must be used.

```
*add_to_list(struct node **list, int n)
```

- `**list` refers to value in `malloc(struct node)`
- `*list` refers to the address of variable (`first`) that points to `malloc(struct node)`
- `&list` refers to address of variable list

15. The answer is `3`.

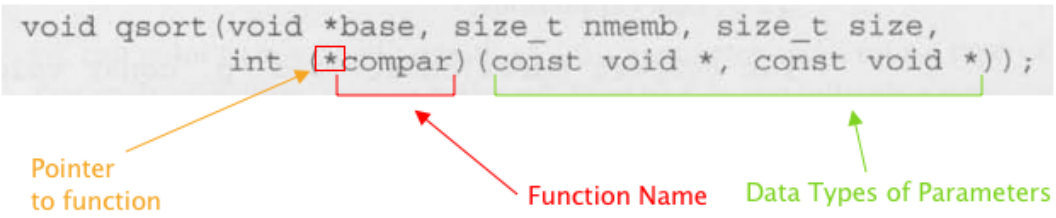
How it Works:

- `f2` is passed to function `f1`
- `f1` iterates while loop starting at `n = 0`
- `n` in while loop is incremented because `0 * 0 + 0 -12 = -12`, and non-zero values are regarded as true.
- textttf1 continues to iterate while loop until `n = 3` (here it stops, becuase `3 * 3 + 3 - 12 = 0`), and 0 is false.

Notes:

- **Pointer to Functions**
  - **Function Pointers to Arguments**

    Example



* The function is passed to another normally like other variables
* The passed function can be used as follows

```
y = (*f)(x)
```

16.
```c
int sum(int (*f)(int), int start, int end) {
    int i = start, total = 0;

    while (i <= end) {
        total += (*f)(i);
        i++;
    }

    return total;
}
```

17.
```c
qsort(&a[50], 50, sizeof(int), compare_parts);
```

Correct Solution:

```c
qsort(&a[50], 50, sizeof(a[0]), compare_parts);
```

Notes

- Learned that the third parameter `size_t size` represents the size of data structure, which in this case is the size of an array slot or `a[0]`.

  In terms of linked list, the `size` is the size of a `struct node`
- Realized that when K. N king said to write `qsort` that sorts the first 50 elements, he meant to replace the parameters with arguments in a function call :(.
- **Quick Sort**

QUICKSORT(A, p, r)
1 **if** $p < r$
2 $q =$ PARTITION(A, p, r)
3 QUICKSORT$(A, p, q-1)$
4 QUICKSORT$(A, q+1, r)$

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

PARTITION(A, p, r)
1 $x = A[r]$
2 $i = p - 1$
3 **for** $j = p$ **to** $r - 1$
4 **if** $A[j] \le x$
5 $i = i + 1$
6 exchange $A[i]$ with $A[j]$
7 exchange $A[i + 1]$ with $A[r]$
8 **return** $i + 1$

18.

```
1  int compare_parts (const *p, const void *q)
2  {
3      const struct part *p1 = p;
4      const struct part *q1 = q;
5
6      if (p1->number > q1->number) {
7          return -1;
8      } else if (p1->number == q1->number) {
9          return 0;
10     } else {
11         return 1;
12     }
13 }
```

Correct Solution:

Assume all numbers in array are positive. Then, we have:

```
1   int compare_parts (const *p, const void *q)
2   {
3       return ((struct part *) q)->number - ((struct part *)p)->
    number;
4   }
```

Notes

- `((struct part *)q)` is for typecasting `q` to `struct part *` instead of the whole `q->number`.

19.
```
1  char buffer[9];
2
3  printf("Add a command name: \n)");
4  scanf("%s", &buffer);
5
6  for (int i = 0; i < 9; i++) {
7      if (strcmp((file_cmd[i])-> name, buffer) == 0) {
8          file_cmd[i].(*cmd_pointer)();
9      }
10 }
```

Correct Solution

```
1   void run_command(char *str) {
2       for (int i = 0; i < sizeof(file_cmd)/sizeof(file_cmd[0]); i
    ++) {
3           if (strcmp((file_cmd[i].cmd_name), str) == 0) {
4               (*file_cmd[i].cmd_pointer)();
5           }
6       }
7   }
```

Notes

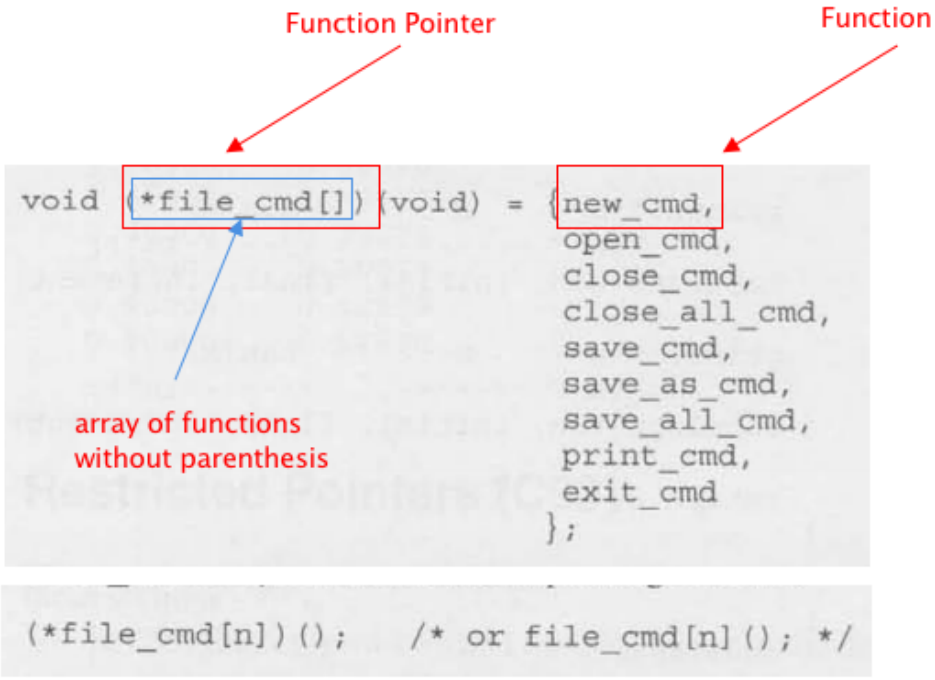
- Learned that to access the pointer of a member variable, `*` is used on the lhs of `file_cmd[i].cmd_pointer`

  Example

  `(*file_cmd[i].cmd_pointer)()`

- **Storing Function Pointers in Array**
  - C treats pointer to functions just like pointers to data

    Example:



20. Please see file `question_20.c`

    Notes

    - **realloc**
      - **Syntax:** `void *realloc(void *ptr, size_t size);`
      - Resizes array allocated by `malloc` or `calloc`
      - `ptr` must point to the memory address of array
      - `size` represents the new size of block

        Example

        ```
        int *p;
        p = malloc(5 * sizeof(int));
        p = realloc(p, 11 * sizeof(int));
        ```

21. Please see file `question_21.c`

22. Please see file `question_22.c`

23. Please see file `question23.c`

    Notes

    - Realized the need to learn gdb

24. Please see file `question24.c`

25. The solution is the same as problem 24.

    Please see file `question25.c`

26. Please see file `question26.c`

    Notes

    - **Flexible Array Members (C99)**
      - **Struct hack**
        * Solves the traditional problem with limit on the length of string, and waste of memory

```
struct vstring {
  int len;
  char chars[N];
};
```

        * Solution
          · By allocating more memory than should
          · Use the remaining space for memory
          · If more memory is required, extend using `realloc`