DATA STRUCTURE

Ohlone College
CS 143-03 Programming with Data Structures

IN ORDER TO UNDERSTAND RECURSION, ONE MUST FIRST UNDERSTAND RECURSION,

Recursion

Recursion is a method of solving a problem by solving a simpler version or versions of the original problem (and perhaps doing some additional computations). A **recursive** solution must break a problem up into one or more simpler versions of the original problem. These might be simpler in that they use a smaller value of a parameter. Also, the chain of recursive calls that gets generated must always end in a **stopping case**, a case where the answer is directly known so that further recursive function calls are not used.

Recursive Function

A recursive function is one that calls itself. This ability enables a recursive function to be repeated with different argument values. You can use recursion as an alternative to iteration (looping). Generally, a recursive solution is less efficient in terms of computer time than an iterative one due to the overhead for the extra function calls. However, in many instances the use of recursion enables us to specify a natural, simple solution to a problem that would otherwise be very difficult to solve. For this reason, recursion is an important and powerful tool in problem solving and programming.

The Nature of Recursion

Problems that lend themselves to a recursive solution have the following characteristics:

- One or more simple cases of the problem (called stopping cases) that have a simple, non-recursive solution.
- For the other cases, there is a process (using recursion) for substituting one or more reduced cases of the problem that are closer to a stopping case.
- The problem can eventually be reduced to stopping cases only, all of which are relatively easy to solve.

The recursive algorithms that we write will generally consist of an *if statement* with the form below.

```
if the stopping case is reached then
    Solve it
else
    Reduce the problem using recursion
```

The Factorial Function

A favorite example for illustrating recursion is the factorial function. In mathematics, one usually defines the factorial of a non-negative integer n as follows, where n! means the factorial of n.

```
0! = 1
n! = n * (n-1) * (n-2) * ... * 2 * 1   for n > 0
```

This easily leads to a normal iterative factorial function. This could be written, for example, as:

```
/* Given:  n  A non-negative integer.
   Task:   To compute the factorial of n.
   Return: The factorial result.
*/
long factorial(int n)
{
   int k;
   long result = 1;

   for (k = 2; k <= n; k++)
      result = result * k;

   return result;
}
```

When this function gets called with a value of 0 or 1 in n, the for loop gets skipped, so that a value of 1 is returned. Trace the execution of this function for the next few integers to verify that the following factorials are computed correctly:

n	**factorial(n)**
0	1
1	1
2	2
3	6
4	24
5	120
6	720

Note that after the first few lines, the factorials start to get pretty big. That is why we used a long to hold the factorials, so that there is less chance of overflow. However, if you use a large enough value of n, overflow will still happen. You will start to get meaningless answers for the factorials, such as negative numbers.

Where is the recursion? Thus far we have none. Is there any way of seeing a simpler version of the computation of a factorial inside of the computation of a factorial? Sure. For example, in computing 4! = 4 * 3 * 2 * 1, we see that it is the same as 4 * 3!, where 3! is a simpler factorial computation. A recursive definition of the factorial function can be written as follows:

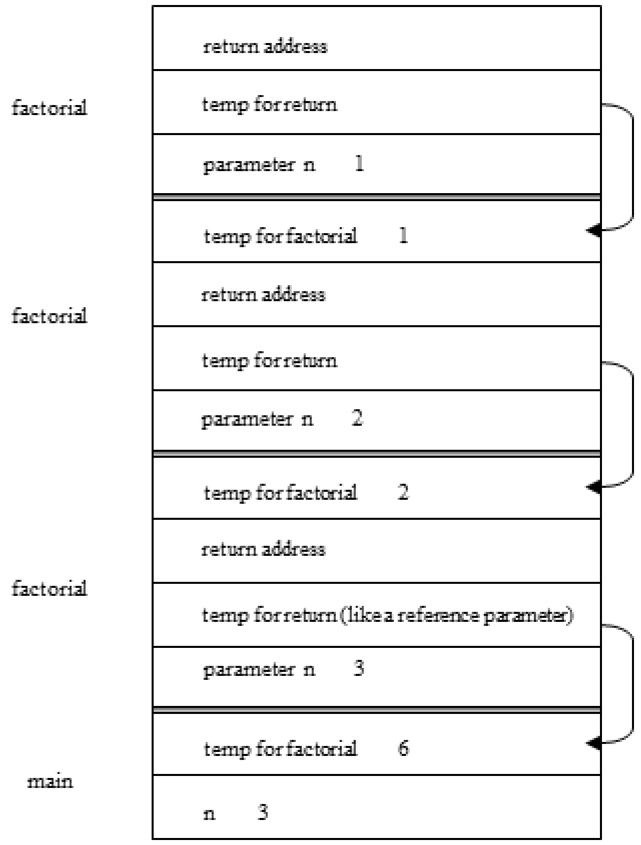
```
0! = 1
n! = n * (n-1)!   for n > 0
```

