Arizona State University SER334: Operating Systems & System Programming
Lecturer Acuña (6), UGTA Lisonbee (2) Revised 1/5/2021

Unit 2 Sample Problems - C Programming II (SOLN)

In this sample problem set, we will practice concepts of the C programming language.

- Length: 50 minutes with discussion.
- Questions: Q1-Q3, Q5, Q7. (optional: Q4, Q6, Q8)

Memory Allocation

1. [Acuña] Consider the following snippet of code. [4 points total]

```
struct point_2d {
    int x;
    int y;
    Color col;
};

void main() {
  point_2d* data[5];
  for(int i = 0; i < 5; i++) {
    point_2d tmp;
    tmp.x = i;
    tmp.y = i * i;
    data[i] = &tmp;
  }
  for(int i = 0; i < 5; i++)
    printf("Point #%d: %d, %d", i, data[i]->x, data[i]->y);
}
```

(a) What will be the output? **Trace this code manually, do not run it.**
Ans: [Acuña]
"Point #0: 4, 16", "Point #1: 4, 16", "Point #2: 4, 16", "Point #3: 4, 16", and "Point #4: 4, 16".

(b) Why does the program give that output? Explain in terms of memory and pointer usage.
Ans: [Acuña]
Each iteration of the loop is getting the address of the same piece of memory. Since the last thing assigned to that place is the 4/16 point, then the display will show that value.

2. [Acuña] Consider the following (incorrect) function which removes the head node in a globally defined list called *my_list*. The struct *grade_node* is used to represent a node containing grade information in a linked list of grades.

```
struct grade_node {
    int value;
    char assignment[255];
    struct grade_node* next;
};

struct grade_node* my_list = ...

void remove_node() {
    free(my_list);
    if(my_list != null)
      my_list = my_list->next;
}
```

(a) What is the syntax error in remove_node?

Ans: [Karaliova]

NULL in C is defined in all caps (because it is a macro), using 'null' will trigger a compiler error.

(b) What is the logical error in remove_node?

Ans: [Karaliova]

Using free(my_list) prevents us from being able to access the pointer to the next element for the head node of the list. In other words, the list is no longer accessible. Instead we should store the pointer to the current head of the list in a temp node first. Then, we need to set head's next node to be new lists's head. Temp node can now be removed:

```
void remove_node() {
    if(my_list != NULL) {
        grade_node* tmp = my_list;

        my_list = my_list->next;
          free(tmp);
          tmp = NULL;
    }
}
```

Defining New Types

3. [Acuña] Consider the problem of padding the following structure, and answer the three questions below. Assume that you are compiling on a system with a 32-bit architecture. [4 points total]

```
struct bmp_header {
      char creator_name[254];
      int width;
      int height;
      char signature_rgb[2];
      int offset_pixels;
};
```

(a) What is the size of this struct as defined?

Ans: [Karaliova]

Total size of structure = (254 + 2) + 4 + 4 + (2 + 2) + 4 = 272 bytes

char creator_name[254]	**(padding)**	int width	int height	char signature_rgb[2]	**(padding)**	int offset_pixels
254 bytes	**2 bytes**	4 bytes	4 bytes	2 bytes	**2 bytes**	4 bytes

(b) How much space would be wasted with word length padding?

Ans: [Karaliova]

Total size of padding = 2 bytes + 2 bytes = 4 bytes

4. [Acuña] One of the issues with using unions is that it can be unclear which element in the union is the one currently being used. Suggest a mechanism to indicate which element is active. [2 points]

Ans: [Acuña]

One idea would be to create an enumeration which contains an element for each of the possible pieces of data that the union contains. Then, a variable of the union type can be wrapped with a struct which also contains a variable of the new enum type. The result will be that the union will have "metadata" attached that will indicated how it is being used.

Parameters, Scope, and Pointers

5. [Acuña] Consider two possible functions for adding together two xyz points:

```
struct point add_points(struct point_3d p1, struct point_3d p2)
struct point* add_points(struct point_3d* p1, struct point_3d* p2)
```

Which of these functions should we expect to operate more efficiently and why? (Assume that you are compiling on a system with a 32-bit architecture.)

Ans: [Acuña]

The second function, which uses pointers, should be a little more efficient. In the first function, the execution environment has to make copies of the entire point structures, about 12-bytes each, to make them available as local variables in a stack frame. In the second, only the address (4-bytes) for each of them will need to be copied.

6. [Acuña] Consider the following function which adds a new node to the front of a list passed as a parameter called *param_list*.

```
struct grade_node {
    //see previous question.
};

void add_node(grade_node* parm_list, grade_node* node) {
    if(node != NULL) {
      node->next = param_list;
```

```
        param_list = node;
    }
}
```

Is it possible for this function ever to work incorrectly? If so, under what conditions does it fail?

Linked Lists

7. [Lisonbee] Using the template provided, **implement** the function 'hide_nums' that traverses a singly linked list of chars, and for every instance of a number (hint: ASCII values 48-57 correspond to numbers 0-9) it is replaced with a space (ASCII value 32). You can assume that 'sequence' is already initialized with a list of chars, with the address of 'sequence' pointing to the head of the list. Furthermore, the end of the list is represented by a node containing a null terminator.

```
struct char_node {
        char data;
        struct char_node* next;
};

struct char_node* sequence = ...

void hide_nums() {
        struct char_node* tmp = sequence; //use tmp var to not lose head of list
        while (tmp->data != '\0') {
                if (tmp->data >= 48 && tmp->data <= 57)
                        tmp->data = 32;
                tmp = tmp->next;
        }
}
```

8. [Lisonbee] Provide a use case for using an array over a linked list, and one for using a linked list over an array. **Explain** why either data structure is better for each use case.

Ans: [Lisonbee] (Answers vary)

One use case where an array is better than a linked list is when you have a data set of a fixed size, and being able to quickly index the data is important. One example would be to create a fixed-size array of structs that contain data for an experiment. The size of the array is the number of samples of the experiment, which doesn't change over time. If you're constantly logging data it is important to quickly access different indicies in the data set. A use case where a linked list is better suited is one where the data set is of an undefined size, and being able to quickly find nodes in the list is not as important. An example would be a list of customers who have shopped at a particular establishment. Every time a new customer checks out they are added to the list (which is fast to do), but in general it is not very important to be able to look up customers quickly as that is something that will rarely be done.