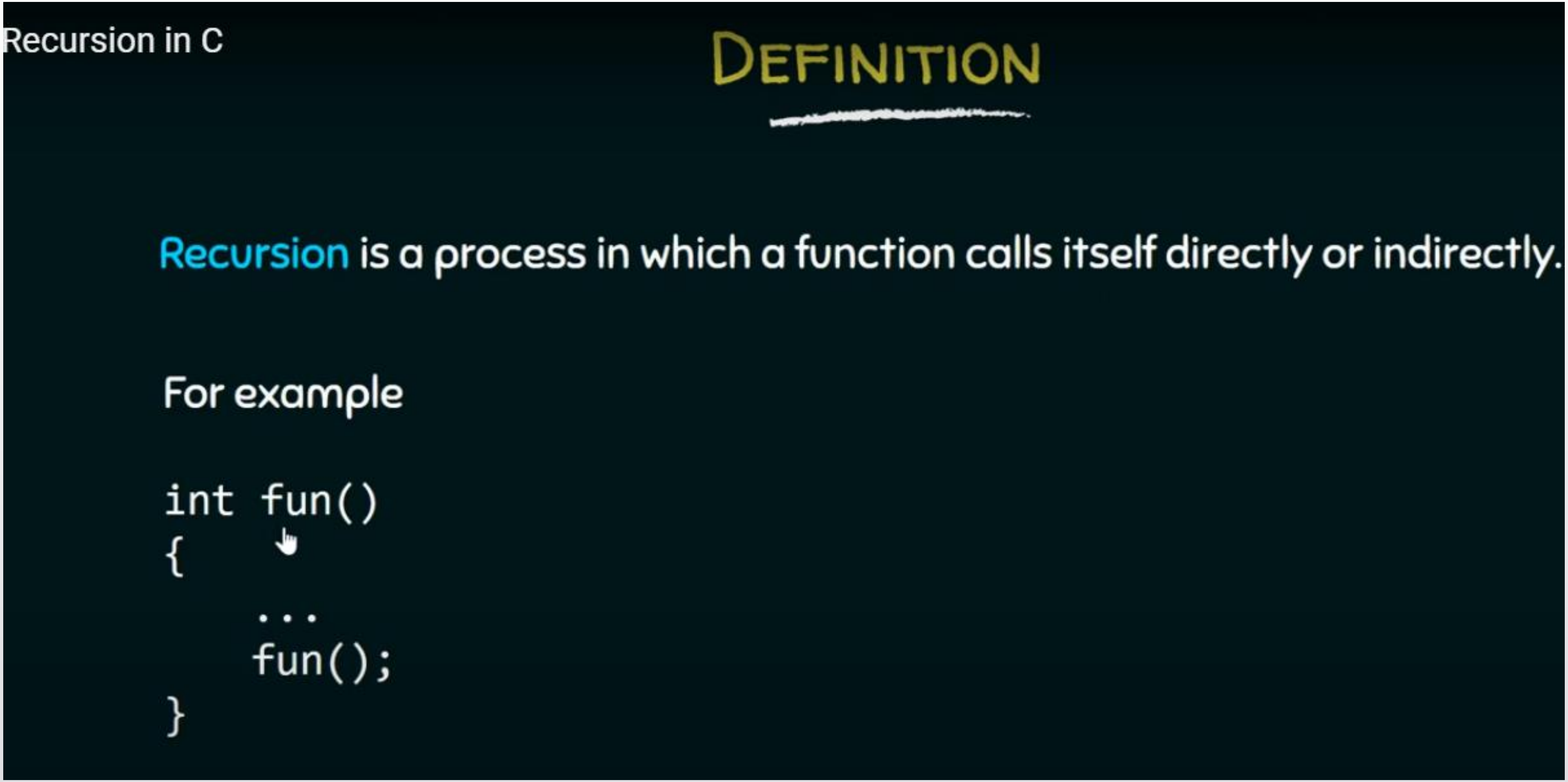
# RECURSION



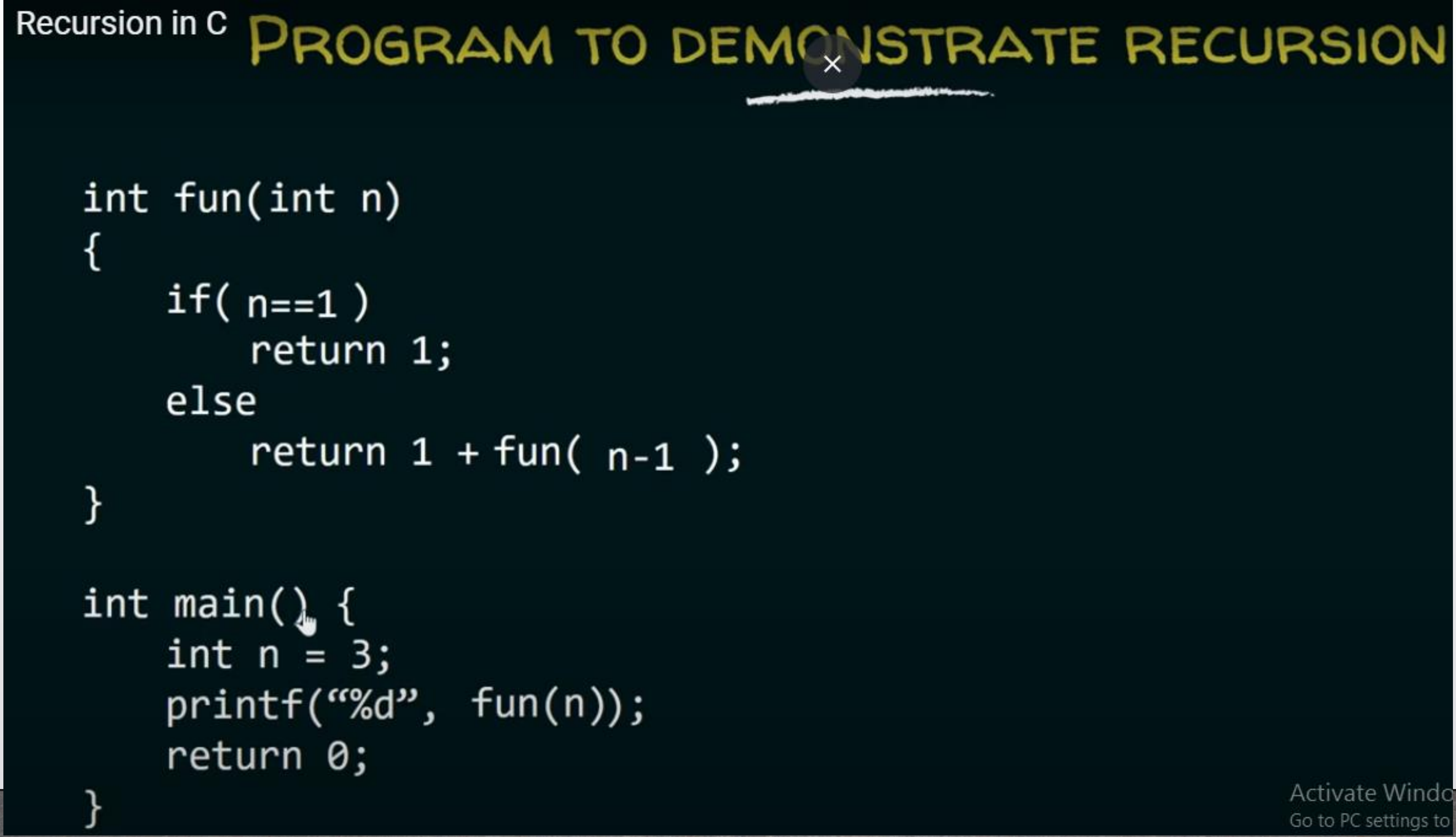
Recursion in C

# DEFINITION

Recursion is a process in which a function calls itself directly or indirectly.

For example

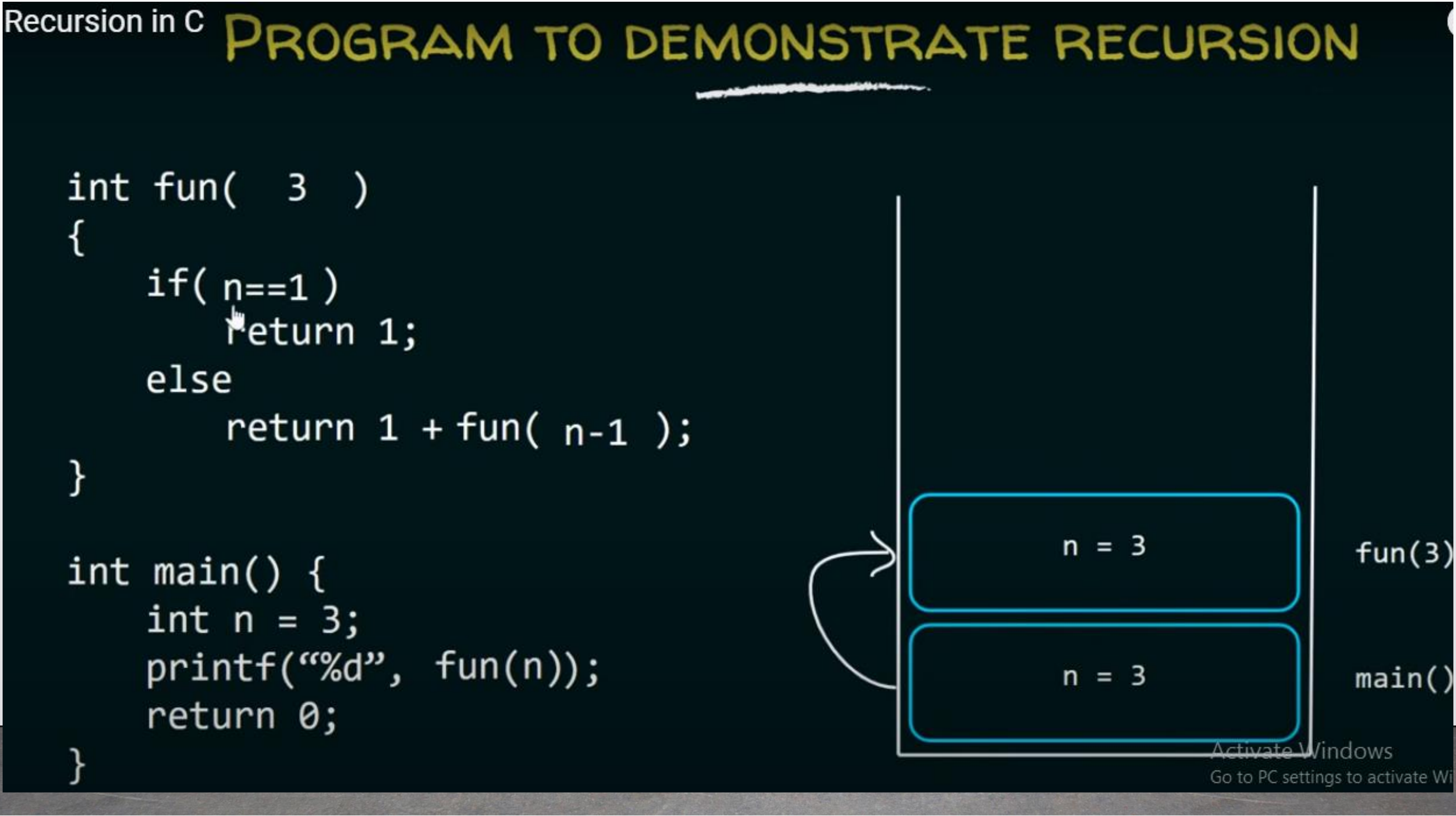
```
int fun()
{
    ...
    fun();
}
```



Recursion in C

# PROGRAM TO DEMONSTRATE RECURSION

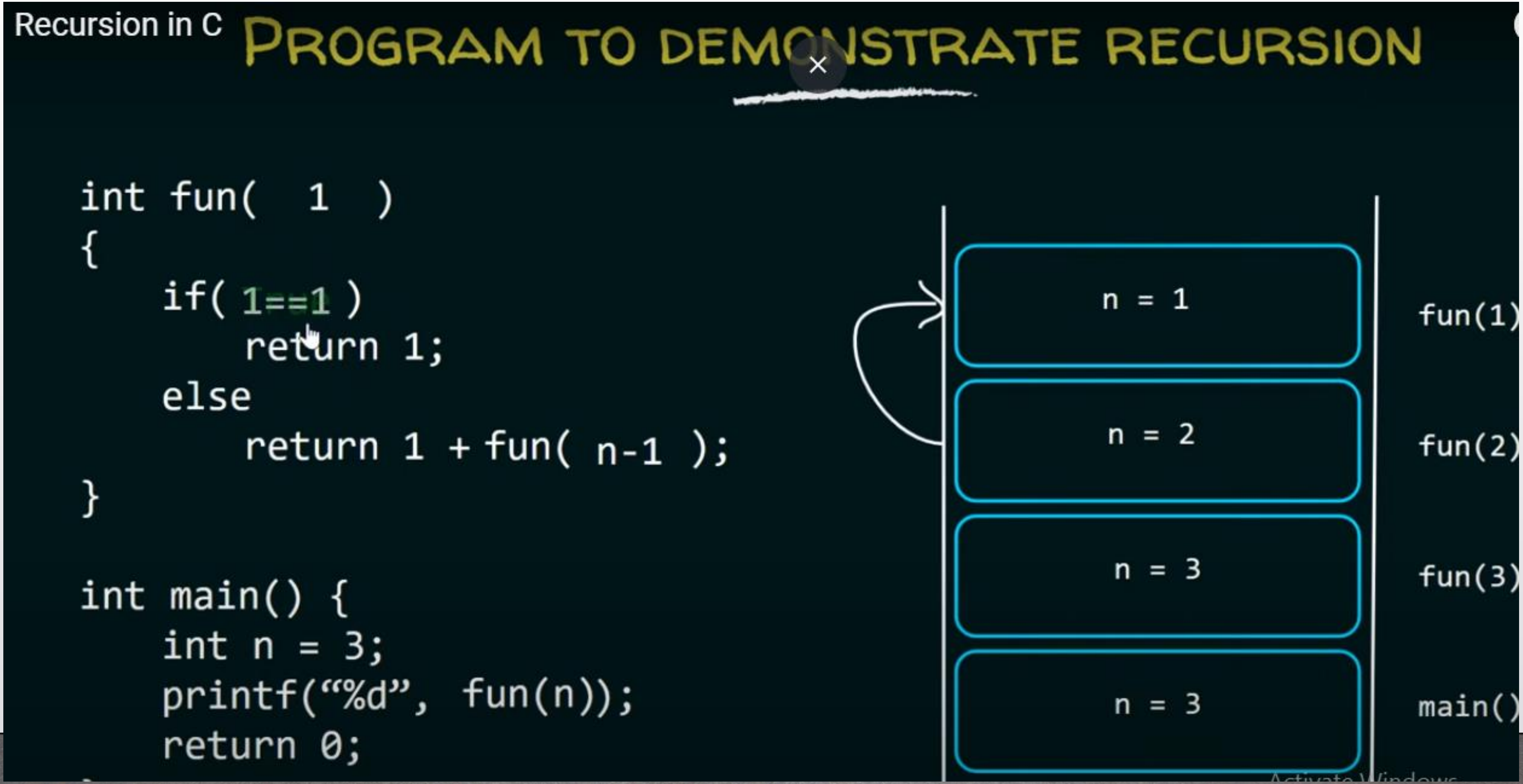
```
int fun(int n)
{
    if( n==1 )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf(“%d”, fun(n));
    return 0;
}
```



