CSC 209 Review 8 Solution

August 29, 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);
```

```c

    printf("Duplicate: %s\n", p);

    free(p);
    return 0;
}


char *duplicate(const char *str) {
    char *p, *q;
    const char *r;

    int n = strlen(str);

    p = (char *)malloc(n + 1);

    if (!p) {
        return p;
    }

    r = str;
    q = p;
    while (r < str + n) {
        *q = *r;
        q++;
        r++;
    }

    *q = '\0';

    return p;
}
```

Correct Solution:

```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, *q;
    ;
    p = duplicate (s);

    printf("Duplicate: %s\n", p);

    free(p);
    return 0;
}

```

```c

char *duplicate(const char *str) {
    char *p, *q;
    const char *r;

    int n = strlen(str);

    p = (char *)malloc(n + 1);

    if (!p) {
        p = ((void*)0);
        return p;
    }

    r = str;
    q = p;
    while (r < str + n) {
        *q = *r;
        q++;
        r++;
    }

    *q = '\0';

    return p;
}
```

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.

```c
int *create_array(int n, int initial_value) {
    int *array;

    array = malloc(n * sizeof(int));

    if (array == NULL) {
        return array;
    }

    for(int i = 0; i < n; i++){
        array[i] = initial_value;
    }

    return array
}
```

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 */
};
```

Pointer to first node
in the linked list

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

– **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.

```
struct node *delete_from_list(struct node *list, int n)
{
    struct node *curr, *to_be_freed;

    for (curr = list; curr != NULL && curr->value != n; curr = curr->next) {
        if (curr->next != NULL && curr->next->value == n) {
            to_be_freed = curr->next;
            curr->next = curr->next->next;
            free(to_be_freed);

            return list;
        }
    }
```

```


        return list;

    }
```

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:

```c
    struct node *to_be_freed;

    for (p = first; p != NULL;) {
        to_be_freed = p;
        p = p->next;
        free(p);
    }
```

8.

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

    struct node {
```

```
        int value;
        struct node *next;
    };

    struct node *top = NULL;

    void make_empty(struct node *top) {
        struct node *temp;

        while (top != NULL) {
            temp = top;
            top = top->next;
            free(temp);
        }
    }

    bool is_empty(void) {
        if (top == NULL) {
            return true;
        }

        return false;
    }

    bool push (int n, struct node *top) {
        struct node *new_node;

        new_node = malloc(sizeof(struct node));

        if (new_node == NULL) {
            return false;
        }

        new_node->value = n;

        if (top == NULL) {
            top = new_node;
        } else {
            new_node->next = top->next;
            top->next = new_node;
        }

        return true;
    }

    int pop(void) {
        struct node *temp;
        int return_val;

        temp = top;
        return_val = temp->value;
        top = top->next;
        free(temp);

```

```
        return return_val;
    }
```

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.
```
    struct part {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
    };

    ...

    void print_part(struct part *p)
    {
        printf("Part number: %d\n", p->number);
        printf("Part name: %s\n", p->name);
        printf("quantity on hand: %d\n", p->on_hand);
    }
```

11.
```
    #include <stddef.h>

    struct node {
        int value;
        struct node *next;
    };

    int count_occurences (struct node *list, int n)
    {
        int count;
        struct node *top;

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

        return count;
    }
```

12.
```
    #include <stddef.h>

    struct node {
        int value;
        struct node *next;
    };

```

```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;
}
```

13.

```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);`

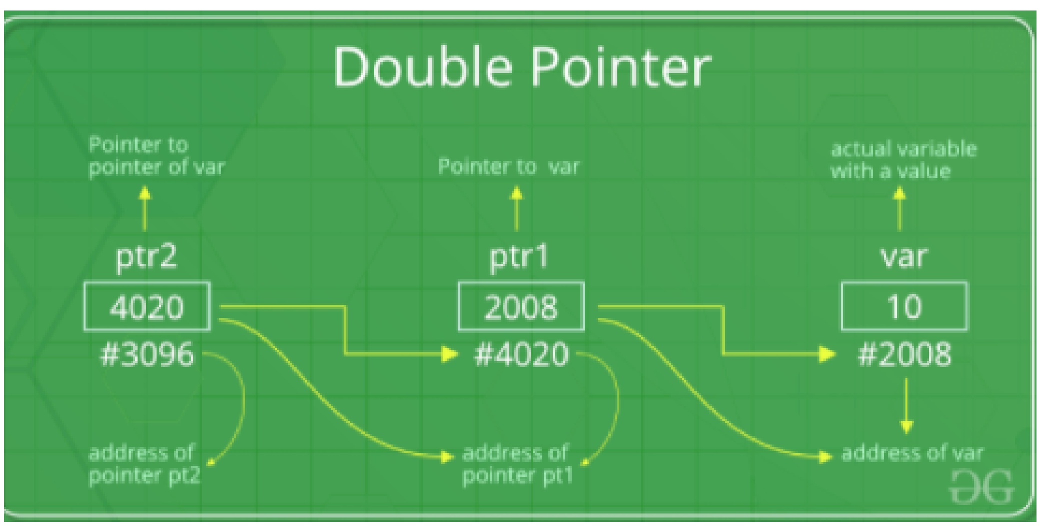
14.

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

    while (cur != NULL) {
```

```c
            if (cur->value == n) {
                break;
            }

            prev = cur;
            cur = cur->next;

        }


        if (prev == NULL) {
            *list = (*list)->next;
        } else {
            prev->next = cur->next;
        }

        free(cur);
    }
```

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`.

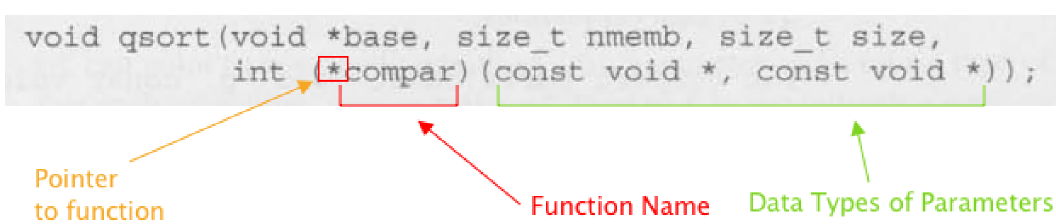
How it Works:

- `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.
- texttt f1 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)

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    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)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

18.

```c
int compare_parts (const *p, const void *q)
{
    const struct part *p1 = p;
    const struct part *q1 = q;

    if (p1->number > q1->number) {
        return -1;
    } else if (p1->number == q1->number) {
        return 0;
    } else {
        return 1;
    }
}
```

Correct Solution:

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

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

Notes

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