CMSC 11 Handout (Pointers and Recursions)
Prepared by: MAADClariño

POINTERS

A pointer is a variable that stores an address/memory location.

It provides indirect access to a value.

Example:

```
int a=5; //a is an int-type variable

int *b; //b is a pointer to an int-type value

b=&a; //b has the address of a
      // &-means address
```

We can also extend this by having a pointer to another pointer.

```
int **c;
c=&b; // c holds the address of b, which is a pointer to an integer.
```

VARIABLE	VALUE	ADDRESS
a	5	@31a
b	@31a	124c
c	124c	458s

Based on the diagram above: (NOTE: & - ADDRESS * - VALUE)

a - 5	**b - @31a**	**c - 124c**
&a - @31a	**&b - 124c**	**&c - 458s**
***a – INVALID since a is not a pointer**	***b - 5**	***c - @31a**
****a – INVALID since a is not a pointer to another pointer**	****b – INVALID since a is not a pointer to another pointer**	****c - 5**

RECURSIONS

A recursion is a function that may call itself directly or indirectly.

Important note:

A recursion has two main parts.
-> Call to itself
-> Base condition or stopping criterion (when to stop calling itself)

```
int funcA(int x){
      int y;
      if(x==1)  //This is the base condition.
            return x;
      else{
            y=x;
            x=funcA(x-1);
            return(x*y);
      }
}

main(){
      int a;
      a=funcA(3);
      printf("%d", a);
}
```

Stack of function calls: The first function to be executed is the main() function.

main() will call funcA(3).

Function calls	Execution of function
funcA(3)	x=3; y=3;
main	

Inside funcA(3), funcA(2) will be called. Execution of funcA(3) will be put on hold.

Function calls	Execution of function
funcA(2)	x=2; y=2;
funcA(3)	x=3; y=3;
main	

Inside funcA(2), funcA(1) will be called. Execution of funcA(2) will be put on

hold.

Function calls	Execution of function
funcA(1)	
funcA(2)	**x=2; y=2;**
funcA(3)	**x=3; y=3;**
main	

Since x is already equal to 1, funcA(1) will return the value of x. The return value will be captured by funcA(2).

Function calls	Execution of function
~~**funcA(1)**~~	~~**Return 1**~~
funcA(2)	**x=2; y=2; Captured value: x=1, Return (x*y) or 2**
funcA(3)	**x=3; y=3;**
main	

The return value of funcA(2) will be captured by funcA(3).

Function calls	Execution of function
~~**funcA(1)**~~	~~**Return 1**~~
~~**funcA(2)**~~	~~**x=2; y=2; Captured value: x=1, Return (x*y) or 2**~~
funcA(3)	**x=3; y=3; Captured value: x=2 Return (x*y) or 6**
main	

The return value of funcA(3) will be captured by main() and stored to *a*.

Function calls	Execution of function
~~**funcA(1)**~~	~~**Return 1**~~
~~**funcA(2)**~~	~~**x=2; y=2; Captured value: x=1, Return (x*y) or 2**~~
~~**funcA(3)**~~	~~**x=3; y=3; Captured value: x=2 Return (x*y) or 6**~~
main	**After the function call ---> a=6**

The printed value is 6.

FUNCTION CALL WITHIN ANOTHER FUNCTION CALL

The idea is same with:

node *p=(node *)malloc(sizeof(node));

The parameter of malloc is a function call. The inner function call must be executed first such that its return value will be the parameter of the outer function.

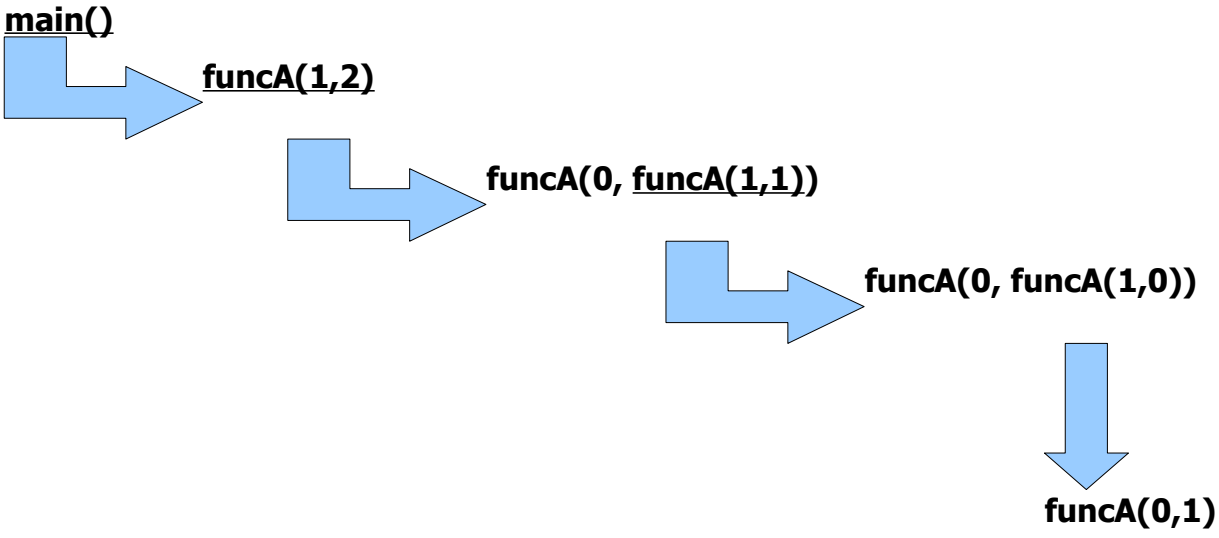
```
int funcA(int x, int y){

      if(x==0)   //This is the base condition.
           return ++y; //increments y first before returning the value
      if((x>0)&&(y==0))
                funcA(x-1,1);
    if((x>0)&&(y>0))
          funcA(x-1,funcA(x,y-1));//Function call within a function call


}

main(){
      int a;
      a=funcA(1,2);
      printf("%d", a);
      scanf("%d",&a);
}
```

This is the flow of function calls.



At this point, funcA(0,1) will return 2 to funcA(1,0) making the second parameter of the previous function call 2. We now execute the outer function call funcA(0,2).

The final return value to main() is 4.

TRY TO FIND OUT WHY...