Recursion in C

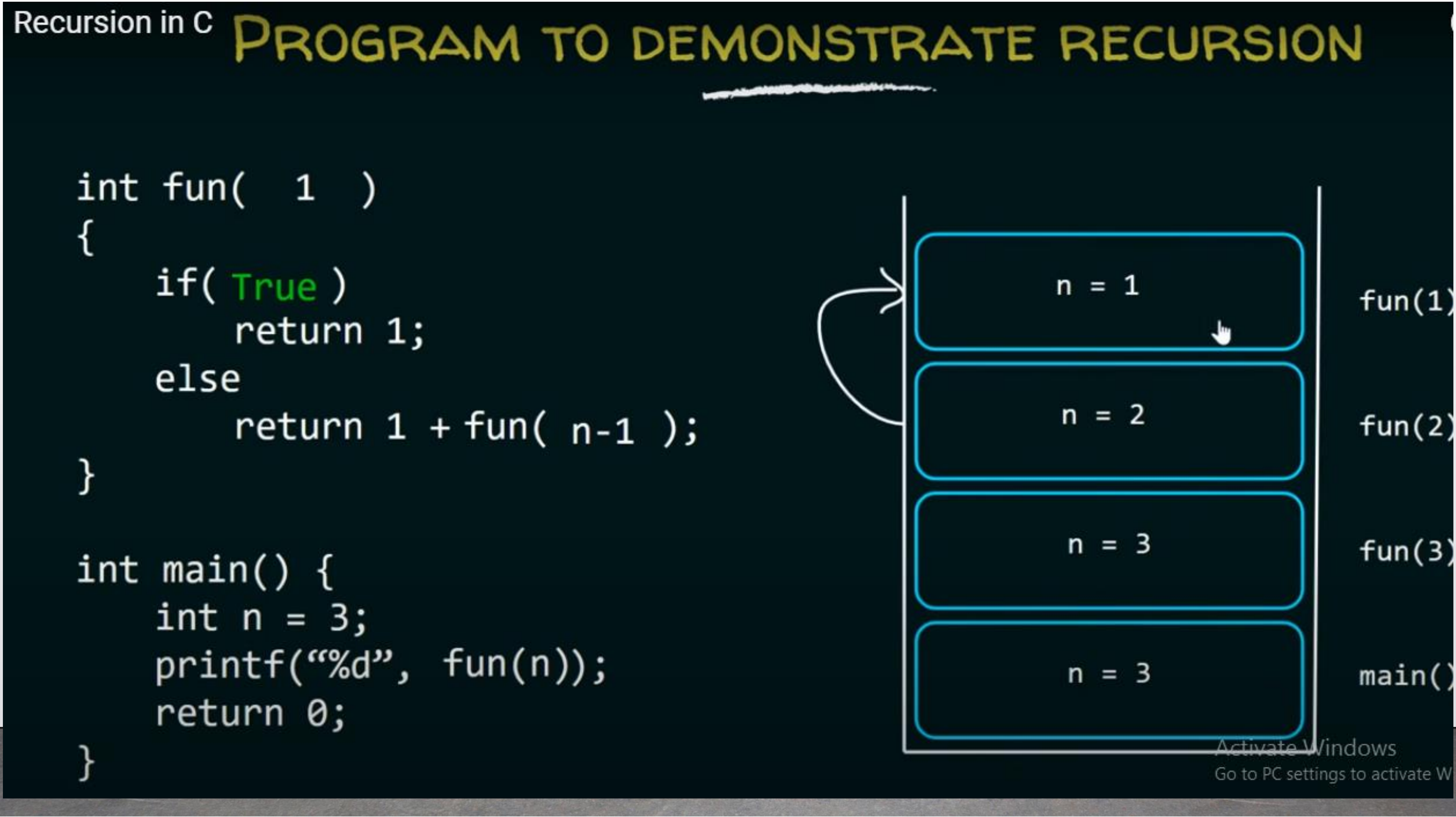
# PROGRAM TO DEMONSTRATE RECURSION

```
int fun(  3  )
{
    if( n==1 )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf(“%d”, fun(n));
    return 0;
}
```


Recursion in C

# PROGRAM TO DEMONSTRATE RECURSION

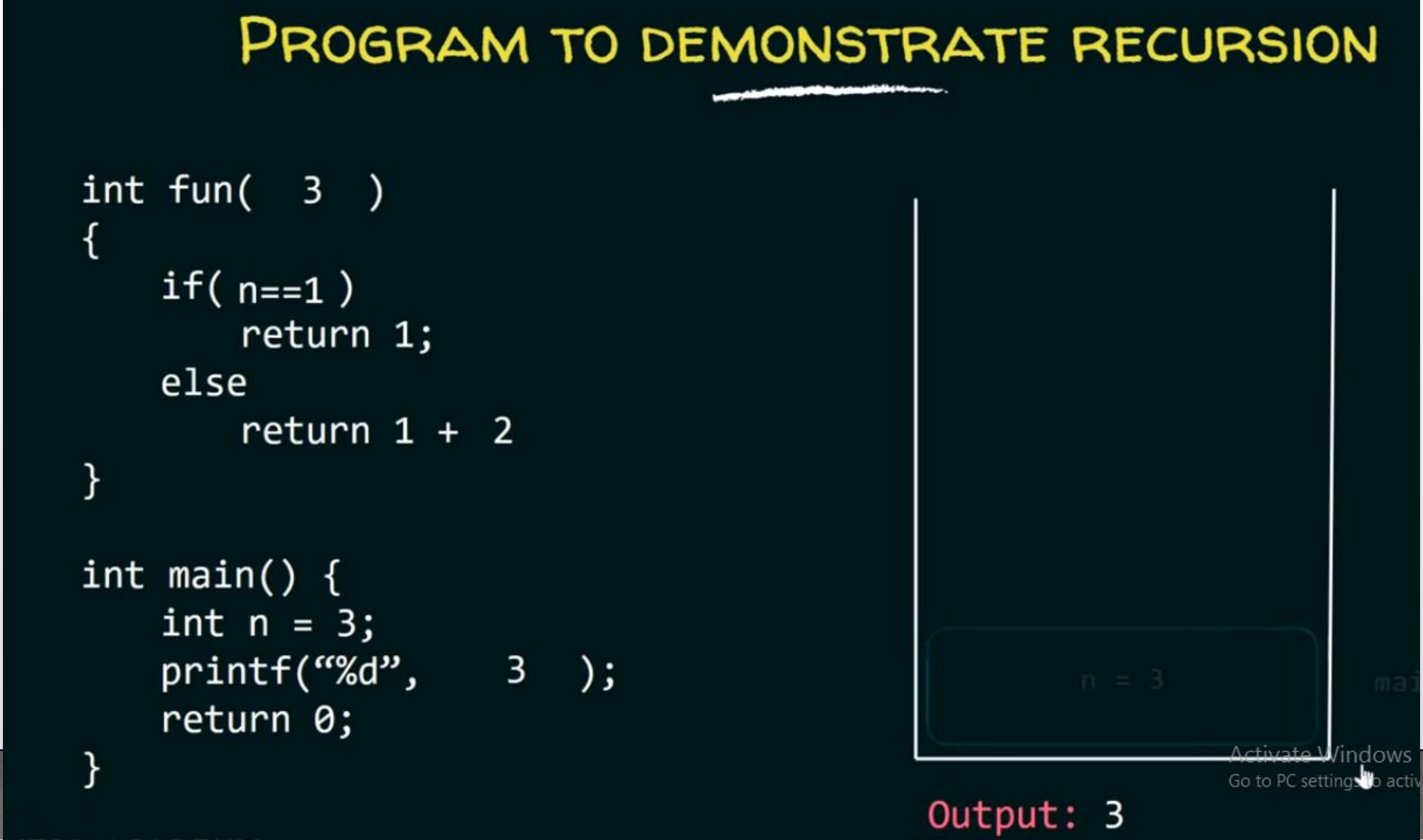
```c
int fun(  1  )
{
    if( 1==1 )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
```



Recursion in C

# PROGRAM TO DEMONSTRATE RECURSION

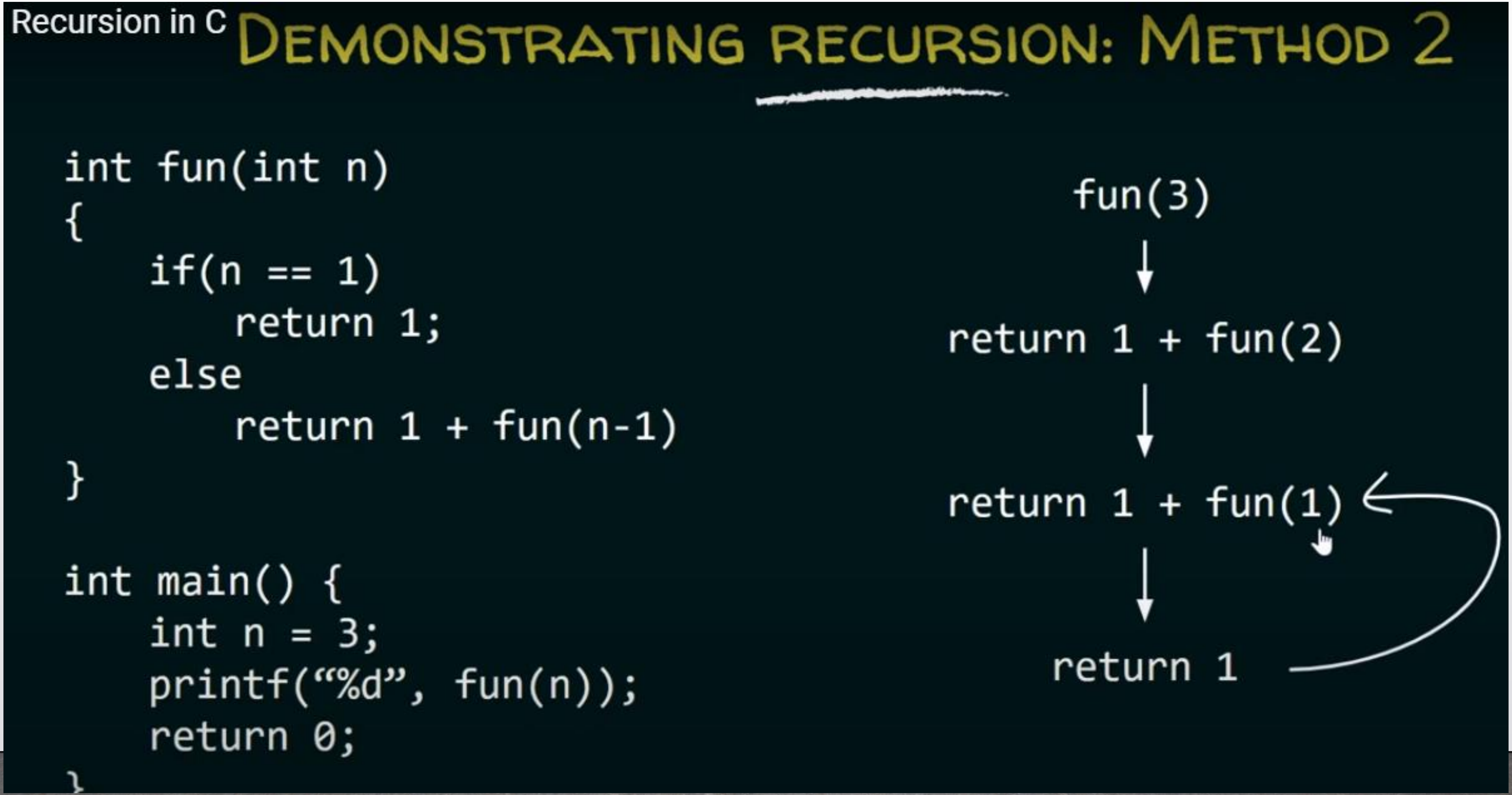
```
int fun(  1  )
{
    if( True )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```



# PROGRAM TO DEMONSTRATE RECURSION

```
int fun(  3  )
{
    if( n==1 )
        return 1;
    else
        return 1 +  2
}

int main() {
    int n = 3;
    printf(“%d”,    3  );
    return 0;
}
```

Output: 3



Recursion in C

# Demonstrating recursion: Method 2

