FUNCTION AND RECURSION

Functions can be invoked in two ways: **Call by Value** or **Call by Reference**. These two ways are generally differentiated by the type of values passed to them as parameters.

The parameters passed to function are called actual parameters whereas the parameters received by function are called formal parameters.

Call By Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

Call by Reference: Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller.

Call By Value	**Call By Reference**
While calling a function, we pass values of variables to it. Such functions are known as "Call By Values".	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as "Call By References.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.

// C program to illustrate
// call by value
 	
#include
 	
// Function Prototype
void swapx(int x, int y);
 	
// Main function
int main()
{
  int a = 10, b = 20;
 	
  // Pass by Values
  swapx(a, b);
 	
  printf("a=%d b=%d\n", a, b);
	// C program to illustrate
	// Call by Reference
	
	#include
	
	// Function Prototype
	void swapx(int*, int*);
	
	// Main function
	int main()
	{
	  int a = 10, b = 20;
	
	  // Pass reference
	  swapx(&a, &b);
	
	  printf("a=%d b=%d\n", a, b);

Call By Value	Call By Reference

return 0;
}
 	
// Swap functions that swaps
// two values
void swapx(int x, int y)
{
int t;
 	
t = x;
x = y;
y = t;
 	
printf("x=%d y=%d\n", x, y);
}
 	
Output:
x=20 y=10
a=10 b=20
	return 0;
	}
	
	// Function to swap two variables
	// by references
	void swapx(int* x, int* y)
	{
	int t;
	
	t = *x;
	*x = *y;
	*y = t;
	
	printf("x=%d y=%d\n", *x, *y);
	}
	
	Output:
	x=20 y=10
	a=20 b=10

Understanding of Recursion

Recursion is a technique in which we break down a problem into one or more subproblems that are similar in form to the original problem. For example, suppose we need to add up all of the numbers in an array. We'll write a function called add_array that takes as arguments an array of numbers and a count of how many of the numbers in the array we would like to add; it will return the sum of that many numbers.

To develop the add_array function, we observe that if we had a function that would add up all but the very last number in the array, then we would simply have to add the last number to that sum and we would be done. But, the add_array function is just what we need for adding up all but the last number (as long as the array contains at least one number). After all, add_array takes an array and a count, and adds up that many array elements. If there are no numbers in the array, then zero is the desired answer. These observations suggest the following definition for the function:

```
int add_array(int arr[], unsigned count)
{
  if (count == 0)
    return 0;
  return arr[count - 1] + add_array(arr, count - 1);
}
```

Notice that the function has two components:

1. a base case, represented by the if and the return 0, in which the function does not call itself. This handles the case where there are no numbers to add.
2. a recursive case that breaks the problem down into a smaller version of the original problem together with an addition. In the recursive case, add_array is used to add together count-1 items; the count-th item is then added to this result (remember that the n-th item of an array is stored at position n-1).

Ensuring that Recursion Will Work

One of the most difficult aspects of programming recursively is the mental process of accepting that the recursive call will do the right thing. The following checklist itemizes the five conditions that must hold for recursion to work. If each of these conditions holds for your recursive function, you know that the recursion will operate correctly:

1. A recursive function must have at least one base case and one recursive case (it's OK to have more than one base case, and more than one recursive case).
2. The test for the base case must execute before the recursive call.
3. The problem must be broken down in such a way that the recursive call is closer to the base case than the top-level call.
4. The recursive call must not skip over the base case.
5. The non-recursive portions of the function must operate correctly.

Let's see whether the recursive fact function meets these criteria:

1. The first condition is met, because if (n==1) return 1 is a base case, while the ``else'' part includes a recursive call (fact(n-1)).
2. If we reach the recursive call, we must have already evaluated if (n==1); this if is the base case test, so criterion 2 is met.
3. The recursive call is fact(n-1). The argument to the recursive call is one less than the argument to the top--level call to fact. Our base case occurs when n is one. The recursive call is therefore closer to the base case as long as n is positive. If n is not positive, the recursive call does not move toward the base case, so the function will not work properly. Since n is unsigned, it cannot be negative.
4. Because n is an integer, and the recursive call reduces n by just one, it is not possible to skip over the base case.
5. Assuming that the recursive call works properly, we must now verify that the rest of the code works properly. We can do this by comparing the code with our second definition of factorial. This definition says that if n is one then n! is one. Our function correctly returns 1 when n is 1. If n is not one, the definition says that we should return (n-1)! * n. The recursive call (which we now assume to work properly) returns the value of n-1 factorial, which is then multiplied by n. Thus, the non--recursive portions of the function behave as required.

Factorial Using Recursion

Now, we will write a factorial program using a recursive function. The recursive function will call itself until the value is not equal to 0.

Now, we will write a C program for factorial using a recursive function. The recursive function will call itself until the value is not equal to 0.

Example :

```
#include <stdio.h>
int main(){
int x = 7;
printf("The factorial of the number is %d", fact(x));
return 0;
}
// Recursive function to find factorial
int fact(int y){
if (y == 0)
return 1;
return y * fact(y - 1);
}
```