Solutions Lab 2: Advanced C (Part 1/2)

2. Dynamic memory allocation

PART 1 Answer: A and B will cause issues. C may cause issues.
Function mystery_a() has undefined behavior. Variable n is a local variable to mystery_a(), and mystery_a () returns a pointer to this variable (i.e., the variable's address). The problem is that n may vanish after mystery_a() has returned as n exists on stack. So, &n may or may not be invalid when it is accessed from outside mystery_a().

Running mystery_b() will result in a Segmentation Fault because pointer variable p is assigned a value without allocating memory to it.

Function mystery_c() works well. Memory is allocated to pointer p using malloc(). So, p will remain in memory after mystery_c() returns, as it is on heap. The only potential issue with this code is causing a memory leak, if p's memory is not eventually freed (see PART 2)

PART 2 Answer: Yes, there is an issue with function f(). The problem is a memory leak, since pointer p is allocated memory (malloc) which is not freed. When the pointer is assigned to NULL, the memory where p was previously pointing becomes inaccessible, creating a leak. Instead, the code sequence should be:

```
free(p); p = NULL;
```

Note that free() can be called for a NULL pointer, so there is no issue with the free() function call.

PART 3 Answer: The program will cause a Segmentation Fault. This is because my_int_malloc() makes a copy of the pointer, so when malloc() is called, it is setting the copied pointer to the memory location, not p. p is pointing to random memory before and after the call to my_int_malloc(), and when it is dereferenced, it will cause a crash.

To fix the issue, we need to pass the address of the pointer (i.e., a double pointer) to my_int_malloc(), as shown below:

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

void my_int_malloc (int **n){
    *n = (int*)malloc(sizeof(int));
}

int main(){
    int *p;
    my_int_malloc(&p);
    *p = 7;
    printf("%d \n",*p);
    return(0);
}
```