```
int fun(int n)
{
    if(n == 1)
        return 1;
    else
        return 1 + fun(n-1)
}

int main() {
    int n = 3;
    printf(“%d”, fun(n));
    return 0;
}
```

fun(3)

↓

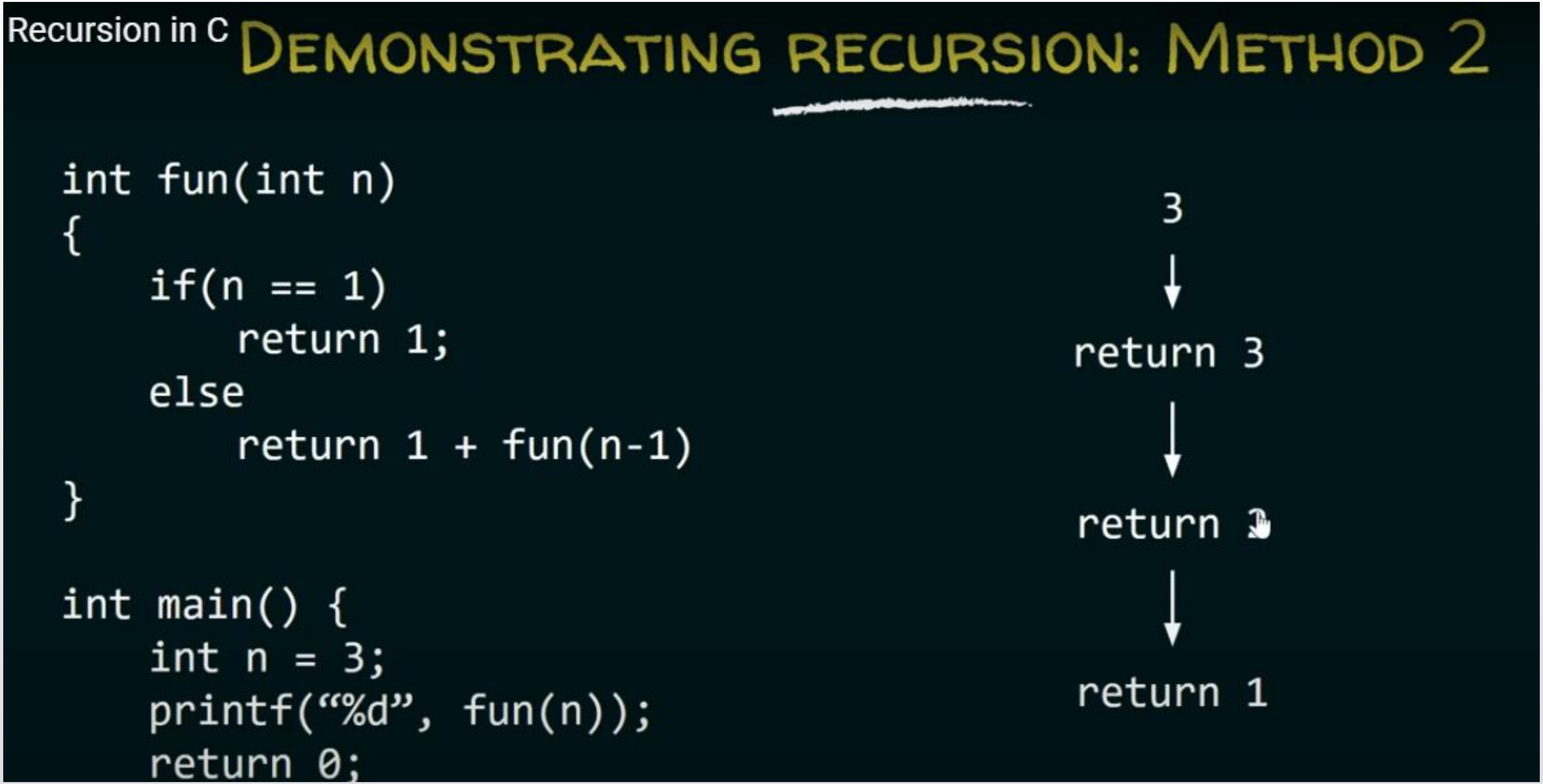
return 1 + fun(2)

↓

return 1 + fun(1)

↓

return 1



Recursion in C

# Demonstrating recursion: Method 2

```c
int fun(int n)
{
    if(n == 1)
        return 1;
    else
        return 1 + fun(n-1)
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
```

3

↓

return 3

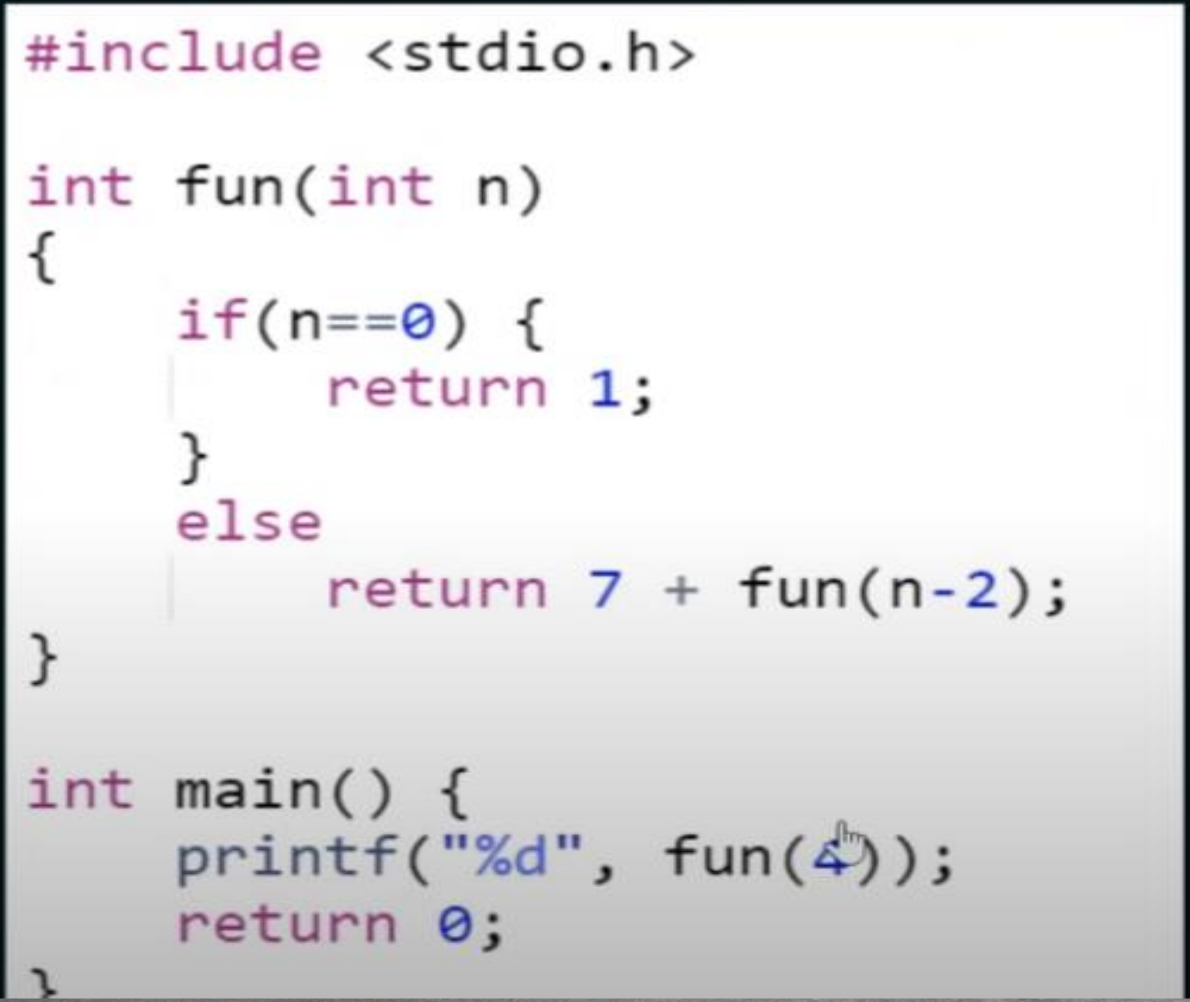
↓

return 2

↓

return 1

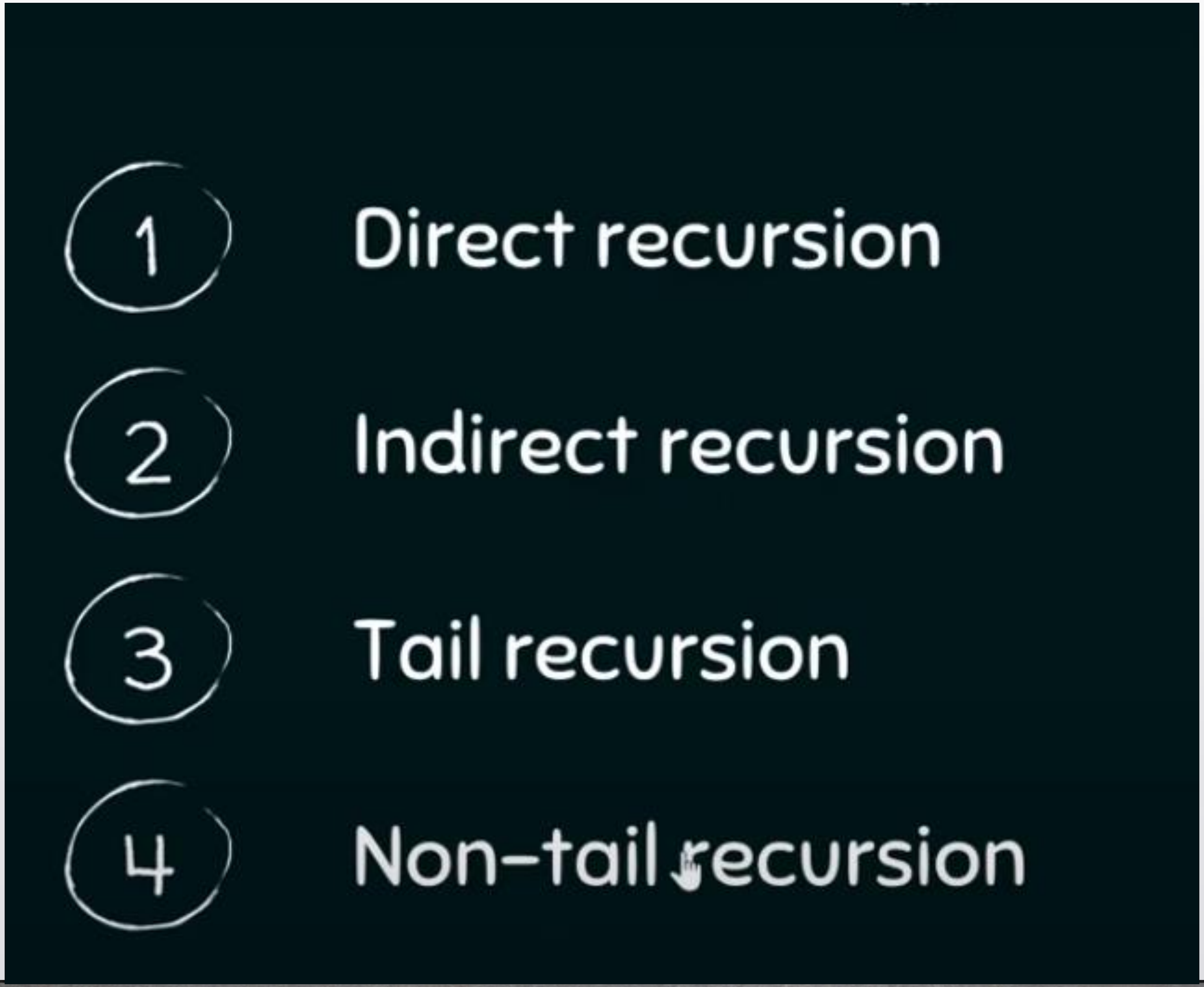
## What is the output of the following C program:



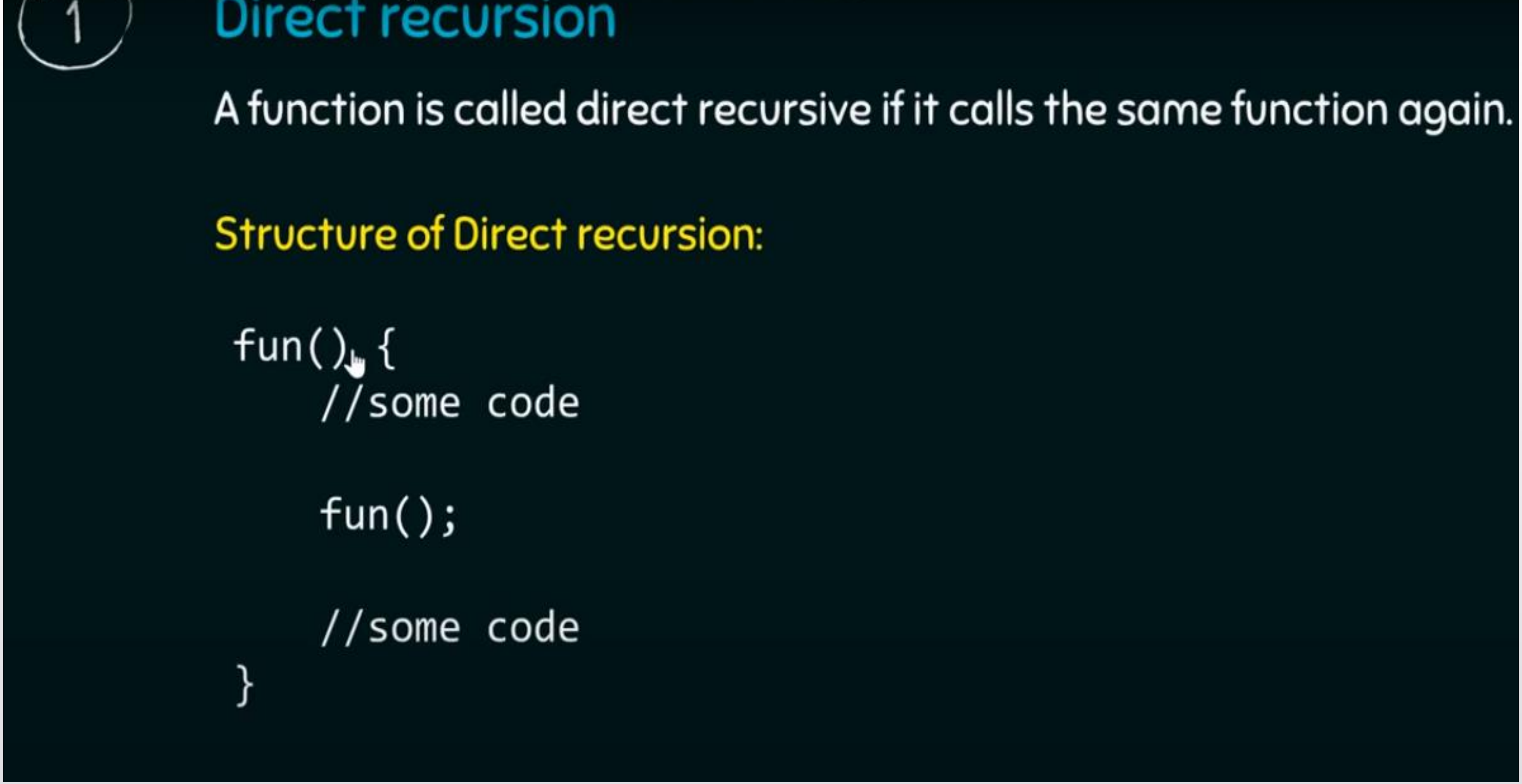
```
#include <stdio.h>