This leads directly to the recursive C++ function found in the example code program **fact.cpp** and copied in below for convenience:

```
/* Given:  n  A non-negative integer.
   Task:   To compute the factorial of n.
   Return: The factorial result.
*/
long factorial(int n)
{
   if ((n == 0) || (n == 1))   // stopping cases
      return 1;
   else                        // recursive case
      return n * factorial(n - 1);
}
```

When the value of n is 0 or 1 we have stopping cases, where we directly return the answer. Otherwise, we have the recursive case, where the factorial function calls itself (but with a parameter that is smaller by 1).

How do we trace the execution of a recursive function? One method is to draw a picture of the run-time stack. If, for example, we had a test program that called factorial(3) we would get a picture of the run-time stack as shown below:

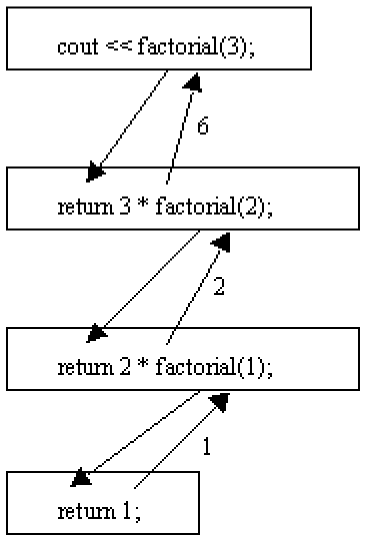


Since our function returns an answer in the function name, we assume as usual that the compiler sets this up to operate like a reference parameter, using a pointer to some temporary location for holding the factorial value. The main function calls factorial so that a stack frame for factorial is put on the stack with a value of 3 in the parameter n. But that invocation of factorial calls factorial again, so another, almost identical, stack frame is put on the stack. This one contains a value of 2 for n, however. This invocation calls factorial yet another time, so another stack frame for factorial is placed on the stack, this time with a value of 1 for n.

Thus you can see how the chain of recursive calls results in the placing of successive stack frames for the same function on the run-time stack. This chain ends when we reach a stack frame that corresponds to a stopping case. Since an answer is known for the stopping case, this answer, 1, is sent back to the previous stack frame by following the pointer to the temporary location for the factorial value. The top stack frame is then removed. Then the previous invocation of factorial can continue. It finishes the suspended computation of 2 * factorial(1) using the 1 we just stored. The product, 2, is then stored using the pointer to the temporary location in the previous stack frame. The current stack frame is removed, so that only one stack frame for factorial remains. This one finishes its suspended computation of 3 * factorial(2) as 3 * 2, giving 6. The 6 is stored by following the pointer to the temporary variable for the factorial value in the stack frame back out to the main function. Finally, the stack frame for factorial is removed.

Notice that the stack frames for factorial first build up on the stack. When a stopping case is reached, the stack frames start coming off with answers sent back at each step by means of the pointer mechanism. Since the picture is fairly complicated with all of the details of stack frames, one wonders if there might be an easier way to trace a recursive function. One method is to skip the run-time stack and simply show in boxes the original function call and then the code that it generates, followed by any code that one generates due to a recursive call, etc. Arrows are used to link each call to the code that it generates. A backwards arrow labeled with a value is used to show a result sent back.

Here is the simplified drawing:



This is much simpler to follow. Try tracing a call of factorial(5) this way. Also try tracing a call of factorial(-1). What happens? We did not really intend that this function would ever be called with a negative parameter value. Still, nothing prevents us from doing this. Clearly, we will first generate the statement return -1 * factorial(-2);, this will generate return -2 * factorial(-3);, etc. It looks like it will run forever, since it produces a value of n that is 1 smaller each time. However, if you know a little computer architecture, you realize that on most computers successively subtracting 1 eventually yields a *positive* integer! This is because computer integers have a limited range. When the lowest integer is reached and we subtract 1, we probably get the largest positive integer due to wrap-around. This is the typical result on most computers. Thus the chain of recursion probably ends once we wrap around and decrement n to 1, but we will get a nonsense answer. Besides, the mathematical factorial function is not defined for negative numbers. Note that one of our product computations will most likely overflow if we ever even get to the point of computing the products. Another possible result is overflow of the run-time stack. This could occur if we run out of stack space before we hit a stopping case. *Interesting topics for another course...*

To learn more:

Data Structure - Recursion Basics