

CSC 209 Review 8 Solution

August 30, 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
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 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  }
```

```
81 #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.

```

101 struct part {
102     int number;
103     char name[NAME_LEN+1];
104     int on_hand;
105 };
106
107 ...
108
109 void print_part(struct part *p)
110 {
111     printf("Part number: %d\n", p->number);
112     printf("Part name: %s\n", p->name);
113     printf("quantity on hand: %d\n", p->on_hand);
114 }

```

```

111 #include <stddef.h>
112
113 struct node {
114     int value;
115     struct node *next;
116 };
117
118 int count_occurences (struct node *list, int n)
119 {
120     int count;
121     struct node *top;
122
123     for (top=list; top != NULL; top = top->next) {
124         if (top->value == n) {
125             count++;
126         }
127     }
128
129     return count;
130 }

```

```

121 #include <stddef.h>
122
123 struct node {
124     int value;
125     struct node *next;
126 };
127

```

```

8  struct node *find_last (struct node *list, int n)
9  {
10     struct node *last = NULL, *top;
11
12     for (top=list; top != NULL; top = top->next) {
13         if (top->value == n) {
14             last = top;
15         }
16     }
17
18     return last;
19 }

```

```

13 #include <stdio.h>
2  #include <stdbool.h>
3  #include <stdlib.h>
4  #include <stddef.h>
5
6  struct node {
7      int value;
8      struct node *next;
9  };
10 struct node *insert_into_ordered_list (struct node *list, struct
node *new_node);
11
12 struct node *insert_into_ordered_list (struct node *list,
13                                         struct node *new_node)
14 {
15     struct node *cur, *prev;
16
17     for (cur = list, prev = NULL;
18         cur != NULL && cur->value < new_node->value;
19         prev=cur, cur = cur->next)
20         ;
21
22     prev->next = new_node;
23     new_node->next = cur;
24     return list;
25 }

```

Notes

- Passing NULL results in segmentation fault

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

```

14 void delete_from_list(struct node **list, int n)
2  {
3      struct node *cur, *prev = NULL;
4      cur = *list;
5
6      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.

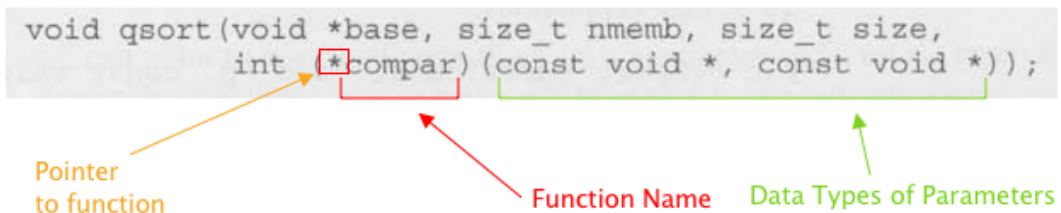
How it Works:

- `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, because $3 * 3 + 3 - 12 = 0$), and 0 is false.

Notes:

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

Example



```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

The diagram shows the function signature for `qsort`. An orange arrow points from the text "Pointer to function" to the `(*compar)` part of the signature. A red arrow points from the text "Function Name" to the `qsort` function name. A green arrow points from the text "Data Types of Parameters" to the parameter list `(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))`.

* The function is passed to another normally like other variables

* The passed function can be used as follows

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

```
16 1 int sum(int (*f)(int), int start, int end) {  
2     int i = start, total = 0;  
3  
4     while (i <= end) {  
5         total += (*f)(i);  
6         i++;  
7     }  
8  
9     return total;  
10 }
```

```
17 1 qsort(&a[50], 50, sizeof(int), compare_parts);
```

Correct Solution:

```
1 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       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )

```

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

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  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 

```

```

181 int compare_parts (const *p, const void *q)
182 {
183     const struct part *p1 = p;
184     const struct part *q1 = q;
185
186     if (p1->number > q1->number) {
187         return -1;
188     } else if (p1->number == q1->number) {
189         return 0;
190     } else {
191         return 1;
192     }
193 }

```

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)->
4      number;
5  }

```

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     ++) {
4         if (strcmp((file_cmd[i].cmd_name), str) == 0) {
5             (*file_cmd[i].cmd_pointer)();
6         }
7     }
8 }

```

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:

The diagram illustrates the syntax and usage of a function pointer. It features two code snippets. The first snippet defines a function pointer array: `void (*file_cmd[]) (void) = { new_cmd, open_cmd, close_cmd, close_all_cmd, save_cmd, save_as_cmd, save_all_cmd, print_cmd, exit_cmd };`. Annotations include a red box around `(*file_cmd[])` with a red arrow pointing to the text 'Function Pointer', and a blue box around `new_cmd` with a red arrow pointing to the text 'Function'. A blue arrow points from the text 'array of functions without parenthesis' to the `file_cmd` part of the array. The second snippet shows the usage: `(*file_cmd[n]) (); /* or file_cmd[n] (); */`.

```
void (*file_cmd[]) (void) = { new_cmd,
open_cmd,
close_cmd,
close_all_cmd,
save_cmd,
save_as_cmd,
save_all_cmd,
print_cmd,
exit_cmd
};

(*file_cmd[n]) (); /* or file_cmd[n] (); */
```

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`