int fun(int n)
{
    if(n==0) {
        return 1;
    }
    else
        return 7 + fun(n-2);
}

int main() {
    printf("%d", fun(4));
    return 0;
}
```

a) 4
b) 7
c) 15
d) 12

# TYPES OF RECURSION



1. Direct recursion
2. Indirect recursion
3. Tail recursion
4. Non-tail recursion

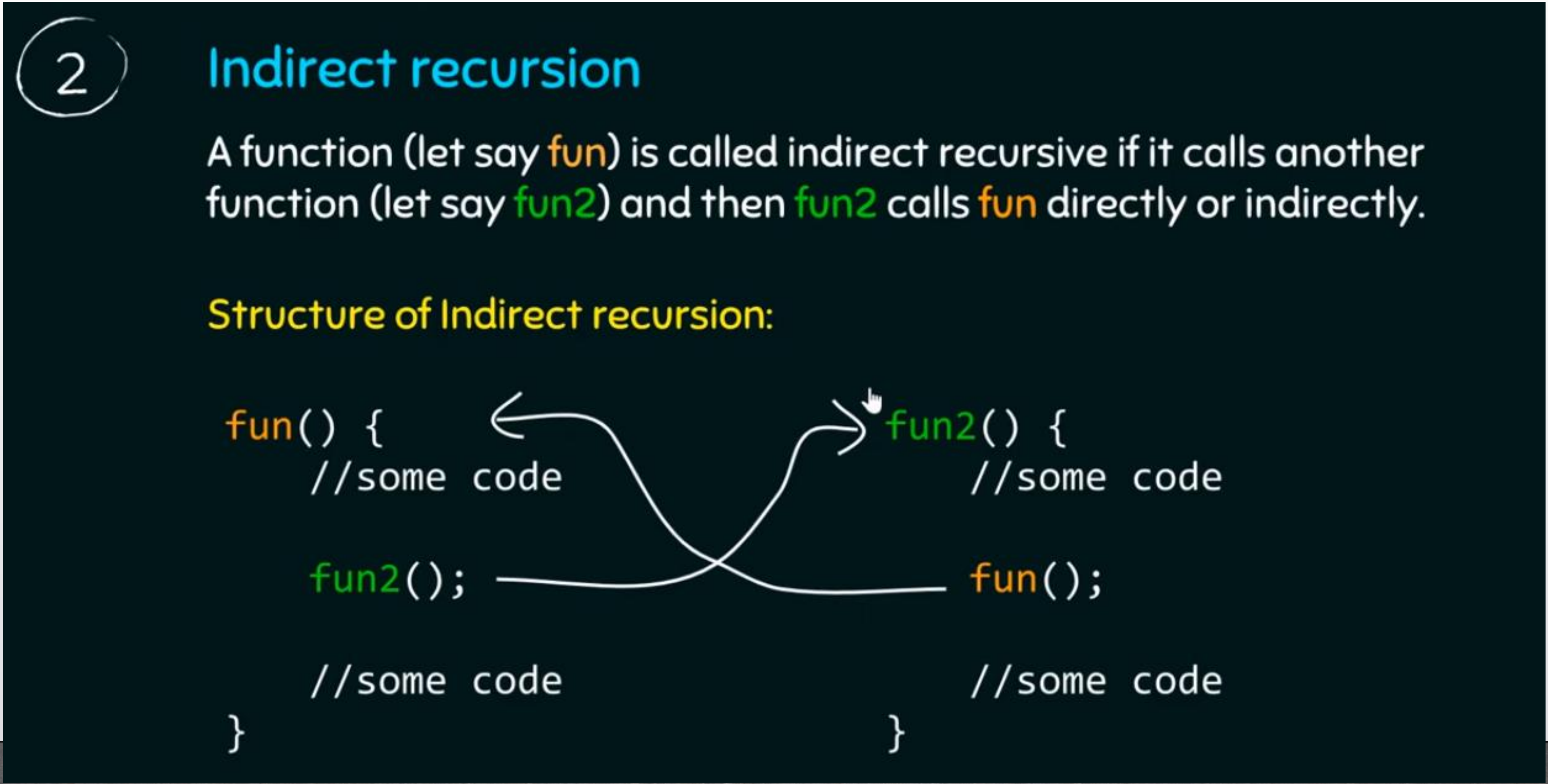


## 1 Direct recursion

A function is called direct recursive if it calls the same function again.

### Structure of Direct recursion:

```
fun() {
    //some code

    fun();

    //some code
}
```



2

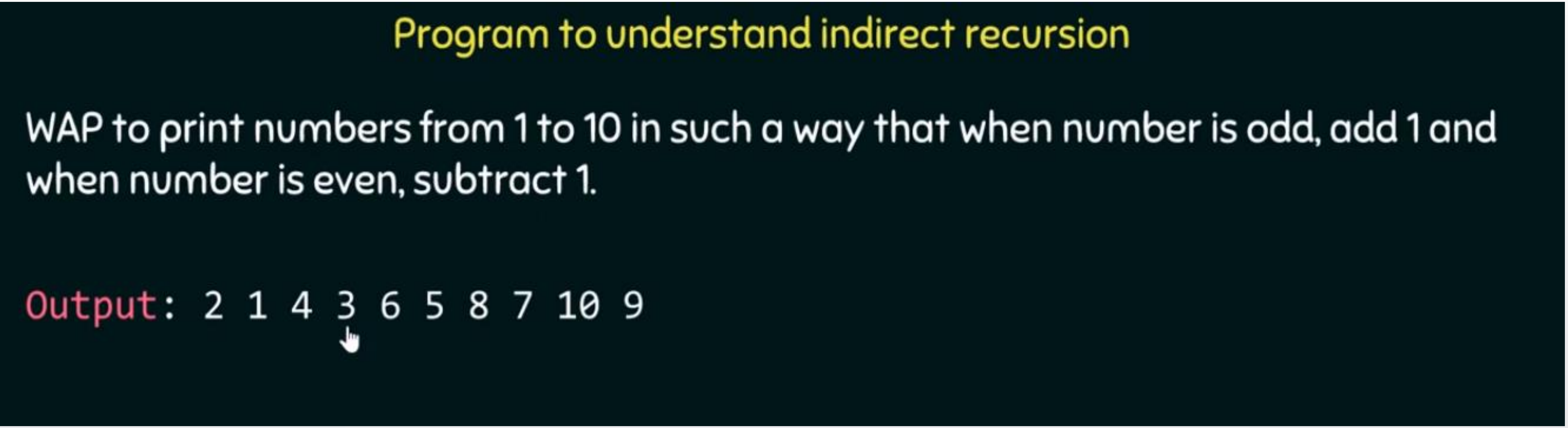
## Indirect recursion

A function (let say fun) is called indirect recursive if it calls another function (let say fun2) and then fun2 calls fun directly or indirectly.

### Structure of Indirect recursion:

```
fun() {
    //some code

    fun2();

    //some code
}
```

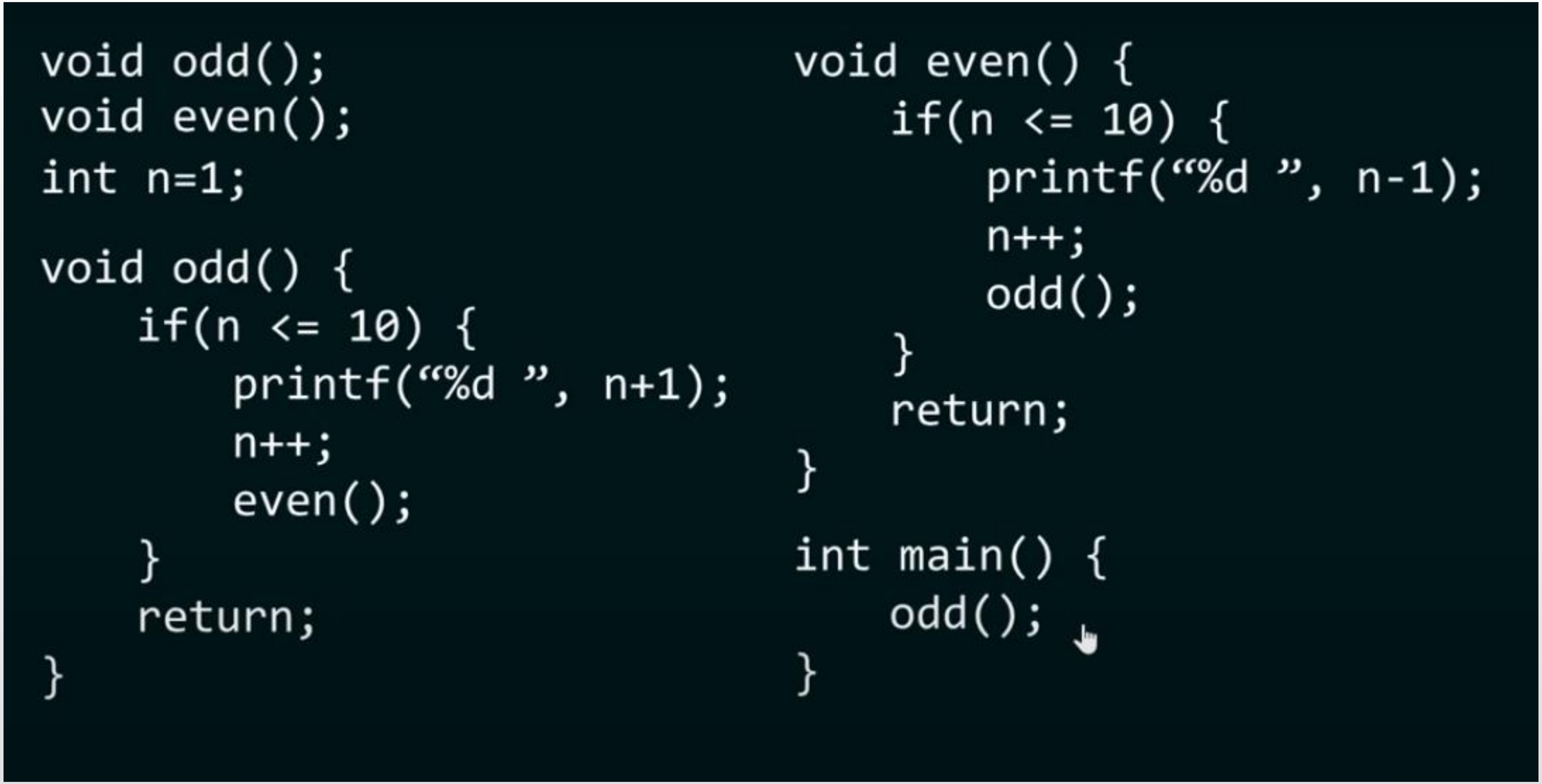
```
fun2() {
    //some code

    fun();

    //some code
}
```



## Program to understand indirect recursion

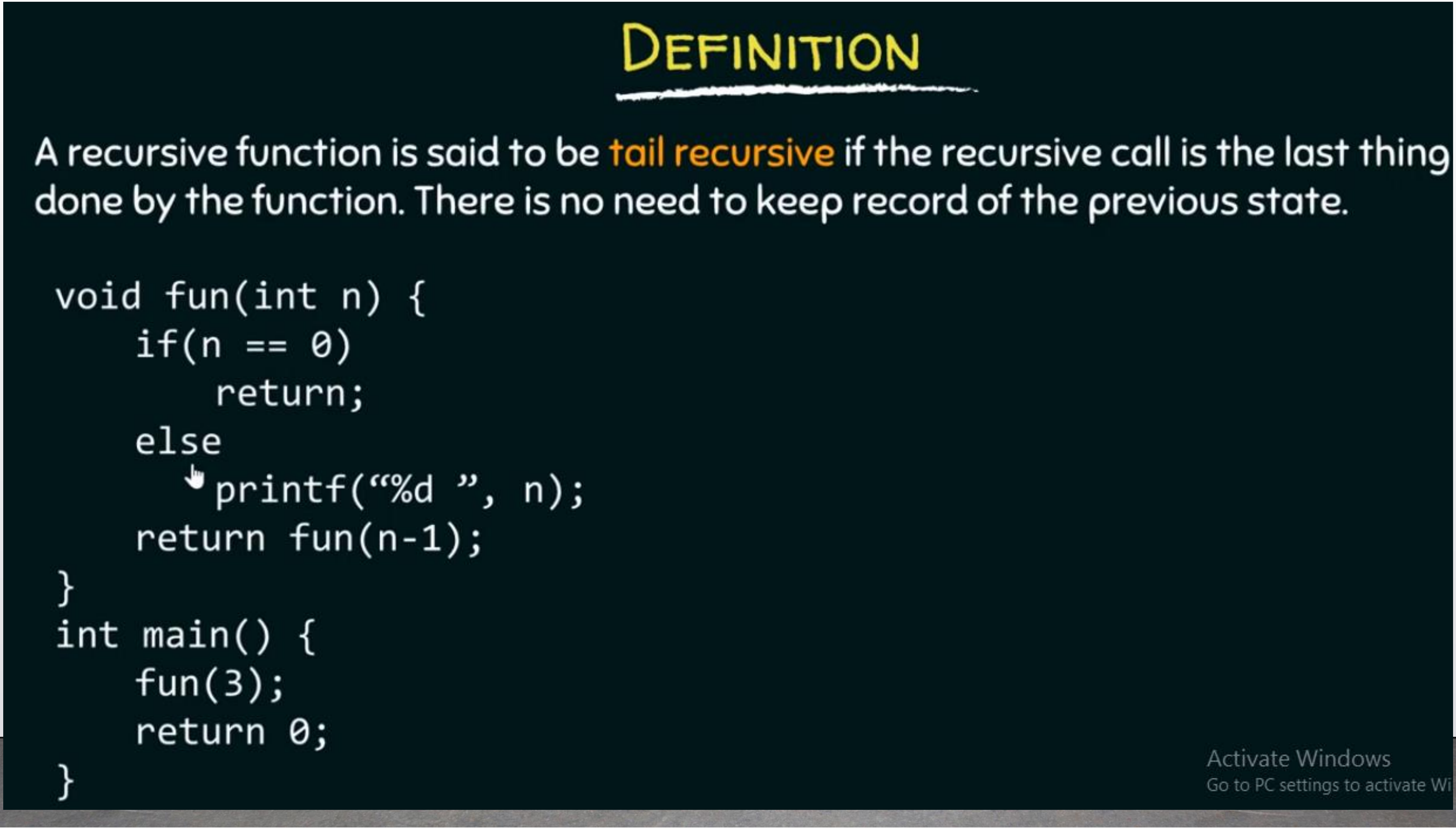
WAP to print numbers from 1 to 10 in such a way that when number is odd, add 1 and when number is even, subtract 1.

