CS1100 - Lecture 21

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

So far we were dealing with simple functions in which one function was called by another function. In this lecture, we will study about functions that call themselves.

Recursive Functions

In mathematics we have studied about functions which are defined in terms of itself. Factorial function is one such function, which is defined as follows.

$$n! = \begin{cases} n \times (n-1)! & : n > 0 \\ 1 & : n = 0 \end{cases}$$

Such functions which are defined in terms of itself are called recursive functions. Another example for a recursive mathematical function is given below.

$$f(n) = \begin{cases} 2 \times f(n-1) & : n > 1 \\ 3 & : n = 1 \end{cases}$$

Functions in mathematics manipulate only the data. They take input parameters and give out the value of the function. A function in a program deals with both data manipulation and manipulation of program control. Just like most other programming languages, C also supports defining recursive functions.

Let us consider writing a recursive function to compute the factorial of a number `n`. The function header of the recursive function is the same as the function header of the simple iterative function for factorial that we explained in the last class. The function takes one integer as input parameter and returns a value of type `long long int`. A first attempt of writing a recursively defined function for factorial is given below.

Attempt 1:

```
long long int factorial(int k)
{
    long long int f;
    f=factorial(k-1);
    f=f*k;
    return f;

}
```

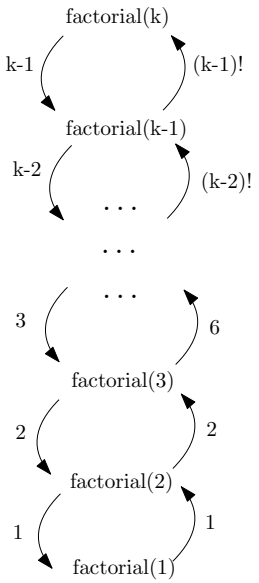
This function definition is not correct because the function does not define a base condition according to which the function stops calling itself recursively and returns some value. If we write the function this way, the execution of this function will never terminate.

It is very important to have the termination condition(s) specified properly in the function definition. A corrected version of the recursive function definition of factorial can be written as follows.

```
long long int factorial(int k)
{
    long long int f;
    if(k<0)
        return (-1);
    if(k==0||k==1)
        return (1);
    else
    {
        f=factorial(k-1);
        f=f*k;
        return f;
    }
}
```

In the above function, if the input parameter `k` is found to be negative, -1 is returned to indicate error. For `k` values 0 or 1, the function returns 1. Otherwise, the function `factorial` calls itself with parameter `k-1`; completes recursively executing `factorial(k-1)` and on return from the recursive call `factorial(k-1)`, the value of $(k-1)!$ is computed and assigned to the variable `f`. Then, `f=f*k;` is executed, so that the value of `f` gets updated to `(k-1)!k = k!` and this value of `f` is returned. Towards the end of this lecture, we will see how to formally prove the correctness of the above function.

The control flow associated with the above function is shown in the figure below.

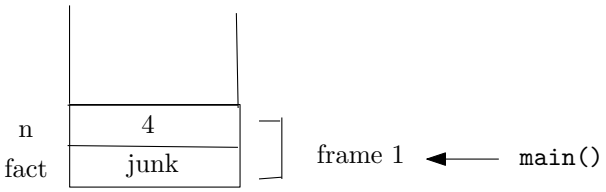


A `main()` function which uses the above function definition can be written as follows.

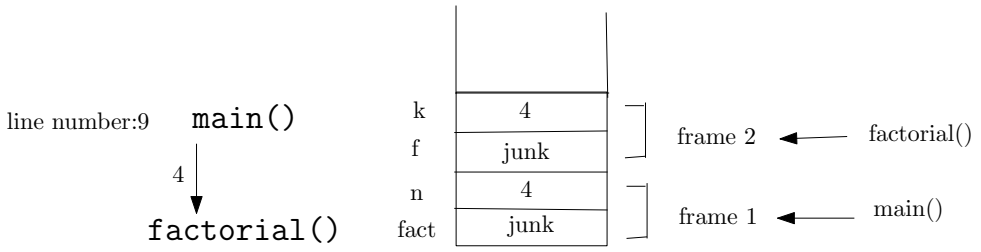
```
int main()
{
  int n;
  long long int fact;
  printf("enter a non-negative number n (<20)\n");
  scanf("%d", &n);
  if (n < 20)
  {
      fact=factorial(n);
      printf("%d! = %lld \n",n, fact);
  }
  else
      printf("n too large \n");
  return (0);
}
```

