Pedram Safaei
CS 326
HW6

1.

a)

The activation record for different instances of foo() occupies the same space in the stack for most implementation of programming languages. This local variable “i” is never initialized but if the stack space has not been used for anything it might inherit a value from previous instances of the routine. So if the stack is created by the space filled by zeros and if the space that is occupied by foo()’s activation record has not been used before the first foo() is called then the local variable “i” will start with the value of 0 for the first iteration, in other words the space allocated for foo() does not have any previous values so memory location consists of all zeros so variable “i” would have the value of zero and as the function foo() is called multiple times the memory location that is allocated for all calls is the same so it remembers what the previous values where hence incrementing and printing and giving us the predicted results, which of course can be different according to what system and compilers we use.

b)

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

void foo()
{
        int i;
        printf("%d ",i++);
}

void main()
{
        int j;
        int * a;
        for (j = 1; j <=10; j++)
        {
                a = malloc (sizeof(int));
                foo();
                free(a);

        }
}
```

Here I am just allocating and deallocating some memory which should not really have any affect on the program but because of reasons we discussed in part a of this question calling other functions like malloc or free will cause the behavior to change since the internal stack is using the same memory location as local variable “i” in the function foo()

2.

For answering this question we need to look at two implementations of this problem, for factorial we can have a recursive or an iterative solution, for a recursive solution we can't do that of course since macros are just preprocessor directives that are handled by the preprocessor before the code goes for compilation and the job of that preprocessor is to include the header and replace the text of the macros with their definition, so it is just a text replacement, so this can't be used since this replacement can't go on until the best case.
Now lets look at the iterative implementation

```
#include <stdio.h>

int i;
int times;

#define fact(n) for (times = 1, i = 1; i <= n; i++) {times *=i;}

int main()
{
        int a = 5;
        fact(a)
        printf("factorial of %d is %d \n", a, times );
        return 0;
}
```

This works fine **but** the question is asking a macro is C that **"Returns"** the factorial and this function does not do that, so to answer the question **we can not have a macro in C for factorial without calling a subroutine.**

3.
if we have this subroutine for example

```
swap(X,Y:int):

var t: int

t:=X;
X:=Y;
Y:=t;

end

Main:
var i, j;
i=:2;
j=:3;

swap(i,j);
```

Now let’s look at them using pass by value and pass by name

a) passing by value

The actual parameters are fully evaluated and the resulting value is then copied into a location being used to hold the formal parameter's value during the function execution. That location is typically a bunch of memory on the runtime stack so the actual argument would not change by calling the parameters using pass by value so the values **will be the same** as they were initialized with, so if we print this we will see that i will be still 2 and j will still be 3 so in this case the subroutine **will not reflect** the function and it shows that **it is impossible** to write the code for the elements of an array.

b) pass by name

By substituting the actual parameters into the function body, the function body can both read and write the given parameters. In this sense the evaluation method is similar to pass-by-reference. The difference is that since with pass-by-name the parameter is evaluated inside the function, a parameter such as **a[i]** depends on the current value of **i** inside the function, rather than referring to the value at a[i] before the function was called. So again we can not use this to swap values (just look at swap(i, a[i]) this won’t act as expected since i will change and messes up a[i])

4.

a) value

as we discussed in previous questions passing by value will not change the variables so we still have the same ones

1 10 11

b) reference

This method will change the actual variables and doesn't just make copies of them so we will go to procedure f and 1 + 1 is 2 but z also changes since so i is now 3 and a[i] is x:=x+1 which was 2 and a[2] is not passed in so it doesn't change

3 10 2

c) value result

using this method is like part b except we don't use the update value inside the function for example if we have

foo(a,b) {

a = a + b // let's say a is 2 and b is 3 the new value of a is 5

….

b = a +b // this is the same reuslts as the previous one it will not use 5 for a

}

so we will have

3 11 3

2 issues with this method

i. if the parameter is mutable (variable, array) no assignment may occur if the actual parameter is non mutable.

ii. value-result is the combination of value (in) and result (out) which uses other things like pass by reference so why use that which uses something else!

d) name

With this method we have i = 1 and a[i] = 10 x:=x+1 changes that to 2 and now we have a[2] which will get z which is 2 and z:z+1 changes that to 3 so we have x=3 y = 2 and z =3 so when we print we will get

3 10 2

5.
With most machines and compiler it won't, but when we think about it not specifying values for the optional parameters will make the compiler to take default values for those and this omits some process of assigning arguments to the parameters, so it should increase the speed of the subroutines since it is omitting some processes that are optional. Although I think that is not the case on most machines and compilers, my final answer would be Yes.