```
Output: 2 1 4 3 6 5 8 7 10 9
```



```
void odd();
void even();
int n=1;

void odd() {
    if(n <= 10) {
        printf(“%d ”, n+1);
        n++;
        even();
    }
    return;
}
```

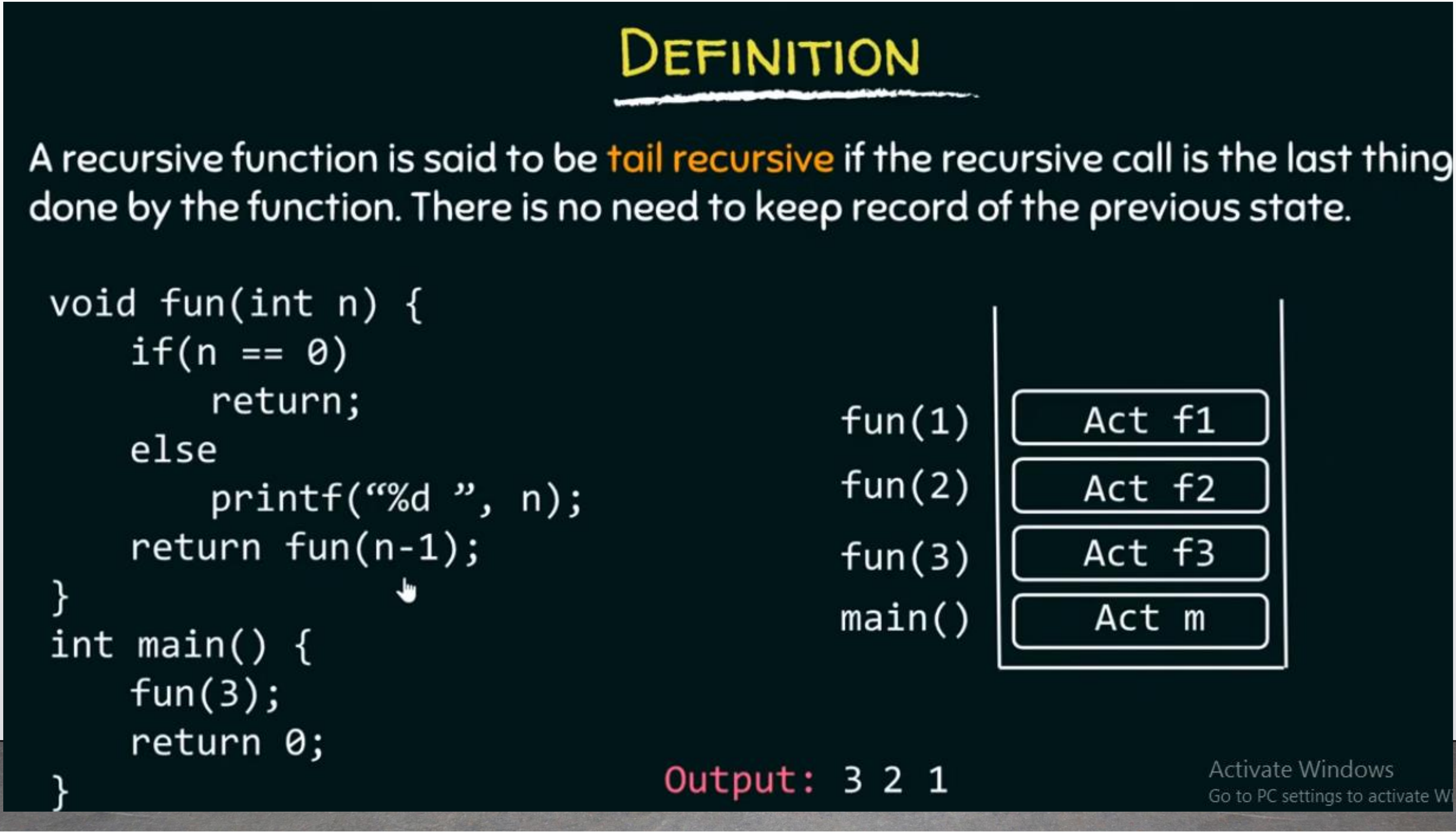
```
void even() {
    if(n <= 10) {
        printf(“%d ”, n-1);
        n++;
        odd();
    }
    return;
}

int main() {
    odd();
}
```



# DEFINITION

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.

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

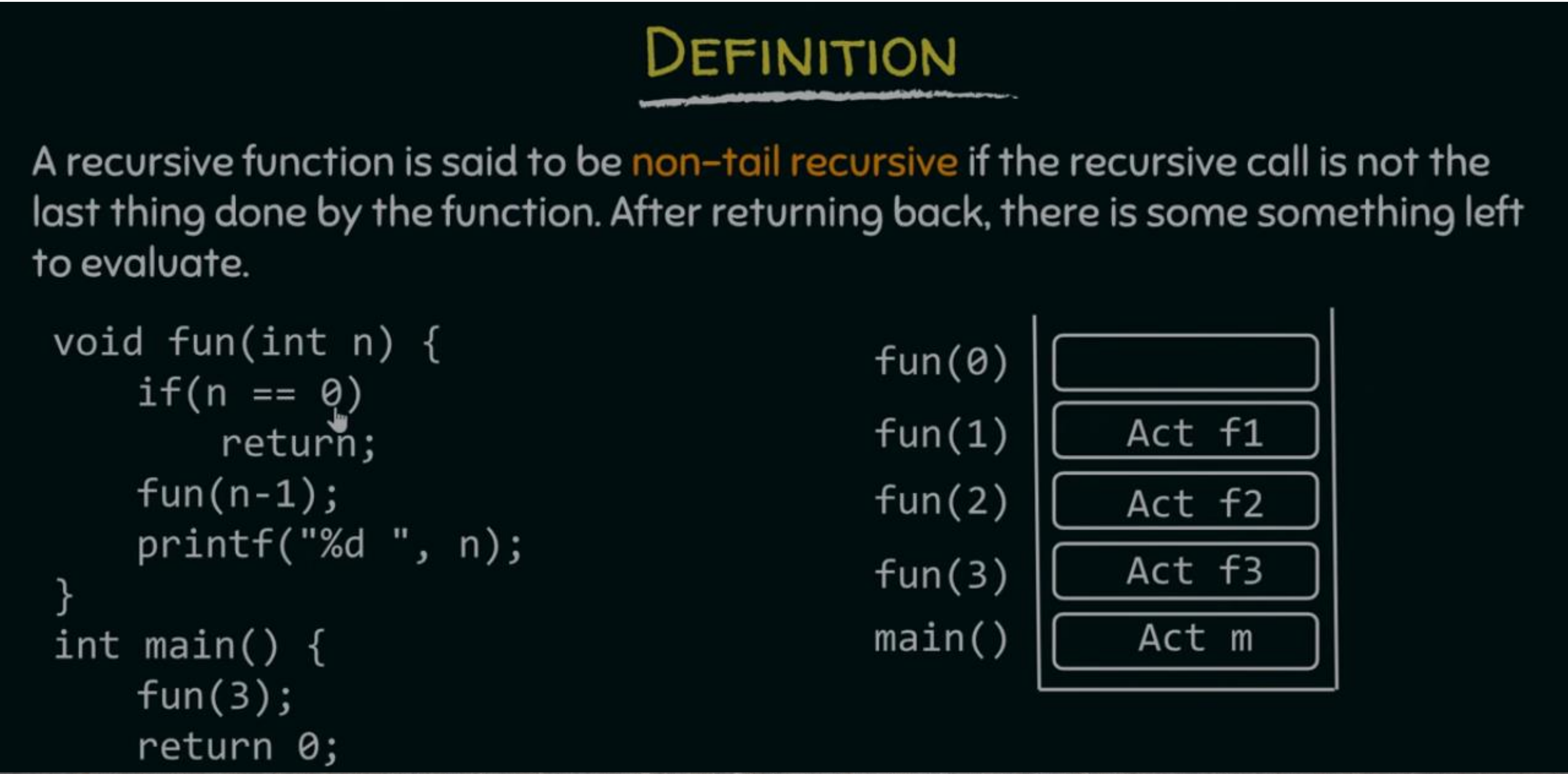


# Definition

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.

```
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 1
```



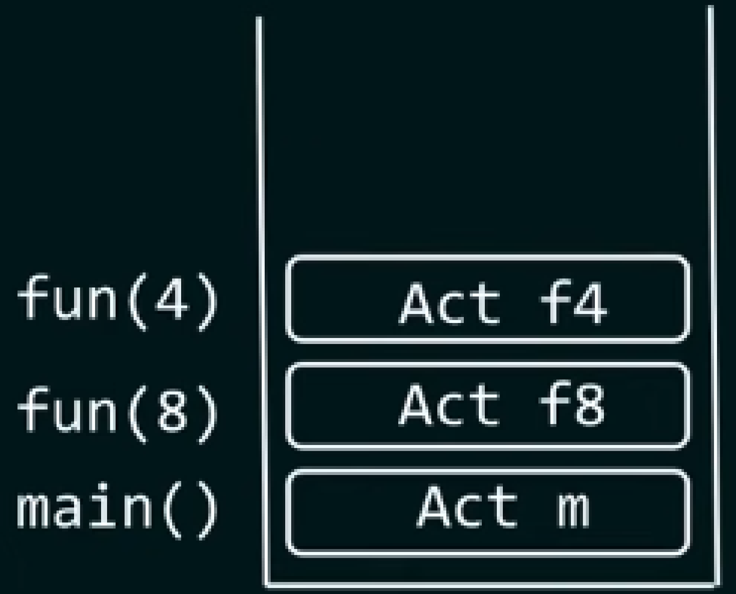
# DEFINITION

A recursive function is said to be non-tail recursive if the recursive call is not the last thing done by the function. After returning back, there is some something left to evaluate.

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

# One more example (non-tail)

```c
int fun(int n) {
    if(n == 1)
        return 0;
    else
        return 1 + fun(n/2);
}
int main() {
    printf("%d", fun(8));
    return 0;
}
```



fun(4) Act f4

fun(8) Act f8

main() Act m

# TAIL VS NON TAIL

The tail recursive functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler.

Compilers usually execute recursive procedures by using a stack. This stack consists of all the pertinent information, including the parameter values, for each recursive call. When a procedure is called, its information is pushed onto a stack, and when the function terminates the information is popped out of the stack. Thus for the non-tail-recursive functions, the stack depth (maximum amount of stack space used at any time during compilation) is more.

The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use (See this for more details).

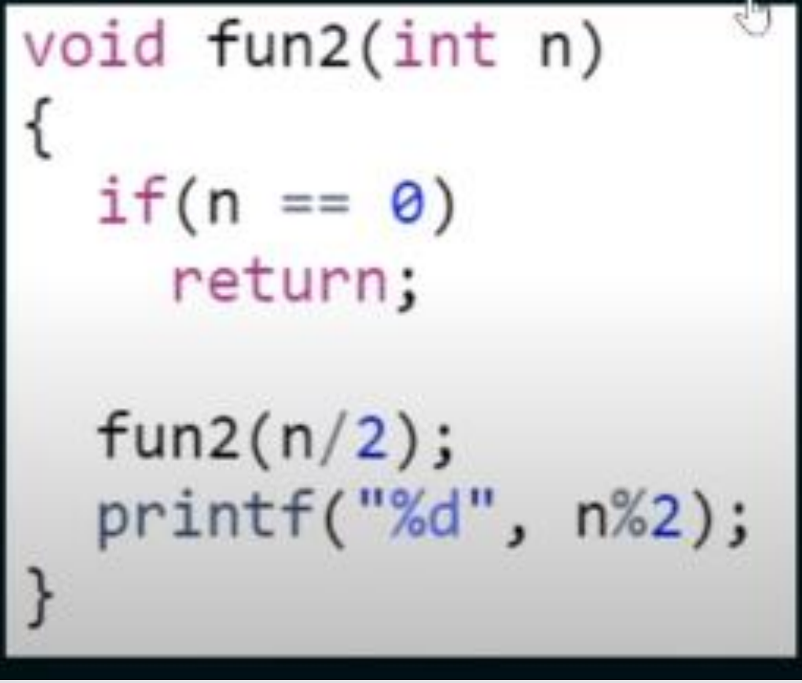
```
// A NON-tail-recursive function. The function is not
tail
// recursive because the value returned by fact(n-1)
is used
// in fact(n) and call to fact(n-1) is not the last thing
// done by fact(n)
 int fact(unsigned int n)
{
	if (n <= 0)
		return 1;

	return n * fact(n - 1);
}

// Driver program to test above function
int main()
{
	printf("%d", fact(5));
	return 0;
}
```

The above function can be written as a tail-recursiv function. The idea is to use one more argument and accumulate the factorial value in the second argument. When n reaches 0, return the accumulated value.

