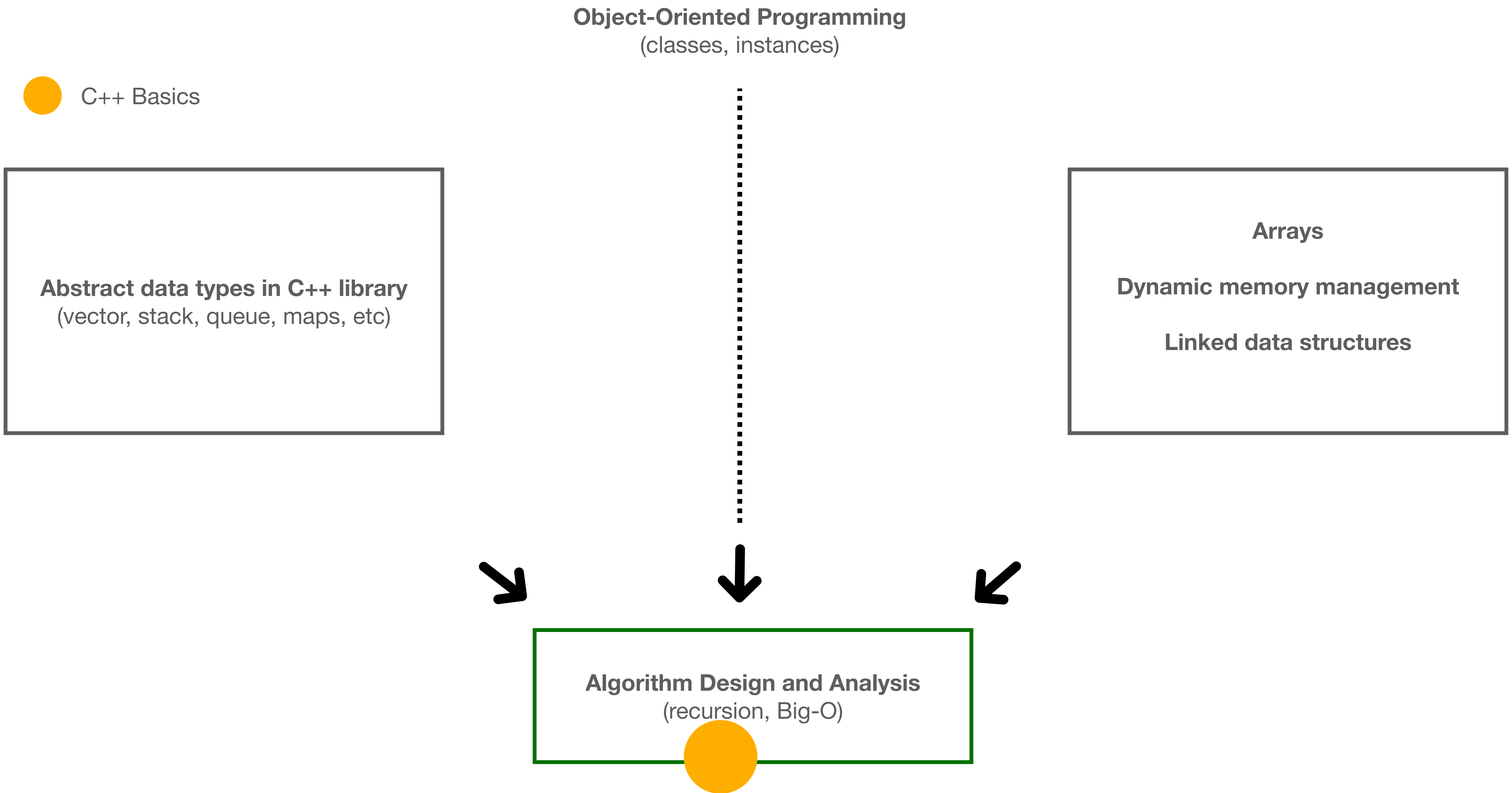
Lecture2: Recursion (Intro)

Xingyu Zhou

01/13 2025

* Part of the materials are inspired or borrowed from the course CS 106B at Stanford

Roadmap



● C++ Basics

Object-Oriented Programming
(classes, instances)

Abstract data types in C++ library
(vector, stack, queue, maps, etc)

Arrays

Dynamic memory management

Linked data structures

Algorithm Design and Analysis
(recursion, Big-O)

Outline of Today’s Class

- Review
- Defining Recursion
- Recursion + Stack frames
- Recursive Problem-Solving

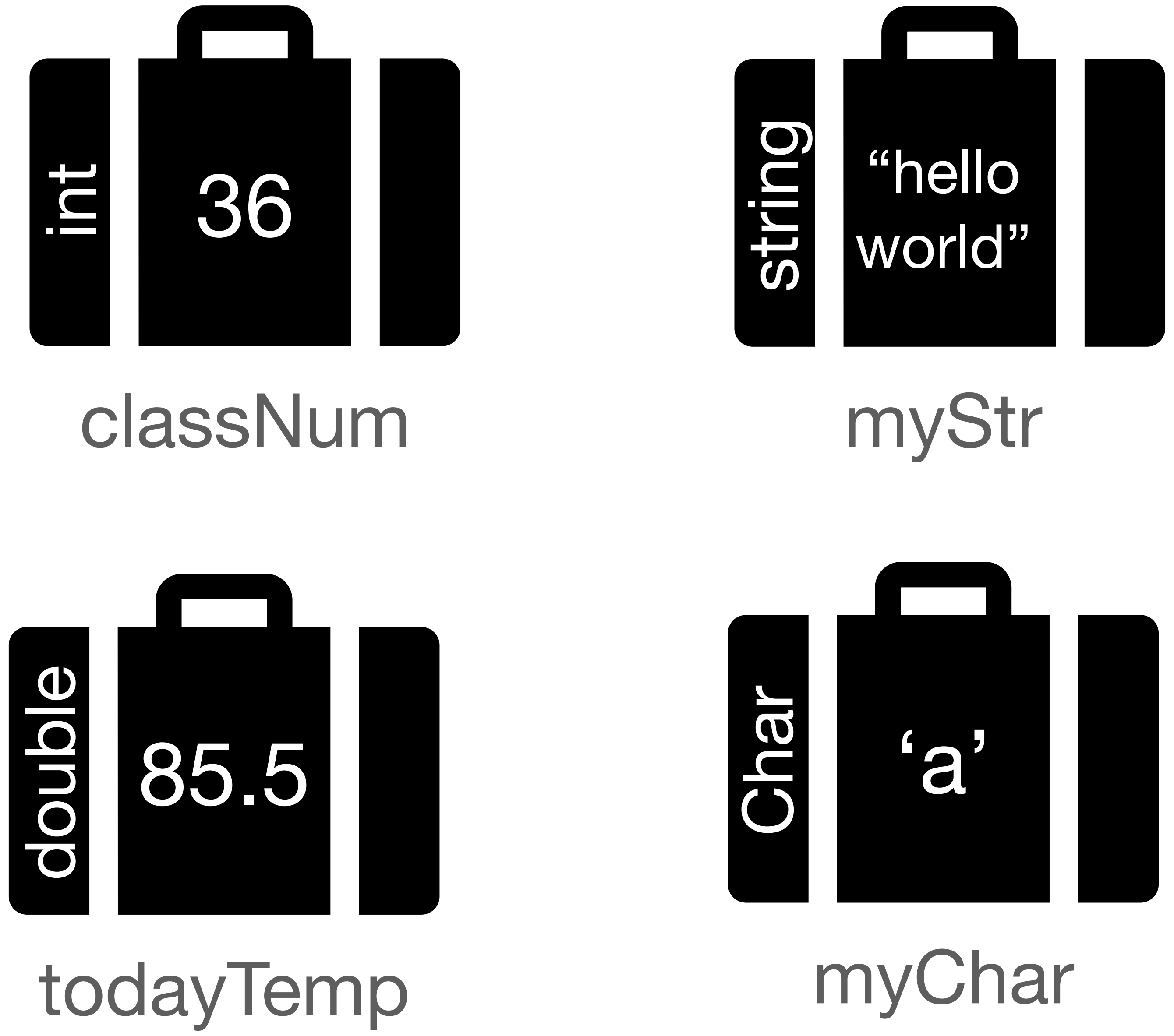
Review…

Review…

Variables

- A way for code to store information by associating a value with a name

We can think of a variable as a *named container storing a value*



In the computer, each variable corresponds to a *piece of memory*

Note: In C++, we use the camelCase naming convention for variables

Review…

Boolean expression

expression	`true` if
`a < b`	`a is less than b`
`a <= b`	`a is less than or equal to b`
`a > b`	`a is larger than b`
`a >= b`	`a is larger than or equal to b`
`a == b`	`a is equal to b`
`a != b`	`a is not equal to b`

operator	`true` if
`exp1 && exp2`	`both exps are true`
`exp1 || exp2`	`at least one is true`
`! exp`	`exp is false`

- **Note:** be careful about == vs. = as well as && vs. &

assignment operator

bit "and" operator

Review…

Conditional statement

- The C++ **`if`** statement tests a boolean expression and runs a block of code if the expression is **`true`**, and, optionally, runs a different block of code if the expression is **`false`**. The **`if`** statement has the following format:

```
if (expression) {
   Statements if expression is true;
} else {
   Statements if expression is false;
}
```

Note: The parentheses around expression are required

- One can also add additional conditions

```
if (expression1) {
   Statements if expression1 is true;
} else if (expression2) {
   Statements if expression1 is false and expression2 is true;
} else {
   Statements if neither expression1 nor expression2 is true;
}
```

Note: C++ explicitly uses curly braces to capture a block (different from Python)

Review...

`While` loop

- Loops repeat the execution of a certain block of codes multiple times
- `while` loops often used when you want to continue executing something until a certain condition is met and *you don't know exactly how many times you want to iterate for*

```
while (expression) {

    Statement;

    Statement;

}
```

Continue until the expression is false

```
int i = 0

while (i < 5) {

    cout << i << endl;

    i++;

}
```

Note: The i++ increments the variable i by 1, and is the reason C++ got its name!

