CSCI 3150 Warm-Up Assignment #1

Getting familiar with pointers and dynamic-allocated memory in C

Sep 7, 2015

Objectives

The objective of this assignment is for you to understand and to handle the horrible manipulations of the pointers. On top of that, the dynamically–allocated memory is always connected to the use of pointers. Thus, incorrect use of the pointers is always mis-interpreted as a wrong use of the `malloc()` function. This warm-up assignment should help you to beef up your skill of the C programming language and acts as a refresher after your summer vacation.

Note: The codes given in this assignment have all been tested under 32-bit Ubuntu 12.04 with gcc v4.6.3.

Marking

The goal of this warm-up assignment is for you to learn something, rather than identifying whether you are good at C or not. Therefore, the marking of this assignment adopts a **hand-waving** style.

- If you have submitted your work and have tried all questions (including sub-questions of course), then you will get all the corresponding marks, no matter your answer is correct or not.
- If you did not submitted your work, then you will get zero marks.
- If you have submitted your work but the content is totally irrelevant (e.g., an empty file, or writing "*I don't know*"), then you will get zero marks.

Notice that the submission relevancy check will be performed by our tutors.

Explanation Tutorial

As you may aware that bugs and concepts in the C programming language is particularly difficult to understand. Although bugs are always related to carelessness, it is always hard to understand why a simple, careless errors would end up in *segmentation fault* errors or even disastrous scenarios. Hence, an explanation tutorial is essential for those who are eager to understand the reasons behind.

Question 1 (0.5%)

```
1  #include <stdio.h>
2
3  int * addition(int a, int b) {
4          int c = a + b;
5          int *d = &c;
6          return d;
7  }
8
9  int main(void) {
10         int result = *(addition(1, 2));
11         int *result_ptr = addition(1, 2);
12 /***
13    Never interchanging printfs?!
14  ***/
15         printf("result = %d\n", *result_ptr);
16         printf("result = %d\n", result);
17         return 0;
18 }
```

Listing 1: Bob writes a function returning a pointer.

A student Bob is writing a simple program to practise the use of pointers. He uses the function `addition()` to calculate the sum of two integers. Instead of returning the value, the function returns a pointer to the result.

Bob used to be very sure in using `printf()`. Yet, something horrible happens: when Lines 15 and 16 are interchanged, the program behaves differently!! Bob is frustrated and his confidence in writing C drops to zero. Please help poor Bob!

(a) Which order of `printf()` statements gives you "`normal`" result?

(b) What's wrong with Bob's program? Please fix it.

(c) Bob insists that `addition()` should return a pointer. Please propose a way to improve his `addition()` function?

Question 2 (0.5%)

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

int count = 0;

int* new_array() {
        int *array = (int*) malloc(sizeof(int)*9);
        int i;
        for(i = 0; i <= 9; i++)
                array[i] = count++;
        for(i = 0; i <= 9; i++)
                printf("%d ",array[i]);
        printf("\n");
        return array;
}

int main() {
        int i, *a;
        for(i = 0; i < 10; i++)
                a = new_array();
        return 0;
}
```

Listing 2: Bob writes a function that cannot be called twice.

Bob writes yet another program to practise the use of pointers. In this time, he writes the function `new_array()` to return a pointer to a dynamically-allocated array. Besides, the function initializes the array and prints the content of the array out. Of course, Bob ends with frustrations again.

(a) Can the program stop? If yes, how many iterations can the program go in before it stops?

(b) Locate the source of the problem. Hint: not because of the missing `free()` call.

Note: Running on different systems will result in different outputs.

Question 3 (0.5%)

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

#define ROWS 10
#define COLS 8

void process_array(int array[ROWS][COLS]) {
        int i, j, count = 0;
        for(i = 0; i < ROWS; i++)
                for(j = 0; j < COLS; j++)
                        array[i][j] = count++;
        /***
          do something else
        ***/
}

int main() {
        int **array = malloc(sizeof(int)*ROWS*COLS);
        process_array(array);
/***
        int i, j;
        for(i = 0; i < ROWS; i++) {
                for(j = 0; j < COLS; j++)
                        printf("%d ", array[i][j]);
                printf("\n");
        }
***/
        return 0;
}
```

Listing 3: Bob's first try using 2-dimensional dynamic array.

Bob is now working on his assignment. He has heard that "*arrays are the same as pointers*", so he thinks that a "*pointer-to-pointers*" and a *two-dimensional array* are equivalent.

Henceforth, he creates a two-dimensional dynamically-allocated array by using `malloc()`. Then, the array is passed it to the function `process_array()` written by his teammate, Alice. The program works fine except that there are warnings reported by gcc.

Questions.

(a) Bob wants to know if Alice's code is correct. He uncomments the code from Line 20 to Line 27 to output the content of the array. What result will he get?

(b) Alice insists her code is absolutely correct (which is true). She makes a threat to Bob, saying that if Bob modified one character of her code, she would hate him. How can he fix the problem if Bob does need to access the array in the main function via the subscripting operators (i.e., `array[i][j]`)?

(c) Alice changes her function signature to

```
void process_array(int **array);
```

What is/are the corresponding modification(s) should Bob make to the main function?

Question 4 (0.5%)

```c
#include <stdio.h>
#include <string.h>
#define SIZE 5

void print_array(int *array, char *name, int len) {
    int i;
    printf("%s = { ", name);
    for(i = 0; i < len; i++)
        printf("%d ", array[i]);
    printf("}\n");
}

int main(void) {
    char string1[SIZE] = {'1','2','3','4','\0'};
    char string2[SIZE], string3[SIZE];
    int array1[SIZE] = {1,2,3,4,5};
    int array2[SIZE], array3[SIZE];

    strncpy(string2, string1, sizeof(string1));
    memcpy(string3, string1, sizeof(string1));
    printf("string2 = %s\n", string2);
    printf("string3 = %s\n", string3);

    strncpy( (char *)array2, (char *)array1,
             sizeof(array1));
    memcpy(array3, array1, sizeof(array1));
    print_array(array2, "array2", SIZE);
    print_array(array3, "array3", SIZE);

    return 0;
}
```

Listing 4: Bob tries comparing `strncpy()` and `memcpy()`.

Bob just learns the functions `strncpy()` (pronounce as *string N copy*) and `memcpy()` (pronounce as *memory copy*). Yet, he is so lazy that he has not read the man pages of the two functions before using them.

He thinks that the two functions are the same. Because of the fact that `strncpy()` only accepts `char *` as parameter, Bob uses a trick: he casts the integer arrays to `char *`.

With a little test over the character arrays (`string2` and `string3`) in the above code, he "*proves*" that both functions are the same. Then, he proceeds to test the two functions over integer arrays. Of course, out of his expectation, the outputs are different.

(a) What are the outputs when he applies both functions on integer arrays?

(b) Please explain why he gets the different printouts for the integer arrays.

Hint. You are not Bob. You should read man pages. Run on your terminal: "`man strncpy`" and "`man memcpy`".

Deadline: 23:59, September 21, 2015 (Monday).

For the submission guidelines, please visit our course homepage:

`http://course.cse.cuhk.edu.hk/~csci3150/`

– END –