Recursion

This lesson explains the concept of recursion using the factorial example

We'll cover the following

Recursion Definition

Recursion is a method of function *calling* in which a function calls *itself* during execution.

Example

Let’s start by showing an example and then discussing it.

```
#include <iostream>
using namespace std;

int factorial(int n)
{
   if(n == 1 || n == 0)      // if n equals 1 or 0 we return 1
   {
      return 1;
   }
   else
   {
      return n*factorial(n-1);    //recursively calling the function if n is other then 1 or 0
   }
}

int main() {
  int temp = factorial(4);    //computing the factorial of 4
  cout << "The value of the factorial computed is: "<< temp << endl;
  return 0;
}
```

This is a classic application of *recursion*. This function calculates the **factorial** of a

number.

> **Note:** The *factorial* of **n**, written **n!**, is the product of every number from **1 to n**. So we can say that **4!** = **4×3×2×1**.

Solution Explained

Let's step through what happens in our *function* when we call `num = factorial(4)`.

- In **line 17**, `factorial(4)` is called
  - Inside the `factorial` function, since, **n=4** we take the `else` path. We `return` `4×factorial(n-1)`, **line 12**.
  - `factorial(3)` is called
    - Since **n=3**, we take the `else` path. We return `3×factorial(n-1)`, **line 12**.
    - `factorial(2)` is called
      - Since **n=2**, we take the `else` path. We return `2×factorial(n-1)`, **line 12**.
      - `factorial(1)` is called
        - Since **n=1**, we take the *first* path, **line 6**, and finally return **1** to the *previous* function.
      - `factorial(1)` returns **1** so `factorial(2)` can return **2×1...2**.
    - `factorial(2)` returns **2** so `factorial(3)` can return **3×2...6**.
  - `factorial(3)` returns **6** so `factorial(4)` can return **4×6...24**.

Many times, a *recursive* solution to a problem is very easy to program.

- The drawback of using *recursion* is that there is a lot of *overhead*.

Every time a function is called, it is placed in *memory*. Since you don't **exit** the `factorial` function until **n** reaches **1**, **n** functions will reside in *memory*. This isn't a problem with the simple `factorial(4)`, but other functions can lead to serious memory requirements.

Well, that was all on *recursion*. Now in the next chapter, we will discuss pointers

and arrays. Stay tuned!