Review…

for loop

- **for** loops are great when you have a known, fixed number of times that you want to execute a block of code
- **for** loop has the following syntax in c++

```
for (initializationStatement; testExpression; updateStatement) {

    Statement;

    Statement;

}
```

```
for (int i = 0; i < 5; i++) {

    cout << i << endl;

}
```

Review…

Function

```
returnType functionName (varType parameter1, varType parameter2, …)
```

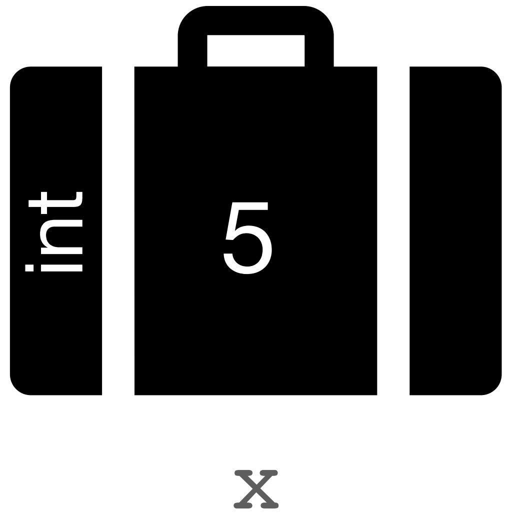
function prototype
(often placed in header file)

```
returnType functionName (varType parameter1, varType parameter2,…) {
    returnType variable = /* Some fancy code. */
    /* Some more code to actually do things. */
    return variable;
}
```

function definition

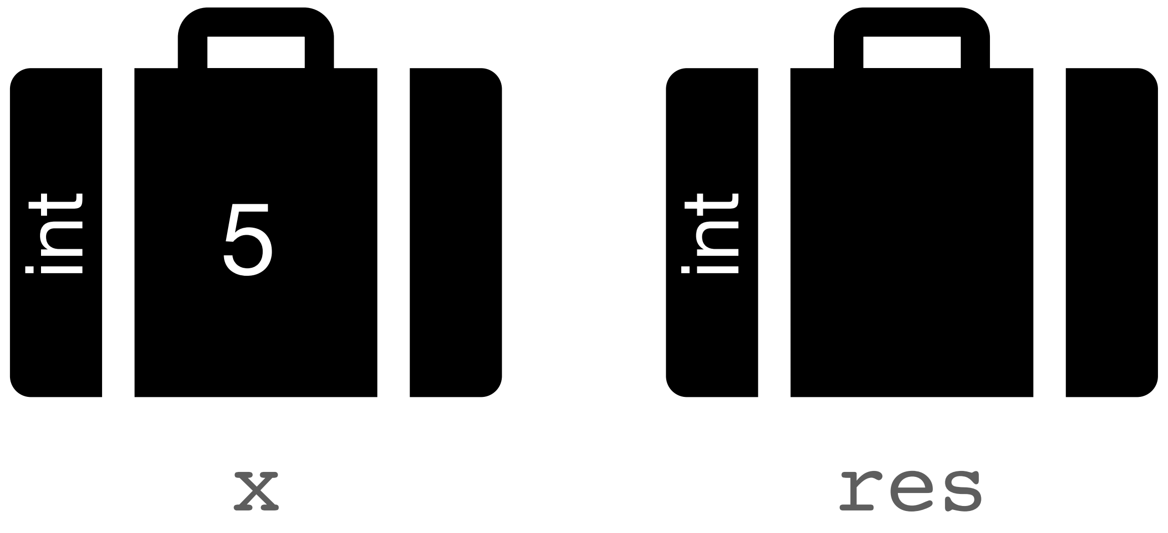
Review…

```
int main () {

    int x = 5;
    int res = doubleValue (x);

    cout << res << endl;
    cout << x << endl;

}
```



Review…

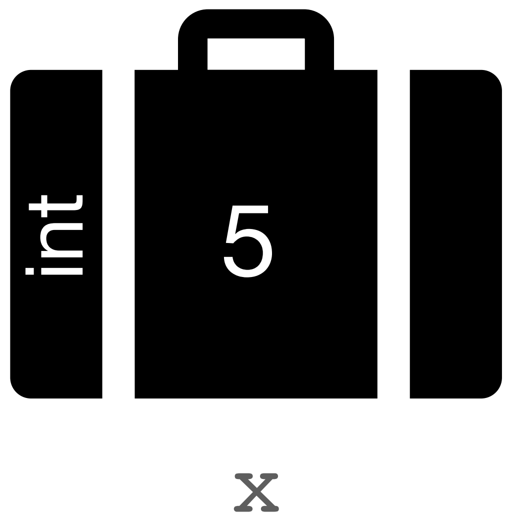
```
int main () {

    int x = 5;
    int res = doubleValue (x);

    cout << res << endl;
    cout << x << endl;

}
```



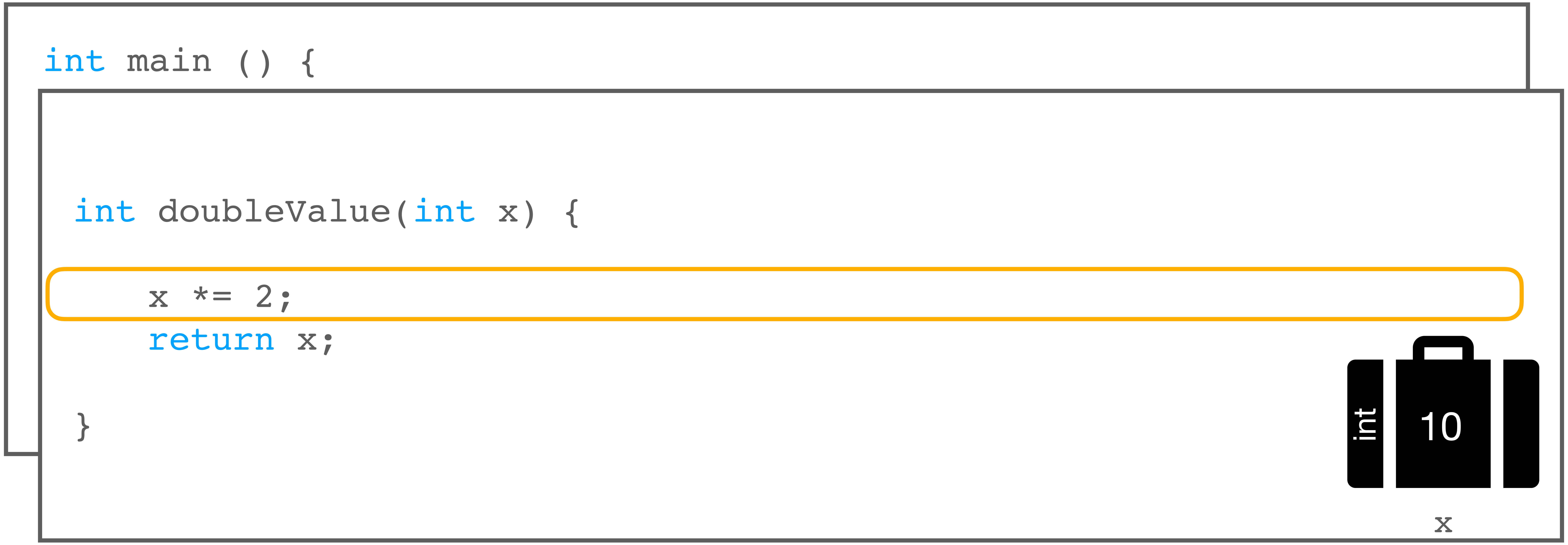
Review…

```
int main () {
```

```
int doubleValue(int x) {

    x *= 2;
    return x;

}
```



Review…



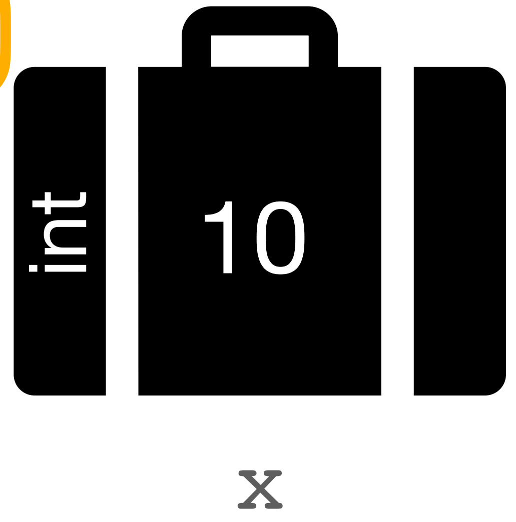
```
int main () {
```

```
int doubleValue(int x) {

    x *= 2;
    return x;

}
```

Review...

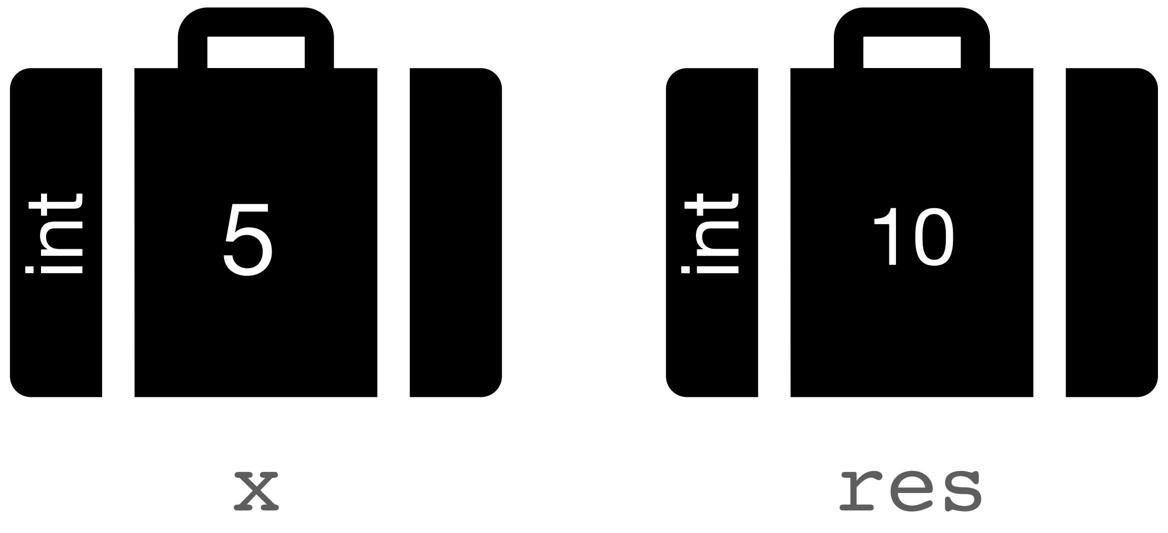
```
int main () {
```

```
int doubleValue(int x) {

    x *= 2;
    return x;

}
```



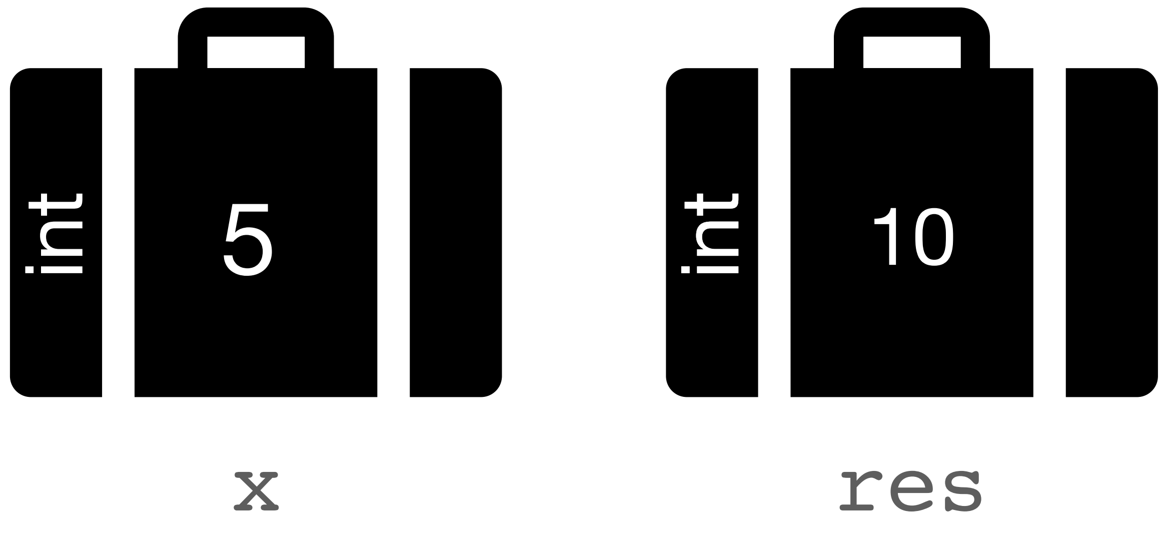
Review…

```
int main () {

    int x = 5;
    int res = doubleValue (x);

    cout << res << endl;
    cout << x << endl;

}
```