Detailed control and data flow of the program for the input n=4:

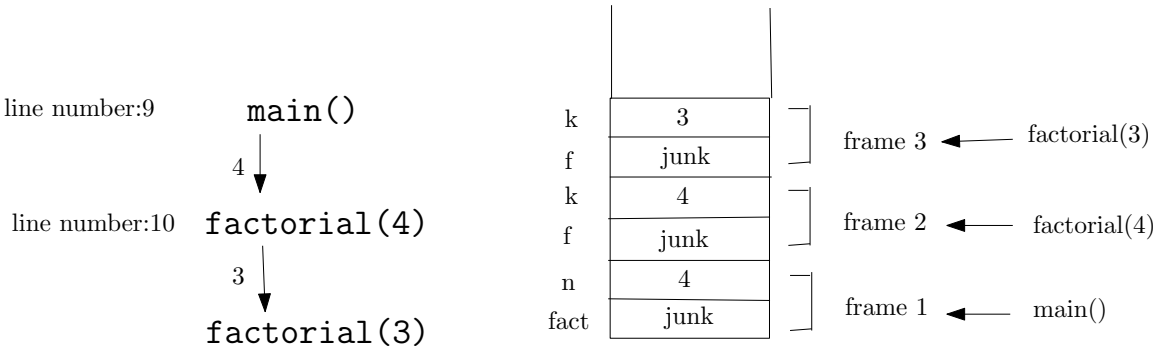
When the program starts execution the activation record of `main()` gets created in memory and it will have memory locations for storing the local variables `fact` and `n`. After reading `n=4` in the `main()` function, the memory diagram is as follows.



When the line `fact=factorial(n);` starts execution, a new activation record for `factorial()` gets created in memory with locations for storing variables `f` and `k`. Since the function call is made with the value of the parameter `n` as 4, the value of the actual parameter `n` in the function call is copied to the location of `k` in the new activation record. After this, the program control passes to the first line of `factorial()` function for executing `factorial(k)` with k=4. The contents of memory locations at this point of execution and the control flow during the function call is as follows.

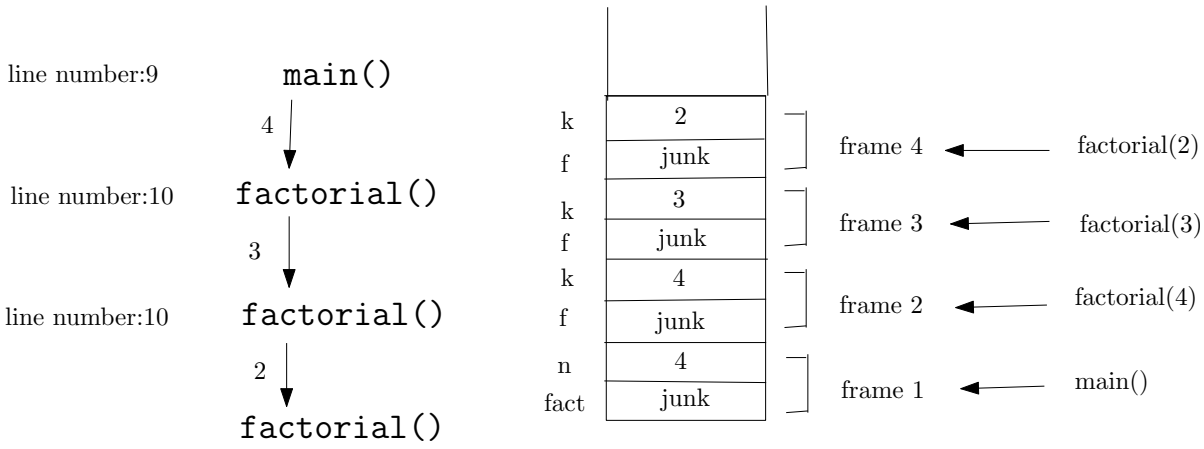


Now the execution of `factorial(k)` with k=4 begins. Since both the `if` conditions are *false*, the instruction `f=factorial(k-1);` (line number 10) will start to execute. For this, again a new activation record for `factorial()` gets created in memory with locations for storing new instances of variables `f` and `k`. Note that these variables in the new activation record are completely different from those in the previous activation record of `factorial`. Since this time the function call `factorial()` is made with the value of the actual parameter `k-1` (which has value 3), the value 3 is copied to the location of `k` in the new activation record. After this, the program control passes to the first line of `factorial()` function for executing `factorial(k)` with k=3. The contents of memory locations at this point of execution and the control flow during the function call is as follows.