```
// A tail recursive function to calculate factorial
 factTR(unsigned int n, unsigned int a)
{
	if (n <= 1)
		return a;

	return factTR(n - 1, n * a);
}

// A wrapper over factTR
int fact(unsigned int n) { return factTR(n, 1); }

// Driver program to test above function
int main()
{
	cout << fact(5);
	return 0;
}
```

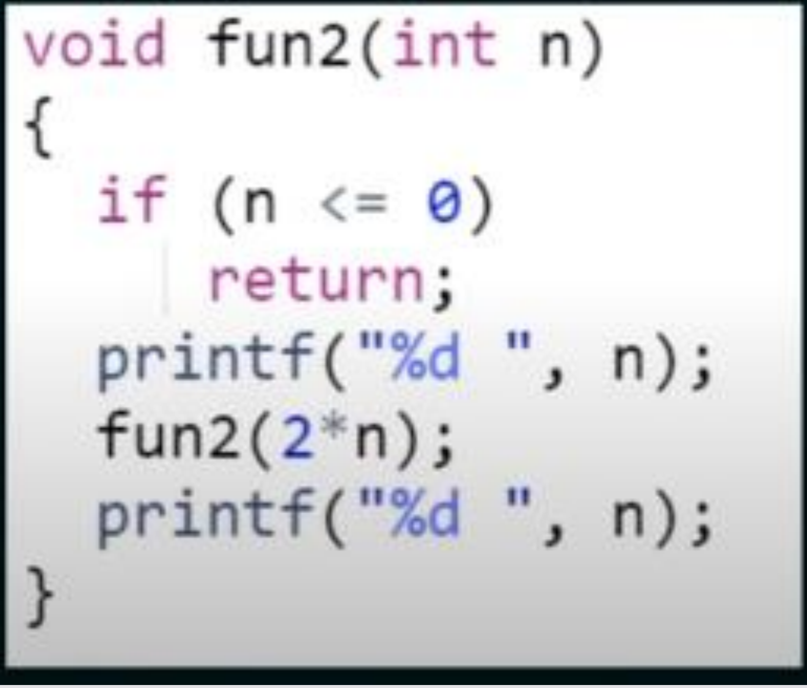
Identify whether the following programs are tail recursive or non tail recursive.

Program 1:



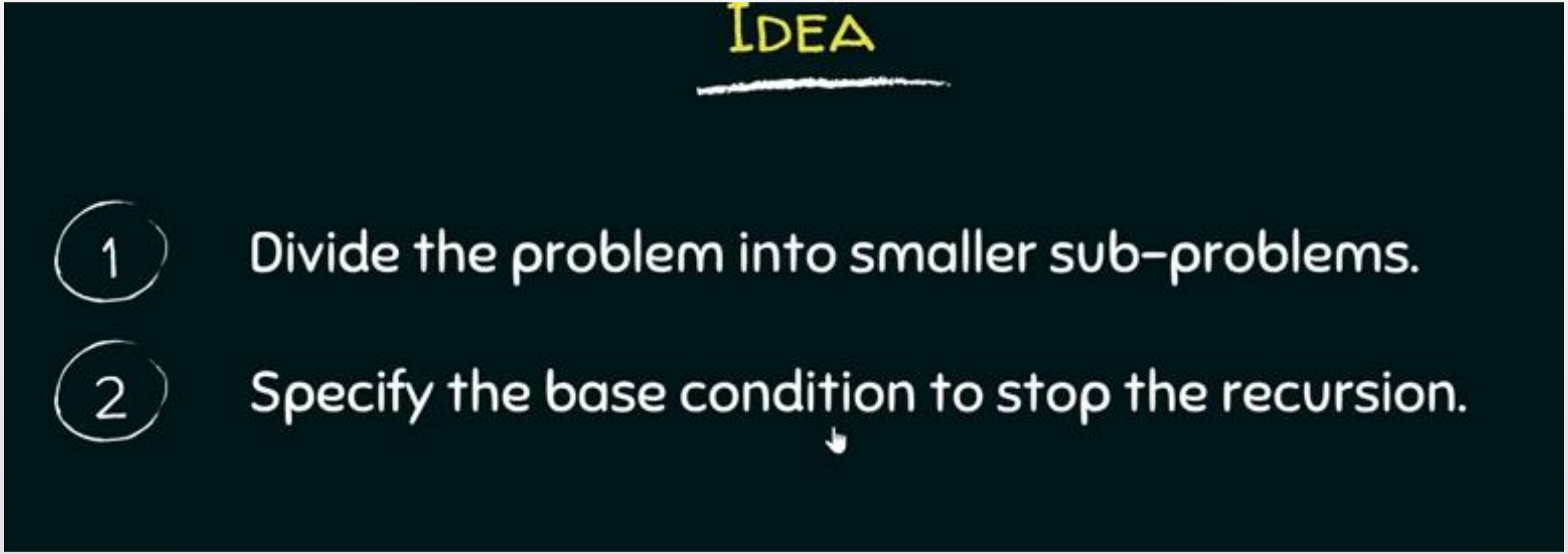
```
void fun2(int n)
{
  if(n == 0)
    return;

  fun2(n/2);
  printf("%d", n%2);
}
```

Program 2:



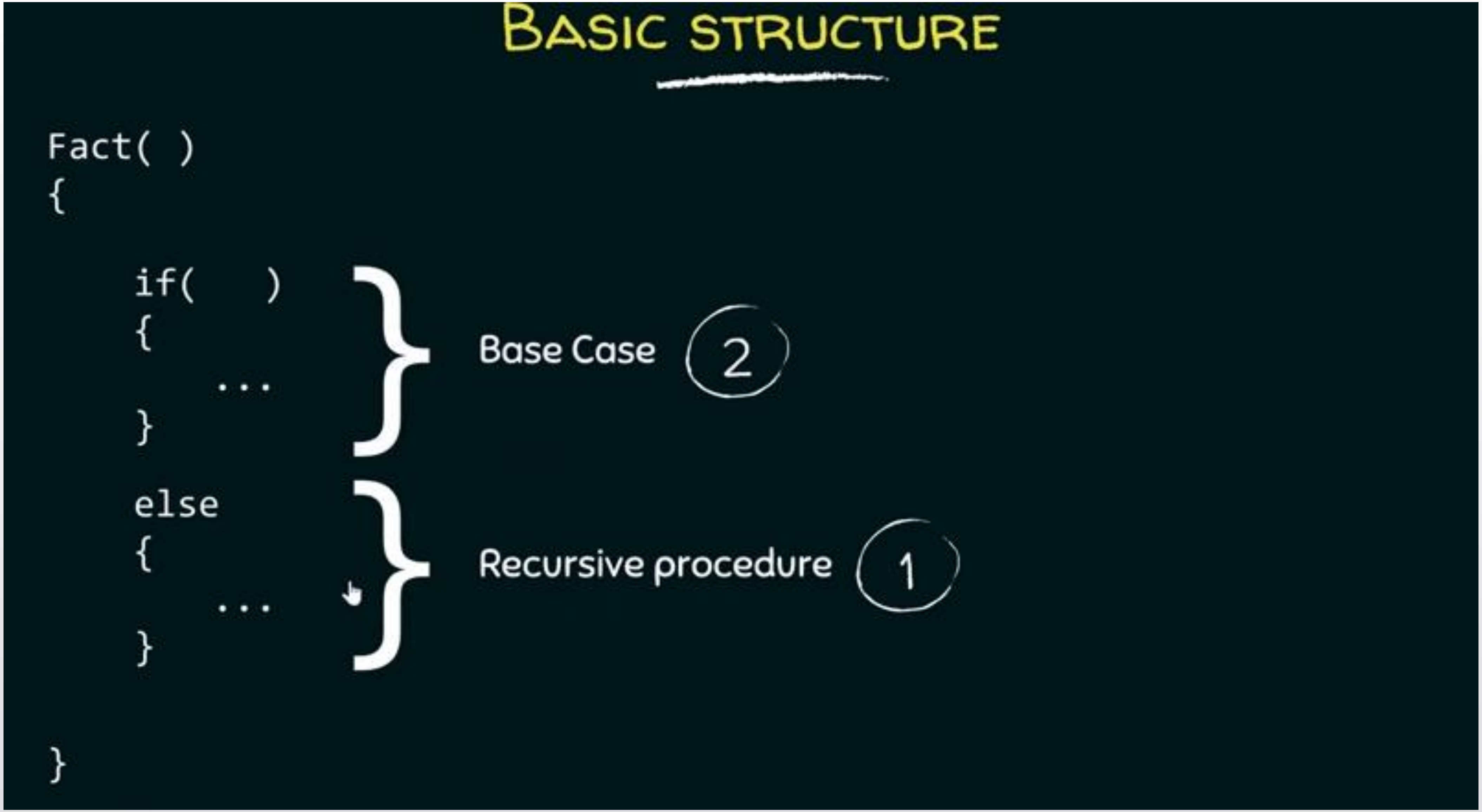
```
void fun2(int n)
{
  if (n <= 0)
     return;
  printf("%d ", n);
  fun2(2*n);
  printf("%d ", n);
}
```

# How to write a recursive function



IDEA

1. Divide the problem into smaller sub-problems.
2. Specify the base condition to stop the recursion.

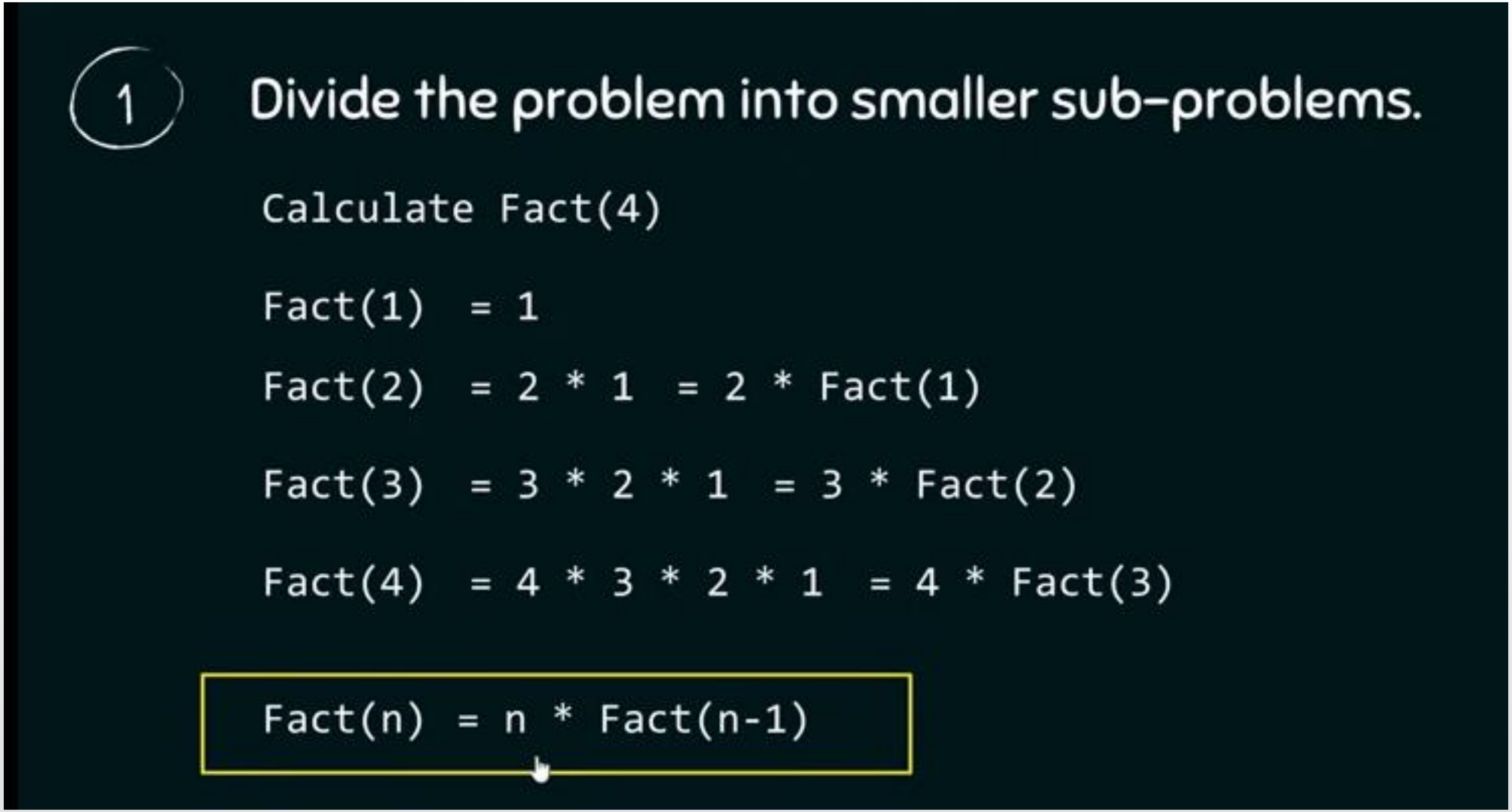


# BASIC STRUCTURE

```
Fact( )
{

    if(    )
    {
        ...
    }

    else
    {
        ...
    }


}
```

Base Case 2

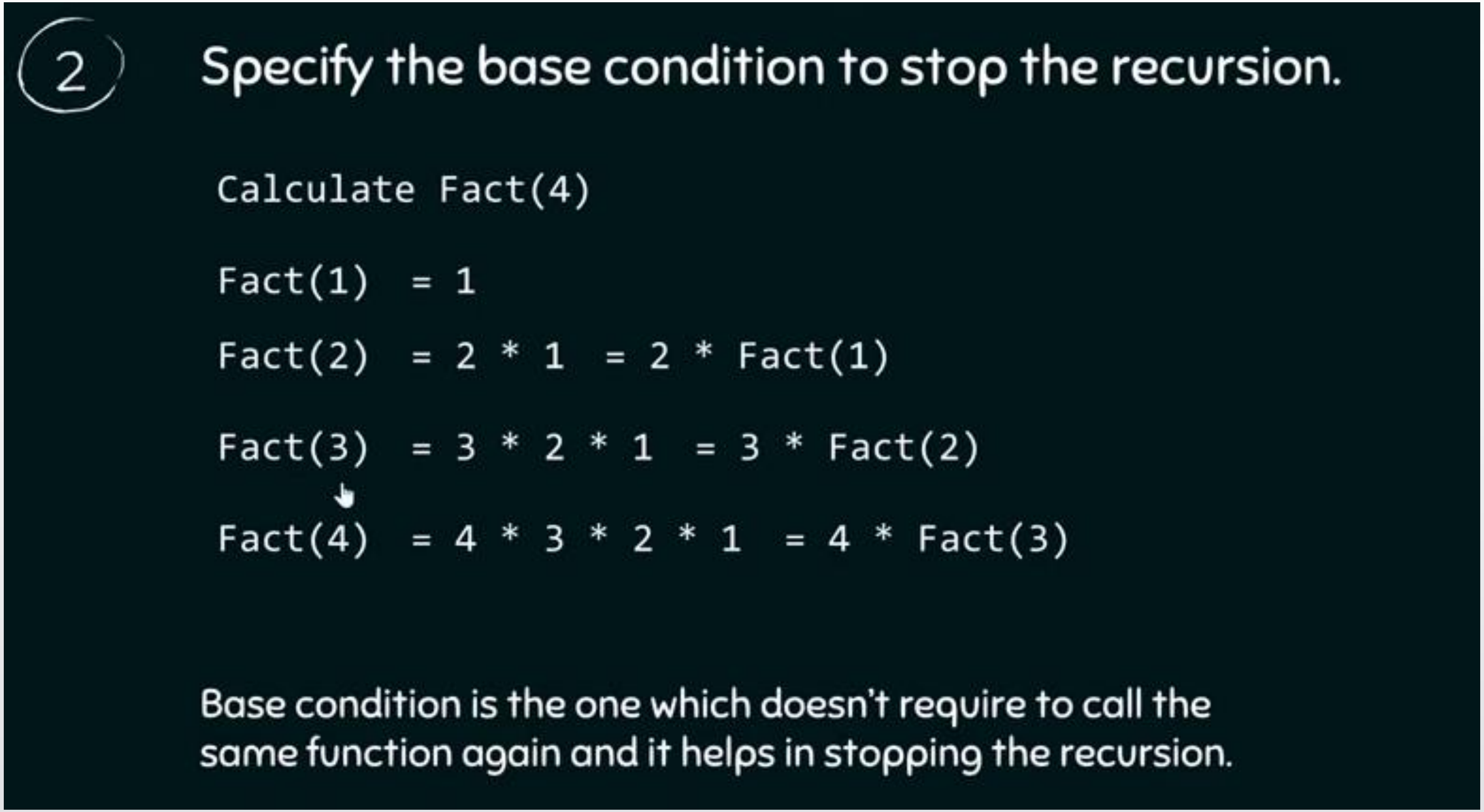
Recursive procedure 1



## 1 Divide the problem into smaller sub-problems.