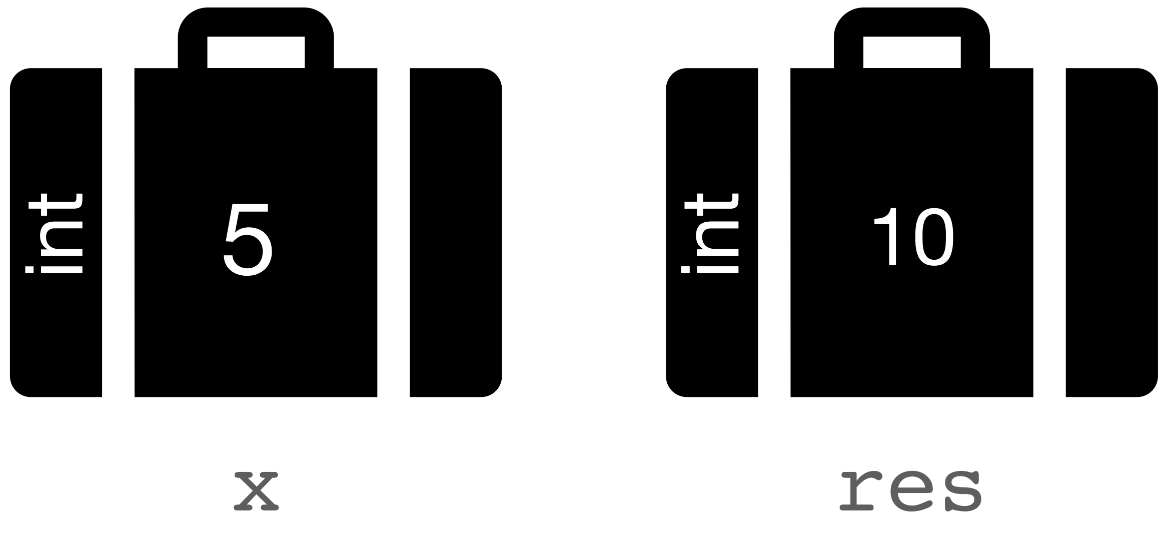
Review…

```
int main () {

    int x = 5;
    int res = doubleValue (x);

    cout << res << endl;
    cout << x << endl;

}
```



Review…

```
int main () {

    int x = 5;
    int res = doubleValue (x);

    cout << res << endl;
    cout << x << endl;

}
```



Review…

```
int main () {

    int x = 5;
    int res = doubleValue (x);

    cout << res << endl;
    cout << x << endl;

}
```

- **Pass-by-value:** copy the value of arguments to the local parameters of a function
  - Changes inside the function will NOT change the values of the arguments

- **Local scope:** each variable only lives (be active) within a range, often given by { }.

Recursion

[A function that calls itself on a smaller input]

Motivating example

`Factorials`

- The factorial of a non-negative integer `n,` given by `n!` is

```
n! = n x (n-1) x (n-2) x … 2 x 1
```

- For example
  - 3! = 3 x 2 x 1 = 6
  - 4! = 4 x 3 x 2 x 1 = 24
  - 5! = 5 x 4 x 3 x 2 x 1 = 120
  - 0! = 1 (by definition)

Live Demo

[Iterative version of factorial]

https://www.onlinegdb.com/edit/WOB_60Mha

Motivating example

`Factorials`

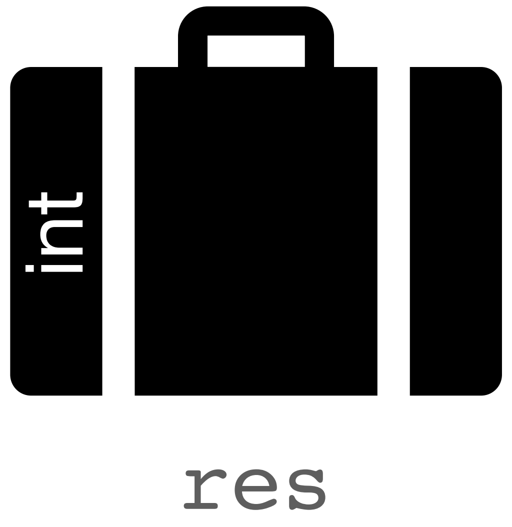
- Another view of factorial `n!`
  - `If n == 0, return 1`
  - `Else return n x (n-1)!`
- We see that the function call itself with a smaller input
- This represents a *recursive leap of faith*.
  - Trust the sub-call to finish the task
  - Do not worry about the detail
  - Focus on how to combine the returned value to calculate the final result

Live Demo

[Recursive version of factorial]

Recursion in action

```
int main () {

    int res = factorial(5);
    cout << “5! = ” << res << endl;
    return 0;

}
```

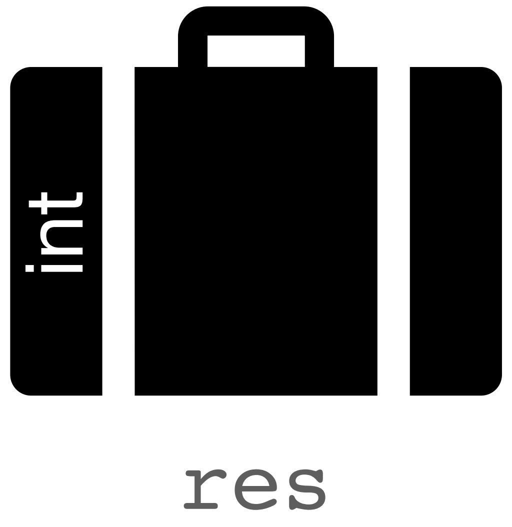


This is a “stack frame.” One gets created each time a function is called.

- The “stack” is where in your computer’s memory the information is stored.
- A “frame” stores all of the data (variables) for that particular function call.

Recursion in action

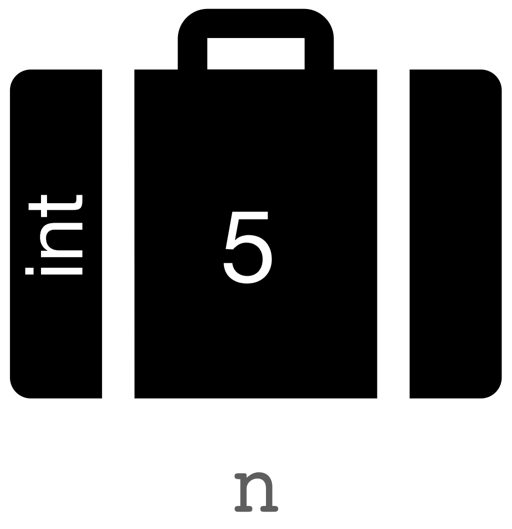
```
int main () {

   int res = factorial(5);
   cout << “5! = ” << res << endl;
   return 0;

}
```



Recursion in action

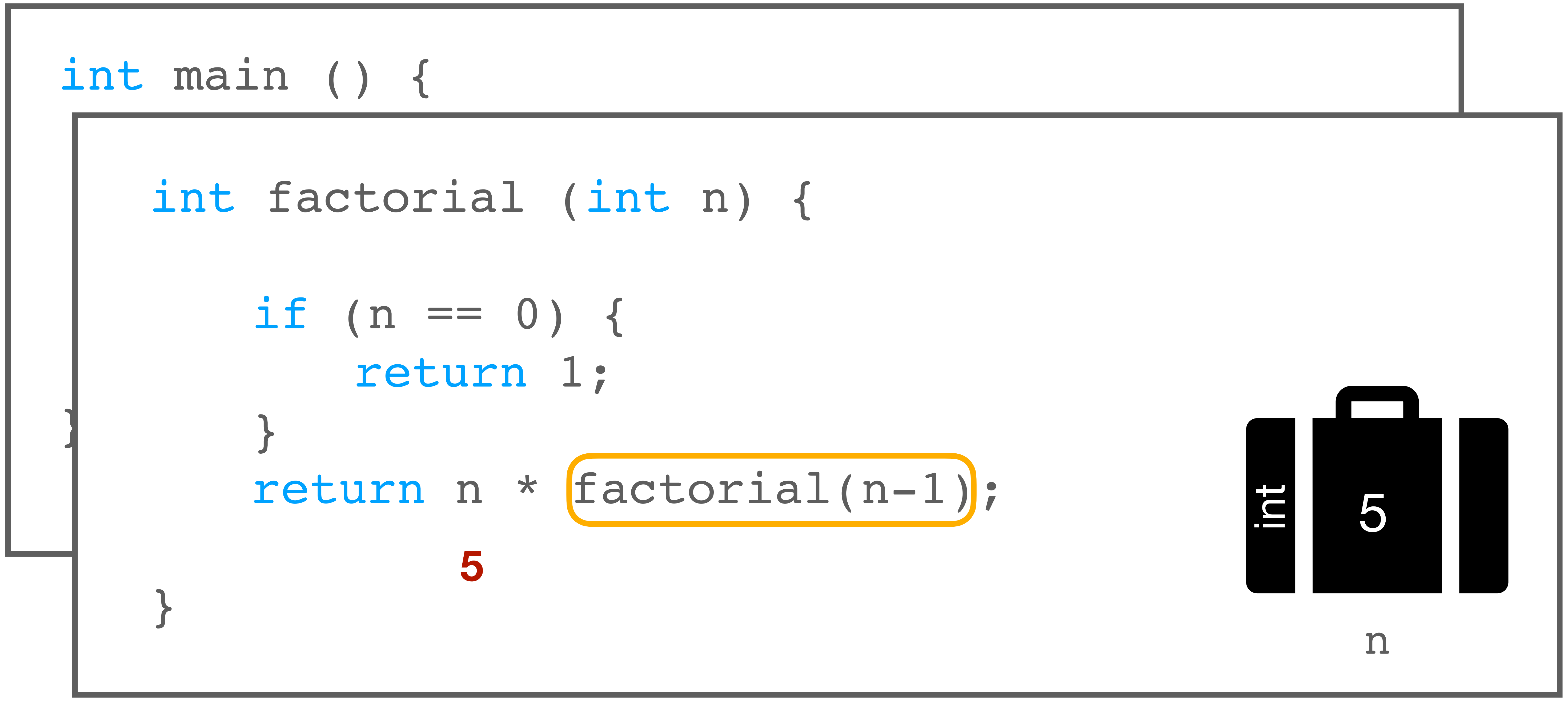
```
int main () {

}
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
            5
}
```