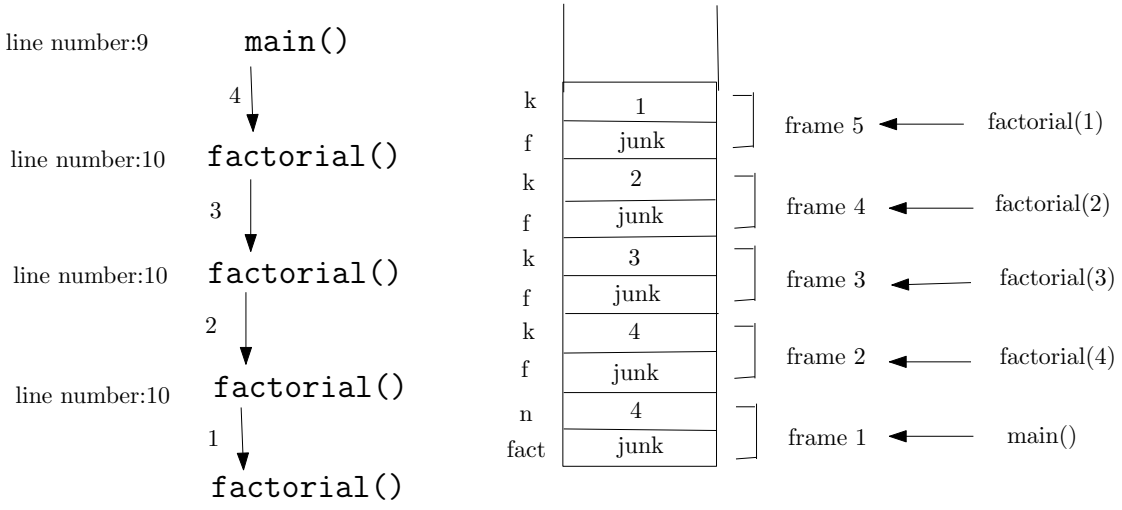


When `factorial(k)` with `k=3` execute, again both the `if` conditions are *false* and the instruction `f=factorial(k-1);` (line number 10) will start to execute.

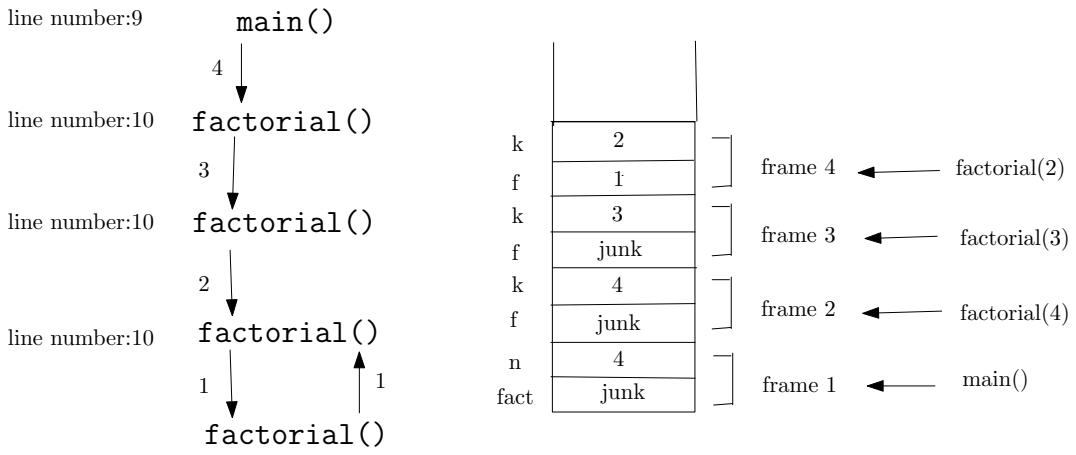
For this, again a new activation record for `factorial()` gets created in memory with locations for storing new instances of variables `f` and `k`. Since the function call `factorial()` is made with the value of the actual parameter `k-1` (which has value 2 as per the activation record of the calling function), the value 2 is copied to the location of `k` in the new activation record. After this, the program control passes to the first line of `factorial()` function for executing `factorial(k)` with k=2. The contents of memory locations at this point of execution and the control flow during the function call is as follows.



