CSCI 3150 Warm-Up Assignment #2

Let's do some debugging.

Objectives

The objective of this assignment is for you to deal with the the `sizeof()` operator. Plus, we include two questions related to what we taught in Chapter 2. The codes given in this assignment have all been tested under 32-bit Ubuntu 12.04 and gcc v4.6.3. If you test them under other environments, there is no guarantee on the outcomes.

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 or 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

Again, we will have the explanation tutorials. It will be held after the deadline.

Question 1 (0.5%)

```c
#include <stdio.h>

#define SIZE (4)

void fill (int array[SIZE]) {
    int i;
    for(i = 0; i < sizeof(array); i++)
        array[i] = i;
}

int main() {
    int array[SIZE];
    fill(array);

/***
At first, the code works fine. But after Bob
changes SIZE to 5, there's something wrong.
Later, Bob adds the following code to verify
the correctness of fill().
***/
    int i;
    printf("=== dump array ===\n");
    for(i = 0; i < SIZE; i++)
        printf("%d ",array[i]);
    printf("\n=== end ===\n");
    return 0;
}
```

Listing 1: A simple but buggy code written by Bob.

Bob was busy working on his programming assignment last week. After the work was done, he found the output of the program is not correct under certain circumstances:

- If `SIZE` is 4, this is no problem on 32-bit Linux machines. Yet, the program behaves badly on 64-bit Linux machines.
- If `SIZE` goes beyond 4, there is always problem on both 32-bit and 64-bit systems.

After some debugging, he doubted it was because of the function `fill()`.

(a) What's wrong with Bob's program? Please locate the line number.

(b) Suggest a way to correct that bug.

Question 2 (0.5%)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define  SIZE  (8)

struct T {
    char flag[4];
    char ch;
};

int main() {
    struct T * array = malloc(sizeof(struct T)*SIZE);
    //Bob's idea: 5 = sizeof(char) * 4 + sizeof(char)
    memset(array, -1, 5 * SIZE);

    printf("=== dump array ===\n");
    int i,j;
    for(i = 0; i < SIZE; i++){
        printf("array[%d]\tflag=",i);
        for(j = 0; j < 4; j++)
            printf("%d",array[i].flag[j]);
        printf("\tch=%d\n",array[i].ch);
    }
    printf("=== end ===\n");
    free(array);
    return 0;
}
```

Listing 2: Version a; code: "q2a.c"

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define  SIZE  (8)

struct T {
    int flag;
    char ch;
};

int main() {
    struct T * array = malloc(sizeof(struct T)*SIZE);

    //Bob: 5 = sizeof(int) + sizeof(char)
    memset(array, -1, 5 * SIZE);

    printf("=== dump array ===\n");
    int i,j;
    for(i = 0; i < SIZE; i++){
        printf("array[%d]\t",i);
        printf("flag=%d\t",array[i].flag);
        printf("ch=%d\n",array[i].ch);
    }
    printf("=== end ===\n");
    free(array);
    return 0;
}
```

Listing 3: Version b; code: "q2b.c"

The two programs above were both written by Bob. They were much the same. Bob came up with version a first. But, he then modified the structure `T` and got version b.

(a) The result of version b was wrong. Can you spot the buggy line?

(b) Suggest a way to correct the buggy line. Note that Bob would like to change the structure `T` from time to time.

Question 3 (0.5%)

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

int person(char* argv0, char* s){
    char * t = strrchr(argv0, (int)'/');
    t = (t == NULL)? argv0 : t+1;
    return strcmp(t, s) == 0;
}

int main(int argc, char* argv[]){
    if(person(argv[0], "bob"))
        printf("I am Bob.\n");
    if(person(argv[0], "alice"))
        printf("I am Alice.\n");
    return 0;
}
```

Listing 4: This program understands the input name; code: "person.c".

Bob come up with the above code. He compiled it using the following command (suppose the source code is person.c):

"`gcc person.c -o alice; gcc person.c -o bob`".

This program is supposed to be magically knowing who the input name is. If `./alice` is called, a line "*I am Alice.*" will be printed. If `./bob` is called, it will output "*I am Bob.*".

Later, Bob used the code in the next page to test it.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
    if(fork() == 0){
        execl("./alice", NULL);
        return 0;
    }
    wait(NULL);
    if(fork() == 0){
        execl("./bob", NULL);
        return 0;
    }
    wait(NULL);
    return 0;
}
```

Listing 5: The test code.

The test code is simple: it tries to call "`./alice`" and "*./bob*". Then, it is expected that the output contains a line "*I am Alice.*" and then another line"*I am Bob.*".

(a) What is the actual output of the test code?

(b) What is/are wrong with the above program(s)?

Question 4 (0.5%)

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

int main(){
    pid_t pid;
    if(pid = fork() == 0)
        printf("child started\n");
    if(pid == 0) {
        // child process
        int i;
        for(i=0; i<10; i++){
            printf("working %d\n", i);
            sleep(1);
        }
        printf("done\n");
        return 0;
    }else{
        // parent process exits immediately
        // so as to allow reparent-ing.
        return 0;
    }
}
```

Listing 6: A daemon.

Bob had just learnt a new thing called **daemon**. A daemon is a program usually running for a long time in the background. For example, a famous daemon is called "`crond`", which runs scheduled tasks (and those tasks are therefore called *cron jobs*).

The first step of making a process into a daemon (i.e., to "*daemonize*") is to termiate its parent immediately after the child process is created.

Then, the init process (pid=1) will become the parent of the child process while the child becomes a daemon. By doing so, the child process then loses control from the user, i.e., at least, it will not be terminated when the user closes the parent shell.

Bob wrote the above code to try the effect out. However, this "*daemon*" was still running in the foreground. Why?

Deadline: 23:59, Nov 2, 2014 (Monday).

For the submission guidelines, please visit our course homepage:

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

– END –