# C Programming
# Lecture 8-2 : Function (advanced)

# Recursive Function (recursion)

- A function that calls itself (in its definition)

- Classic example : factorial

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

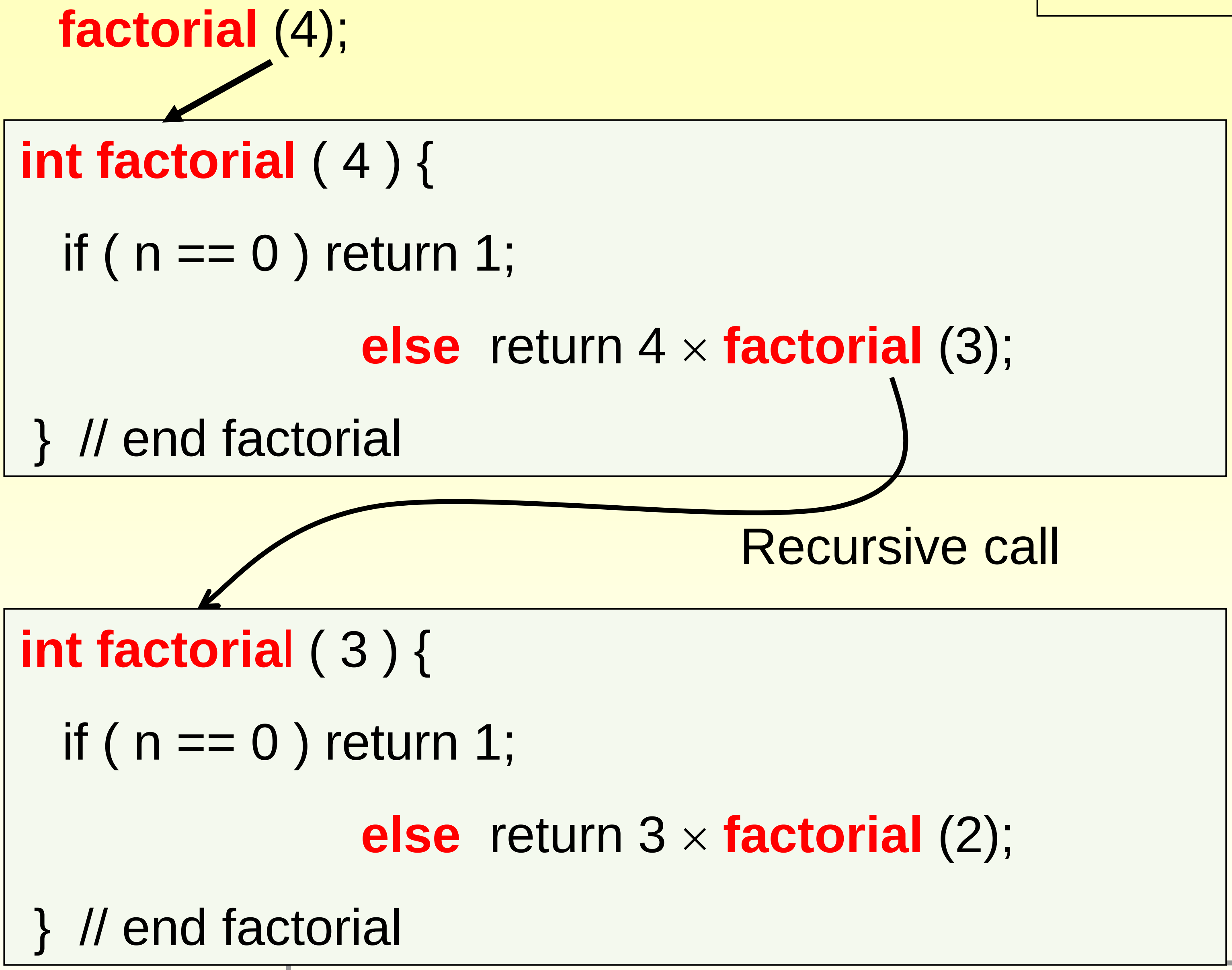
# Recursion example : factorial

$$f(N) = N!$$

$$f(N) = N \times f(N-1)$$

```
// recursive function
int factorial (int n) {       // we assume n is positive integer
   if ( n == 0 )
      return 1;
   else  return n × factorial (n - 1);   // recursive calling
 }  // end factorial
```

```
   . . . . .
printf(“%d”, factorial (4));
   . . . . .
```



factorial (4);

```
int factorial ( 4 ) {

   if ( n == 0 ) return 1;

                                   else  return 4 × factorial (3);

 }  // end factorial
```