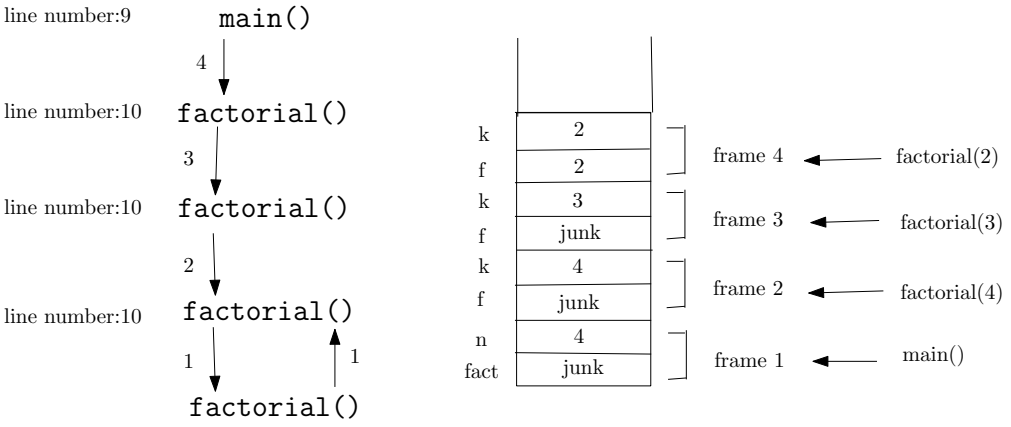
In the same way, while executing line number 10 of `factorial(k)` with `k=2`, a call `factorial(k-1)` is made which invokes starting execution of a new instance of `factorial(k)` with `k=1`. The contents of memory locations at this point of execution and the control flow during the function call is as follows.



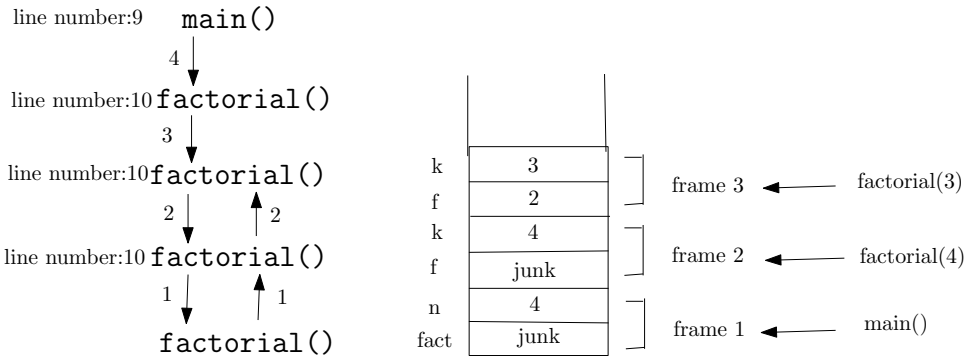
Now, the function `factorial(k)` with `k=1` starts execution and since the condition `if(k==0||k==1)` is *true*, the function returns the value 1 to the calling function. The return value is copied to the location of the variable `f` of the calling function (`factorial(k)` with `k=2`) and the stack frame of the function call `factorial(1)` is deleted from the memory. After this the program control continues from line number 11 of `factorial(k)` with `k=2`. The contents of memory locations at this point of execution and the control flow diagram is as follows.



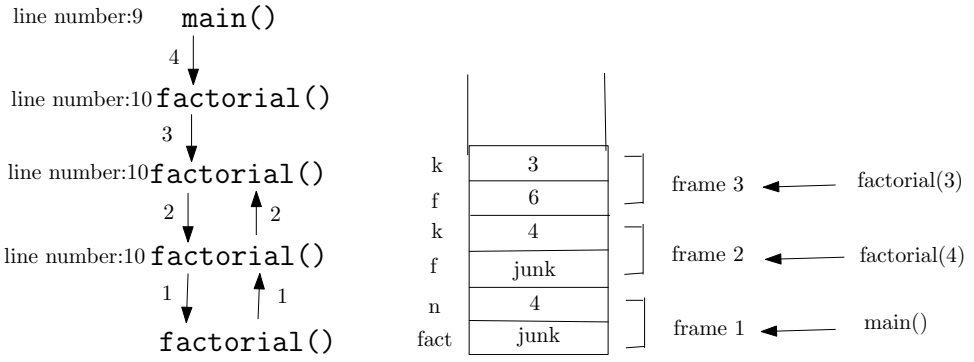
In line number 11 of `factorial(k)` with `k=2`, the value of `f` gets updated to `f*k` which is $1 \times 2{=}2$. In the next line, the new value of `f` will be returned to the calling function (`factorial(k)` with `k=3`). The memory diagram just before the `return` statement is as given below.