A new frame is created when a function is called

Recursion in action

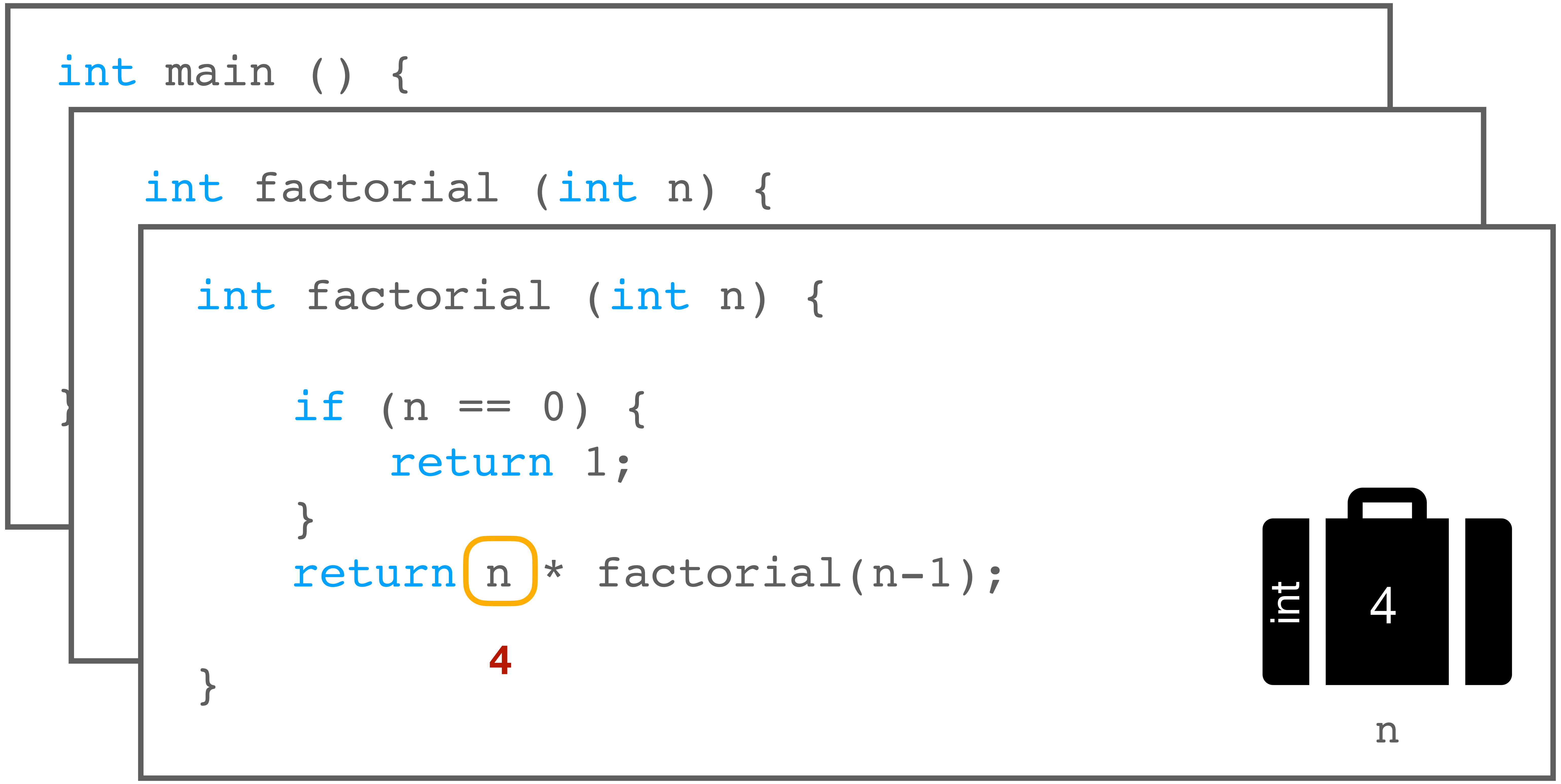


```
int main () {
}
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
           5
}
```

A new frame is created when a function is called

Recursion in action

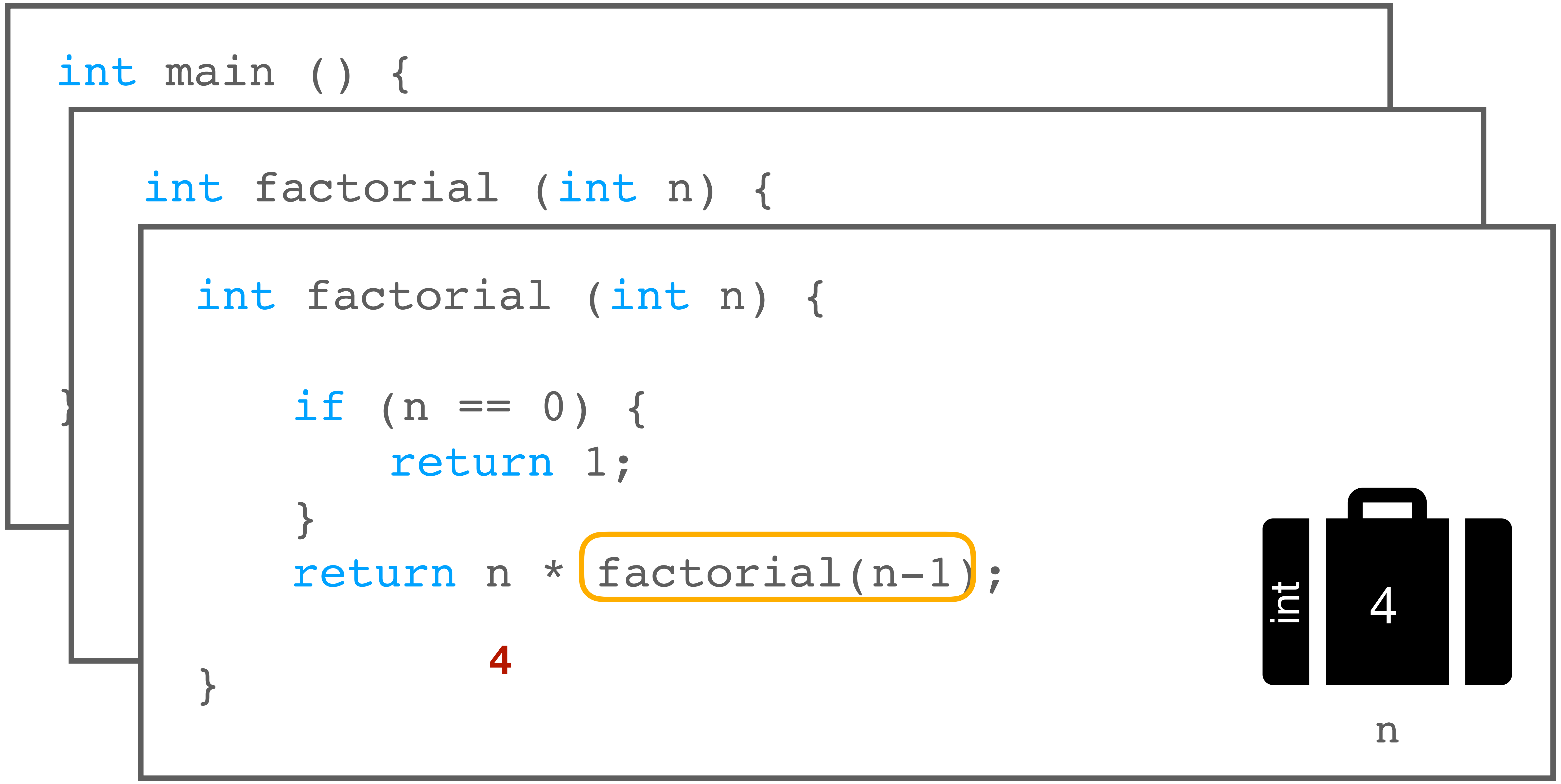


```
int main () {

}
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);

}
```

Recursion in action

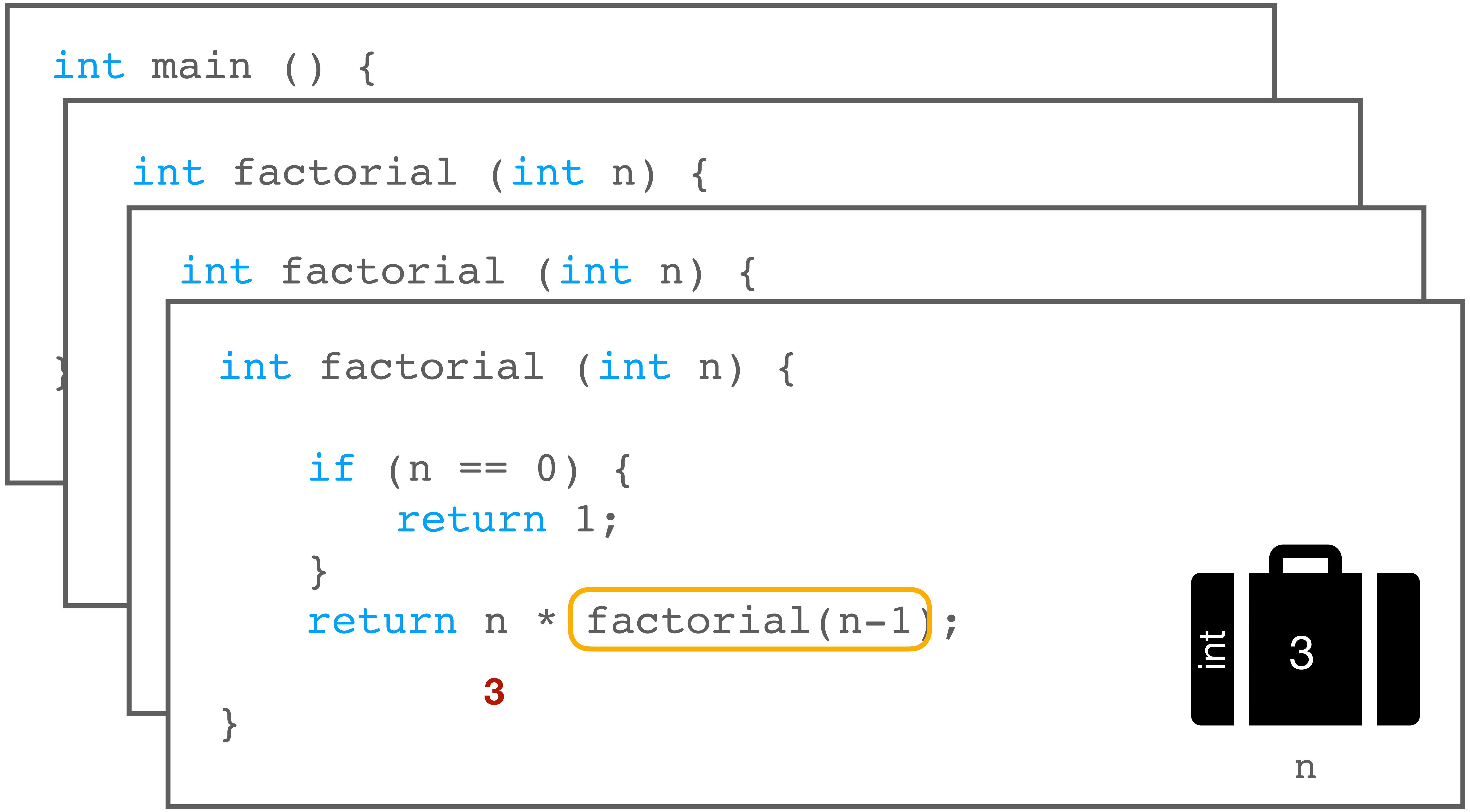


```
int main () {

}
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
              4
}
```