Recursive call

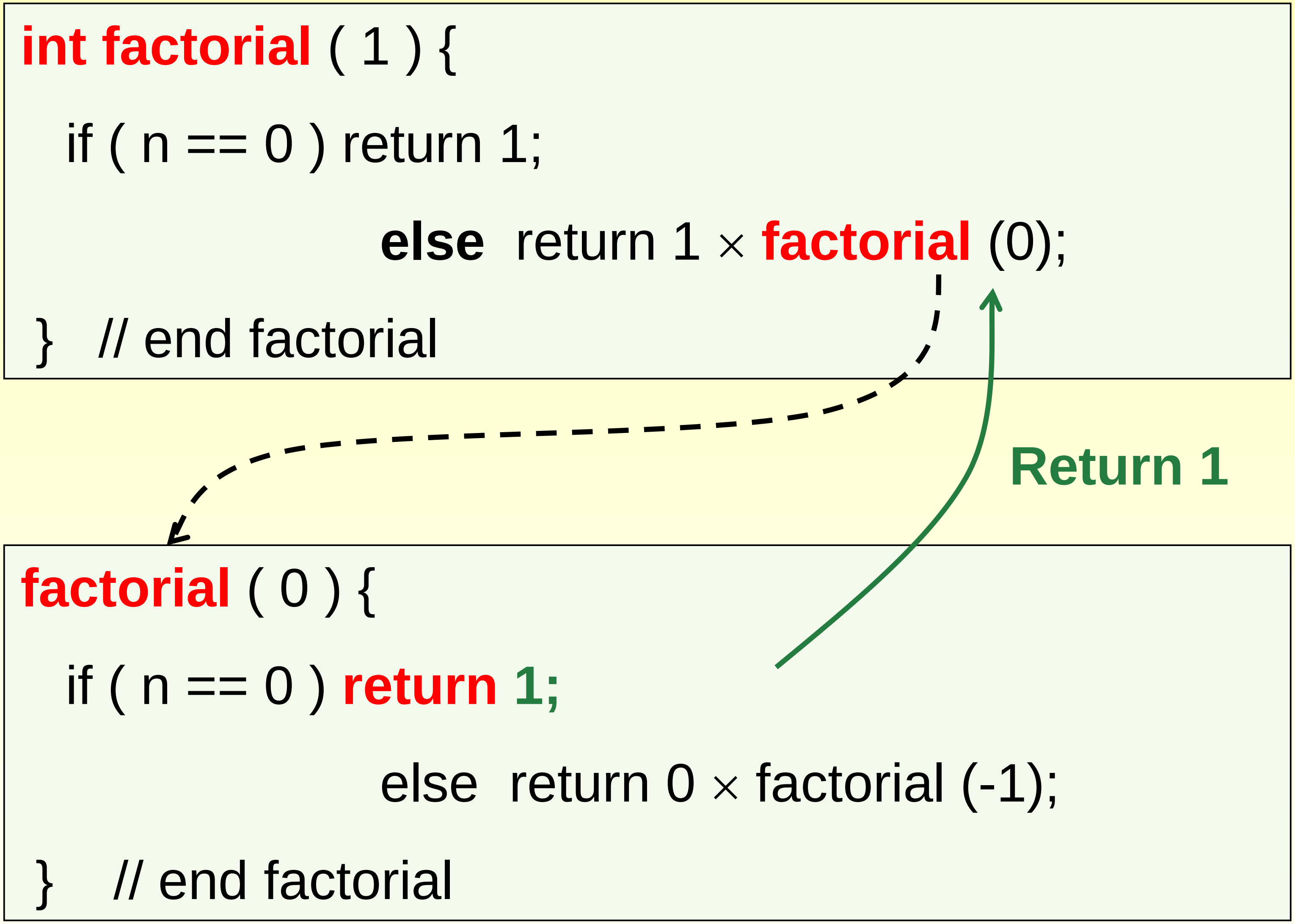
```
int factorial ( 3 ) {

   if ( n == 0 ) return 1;

                                   else  return 3 × factorial (2);

 }  // end factorial
```

recursion

```
int factorial ( 2 ) {
    if ( n == 0 ) return 1;
                  else  return 2 × factorial (1);
 }   //end factorial
```

