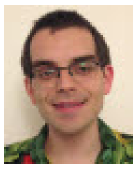
Lesson 16: Recursion in C


By Alex Allain

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it *is* similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, then I will first build a 9 foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.

A simple example of recursion would be:

```
void recurse()
{
    recurse(); /* Function calls itself */
}

int main()
{
    recurse(); /* Sets off the recursion */
    return 0;
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

```
#include <stdio.h>

void recurse ( int count ) /* Each call gets its own copy of count */
{
    printf( "%d\n", count );
    /* It is not necessary to increment count since each function's
       variables are separate (so each count will be initialized one greater)
     */
    recurse ( count + 1 );
}

int main()
{
  recurse ( 1 ); /* First function call, so it starts at one */
  return 0;
}
```

This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it was previous by passing in count + 1. Keep in mind that it is not a function call restarting itself; it is hundreds of function calls that are each unfinished.

The best way to think of recursion is that each function call is a "process" being carried out by the computer. If we think of a program as being carried out by a group of people who can pass around information about the state of a task and instructions on performing the task, each recursive function call is a bit like each person asking the next person to follow the same set of instructions on some part of the task while the first person waits for the result.

At some point, we're going to run out of people to carry out the instructions, just as our previous recursive functions ran out of space on the stack. There needs to be a way to avoid this! To halt a series of recursive calls, a recursive function will have a condition that controls when the function will finally stop calling itself. The condition where the function will not call itself is termed the base case of the function. Basically, it will usually be an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again. (Or, it could check if a certain condition is true and only then allow the function to call itself).

A quick example:

```
void count_to_ten ( int count )
{
    /* we only keep counting if we have a value less than ten
       if ( count < 10 )
       {
           count_to_ten( count + 1 );
       }
}
int main()
{
  count_to_ten ( 0 );
}
```

This program ends when we've counted to ten, or more precisely, when count is no longer less than ten. This is a good base case because it means that if we have an input greater than ten, we'll stop immediately. If we'd chosen to stop when count equaled ten, then if the function were called with the input 11, it would run out of memory before stopping.

Notice that so far, we haven't done anything with the result of a recursive function call. Each call takes place and performs some action that is then ignored by the caller. It is possible to get a value back from the caller, however. It's also possible to take advantage of the side effects of the previous call. In either case, once a function has called itself, it will be ready to go to the next line after the call. It can still perform operations. One function you could write could print out the numbers 123456789987654321. How can you use recursion to write a function to do this? Simply have it keep incrementing a variable passed in, and then output the variable twice: once before the function recurses, and once after.

```
void printnum ( int begin )
{
    printf( "%d", begin );
    if ( begin < 9 )         /* The base case is when begin is no longer */
    {                           /* less than 9 */
        printnum ( begin + 1 );
    }
    /* display begin again after we've already printed everything from 1 to 9
     * and from 9 to begin + 1 */
    printf( "%d", begin );
}
```

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function from 9 to begin.

This is, however, just touching on the usefulness of recursion. Here's a little challenge: use recursion to write a program that returns the factorial of any number greater than 0. (Factorial is number * (number - 1) * (number - 2) ... * 1).

Hint: Your function should recursively find the factorial of the smaller numbers first, i.e., it takes a number, finds the factorial of the previous number, and multiplies the number times that factorial...have fun. :-)

Still not getting it? Ask an expert!
Previous: Linked Lists
Next: Functions with variable arguments
Back to C Tutorial Index