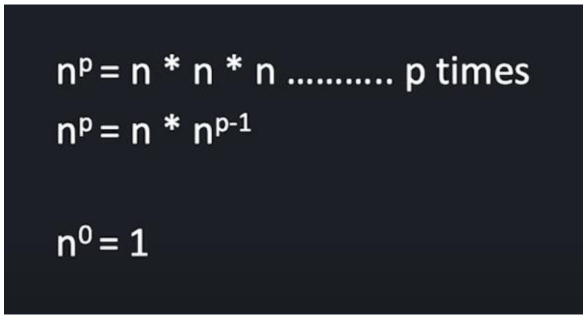
RECURSION

- When a function calls itself to make the problem smaller.
- A Base Case
  - Terminate all recursive calls
  - Clearly identifiable
  - A Recurrence Progression

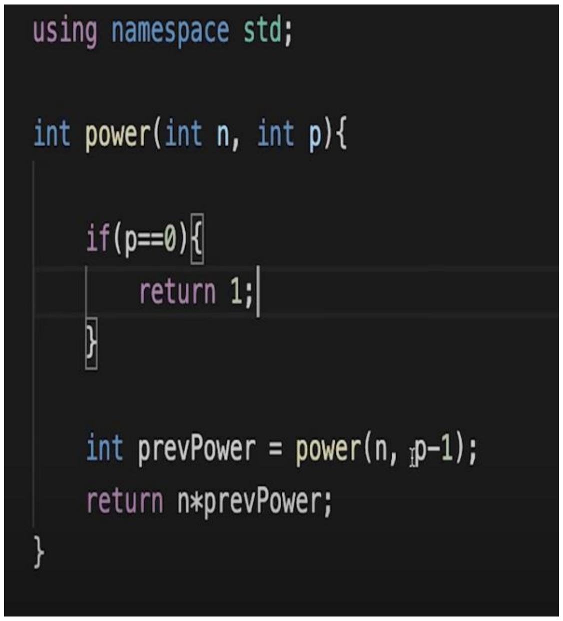
- Stack frame – **our activation record**
- keeps the information about local variables, formal parameters, return address and all information passed to the caller function.
-



