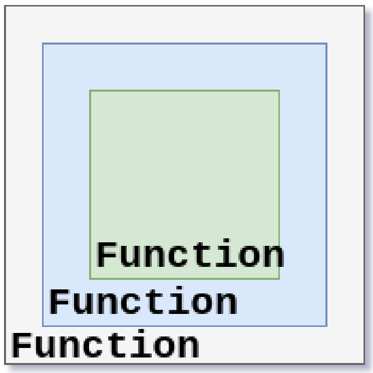
Recursion



Recursion occurs when a function invokes itself. When breaking tasks into subtasks, you may realize that you come up with a smaller example of the same task. Recursion can be utilized to accomplish said tasks. For example, recursion is useful for keeping a Binary Search Tree in tact (keeping all nodes connected during insertion and deletion), rehepifying a Heap, displaying tree structures in various ways, finding the N-th term of the Fibonacci sequence, etc. See **Binary Search Tree** and **Heap** for further information on those topics.

A recursive function starts with a function invocation elsewhere in the program with the appropriate argument(s) just like any normal function invocation you have seen before. However, within the function body, you will notice there is an invocation to the same function. An impractical example of this is:

```
void function(int x)
{
    //Code
    function(x-1);  //Function invocation to same function
}
```

When using recursion, there needs to be a function invocation in which there are no further recursive calls. This is called the **base case** or **stopping case**. Much like a loop needs to have a sentinel value (special value to terminate a loop), a recursive call needs a **base case** to prevent infinite recursion. For the examples here, the **base case** is a small number, namely 0 or 1. With each recursive call, one of the arguments is decremented by one and will eventually match the **base case** value to officially stop the recursion. A full example is:

```
long power(int base, int exp) //Assume exp is >= 0
{
    if(exp == 0) //Base case
        return 1;
    return base * power(base, exp-1); //Recursive invocation
}
```

This function computes a number (base) raised to a power (exp). This can be done recursively, because each subtask is a smaller version of the main task at hand. For example, 5^3 is 5 * 5 * 5 * 1. In terms of a recursive function, this can be thought of at 5 * 5^2. However, 5^2 is really 5 * 5^1. However, 5^1 is really 5 * 5^0. However, 5^0 is really 1. As you can see a lot of repetition was typed with a minor change of the exponent decreasing by one with each iteration.

You see that as long as the exponent is greater than 0 there will be a **return** statement executed. This **return** statement takes the base number and multiplies it by the result of base^(exp-1). When the exponent is 0, the function returns 1. This is the **base case** and no further recursive calls are made. You can now think of the function working backward to get to the first function invocation. Here is an example utilizing 5^3:

```
Example: power(5,3)
Forward Sequence to get to Base Case:
power(5,3) = 5 * power(5,2)
power(5,2) = 5 * power(5,1)
power(5,1) = 5 * power(5,0)
power(5,0) = 1

Backward Sequence from Base Case:
   1) Replace power(5,0) with 1
      power(5,3) = 5 * power(5,2)
      power(5,2) = 5 * power(5,1)
      power(5,1) = 5 * 1
   2) Replace power(5,1) with 5
      power(5,3) = 5 * power(5,2)
      power(5,2) = 5 * 5
   3) Replace power(5,2) with 25
      power(5,3) = 5 * 25
   4) Original function invocation
      return 125
```

Here is another similar example to compute the factorial of a number (the maximum number is 20 factorial if you are returning type long). Factorial 4 is really 4 * factorial(3). Factorial 3 is really 3 * factorial(2). Factorial 2 is really 2 * factorial(1). Factorial(1) is really 1 (**base case**). Here is an example of factorial(4):

```
long factorial(int n) //Assume n is >= 0
{
    if(n <= 1) //Base Case 0 or 1 assuming the above statement
        return 1;
    return n * factorial(n-1); //Recursive invocation
}


Example: factorial(4)
Forward Sequence to get to Base Case
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
```

```
Backward Sequence from Base Case
   1) Replace factorial(1) with 1
      factorial(4) = 4 * factorial(3)
      factorial(3) = 3 * factorial(2)
      factorial(2) = 2 * 1
   2) Replace factorial(2) with 2
      factorial(4) = 4 * factorial(3)
      factorial(3) = 3 * 2
   3) Replace factorial(3) with 6
      factorial(4) = 4 * 6
   4) Original function invocation
      return 24
```

Not all recursive functions need to return a value. As long as there is a **base case** somewhere in the function, recursion will work. Here is an example of a function that displays the numbers 3 - 1 inclusive, Base Case, and then 1 - 3 inclusive.

```
void example(int x) //Assume x >= 0
{
    if(x == 0)
        cout << "Base Case ";
    else {
        cout << x << ' ';
        example(x-1); //Recursive invocation
        cout << x << ' '; //Executes after recursive calls complete
    }
    return; //Return to previous function invocation (optional to put)
}
```

```
Example: example(3)
   1) cout 3 and invoke example(2)
   2) cout 2 and invoke example(1)
   3) cout 1 and invoke example(0)
   4) cout Base Case and return to previous invocation
   3) cout 1 and return to previous invocation
   2) cout 2 and return to previous invocation
   1) cout 3 and end recursive function
Output: 3 2 1 Base Case 1 2 3
```

Recursion is similar to looping. Just as a loop can execute infinitely, a recursive function can do the same without a proper **base case**. A **stack overflow** may occur if your function never stops recursively calling itself. A **stack overflow** is when you try to use more memory on the stack but all of the stack's memory is taken up. Some compilers will issue a warning stating that the recursive function will never stop calling itself to prevent this outcome.