Recursive call

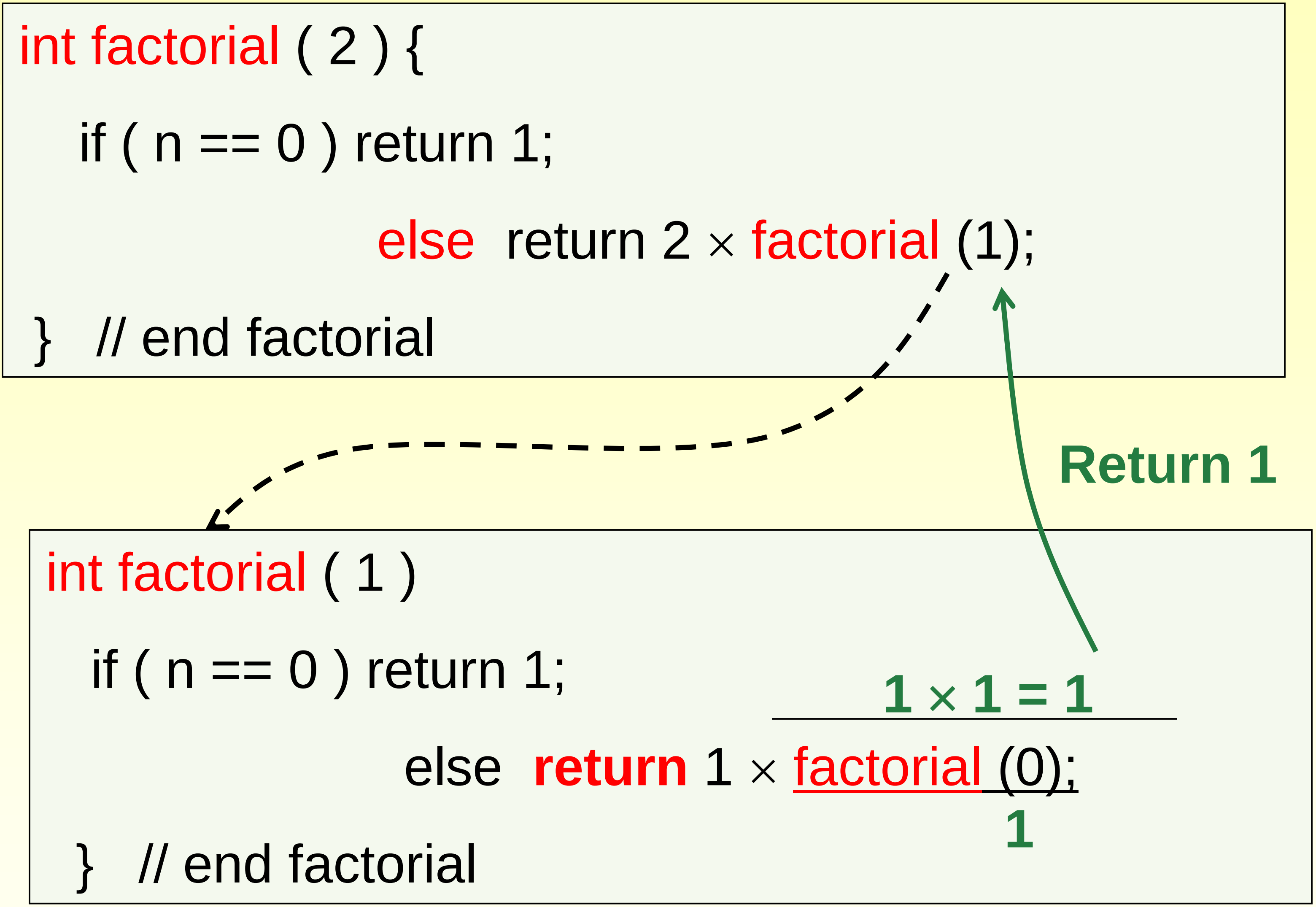
```
int factorial ( 1 ) {
   if ( n == 0 ) return 1;
                  else  return 1 × factorial (0);
  }   // end factorial
```



```
int factorial ( 1 ) {
   if ( n == 0 ) return 1;
                     else  return 1 × factorial (0);
 }   // end factorial
```

