CSC 209 Review 8 Solution

September 11, 2020

1.
```c
int *my_malloc (int n) {
    int *res;

    res = malloc(n * sizeof(int));
    if (res == NULL) {
        perror("Allocation failed.");
    }

    return res;
}
```

Please see `question_1.c` for details.

2.
```c
char *duplicate(char *str) {
    char *res;

    res = malloc(strlen(str) + 1);
    if (res == NULL) {
        return res;
    }

    strcpy(res, str);

    return res;
}
```

Please see `question_2.c` for details.

3.
```c
int *create_array(int n, int initial_value) {
    int *p, *res;

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

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

```

```
        for (p = res; p < res + n; p++) {
            *p = initial_value;
        }

        return res;
    }
```

Please see `question_3.c` for details.

4.
```
        int main(void) {
            struct point {int x, y};
            struct rectangle {struct point upper_left, lower_right};
            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\n", p.upper_left.x, p.upper_left.y);
            printf("%d %d\n", p.lower_right.x, p.lower_right.y);

            free(p);

            return 0;
        }


```

Please see `question_4.c` for details.

5. `b)`, `c)` and `d)` are legal.

Correct Solution

`b)`, `c)` are legal.

Notes

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

    Example

    `p->upper_left.x`

6.
```c
struct node *delete_from_list(struct node *list, int n)
{
    struct node *cur = list, *temp;

    if (cur->value == n) {
        list = cur->next;
        return list;
    }

    for (cur = list; cur != NULL; cur = cur -> next) {

        if (cur->next != NULL && cur->next->value == n) {
        break;
        }
    }

    if (cur == NULL) {
        return list;
    }

    temp = cur->next;
    cur->next = cur->next->next;

    free(temp);
    return list;
}
```

7. It's incorrect because it's deleting the node before moving to next.

   To fix this bug, p must move to the next node before removing the current.

```c
struct node *temp;
p = first;
while (p != NULL) {
    temp = p;
    p = p -> next
    remove(temp);
}
```

8. Please see file `question_8/stack.h`, `question_8/stack.c`, `question_8/calc.c` for details.

9. True.

   By definition, `(&x)->a` is the same as `(*(&x)).a`.

   Since `(*(&x)) = x`, we can write `(&x)->a` is the same as x.a.

10.
```c
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. Please see `question_11.c` for details.

12.
```c
struct node {
    int value;

}

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);
}
```

13.
```c
struct node {
    int value;
    struct node *next;
};

struct node *find_last(struct node *list, int n)
{
    struct node *res = NULL, *p;

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

    return res;
}
```

Please see file `question_12.c` for details.

14.
```c
struct node {
    int value;
    struct node *next;
};

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

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

    while (cur != NULL && cur->value <= new_node ->value) {
        prev = cur;
        cur = cur->next;
    }
```

```
19
20          prev->next = new_node;
21          new_node->next = cur;
22          return list;
23      }
```

```
15.1    struct node *delete_from_list(struct node *list, int n)
2       {
3           struct node *cur, *prev;
4
5           for (cur = list, prev = NULL;
6                cur != NULL && cur->value != n;
7                prev = cur, cur = cur->next)
8
9                ;
10
11          if (curr == NULL) {
12              return list;
13          }
14
15          if (prev == NULL) {
16              list = list->next;
17          } else {
18              prev->next = cur->next;
19          }
20
21          free(cur);
22          return list;
23      }
```

Please see `question_14.c` for details.