```
Calculate Fact(4)

Fact(1)  = 1
Fact(2)  = 2 * 1  = 2 * Fact(1)
Fact(3)  = 3 * 2 * 1  = 3 * Fact(2)
Fact(4)  = 4 * 3 * 2 * 1  = 4 * Fact(3)
```

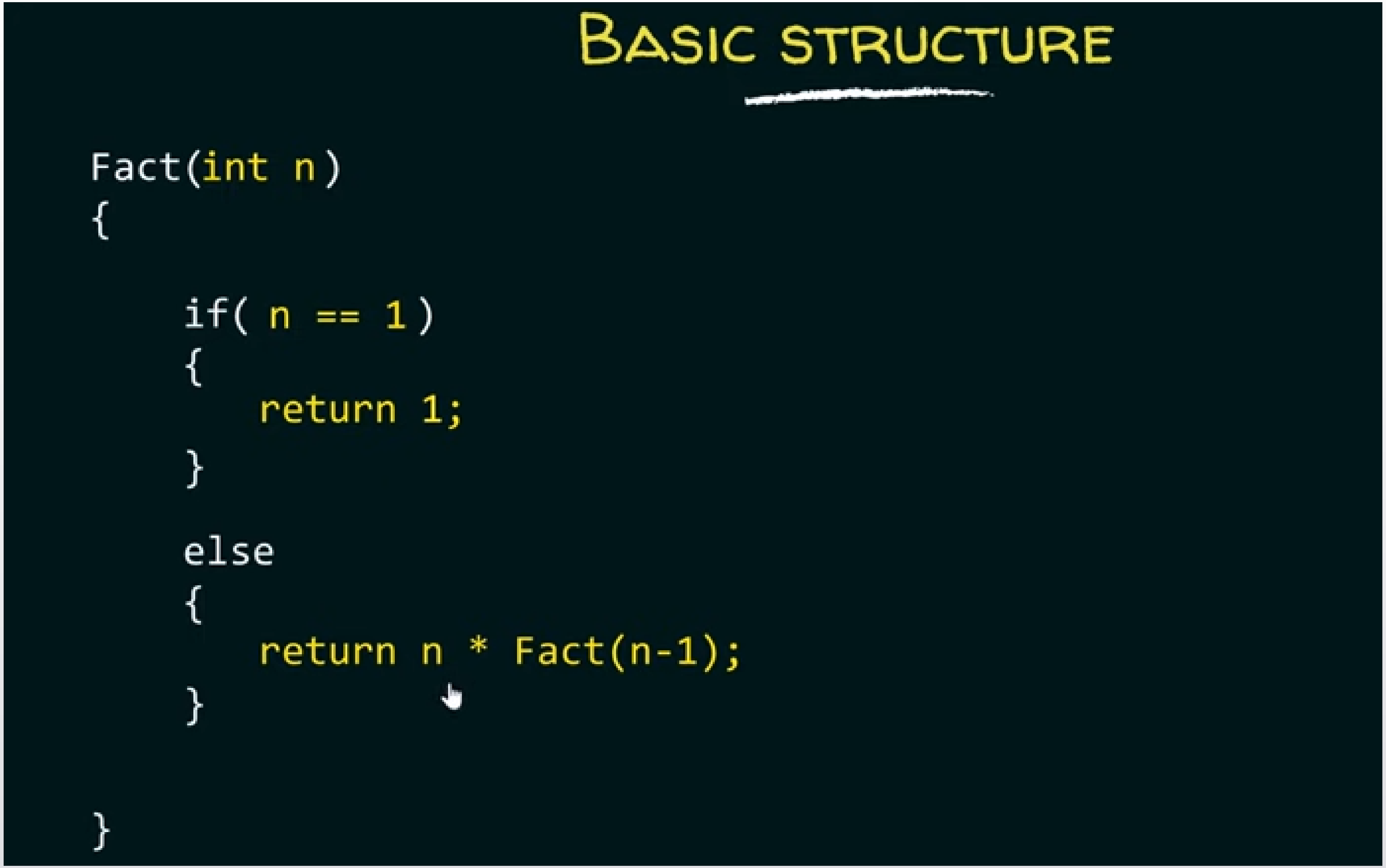
```
Fact(n) = n * Fact(n-1)
```



## 2 Specify the base condition to stop the recursion.

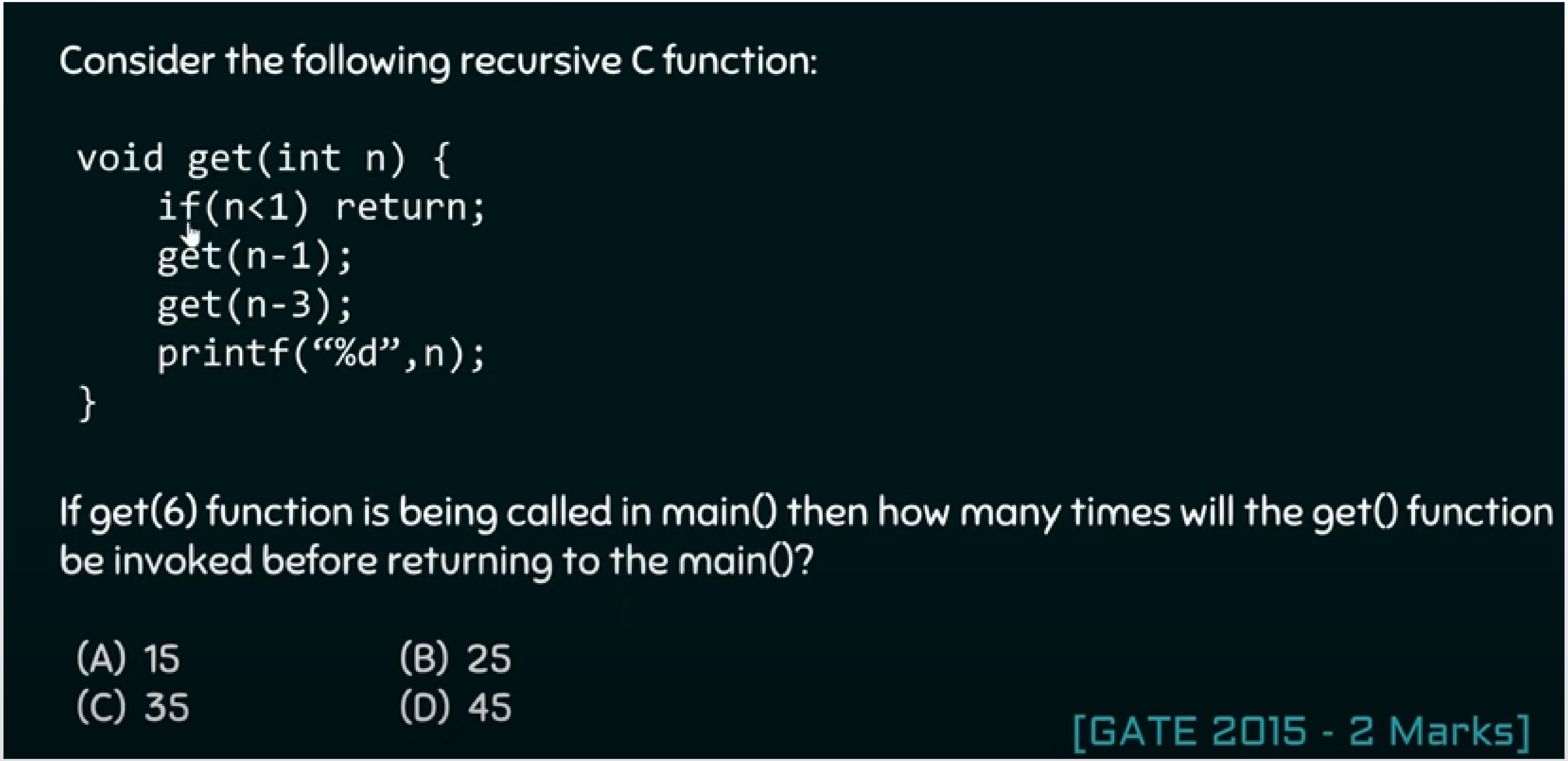
```
Calculate Fact(4)

Fact(1)  = 1

Fact(2)  = 2 * 1  = 2 * Fact(1)

Fact(3)  = 3 * 2 * 1  = 3 * Fact(2)

Fact(4)  = 4 * 3 * 2 * 1  = 4 * Fact(3)
```

Base condition is the one which doesn't require to call the same function again and it helps in stopping the recursion.



# Basic structure

```
Fact(int n)
{

    if( n == 1 )
    {
       return 1;
    }

    else
    {
       return n * Fact(n-1);
    }


}
```



Consider the following recursive C function:

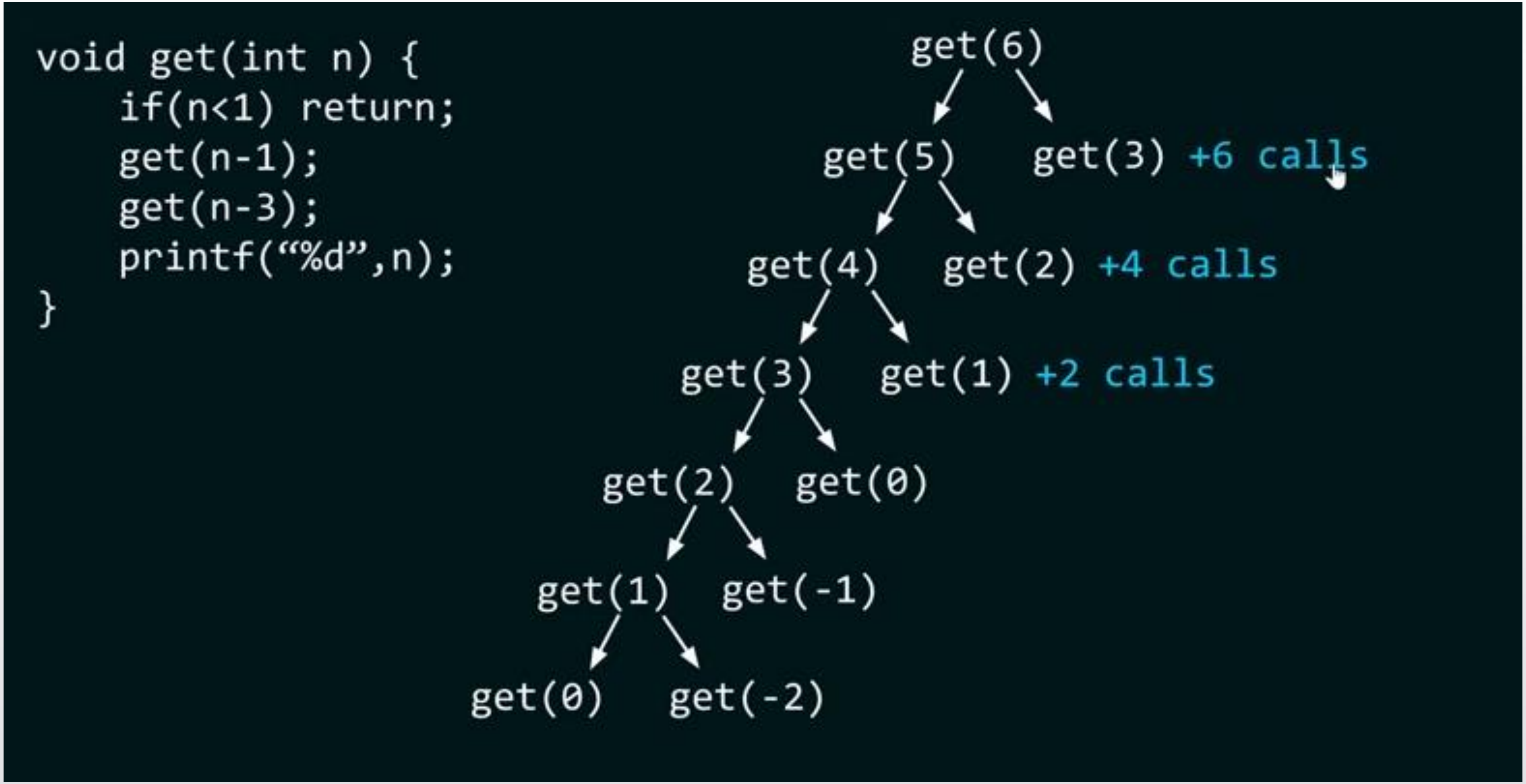
```
void get(int n) {
    if(n<1) return;
    get(n-1);
    get(n-3);
    printf("%d",n);
}
```

If get(6) function is being called in main() then how many times will the get() function be invoked before returning to the main()?

(A) 15 (B) 25

(C) 35 (D) 45

[GATE 2015 - 2 Marks]