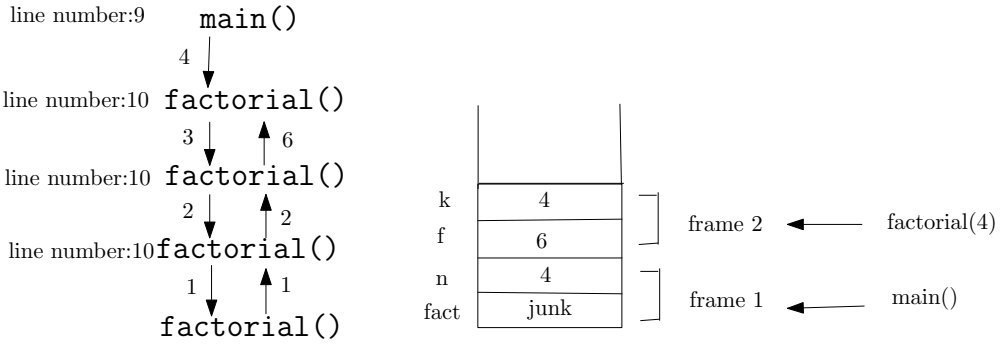
Now, the function `factorial(k)` with `k=2` returns the value 2. The return value is copied to the location of the variable `f` of the calling function (`factorial(k)` with `k=3`) and the stack frame of the function `factorial(2)` is deleted from the memory. After this the program control continues from line number 11 of `factorial(k)` with `k=3`. The contents of memory locations at this point of execution and the control flow diagram is as follows.



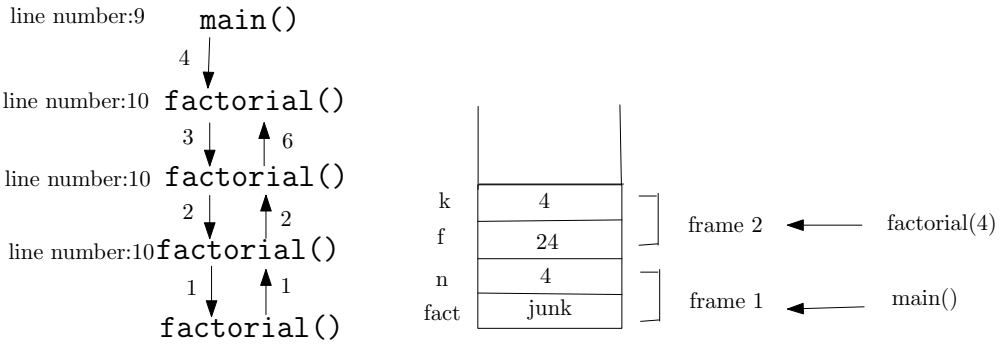
In line number 11 of `factorial(k)` with `k=3`, the value of `f` gets updated to `f*k` which is $2 \times 3=6$. In the next line, the new value of `f` will be returned to the calling function (`factorial(k)` with `k=4`). The memory diagram just before the `return` statement is as given below.



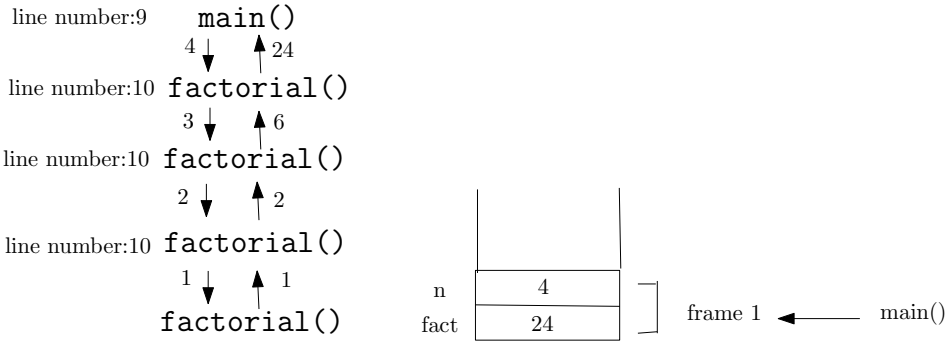
Now, the function `factorial(k)` with `k=3` returns the value 6. The return value is copied to the location of the variable `f` of the calling function (`factorial(k)` with `k=4`) and the stack frame of the function `factorial(3)` is deleted from the memory. After this, the program control continues from line number 11 of `factorial(k)` with `k=4`. The contents of memory locations at this point of execution and the control flow diagram is as follows.



