Recursion

Definition: Any time the body of a function contains a call to the function itself.

So, just as you are allowed to call function B from within function A, you are ALSO allowed to call function A from within function A!

Potential Problem: But if my function calls itself, how can we ever finish executing the original function?

What this means is that some calls to the function MUST NOT result in a recursive call. I think this can best be seen from an example.

// Pre-conditions: e is greater than or equal to 0.
// Post-conditions: returns b^e.
int Power(int base, **int** exponent**) {**

    if (exponent == 0 **)**
        return 1;
    else
        return (base***Power(**base**,** exponent-1**));**
}

To convince you that this works, let's look at an example:

Say we were trying to evaluate the expression

Power(5, 2)

Power(5,2) returns 5***Power(**5,2-1) = 5***Power(**5,1**)**

To evaluate this expression we must evaluate:

Power(5,1**)** returns 5***Power(**5, 0**)**

Finally, we have:

Power(5,0**) returns 1.**

Thus, Power(5,1**) returns 5*1, which is 5.**

Finally, Power(5,2) returns 5***Power(**5,2-1) = 5*5 = 25, and we are done.**

General Structure of Recursive Functions

What we can determine from the example above is that in general, when we have a problem, we want to break it down into chunks, where one of the chunks is a smaller version of the same problem.

(In the case of Power, we utilized the fact that $x^y = x * x^{y-1}$, and realized that x^{y-1} is in essence an easier version of the original problem.)

Eventually, this means that we break down our original problem enough that our sub-problem is quite easy to solve. At this point, rather than making another recursive call, directly return the answer, or complete the task at hand.

So, ideally, a general structure of a recursive function has a couple options:

1) Break down the problem further, into a smaller subproblem
2) OR, the problem is small enough on its own, solve it.

When we have two options, we often use an if statement. This is typically what is done with recursion. Here are the two general constructs of recursive functions:

Construct 1

```
if (terminating condition) {
      DO FINAL ACTION
}
else {
      Take one step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

Construct 2

```
if (!(terminating condition) ) {
      Take a step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

Typically, functions that return values take on the first construct, while void recursive functions use the second construct. Note that these are not the ONLY layouts of recursive programs, just common ones.

Example using construct 1

Let's write a function that takes in one positive integer parameter n, and returns the sum 1+2+...+n. (You probably did this is C numerous times, yet surprisingly it never gets old!)

Step 1: We need to solve this problem is such a way that part of the solution is a sub-problem of the exact same nature.

Let f(n) denote the value 1+2+3+...+n
Using our usual iterative logic, we have

f(n) = 1 + (2 + 3 + ... + n)

But, 2+3+... + n IS NOT a sub-problem of the form 1+2+...+n.

But, let's try this:

f(n) = 1 + 2 + ... + n = n + (1 + 2 + ... + (n-1))

So, here we have gotten the expression 1 + 2 + ... + (n-1) which IS a sub-problem we were looking for. Hence, we have

 f(n) = 1 + 2 + ... + n = n + (1 + 2 + ... + (n-1)) = n + f(n-1)

If we look at the construct 1, the first thing we need to determine is a terminating condition. We know that f(0) = 0, so our terminating condition can be n=0.

Furthermore, our recursive call needs to be returning an expression for f(n) in terms of f(k), for some k<n. In this case, we just found that f(n) = n + f(n-1). So, now we can write our function:

```
// Pre-condition: n is a positive integer.
// Post-condition: Function returns the sum 1+2+...+n
int Triangle_Number(int n) {

    if ( n == 0 )
        return 0;
    else
        return (n + Triangle_Number(n-1));
}
```

Let's compare this to the ITERATIVE version you would have written in COP3223:

```
// Pre-condition: n is a positive integer.
// Post-condition: Function returns the sum 1+2+...+n
int Triangle_Number(int n) {
    int index, sum = 0;
    for (index=1; index <=n; index++)
        sum = sum + index;

    return sum;
}
```

Example using construct 2

We must write a function for an example of construct 2. A couple lectures ago, when I introduced loops, we wrote an algorithm that printed out a tip chart. We could have just as easily made that a function that takes in the lowest dollar value and highest dollar value on the chart. The method header of the function would look like:

// The function prints out a chart with the appropriate tips
// for meals ranging from first_val to last_val number of
// dollars, for every whole dollar amount. Both parameters
// must be positive integers.
void Tip_Chart (int first_val, int last_val)

Now, let's consider writing this function recursively.