```
void get(int n) {
    if(n<1) return;
    get(n-1);
    get(n-3);
    printf("%d",n);
}
```

get(6)

get(5) get(3) +6 calls

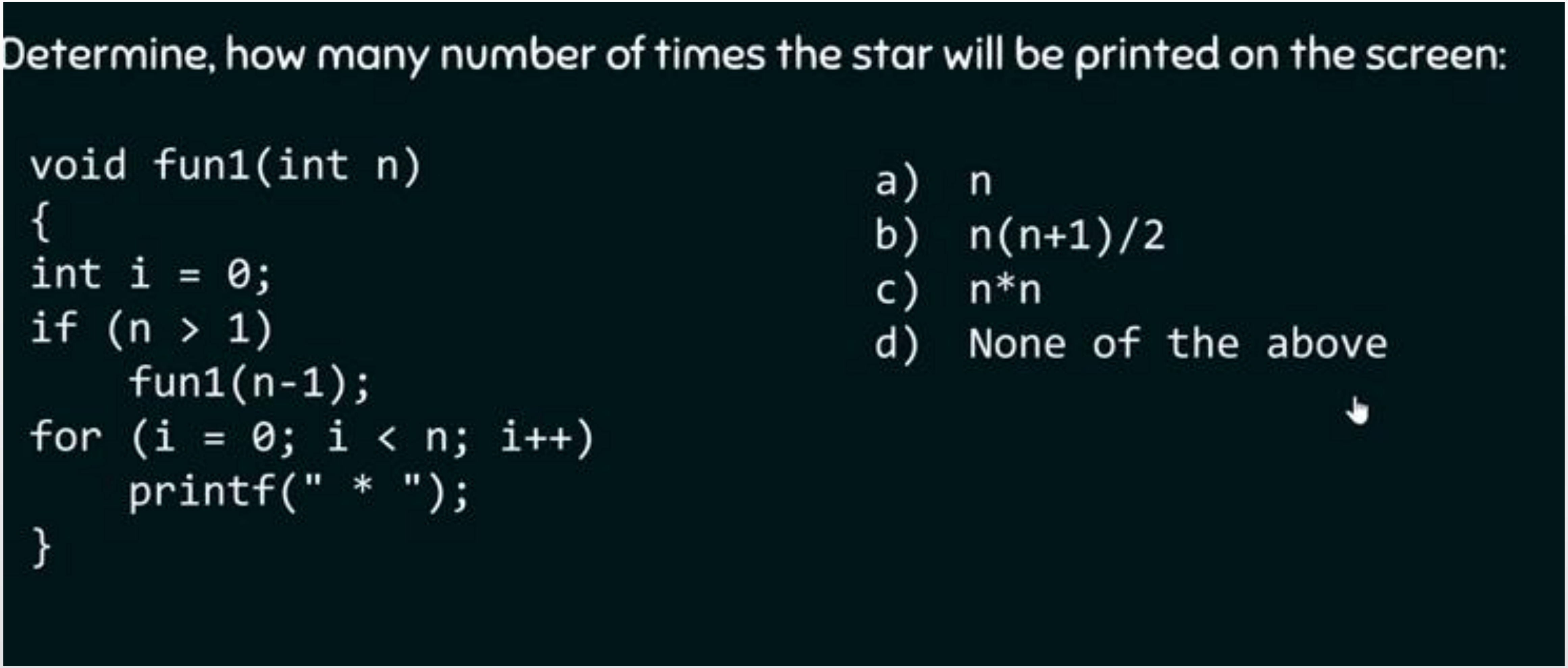
get(4) get(2) +4 calls

get(3) get(1) +2 calls

get(2) get(0)

get(1) get(-1)

get(0) get(-2)



Determine, how many number of times the star will be printed on the screen:

```
void fun1(int n)
{
int i = 0;
if (n > 1)
    fun1(n-1);
for (i = 0; i < n; i++)
    printf(" * ");
}
```

a) n

b) n(n+1)/2

c) n*n

d) None of the above

# Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

Fn = Fn-1 + Fn-2
with seed values

F0 = 0 and F1 = 1.

Given a number n, print n-th Fibonacci Number.

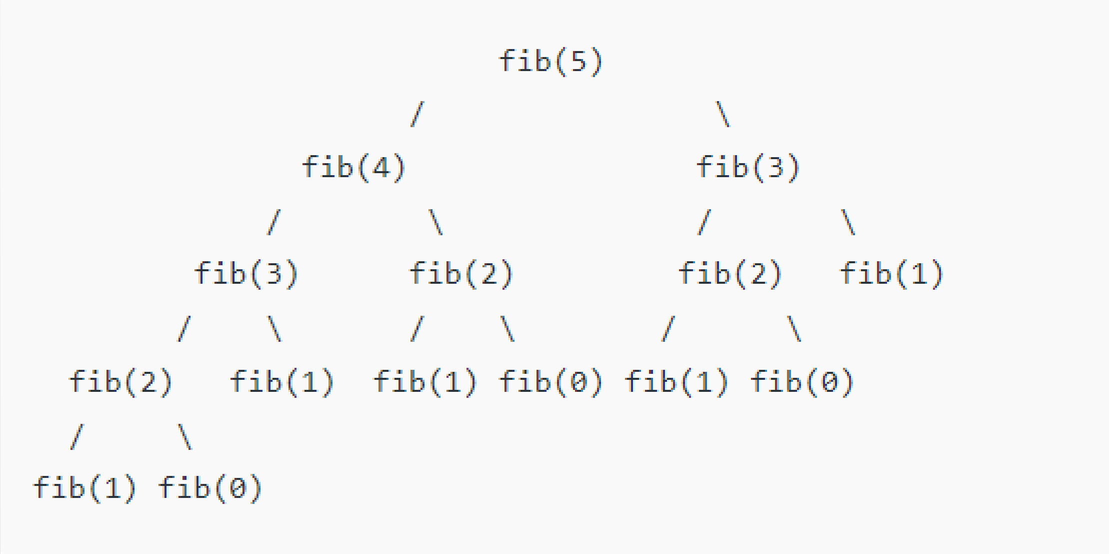
Examples:

Input  : n = 2
Output : 1

Input  : n = 9
Output : 34

```
// Fibonacci Series using Recursion
#include <stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

Time Complexity: Exponential, as every function calls two other functions.



```
                          fib(5)
                     /                \
               fib(4)                fib(3)
             /        \              /       \
         fib(3)      fib(2)         fib(2)   fib(1)
        /    \       /    \        /      \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /     \
fib(1) fib(0)
```

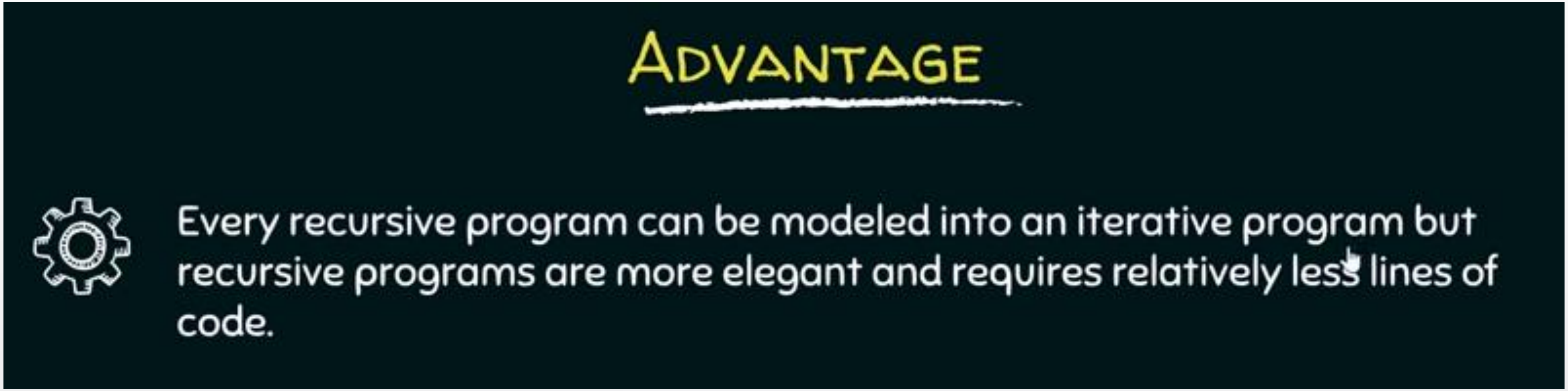
# Method 2: (Use Dynamic Programming)

```
//Fibonacci Series using Dynamic Programming
#include<stdio.h>
int fib(int n)
{
/* Declare an array to store Fibonacci numbers. */
int f[n+2]; // 1 extra to handle case, n = 0
int i;
/* 0th and 1st number of the series are 0 and 1*/
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
        /* Add the previous 2 numbers in the series
                and store it */
        f[i] = f[i-1] + f[i-2];
}
return f[n];
}
```

```
int main ()
{
int n = 9;
printf("%d", fib(n));
getchar();
return 0;
}
```

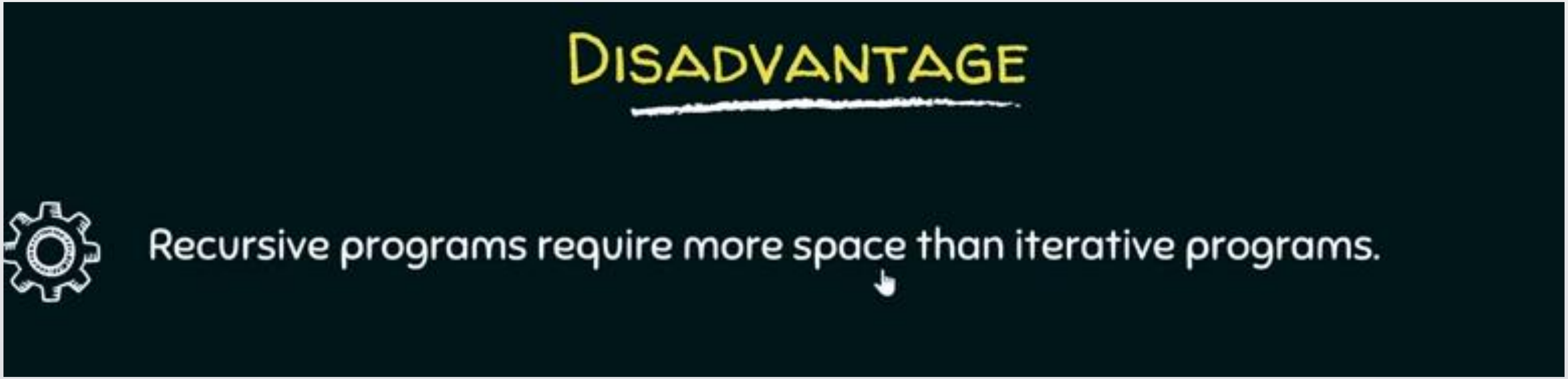
Time complexity: O(n) for given n

# Program to find GCD or HCF of two numbers



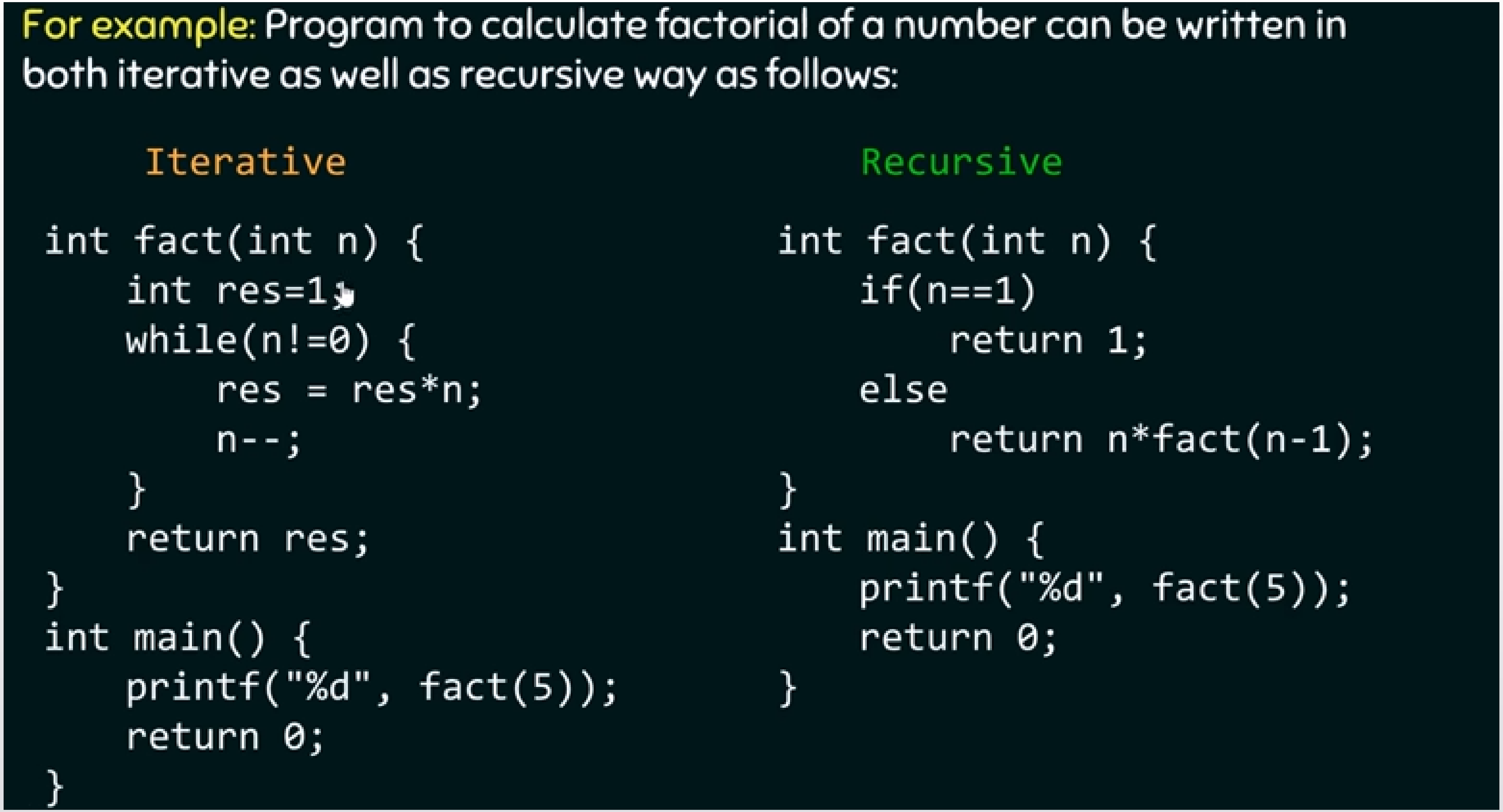
## Advantage

- Every recursive program can be modeled into an iterative program but recursive programs are more elegant and requires relatively less lines of code.



## Disadvantage

- Recursive programs require more space than iterative programs.



For example: Program to calculate factorial of a number can be written in both iterative as well as recursive way as follows:

Iterative

```
int fact(int n) {
    int res=1;
    while(n!=0) {
        res = res*n;
        n--;
    }
    return res;
}
int main() {
    printf("%d", fact(5));
    return 0;
}
```

Recursive

```
int fact(int n) {
    if(n==1)
        return 1;
    else
        return n*fact(n-1);
}
int main() {
    printf("%d", fact(5));
    return 0;
}
```