In line number 11 of `factorial(k)` with `k=4`, the value of `f` gets updated to `f*k` which is $6 \times 4=24$. In the next line, the new value of `f` will be returned to the calling function `main()`. The memory diagram just before the `return` statement is as given below.



The value 24 is returned to the `main()` function by `factorial(k)` with `k=4` and the activation record of `factorial(k)` with `k=4` is removed from memory. The return value is copied to the location of `fact` in the `main()` function. The memory diagram is as given below.



The program execution continues in `main()` and it displays `4!=24` to the user.

The `factorial()` function can be also re-written in a more concise way as follows.

```
long long int factorial(int k)
//returns k! for any input k such that 0 <= k <= 19
{
      if(k<0)
        return(-1);
      if(k==1 || k ==0)
        return(1);
      else
      {
        return(k * factorial(k-1));
      }
}
```

Proof of correctness of a recursive function

As we saw in the example of `factorial()` function, the control flow can go very deep and there can be simultaneously many activation records for the same recursive function in memory. This makes keeping track of the data and control manipulation a difficult task for human beings. Therefore, we need a different way of proving the correctness of a recursively defined function.

Usually the proof of correctness can be done using mathematical induction. Instead of worrying about deeper and deeper levels of a recursive function call, we do the following:

- Induction base case: By analysing the function, prove that the function works correctly for the parameter values for the base cases. This will be usually simple to do.
- Induction hypothesis: Assume that the function works correctly for smaller values of the parameters than the formal parameters given in the function header.
- Inductive step: With the above assumption, analyse the function definition and prove that the function works correctly for the parameter value given by the formal parameter.

In this method of proof, we need not go too deep to analyse the correctness. Often just assuming correctness of the previous level of the recursive function call will be enough.

Using the above recipe, the proof of correctness of the `factorial()` function can be given as follows.

- It is clear that for `k=0`, `k=1` and `k<0`, the `factorial()` function terminates and returns the correct value.
- Now assume that the `factorial()` function works correctly (and terminates) for parameter values less than the value of `k`. In particular, by this assumption the function call `factorial(k-1)` terminates and correctly returns the value (k-1)!.
- In the `factorial()` function, the return value of `factorial(k-1)` is multiplied with the value of `k` and the result of this computation is the return value of the function call `factorial(k)`. It is clear that the return value is k*(k-1)!=k! which is the correct result. This also shows that the function terminates for all input parameter values.

In general, during the inductive proof it is not necessary that the formal parameter values directly reduce. But it can be some other hidden parameter that may be reducing so that finally the termination of the function is guaranteed when the value of this hidden parameter reaches some base values. We will see some examples for this in upcoming lectures.

Exercise: By looking at the following programs, predict their outputs. Execute these programs and verify whether your guess is correct. Prove that the functions work as intended.

Program 1:

```
#include<stdio.h>
void triangle2(int);
int main()
{
   int n, r;
   printf("enter n \n");
   scanf("%d",&n);
   if(n>0)
   {
       triangle2(n);
   }

   return 0;
}

void triangle2(int n)
/* Explain what this does */
{
   int i;
   if (n<=0)
       return;
   else
   {
       triangle2(n-1);
       for(i=0; i<n; i++)
       {
          printf("0 ");
       }
       printf("\n");
       return;
   }
}
```

Program 2:

```
#include<stdio.h>
void triangle3(int);
int main()
{
   int n, r;
   printf("enter n \n");
   scanf("%d",&n);
   if(n>0)
   {
       triangle3(n);
   }

   return 0;
}

void triangle3(int n)
/* Explain what this does */
{
   int i;
   if (n<=0)
       return;
   else
   {
       for(i=0; i<n; i++)
       {
          printf("0 ");
       }

       printf("\n");

       triangle3(n-1);
       return;
   }
}
```