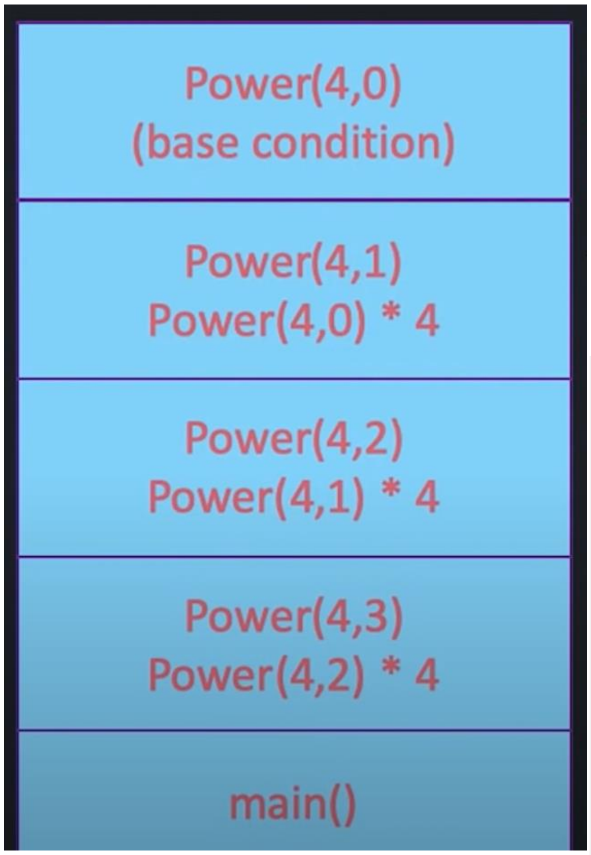
$n^p = n * n * n$ .... p times

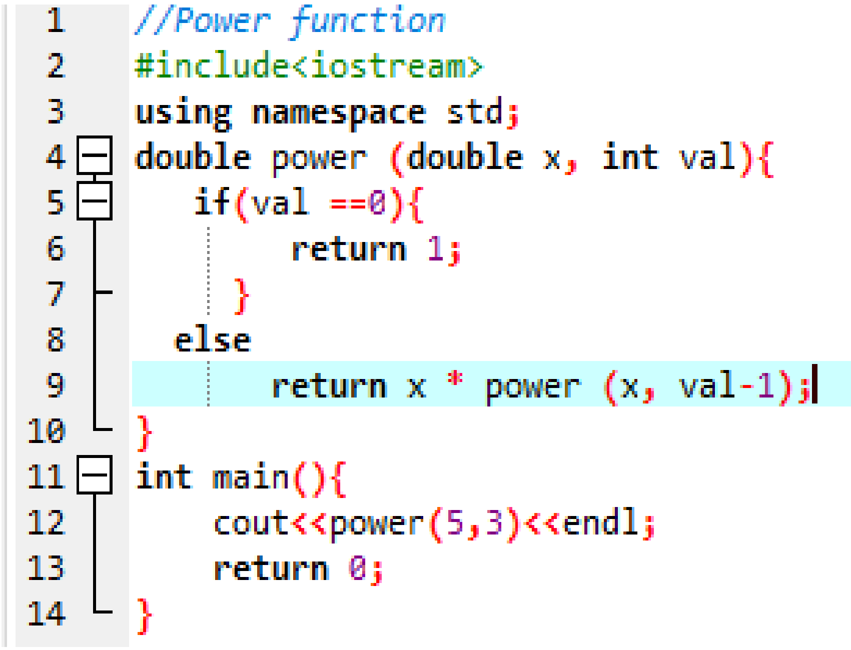
$n^p = n * n^{p-1}$

$n^0 = 1$



```
using namespace std;

int power(int n, int p){

    if(p==0){
        return 1;
    }

    int prevPower = power(n, p-1);
    return n*prevPower;
}
```





```
//Power function
#include<iostream>
using namespace std;
double power (double x, int val){
   if(val ==0){
        return 1;
      }
   else
       return x * power (x, val-1);
}
int main(){
     cout<<power(5,3)<<endl;
     return 0;
}
```

Factorial Program Using Recursion In C++

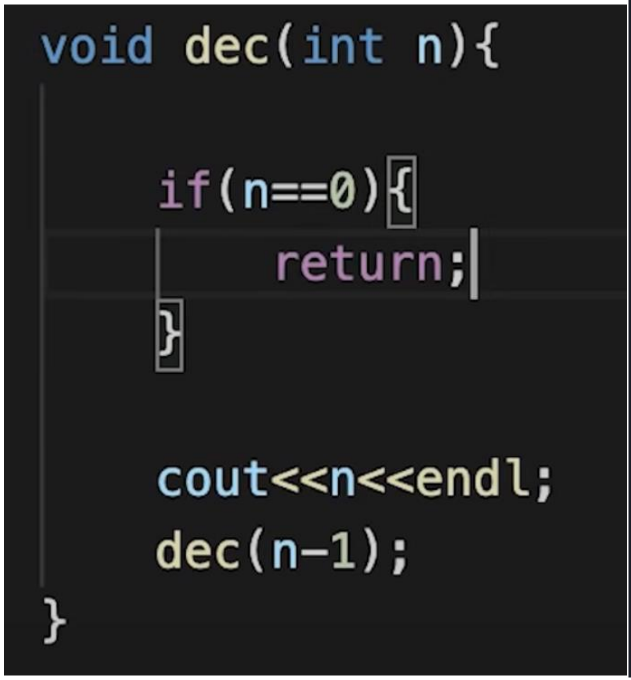
Factorial is the product of an integer and all other integers below it. For example, the factorial of 5 (5!) is equal to 5x4x3x2x1 i.e. 120

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

int fact(int n);

int main()
{
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;

    cout << "Factorial of " << n << " = " << fact(n);

    return 0;
}

int fact(int n)
{
    if(n > 1)
        return n * fact(n - 1);
    else
        return 1;
}
```

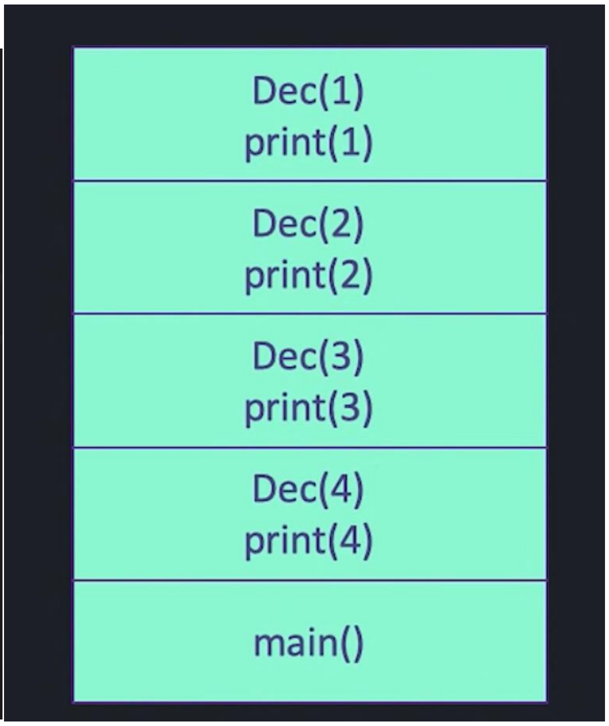
Reverse A Number Using Recursion

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

int reverseNumber(int n) {

   static temp,sum;

   if(n>0){

      temp = n%10;
      sum = sum*10 + temp;

      reverseNumber(n/10);

   } else {
        return sum;
    }

}

int main() {

  int n,reverse;

   cout<<"Enter number";
   cin >> n;

   reverse = reverseNumber(n);

   cout << "Reverse of number is" << reverse;

   return 0;
}
```

- Decreasing order



```
void dec(int n){

    if(n==0){
        return;
    }

    cout<<n<<endl;
    dec(n-1);
}
```



Direct Recursion:

A C function is *directly* recursive if it contains an explicit call to itself. For example, the function

```
int foo(int x) {
   if (x <= 0) return x;

   return foo(x - 1);
}
```

includes a call to itself, so it's directly recursive. The recursive call will occur for positive values of `x`.