First, we need to determine a terminating condition. That seems to be relatively simple: if the lowest dollar amount is greater than the highest dollar amount, then the chart is empty and we are done.

Next, we need to figure out how to break down our task into two steps: one that does part of the job, and secondly a recursive step.

1) It is clear that the first value we have to print of the chart is associated with the lowest dollar value, since that must appear on the chart first. Printing out this line of the tip chart can be the part of the job we accomplish.
2) Now, are we left with a sub-problem of the same nature? Yes! We simply now need to print a tip chart from the lowest dollar value+1 to the highest dollar value.

So, now, we can use these steps to come up with the function:

#define   TIP_RATE   0.15

// Pre-condition: Both parameters are integers with the first
//   one being less than or equal to the second one.
// **Post-condition:** A tip chart will be printed out, with a row for
//   each dollar amount ranging in between the
//   value of the two parameters.
void Tip_Chart (int first_val, **int** last_val*)*


```
        if ( !(first_val > last_val) ) {
                printf("On a meal of $%d", first_val);
                printf(" you should tip $%lf\n", first_val*TIP_RATE);
                Tip_Chart(first_val + 1, last_val)
        }
}
```

Using a Stack to Trace Recursive Code

A stack is a construct that can be used to store and retrieve items. It works just like a stack of plates in the cafeteria: The last plate placed on the top is the first one that must be removed. Essentially, it's a Last In, First Out(or LIFO) system.

Stacks can help us trace calls to recursive functions. Consider computing Power(8,3) using the recursive definition of Power given in lecture. We can put a line of code from our main algorithm as the first item on the stack:

Main algorithm: Unfinished: "total = Power(8,3)

Now, we need to compute that value, so the function call Power(8,3) is placed above this statement in the stack.

Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Now we repeat the process...

Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Again...

Power(3rd) b=8,e=1, Unfinished: "Power returns 8*Power(8,0)"
Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Finally,

Power(4th) b=8,e=0, returns 1
Power(3rd) b=8,e=1, Unfinished: "Power returns 8*Power(8,0)"
Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Now, we are ready to "collapse" the stack:

Power(3rd) b=8,e=1, "Power returns 8*1"
Power(2nd) b=8,e=2, Unfinished:"Power returns 8*Power(8,1)"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Power(2nd) b=8,e=2, "Power returns 8*8"
Power(1st) b=8,e=3, Unfinished: "Power returns 8*Power(8,2)"
Main algorithm: Unfinished: "total = Power(8,3)"

Power(1st) b=8,e=3, "Power returns 8*64"
Main algorithm: Unfinished: "total = Power(8,3)"

Main algorithm: "total = 512"

Practice Problem

Write a recursive function that takes in two non-negative integer parameters, and returns the product of these parameters.

// Precondition: Both parameters are non-negative integers.
// Postcondition: The product of the two parameters is retrned.
function Multipy returnsa *Num* **(first, second isoftype** *in Num***) {**

   if ((second == 0 **) || (** first = 0 **))**
      return 0;
   else
      returns (first **+Multiply(**first**,** second-1**));**
}

How can we adapt this solution to work for negative numbers?

// Precondition: None
// Postcondition: The product of the two parameters is retrned.
function Multipy returnsa *Num* **(first, second isoftype** *in Num***) {**

   if (second < 0 **)**
      return -Multiply(first,-second);
   else if (first < 0 **)**
      return -Multiply(-first, second);
   else if ((second == 0 **) || (** first = 0 **))**
      return 0;
   else
      return (first **+Multiply(**first**,** second-1**));**
}

Two more exercises for next time:

1) Write a recursive function that determines the nth Lucas number. Here is the definition of the Lucas numbers:

$L_1 = 1$, $L_2 = 3$, $L_n = L_{n-1} + L_{n-2}$, for all integers n > 2.

// Pre-condition: n > 0
// Post-condition: Returns the nth Lucas number, L_n.
int Lucas(int n);

2) Binomial coefficients, denoted C(n,k), can be computed as follows:

C(n,0) = C(n,n) = 1, for all non-negative integers n.
C(n,k) = C(n-1, k-1) + C(n-1, k), for all integers n > 0 with
$$0 < k < n.$$

// Pre-condition: n ≥ 0, and 0 ≤ k ≤ n
// Post-condition: Returns C(n,k).
int bin_coeff(int n, int k);