Recursion in action



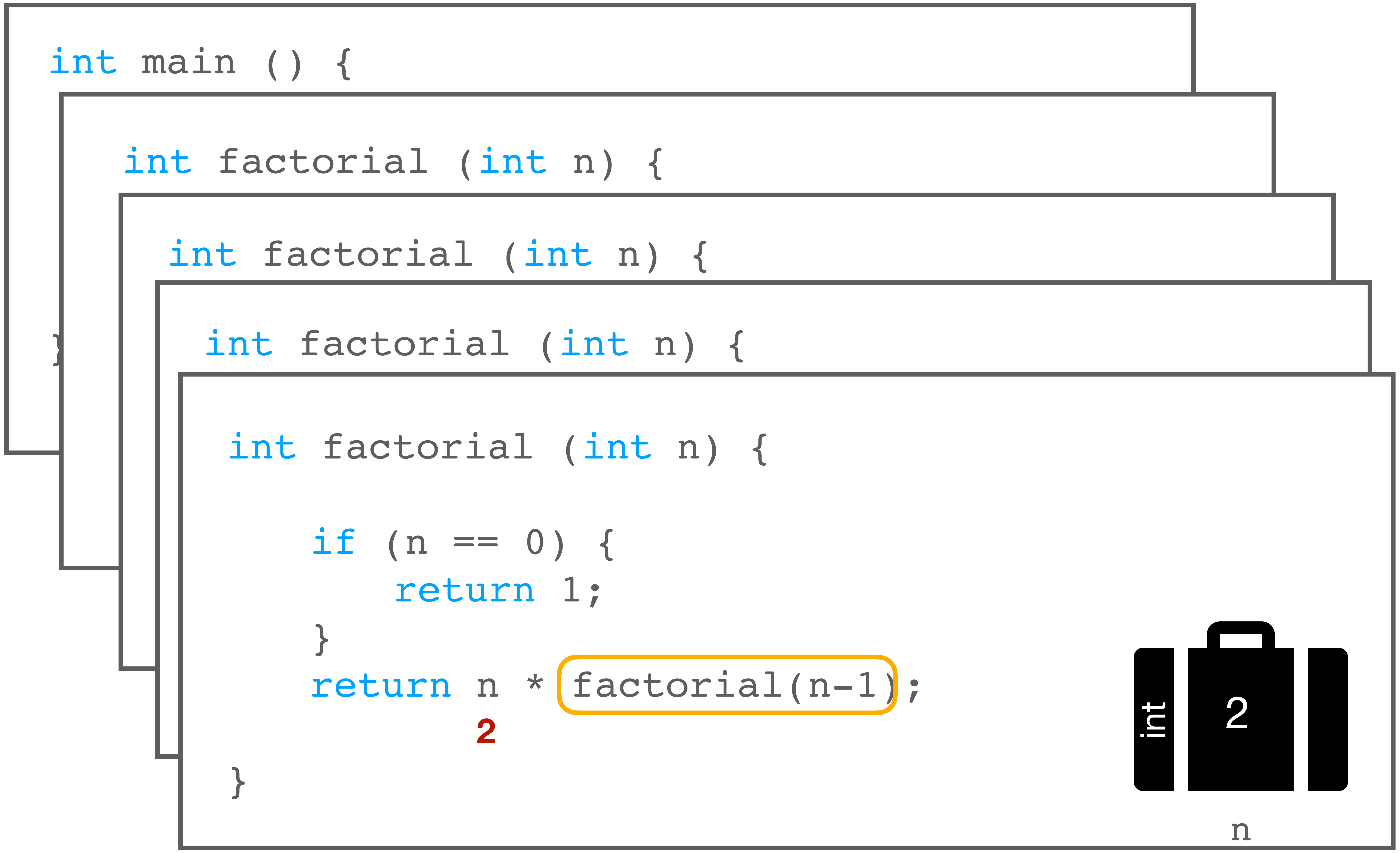
```
int main () {

}
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);

}
```

Recursion in action



```
int main () {

}
```

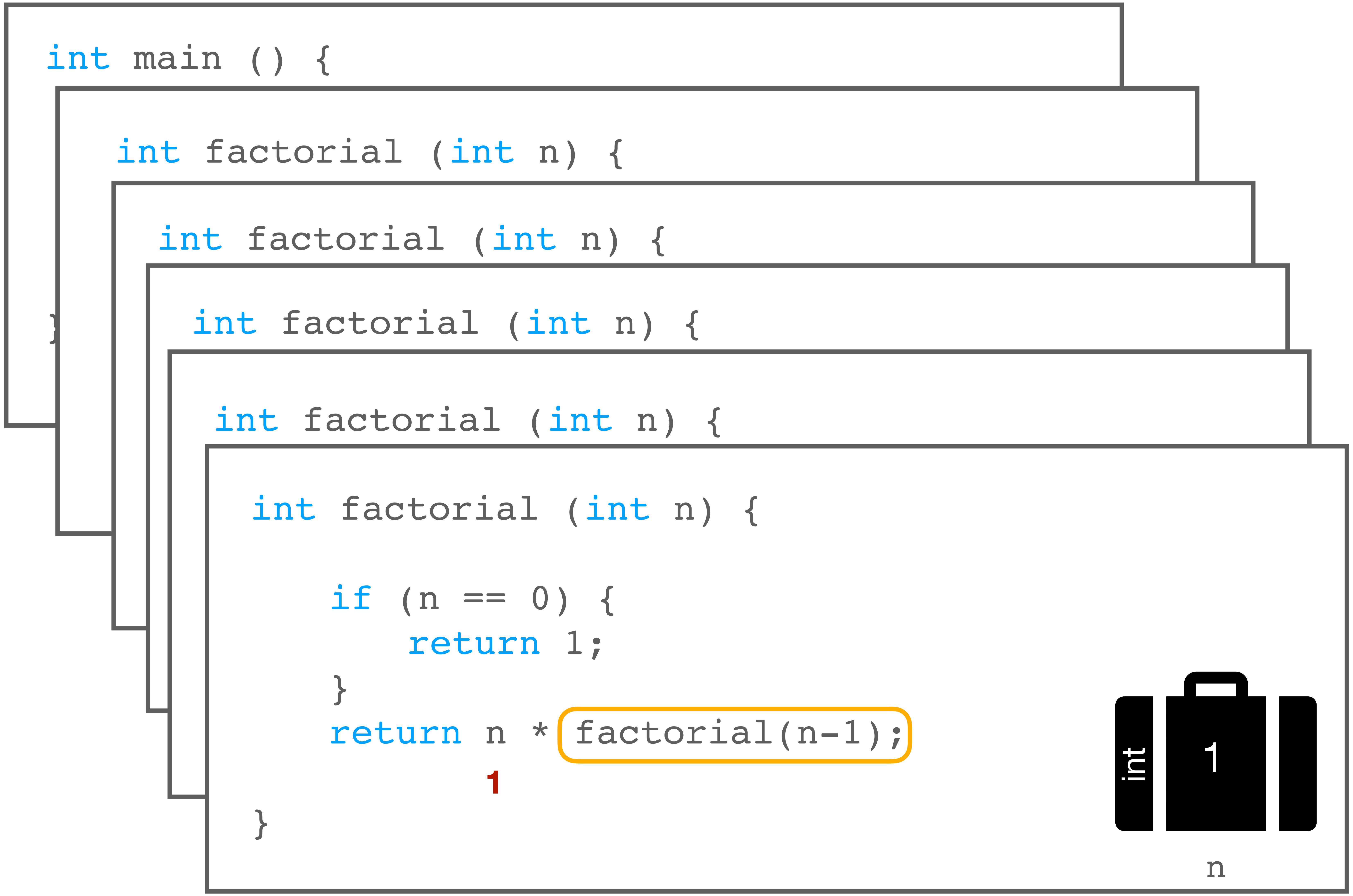
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
           2
}
```

Recursion in action



```
int main () {
}
```

```
int factorial (int n) {
```

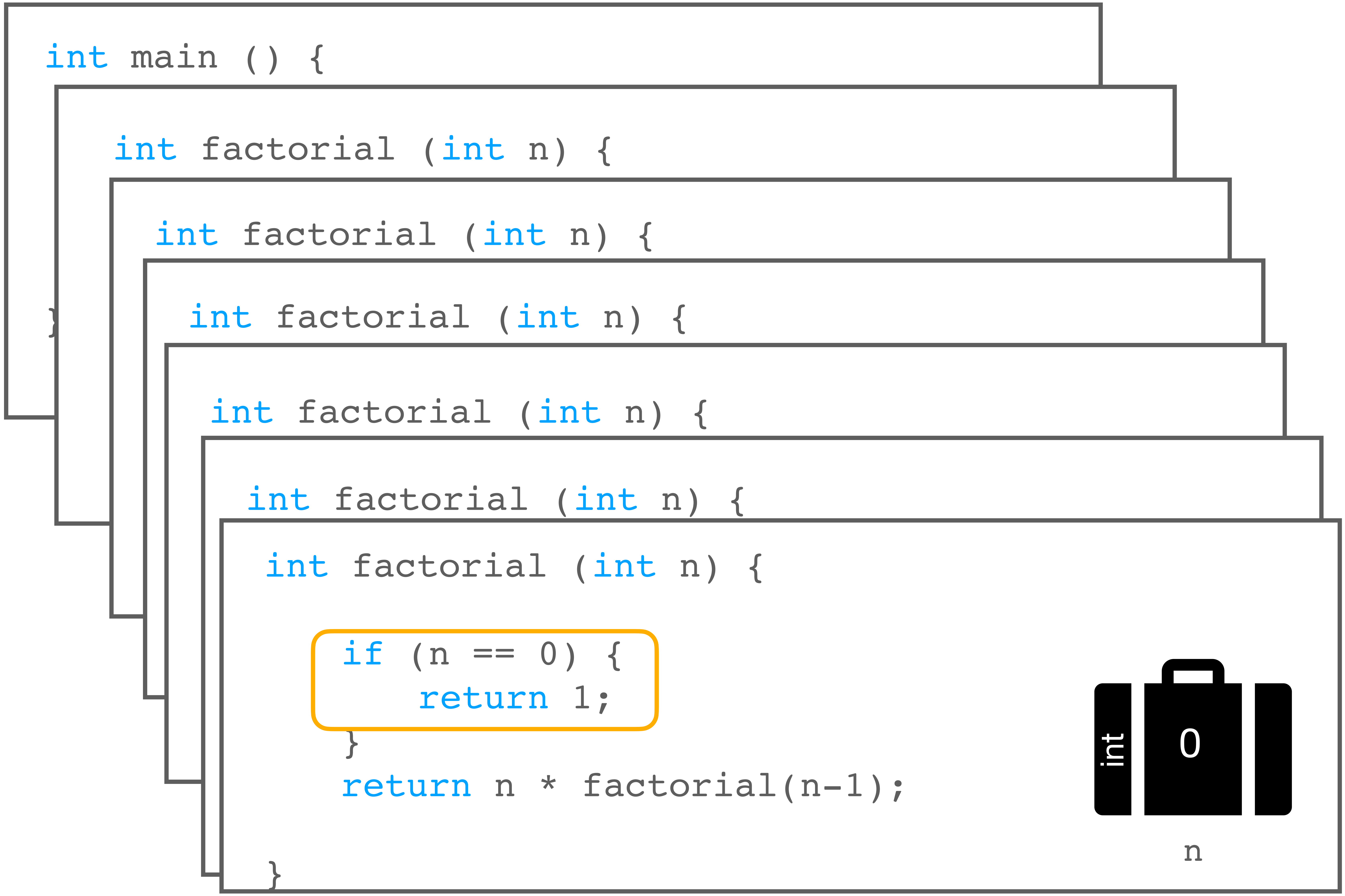
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
               1
}
```