Indirect Recursion:

A C function `foo` is *indirectly* recursive if it contains a call to another function which ultimately calls `foo`.

The following pair of functions is indirectly recursive. Since they call each other, they are also known as *mutually recursive* functions.

```
int foo(int x) {

   if (x <= 0) return x;

   return bar(x);
}

int bar(int y) {
   return foo(y - 1);
}
```

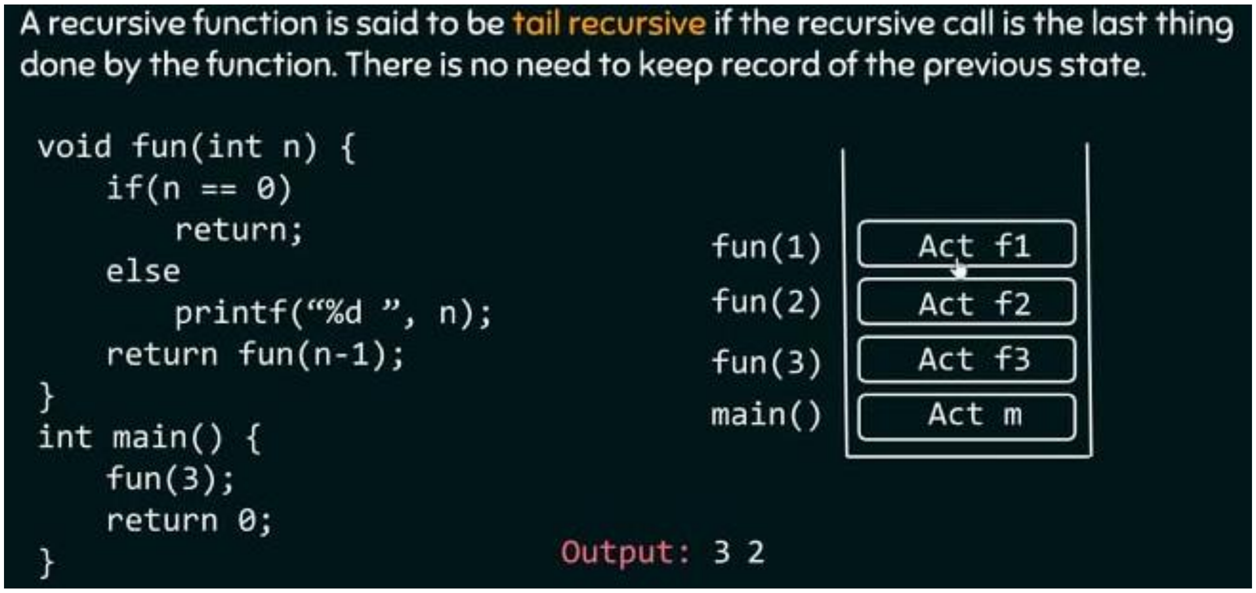
Tail Recursion:

A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call.

Tail recursive functions are often said to "return the value of the last recursive call as the value of the function." Tail recursion is very desirable because the amount of information which must be stored during the computation is independent of the number of recursive calls. Some modern computing systems will actually compute tail-recursive functions using an iterative process.

The "infamous" factorial function `fact` is usually written in a non-tail-recursive manner:

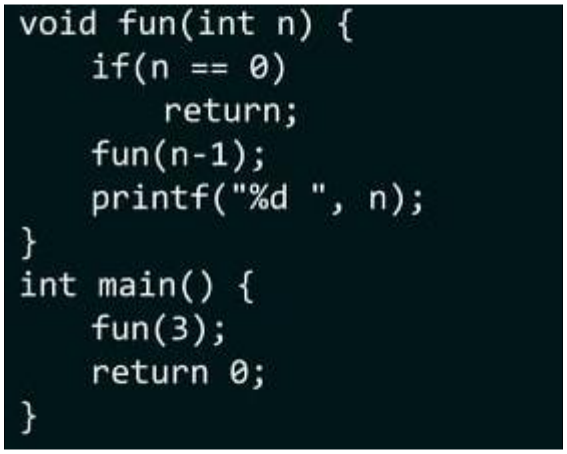
```
int fact (int n) { /* n >= 0 */

   if (n == 0)  return 1;

   return n * fact(n - 1);
```



A recursive function is said to be tail recursive if the recursive call is the last thing done by the function. There is no need to keep record of the previous state.

```c
void fun(int n) {
    if(n == 0)
        return;
    else
        printf("%d ", n);
    return fun(n-1);
}
int main() {
    fun(3);
    return 0;
}
```

```
Output: 3 2
```



```c
void fun(int n) {
    if(n == 0)
        return;
    fun(n-1);
    printf("%d ", n);
}
int main() {
    fun(3);
    return 0;
}
```