Return 1

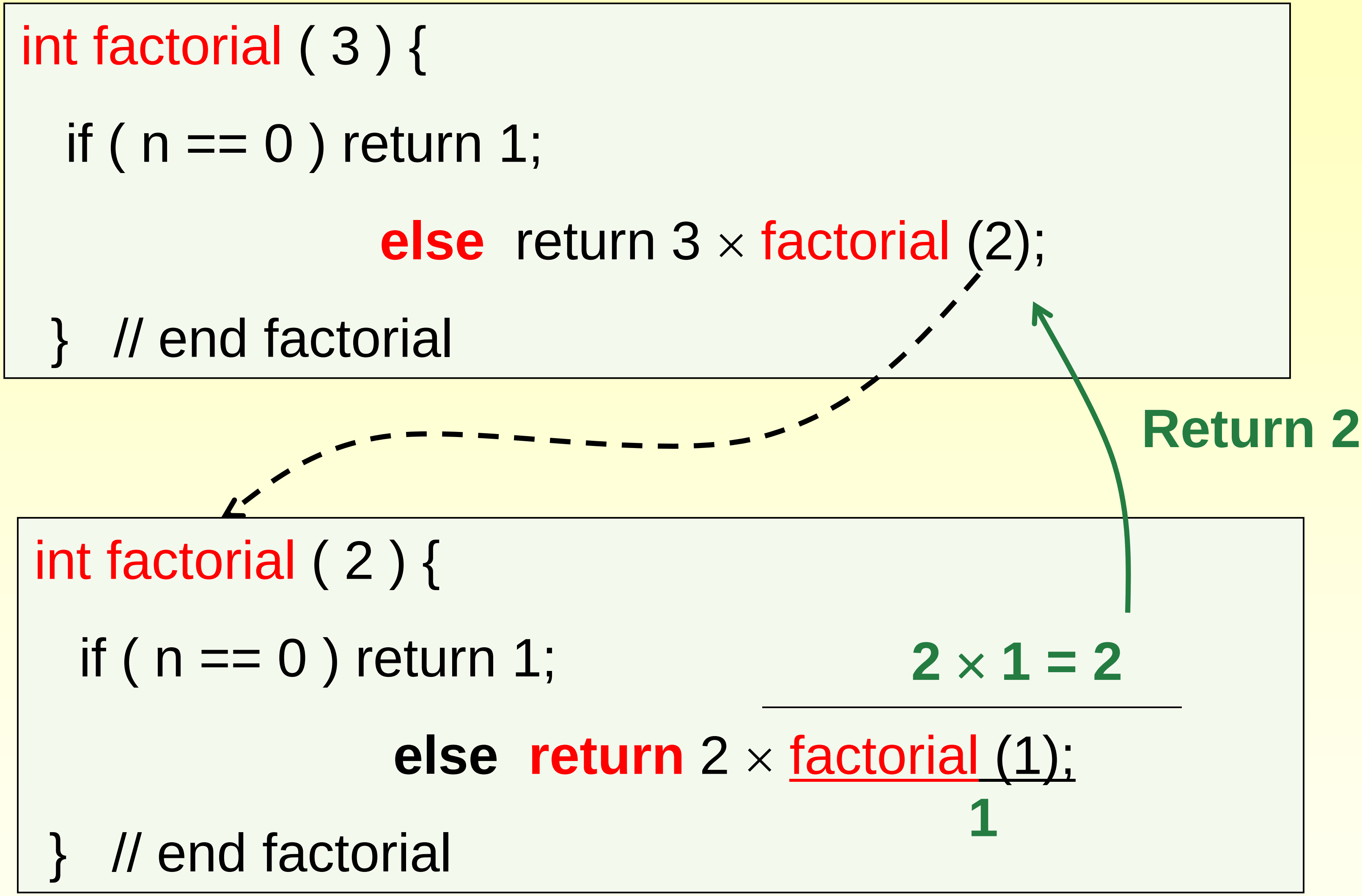
```
factorial ( 0 ) {
   if ( n == 0 ) return 1;
                     else  return 0 × factorial (-1);
 }    // end factorial
```



```
int factorial ( 2 ) {
   if ( n == 0 ) return 1;
                  else  return 2 × factorial (1);
}   // end factorial
```

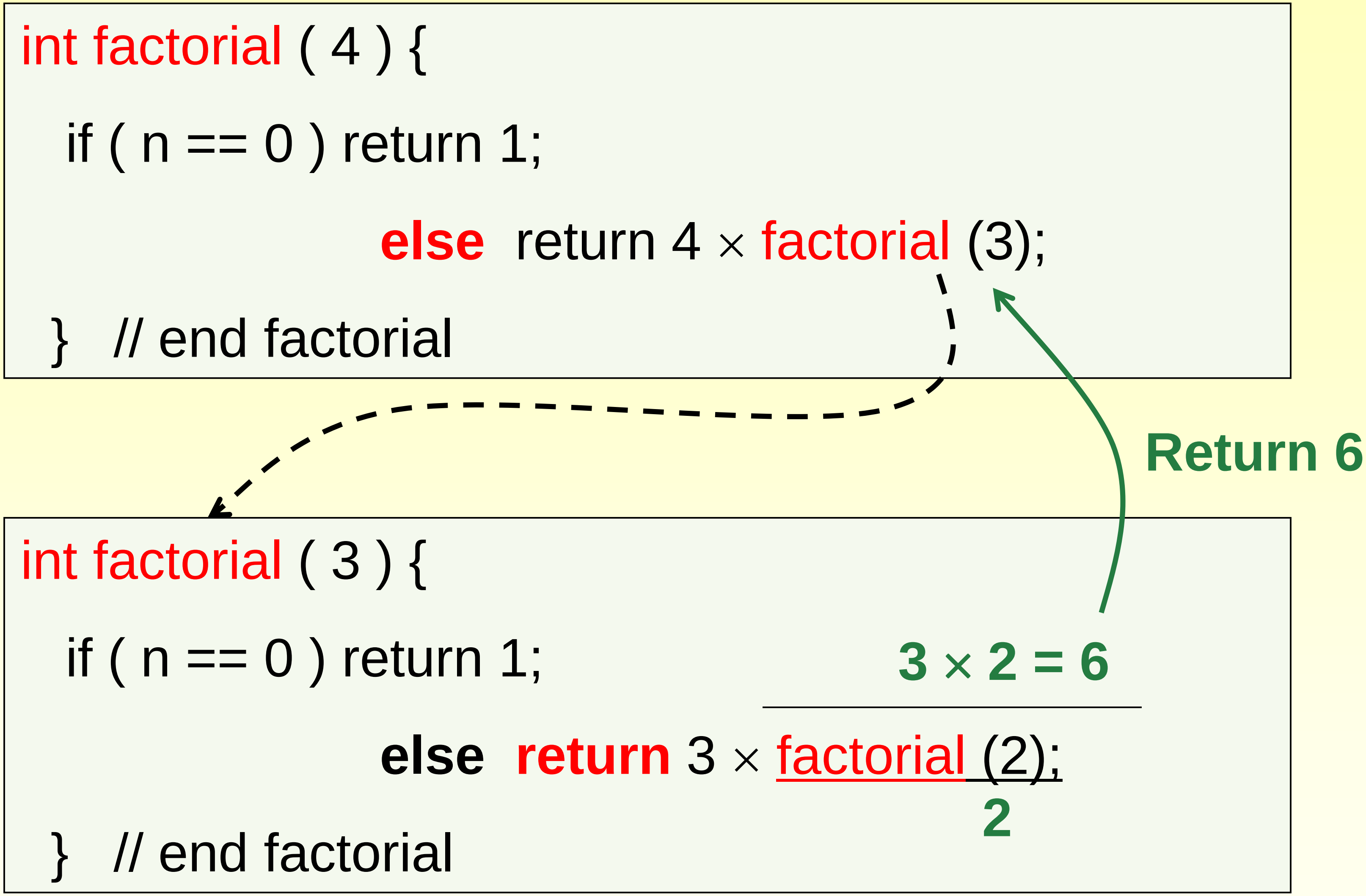
Return 1

```
int factorial ( 1 )
   if ( n == 0 ) return 1;
                                        1 × 1 = 1
                  else  return 1 × factorial (0);
                                             1
}   // end factorial
```



```
int factorial ( 3 ) {
   if ( n == 0 ) return 1;
                        else  return 3 × factorial (2);
  }   // end factorial
```

```
int factorial ( 2 ) {
   if ( n == 0 ) return 1;
                        else  return 2 × factorial (1);
  }   // end factorial
```



```
int factorial ( 4 ) {

   if ( n == 0 ) return 1;

                    else  return 4 × factorial (3);

  }   // end factorial
```

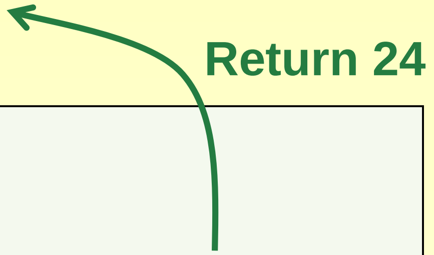
Return 6

```
int factorial ( 3 ) {

   if ( n == 0 ) return 1;

                    else  return 3 × factorial (2);

  }   // end factorial
```

3 × 2 = 6

2

```
                cout << factorial (4);   //output 24
                         24
```



Return 24

```
int factorial ( 4 ) {

   if ( n == 0 ) return 1;
                                         4 × 6 = 24
          else  return 4 × factorial (3)
                                      6
 }   // end factorial
```

# Exercise : fibonacci numbers

- Code?

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geqq 2 \end{cases}$$

# Call-by-Value

- The value of Argument variable will be copied to parameter variable
- The value of Argument variable is not affected during the processing of function
- Advantage : we can avoid(exclude) unwanted side-effects
- C provides call-by-value mechanism.

# Example : swap function

```
#include <stdio.h>

void swap(int a , int b)
{
          int temp;
          temp=a;
          a=b;
          b=temp;
}

int main()
{
          int x=3, y=2;
          printf(“before: x=%d, y=%d\n”,x,y);
          swap(x,y);
          printf(“after : x=%d, y=%d\n”,x,y);
}
```

```
#include <stdio.h>

void swap(int* a , int* b)
{
          int temp;
          temp=*a;
          *a=*b;
          *b=temp;
}

int main()
{
          int x=3, y=2;
          printf(“before: x=%d, y=%d\n”,x,y);
          swap(&x,&y);
          printf(“after : x=%d, y=%d\n”,x,y);
}
```

```
Output :
```

```
Output :
```

# Macro function

- Effective when a function is short and simple

- `#define min(x,y) ( (x<y) ? (x) : (y) )`
- `#define max(x,y) ( (x>y) ? (x) : (y) )`

- Advantage?
  - No overhead for function call & return

# Example : MAX

```
#include <stdio.h>

#define MAX(x, y) (x > y)? x: y

int main()
{
    int i, j;
    int max;

    printf("get two integers : ");
    scanf("%d %d", &i, &j);
    max = MAX(i, j);
    printf("MAX(%d, %d) = %d\n", i, j, max);

    return 0;
}
```

# Inline function

- the compiler will insert the complete body of the inline function in every place in the code where that function is used.

- Reduce overhead for function call & return
- Effective when a function is short and simple

```
inline int cube( int n )
{
       return n * n * n;
}
```

# Static (additional)

```
#include <stdio.h>

int count()
{
    static int n = 0;

    return ++n;
}

int main()
{
    int i;

    for (i = 0; i < 5; ++i)
        printf("count = %d\n", count());

    return 0;
}
```