Recursion in action



```
int main () {

}
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

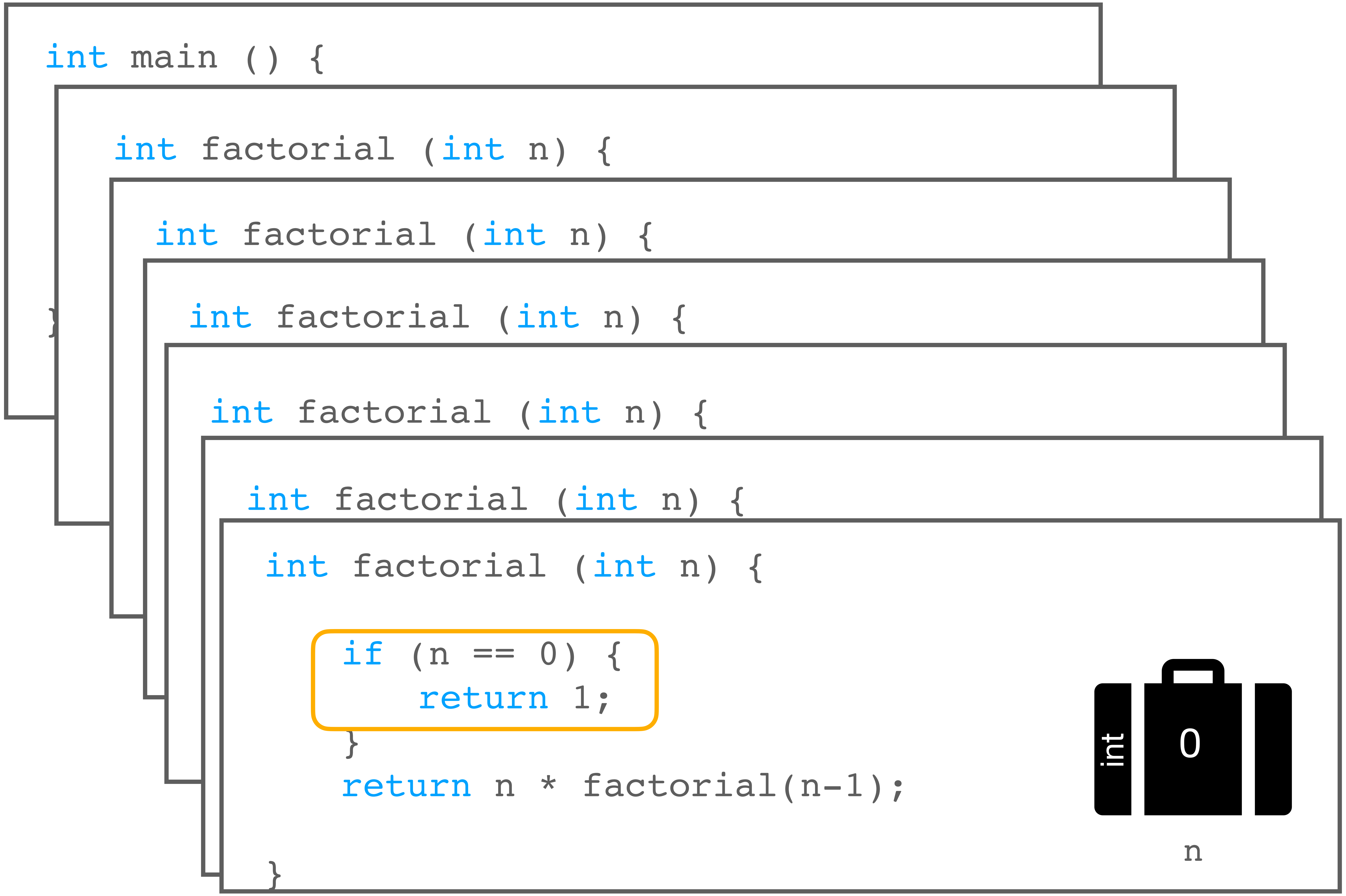
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);

}
```

Stack frame clears up in memory when the function returns or ends

Recursion in action



```
int main () {

}
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

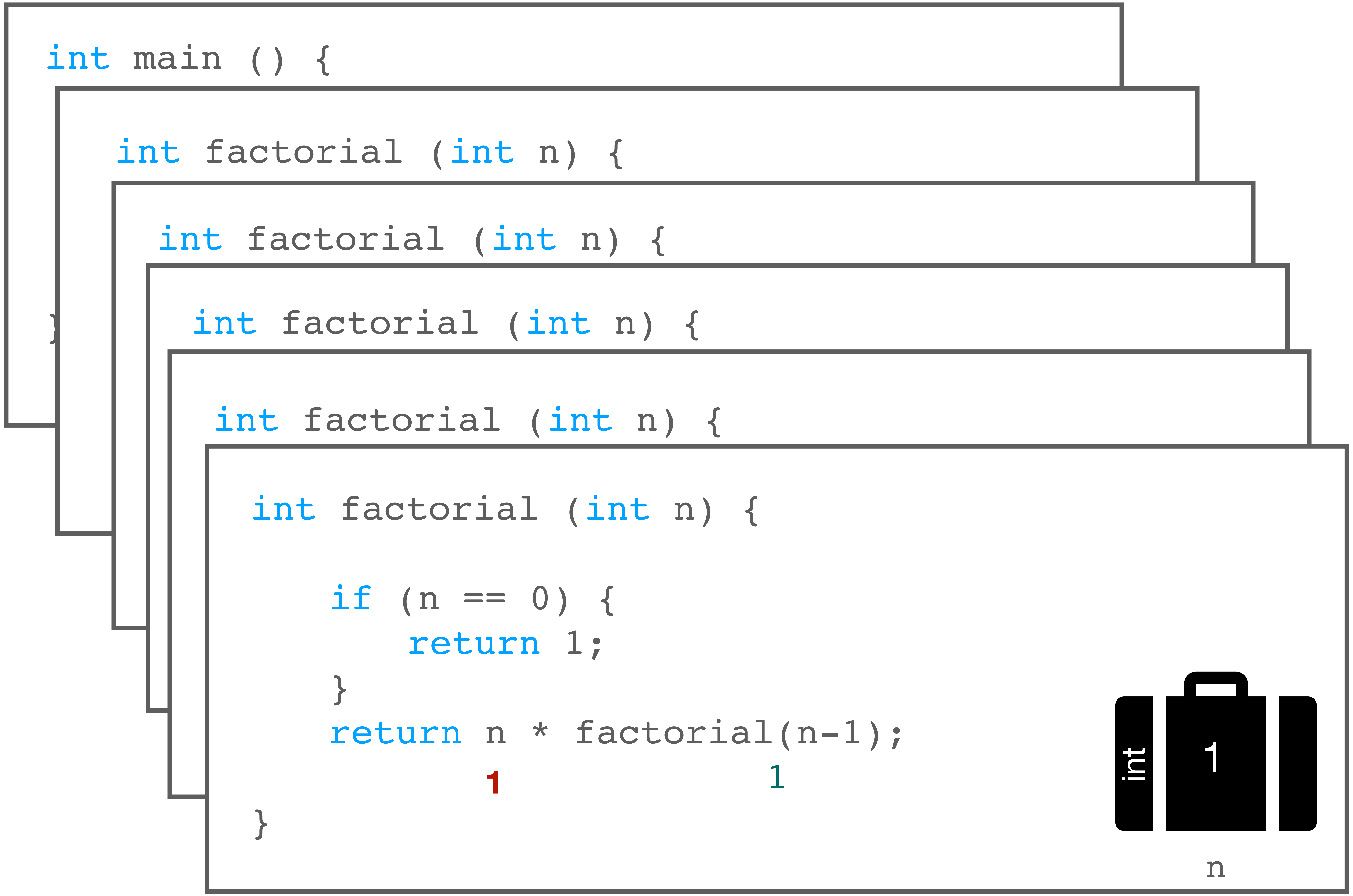
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);

}
```

Recursion in action



```
int main () {

}
```

```
int factorial (int n) {
```

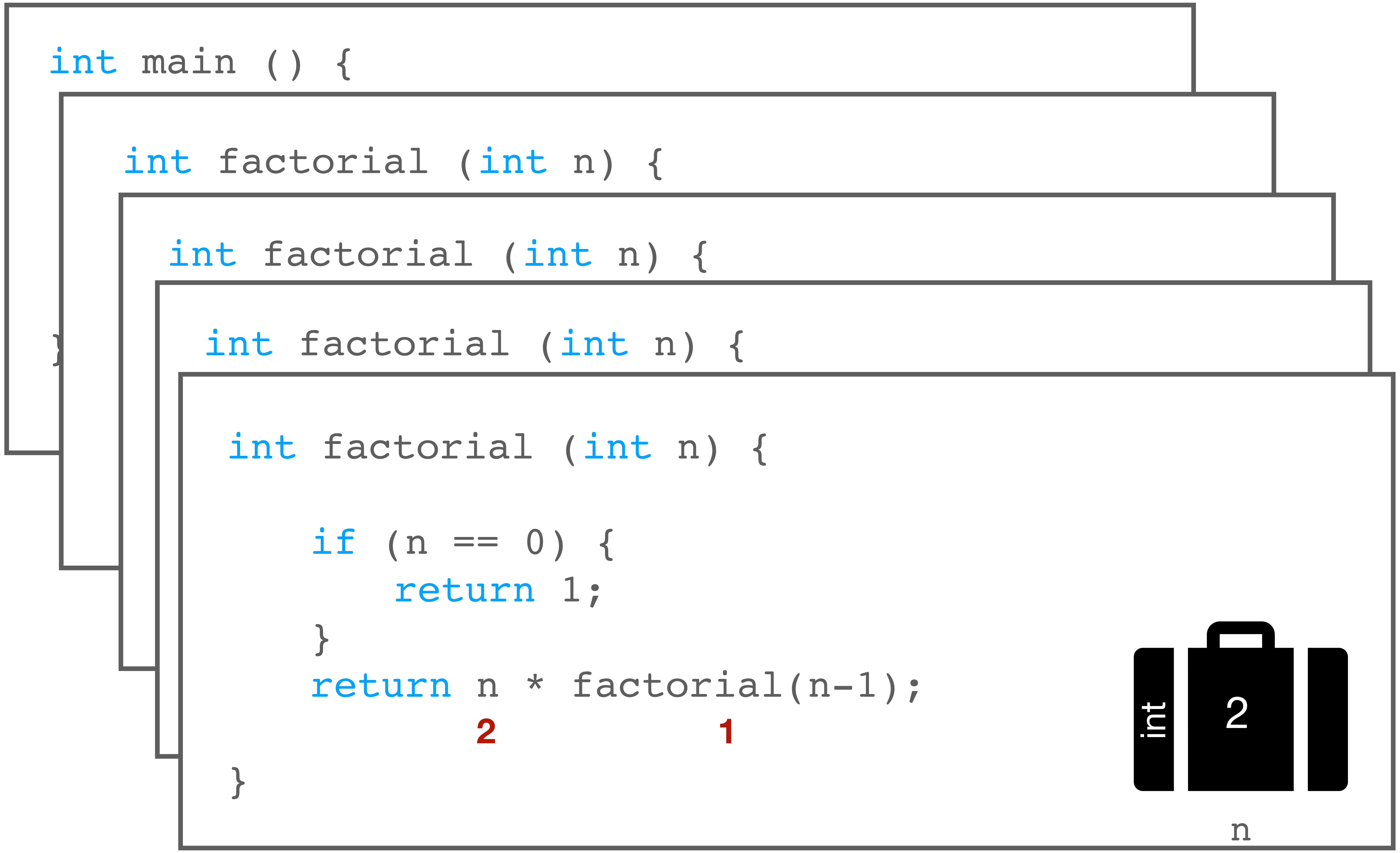
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
           1             1
}
```

Recursion in action



```
int main () {

}
```

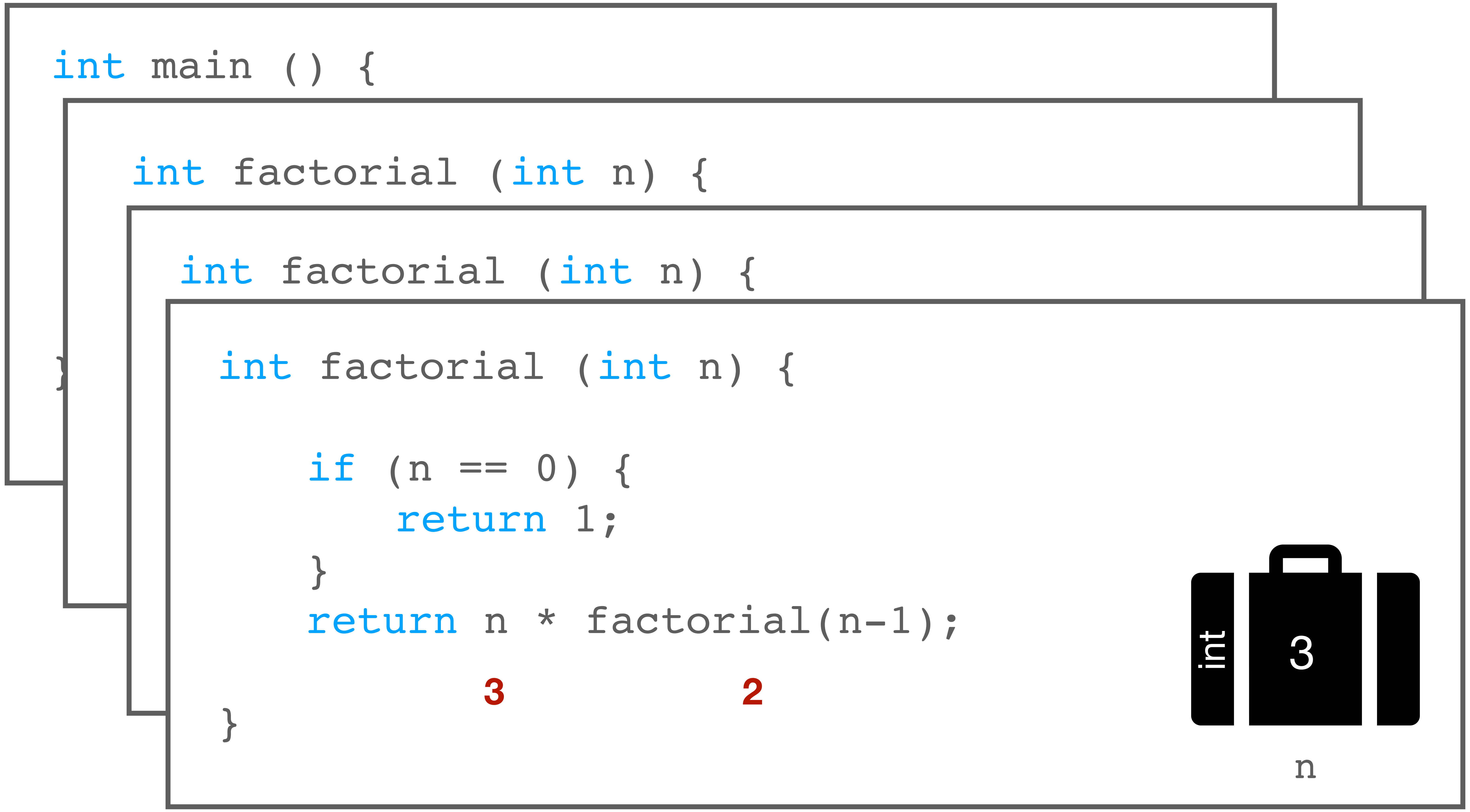
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
           2             1
}
```

Recursion in action



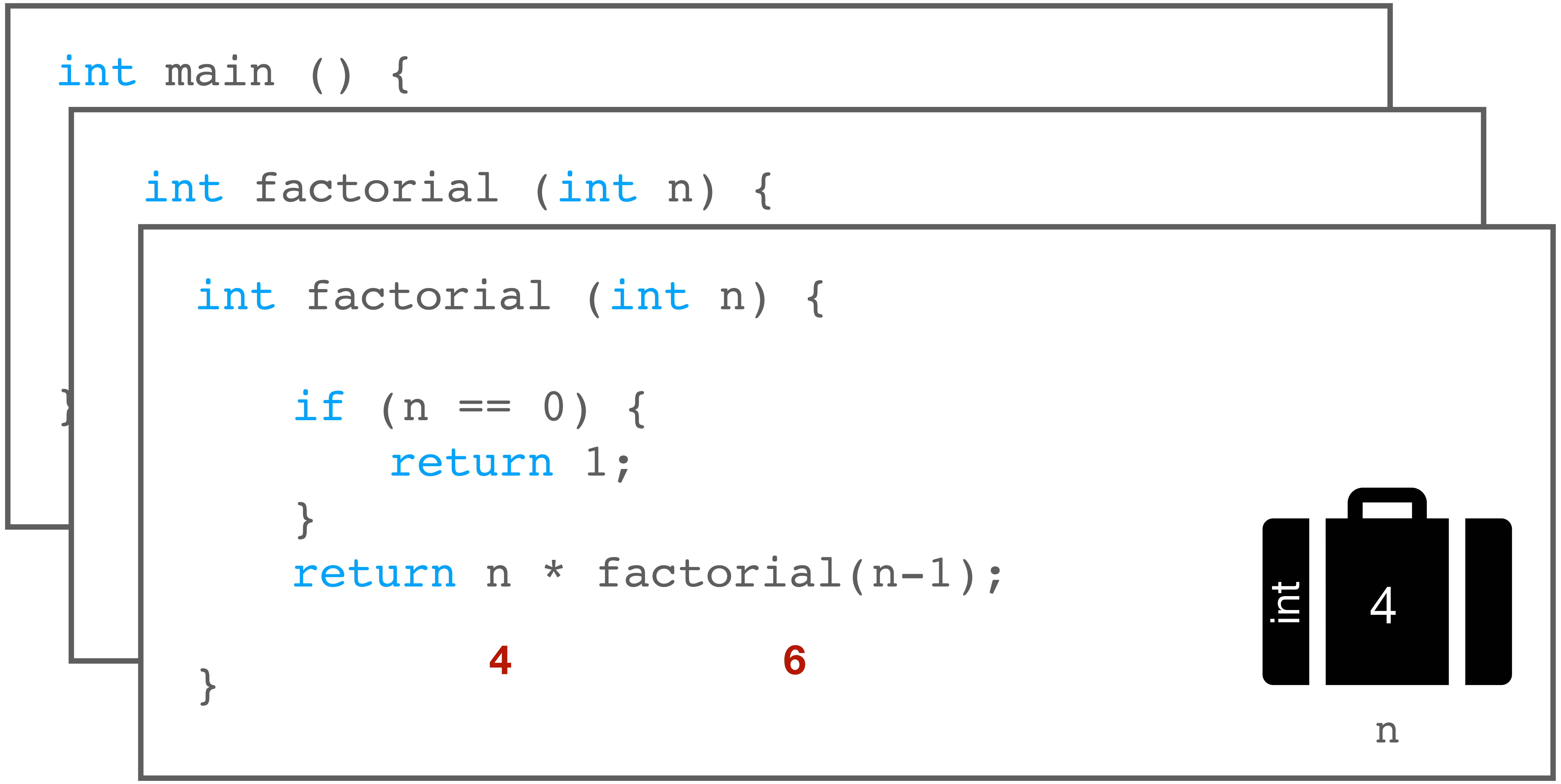
```
int main () {

}
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);

}
```

Recursion in action



```
int main () {

}
```

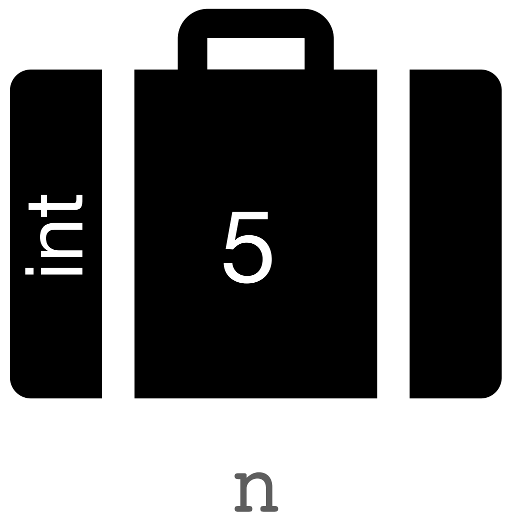
```
int factorial (int n) {
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
              4            6
}
```

Recursion in action

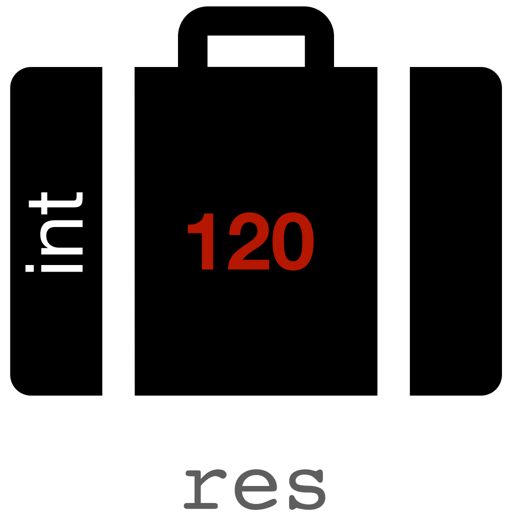
```
int main () {

}
```

```
int factorial (int n) {

    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
           5           24
}
```



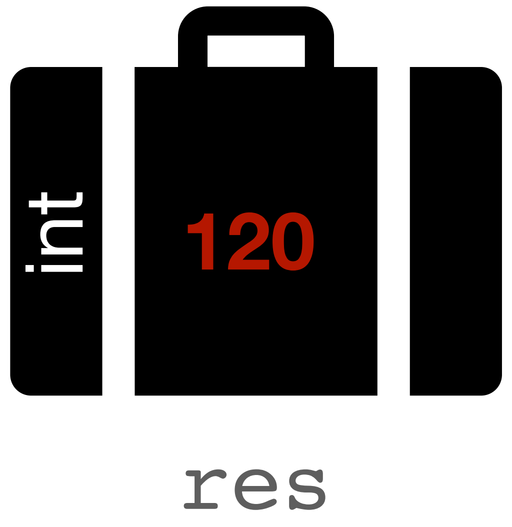
Recursion in action

```
int main () {

   int res = factorial(5);
   cout << “5! = ” << res << endl;
   return 0;

}
```



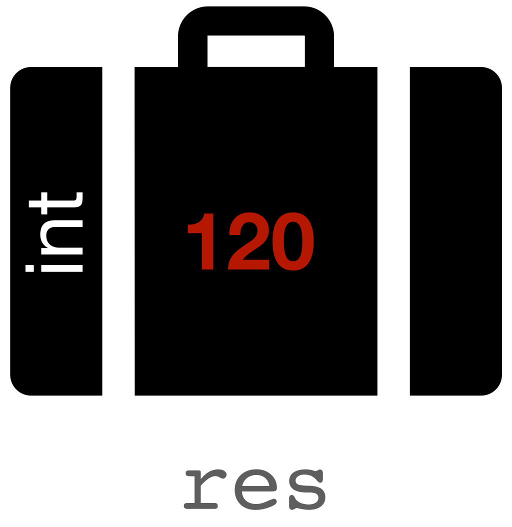
Recursion in action

```
int main () {

    int res = factorial(5);
    cout << “5! = ” << res << endl;
    return 0;

}
```



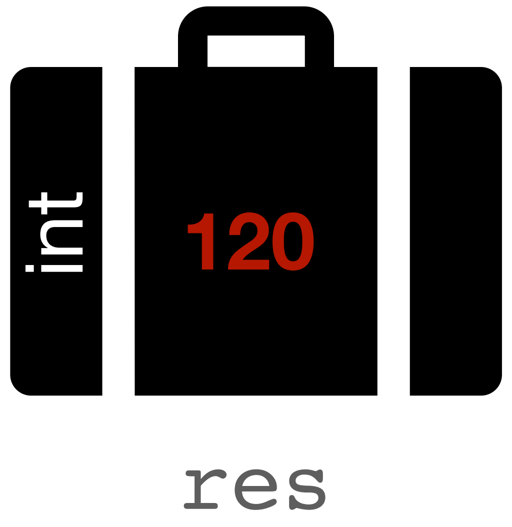
Recursion in action

```
int main () {

    int res = factorial(5);
    cout << “5! = ” << res << endl;
    return 0;

}
```



Recursion in action

```
int main () {

    int res = factorial(5);
    cout << “5! = ” << res << endl;
    return 0;

}
```



Thinking recursively

```
If (the problem is simple) {
   Directly solve the problem.
   return the solution

} else {
   1. Split the problem into one or more
   smaller problems with the same structure.
   2. Solve each of those smaller problems.
   3. Combine the results to get the overall solution.

   return the solution
}
```

The base case

The recursive cases

Another example

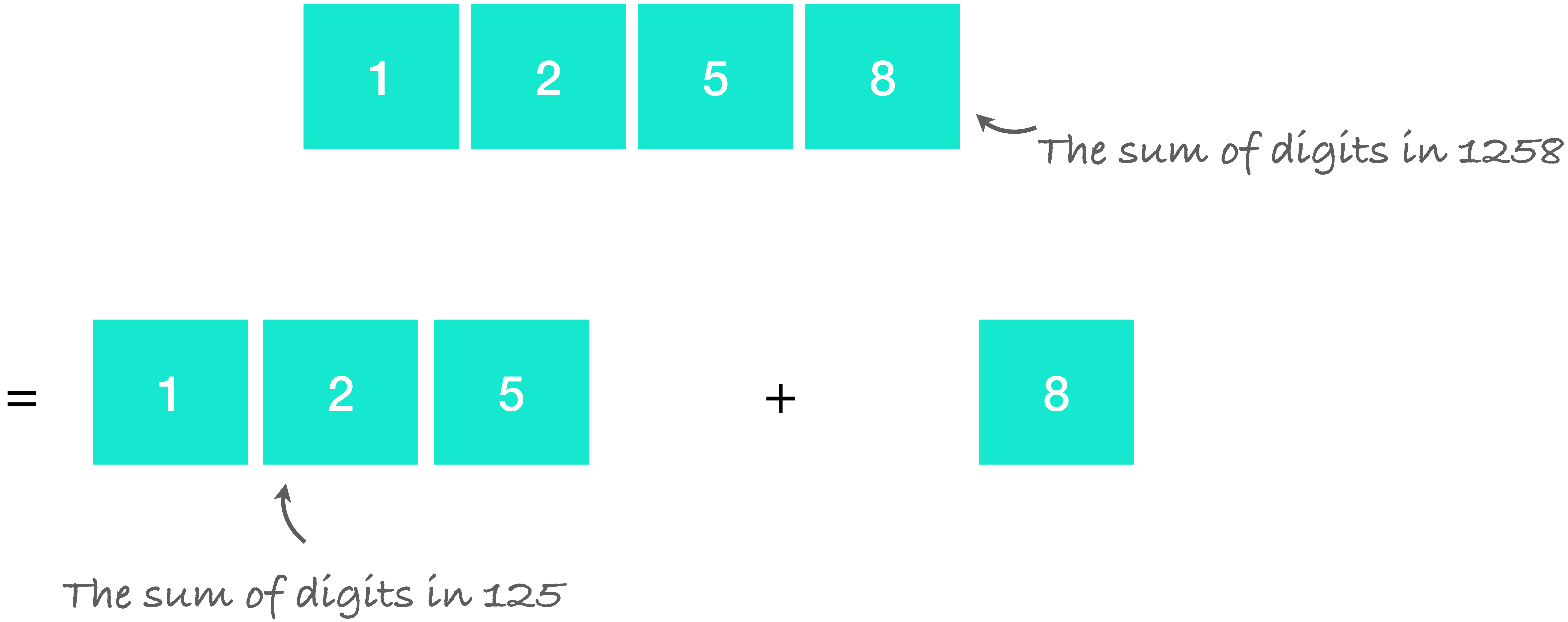
`Sum of digits`

- Given `n = 1258`, returns `16` (i.e., 1+2+5+8)
- We have solved it using iteration last week
- Let us see how to solve it using recursion

1	2	5	8

Another example

Sum of digits



1 2 5 8

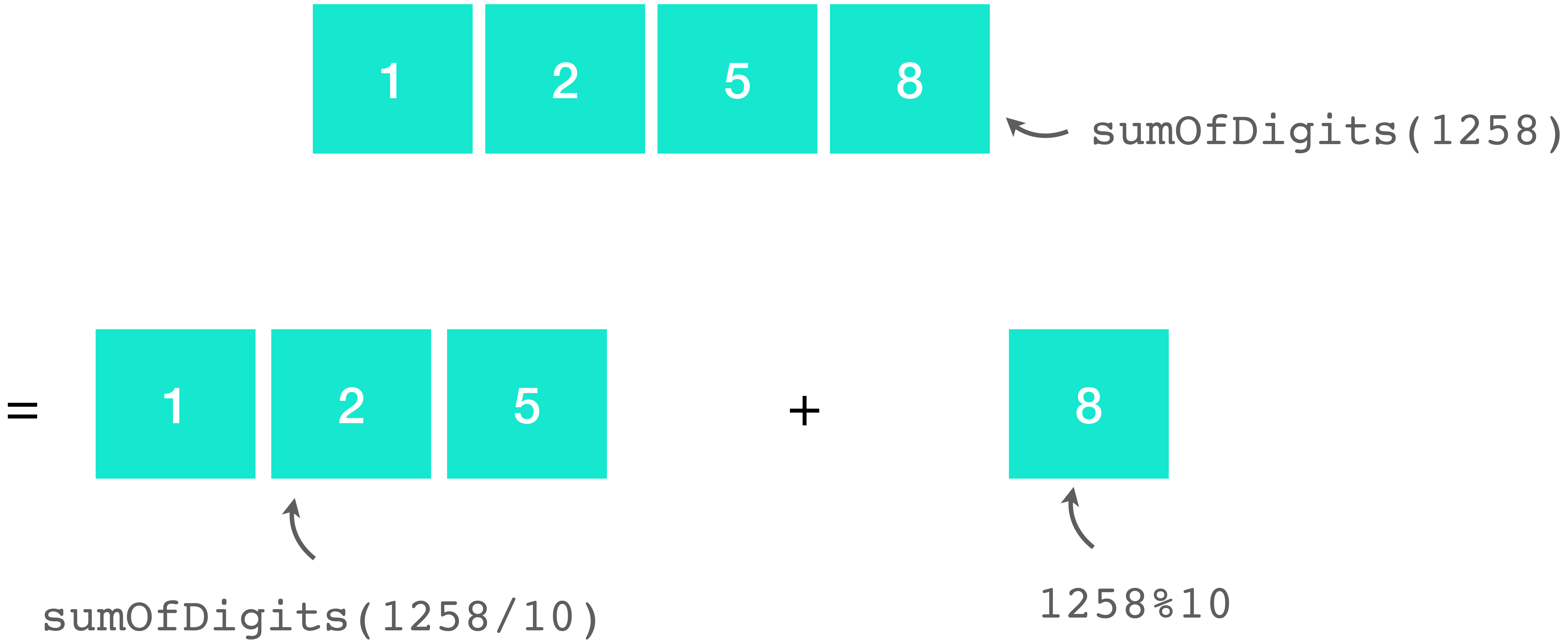
The sum of digits in 1258

= 1 2 5 + 8

The sum of digits in 125

Another example

Sum of digits



sumOfDigits(1258)

sumOfDigits(1258/10)

1258%10

One more example…

Digital root

- The digital root of an integer is the number you get by repeatedly summing the digits of a number until you're down to a single digit
- What is digital root of 5?

digitRoot(n) = digitRoot(sumofDigits(n))

- What is digital root of 27?
- What is digital root of 1258?
  - 1+2+5+8 = 16
  - 1+6 = 7
  - Return 7

In-class problem

1. **Fork the following project**
2. **Finish the implementations of three functions (*i.e., cpp file*)**
   a. `sumOfDigitsRec – recursive version`
   b. `digitalRootRec – recursive version for digital root`
   c. `digitalRoot – iterative version for digital root`
3. **Share the link of your finished project to this Google sheet**