RECURSIVE FUNCTIONS IN CPP

Recursion

In C/C++, a function can call itself. A function is said to be *recursive* if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.

A simple example of a recursive function is **factr( )**, which computes the factorial of an integer. The factorial of a number **n** is the product of all the whole numbers between 1 and **n**. For example, 3 factorial is 1 x 2 x 3, or 6. Both **factr( )** and its iterative equivalent are shown here:

```
/* recursive */
int factr(int n) {
int answer;
if(n==1) return(1);
answer = factr(n-1)*n; /* recursive call */
return(answer);
}
/* non-recursive */
int fact(int n) { int
t, answer; answer
= 1; for(t=1; t<=n;
t++)
answer=answer*(t);
return(answer);
}
```

The nonrecursive version of **fact( )** should be clear. It uses a loop that runs from 1 to **n** and progressively multiplies each number by the moving product. The operation of the recursive **factr( )** is a little more complex. When **factr( )** is called with an argument of 1, the function returns 1. Otherwise, it returns the product of **factr(n−1)*n**. To evaluate this expression, **factr( )** is called with **n−1**. This happens until **n** equals 1 and the calls to the function begin returning.

Computing the factorial of 2, the first call to **factr( )** causes a second, recursive call with the argument of 1. This call returns 1, which is then multiplied by 2 (the original **n** value). The answer is then 2. Try working through the computation of 3 factorial on your own. (You might want to insert **printf( )** statements into **factr( )** to see the level of each call and what the intermediate answers are.) When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the

values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function. Recursive functions could be said to "telescope" out and back.

Often, recursive routines do not significantly reduce code size or improve memory utilization over their iterative counterparts. Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls. In fact, many recursive calls to a function could cause a stack overrun. Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack could be exhausted. However, you probably will not have to worry about this unless a recursive function runs wild.

The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms. For example, the Quicksort algorithm is difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively. When writing recursive functions, you must have a conditional statement, such as an **if,** somewhere to force the function to return without the recursive call being executed. If you don't, the function will never return once you call it. Omitting the conditional statement is a common error when writing recursive functions. Use **printf( )** liberally during program development so that you can watch what is going on and abort execution if you see a mistake.

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf