Functions in C++

Outline for Today

- ***Functions in C++***
  - How C++ organizes code.
- ***Some Simple Functions***
  - Getting comfortable with the language.
- ***Intro to Recursion***
  - A new perspective on problem-solving.

Functions in C++

```
/*              C++ Version              */
double areaOfCircle(double r) {
    return M_PI * r * r;
}

int maxOf(int first, int second) {
    if (first > second) {
        return first;
    }
    return second;
}

void printNumber(int n) {
    cout << "I like " << n << endl;
}
```

```
"""             Python Version             """
def areaOfCircle(r):
    return math.pi * r * r


def maxOf(first, second):
    if first > second:
        return first
    return second



def printNumber(n):
    print("I like " + str(n))
```

```
/*              Java Version              */
private double areaOfCircle(double r) {
    return M_PI * r * r;
}

private int maxOf(int first, int second) {
    if (first > second) {
        return first;
    }
    return second;
}

private void printNumber(int n) {
    System.out.println("I like " + n);
}
```

```
//       JavaScript Version
function areaOfCircle(r) {
   return Math.PI * r * r;
}

function maxOf(first, second) {
   if (first > second) {
       return first;
   }
   return second;
}

function printNumber(n) {
   console.log("I like " + n);
}
```

```cpp
/*                C++ Version                */
double areaOfCircle(double r) {
    return M_PI * r * r;
}

int maxOf(int first, int second) {
    if (first > second) {
        return first;
    }
    return second;
}

void printNumber(int n) {
    cout << "I like " << n << endl;
}
```

```python
"""          Python Version          """
def areaOfCircle(r):
    return math.pi * r * r


def maxOf(first, second):
    if first > second:
        return first
    return second



def printNumber(n):
    print("I like " + str(n))
```

```java
/*                Java Ve
private double areaOf
    return M_PI * r *
}

private int maxOf(int
    if (first > secon
        return first;
    }
    return second;
}

private void printNum
    System.out.println("I like " + n);
}
```

```javascript
//          JavaScript Version

function printNumber(n) {
    console.log("I like " + n);
}
```

Functions in C++ work like functions in Python/JavaScript or like methods in Java. They (optionally) take in parameters, perform a calculation, then (optionally) return a value.

```
/*                 C++ Version                 */
double areaOfCircle(double r) {
    return M_PI * r * r;
}

int maxOf(int first, int second) {
    if (first > second) {
        return first;
    }
    return second;
}

void printNumber(int n) {
    cout << "I like " << n << endl;
}
```

```
"""           Python Version            """
def areaOfCircle(r):
    return math.pi * r * r


def maxOf(first, second):
    if first > second:
        return first
    return second



def printNumber(n):
    print("I like " + str(n))
```

```
/*                 Java Ve
private double areaOf
    return M_PI * r *
}

private int maxOf(int
    if (first > secon
        return first;
    }
    return second;
}

private void printNum
    System.out.println("I like " + n);
}
```

```
    console.log("I like " + n);
}
```

All variables in C++ need a type. Some common types include **int** (integer), **double** (real number), and **bool** (true/false),

```
/*              C++ Version             */
double areaOfCircle(double r) {
    return M_PI * r * r;
}

int maxOf(int first, int second) {
    if (first > second) {
        return first;
    }
    return second;
}

void printNumber(int n) {
    cout << "I like " << n << endl;
}
```

```
"""          Python Version          """
def areaOfCircle(r):
    return math.pi * r * r


def maxOf(first, second):
    if first > second:
        return first
    return second



def printNumber(n):
    print("I like " + str(n))
```

```
/*              Java Ve
private double areaOf
    return M_PI * r *
}

private int maxOf(int
    if (first > secon
        return first;
    }
    return second;
}

private void printNum
    System.out.println("I like " + n);
}
```

```
    console.log("I like " + n);
}
```

You define a function by writing

```
return-type fn-name(args) {
    // … code goes here …
}
```

```cpp
/*                C++ Version               */
double areaOfCircle(double r) {
    return M_PI * r * r;
}

int maxOf(int first, int second) {
    if (first > second) {
        return first;
    }
    return second;
}

void printNumber(int n) {
    cout << "I like " << n << endl;
}
```

```python
"""          Python Version           """
def areaOfCircle(r):
    return math.pi * r * r


def maxOf(first, second):
    if first > second:
        return first
    return second


def printNumber(n):
    print("I like " + str(n))
```

```java
/*                Java Ve
private double areaOf
    return M_PI * r *
}

private int maxOf(int
    if (first > secon
        return first;
    }
    return second;
}

private void printNum
    System.out.println("I like " + n);
}
```

```javascript
//          JavaScript Version

    console.log("I like " + n);
}
```

If a function does not return a value, its return type should be the cool-but-scary-sounding **void**.

The `main` Function

- A C++ program begins execution in a function called `main` with the following signature:

```
int main() {
    /* … code to execute … */
    return 0;
}
```

- By convention, `main` should return `0` unless the program encounters an error.
- Curious why `main` returns an `int`? Come chat with me after class today!

A Simple C++ Program

Hip hip, hooray!

Hip hip, hooray!
Hip hip, hooray!
Hip hip, hooray!

What Went Wrong?

One-Pass Compilation

- When you compile a C++ program, the compiler reads your code from top to bottom.
- If you call a function that you haven’t yet written, the compiler will get Very Upset and will say mean things to you.
- You will encounter this issue. What should you do?

Dramatic Reenactment

Option 1: Reorder Your Functions

Option 2: Use Forward Declarations

Forward Declarations

- A ***forward declaration*** is a statement that tells the C++ compiler about an upcoming function.
  - The textbook calls these ***function prototypes***. It's different names for the same thing.
- Forward declarations look like this:

  return-type function-name(***parameters***);

- Essentially, start off like you're defining the function as usual, but put a semicolon instead of the function body.
- Once the compiler has seen a forward declaration, you can go and call that function as normal.

Some More Functions

Summing Up Digits

- Ever seen that test for divisibility by three?

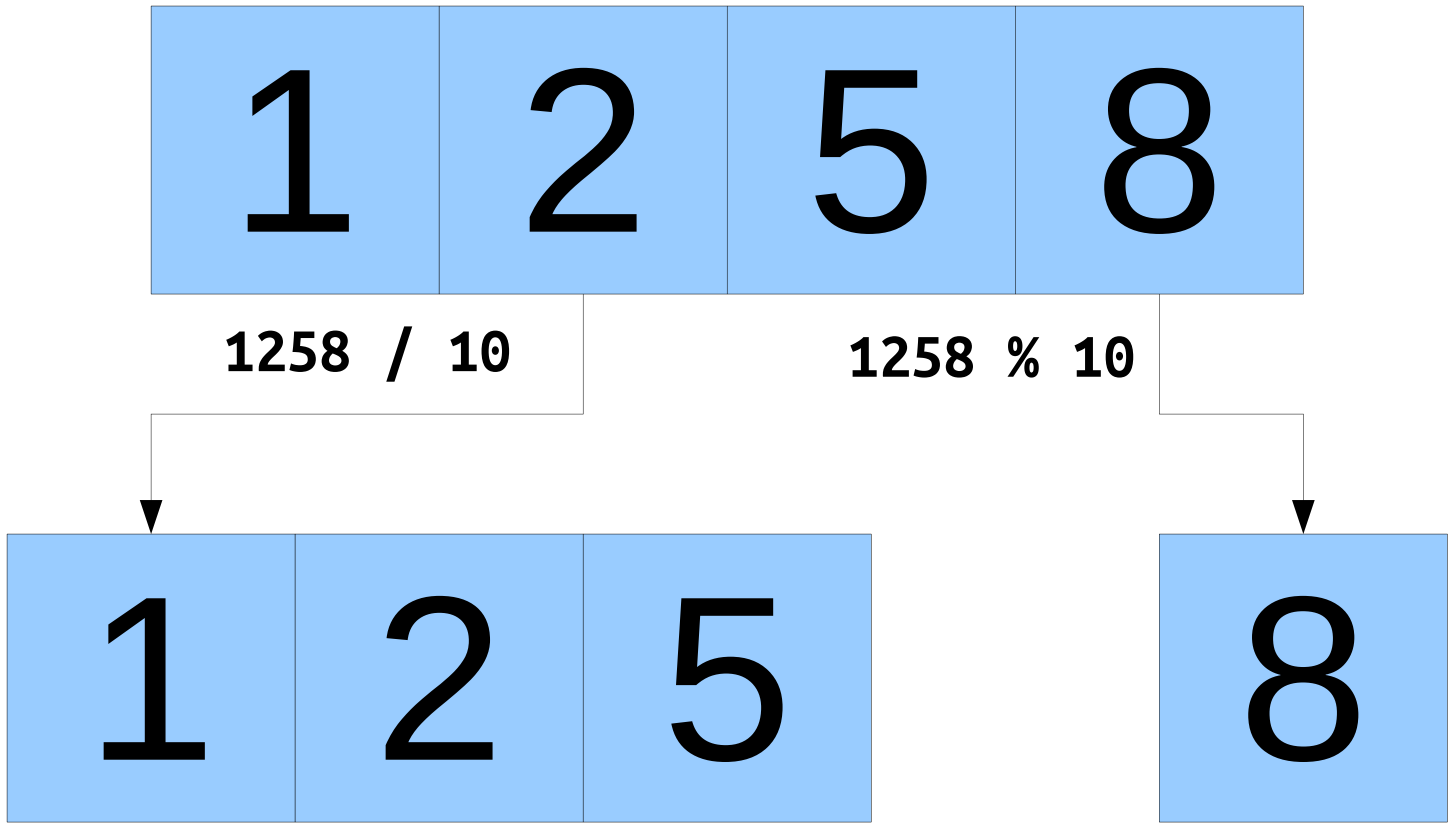
  Add the digits of the number; if the sum is divisible by three, the original number is divisible by three (and vice-versa).

- Let's write a function

  ```
  int sumOfDigitsOf(int n)
  ```

  that takes in a number and returns the sum of its digits.

Working One Digit at a Time



Dividing two integers in C++ ***always*** produces an integer by dropping any decimal value. Check the textbook for how to override this behavior.

Functions in Action

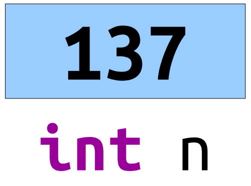
```
int main() {
   int n = getInteger("Enter an integer: ");
   int digitSum = sumOfDigitsOf(n);
   cout << n << " sums to " << digitSum << endl;

   return 0;
}
```

Functions in Action

```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```

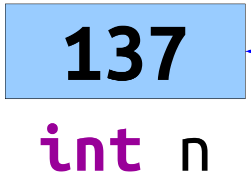
Functions in Action

```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```



Functions in Action

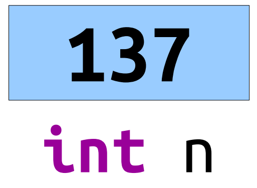
```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```



The variable n actually is an honest-to-goodness integer, not a pointer to an integer that lives somewhere else. In C++, all variables stand for actual objects unless stated otherwise. (More on that later.)

Functions in Action

```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```



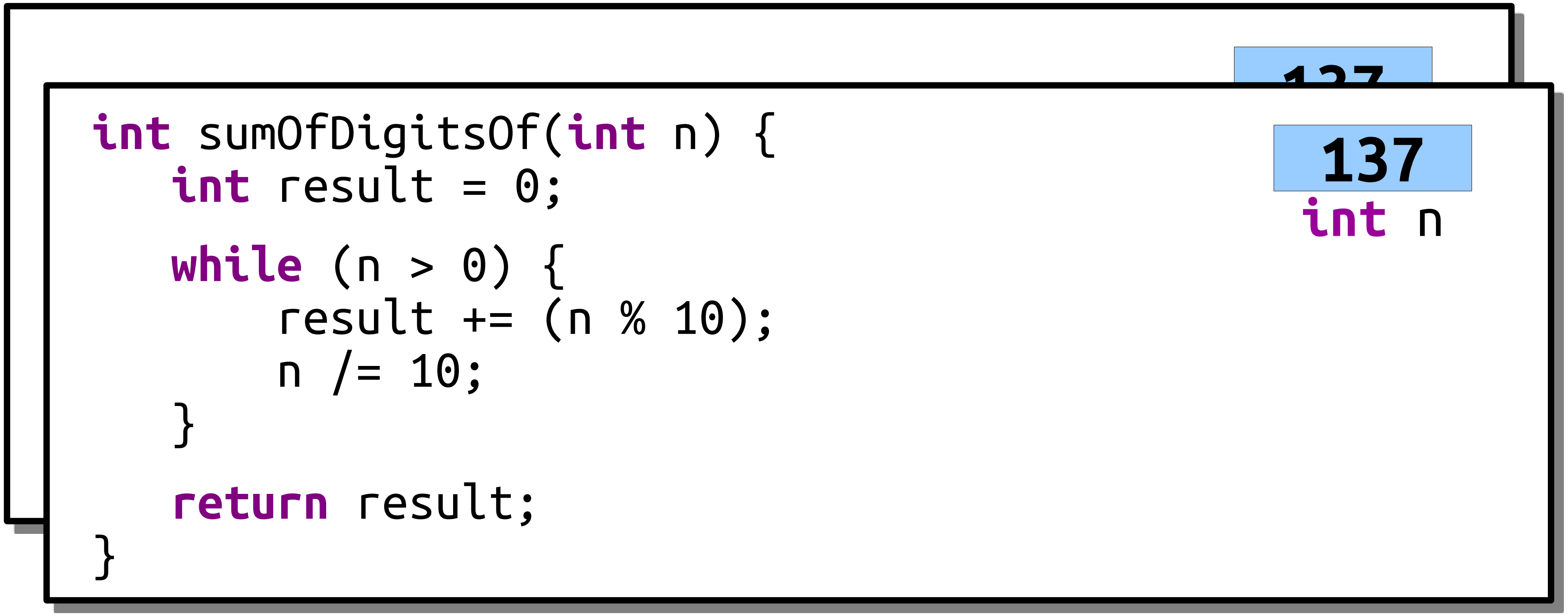
Functions in Action

```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```

137

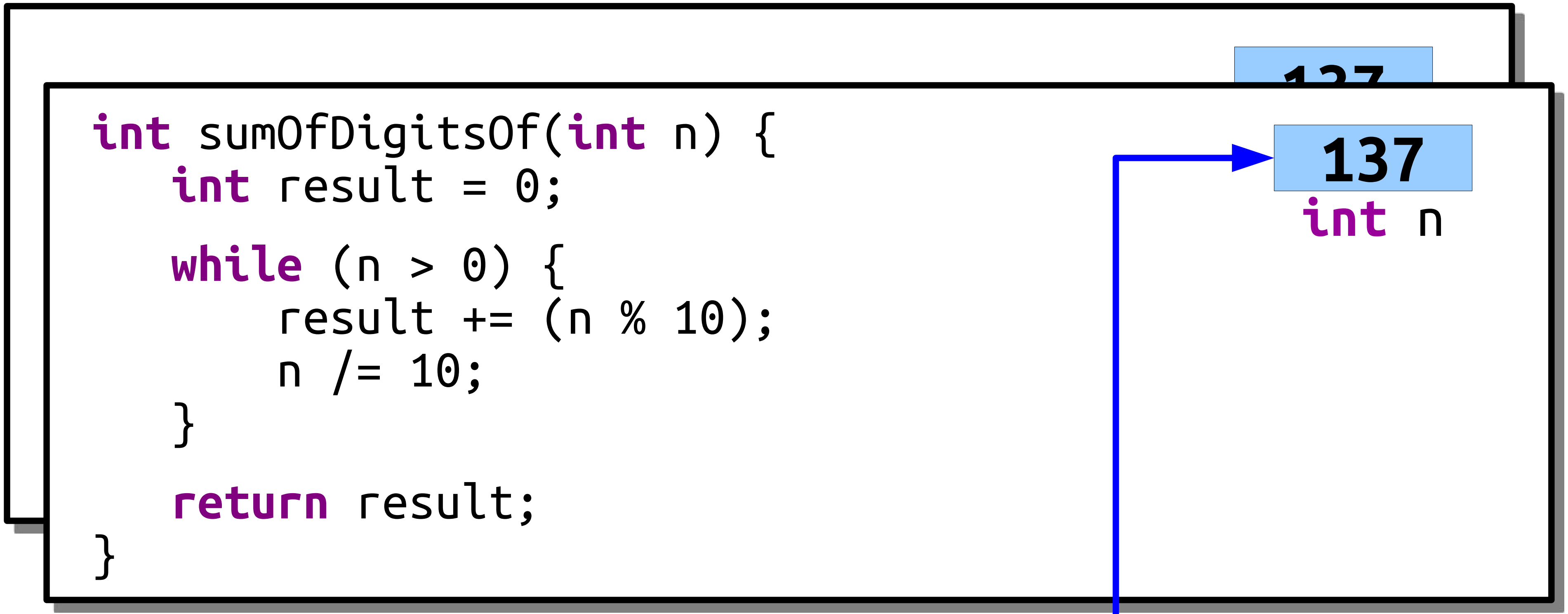
int n

Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

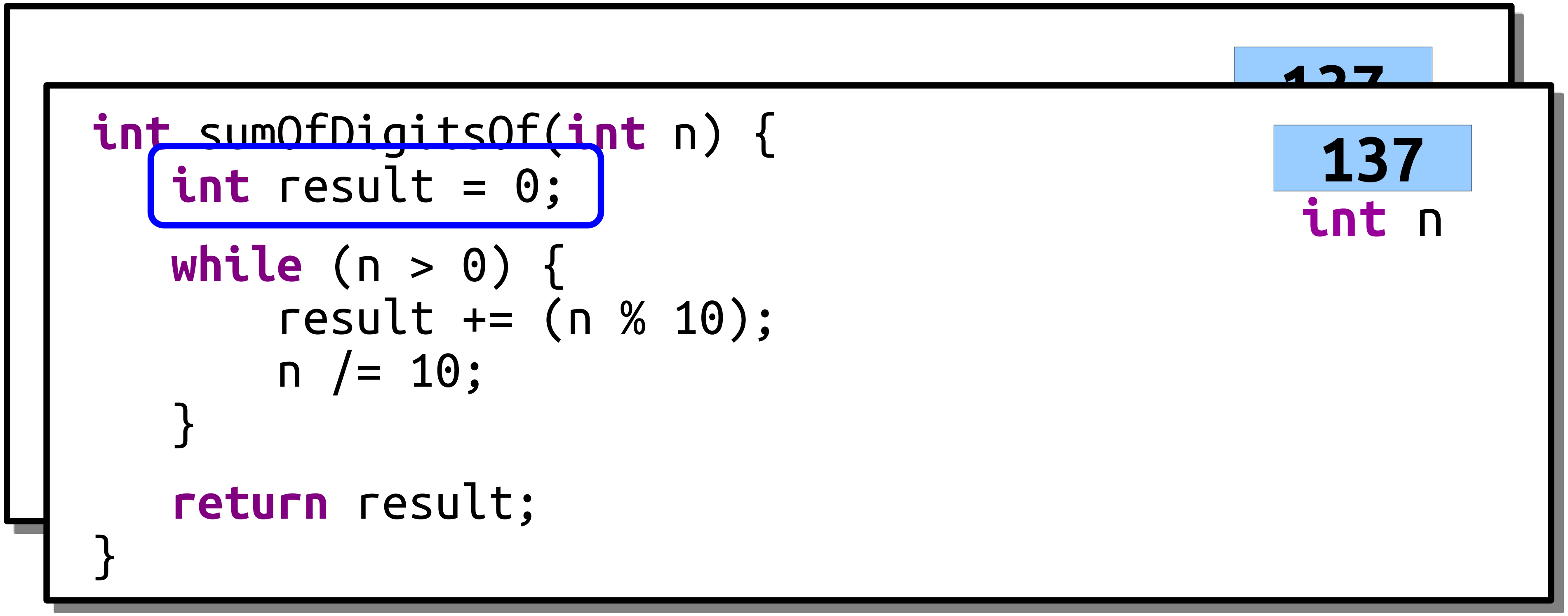
Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

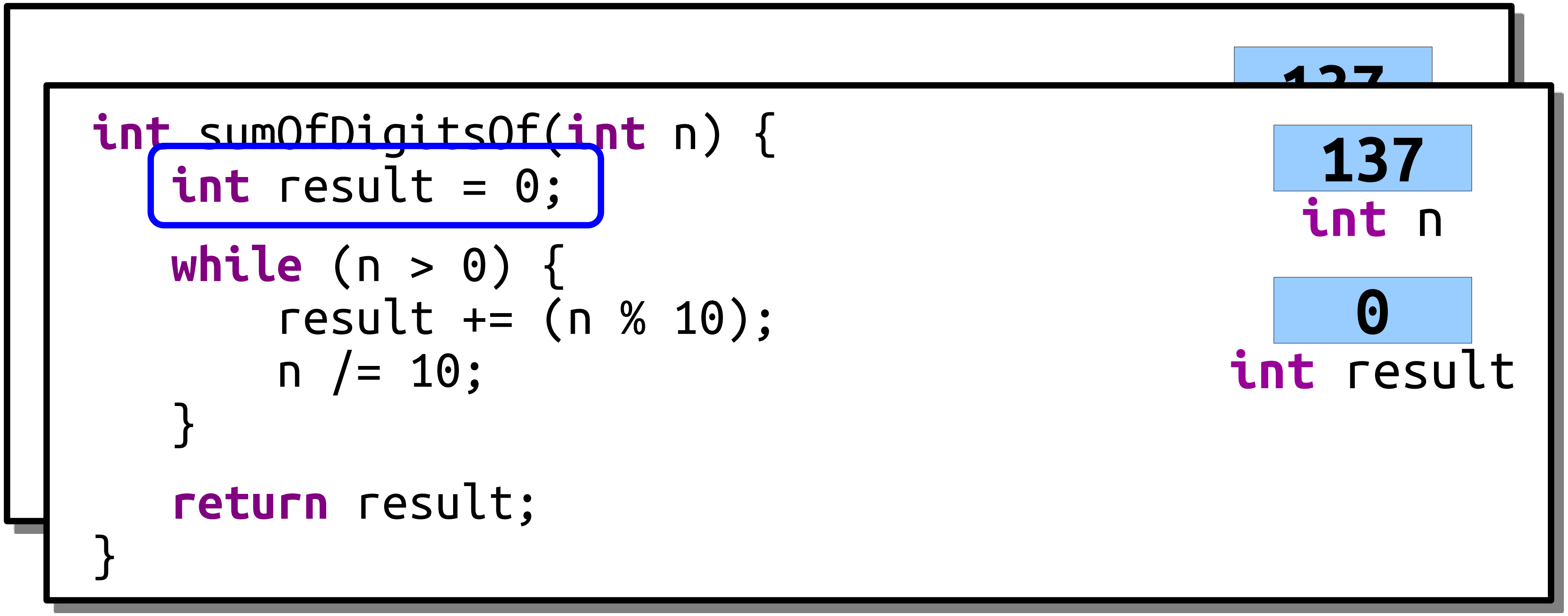
When we call `sumOfDigitsOf`, we get our own variable named `n`. It’s separate from the variable `n` in `main()`, and changes to this variable `n` don’t reflect back in `main`.

Functions in Action



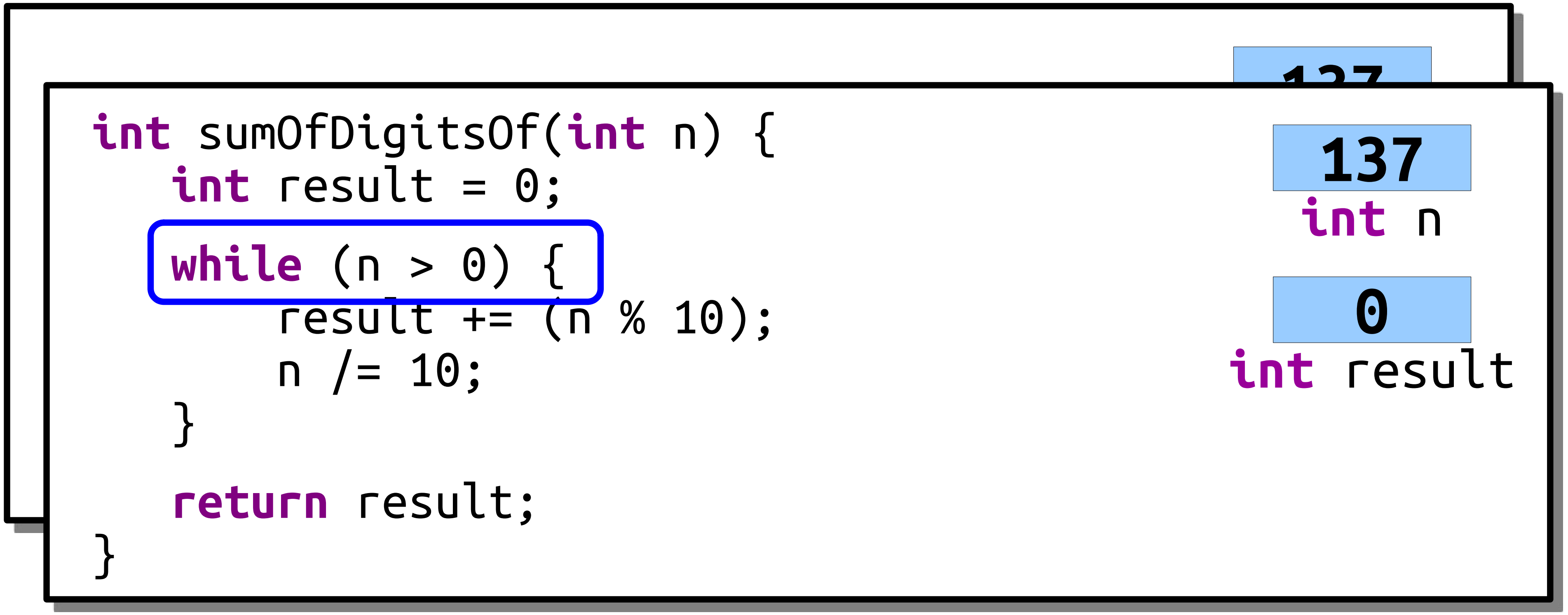
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



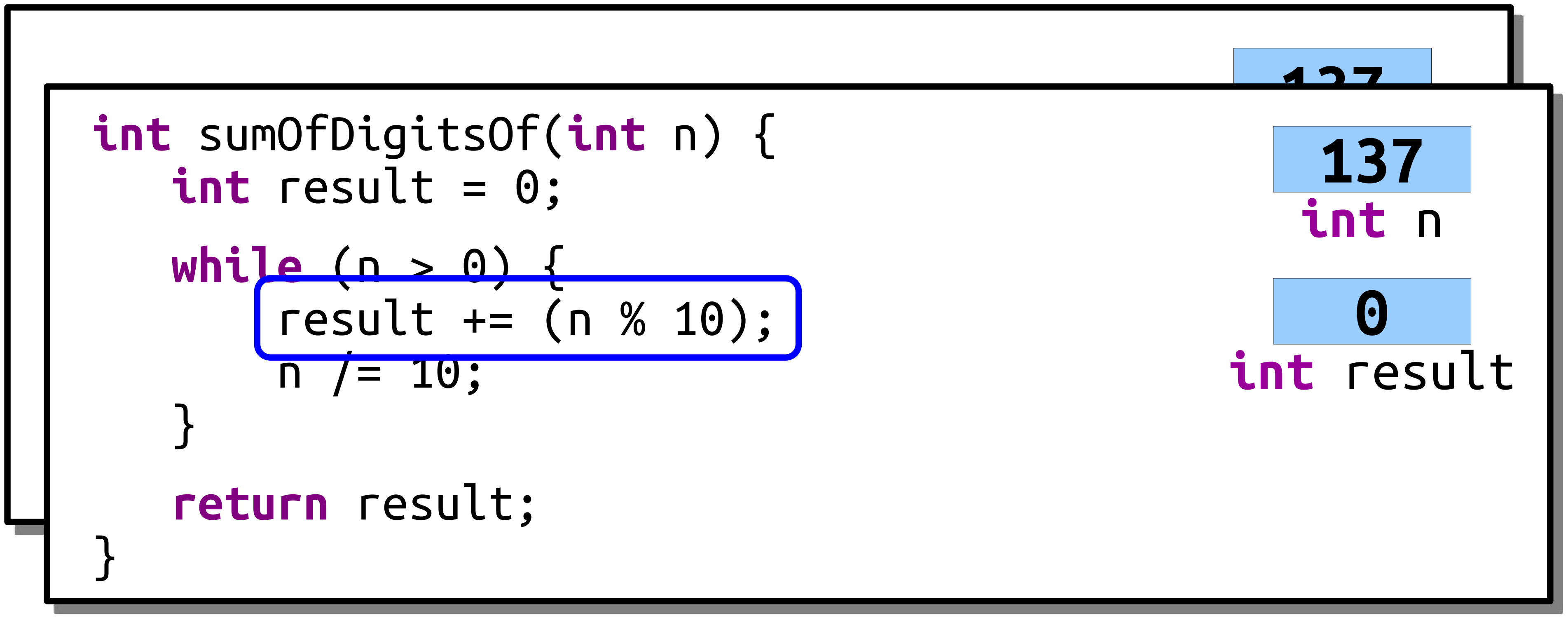
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



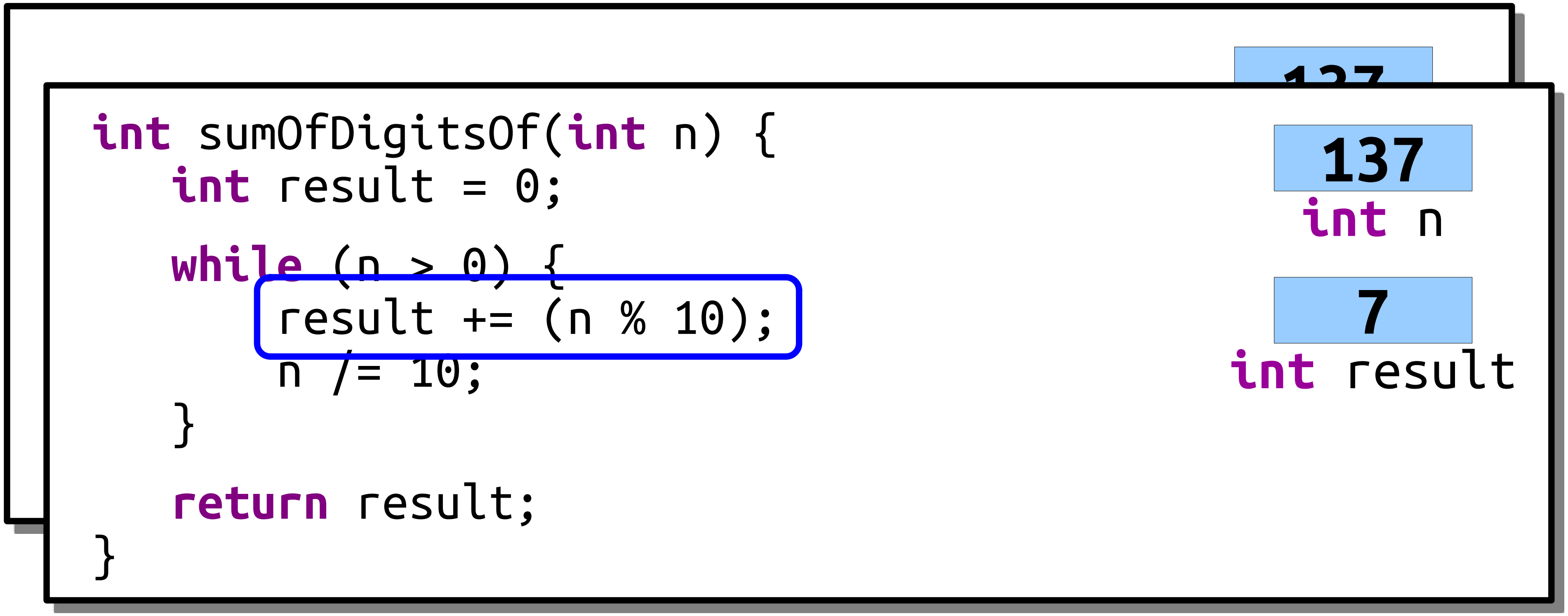
```
int sumOfDigitsOf(int n) {
    int result = 0;
    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }
    return result;
}
```

Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

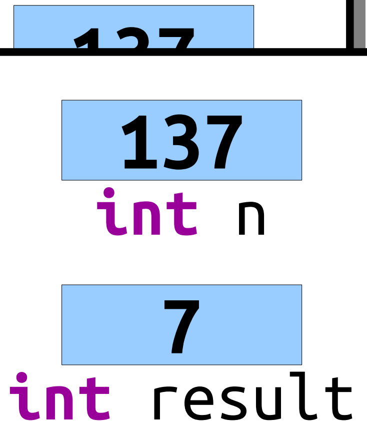
Functions in Action



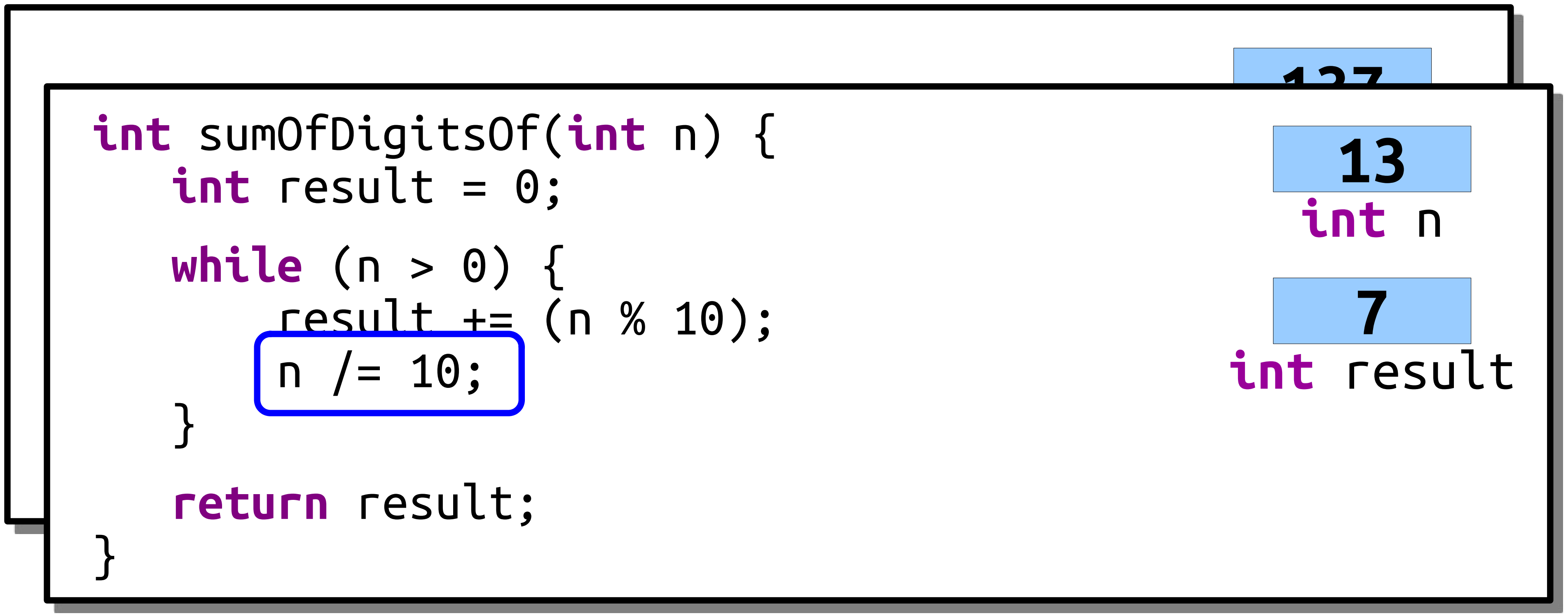
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action

```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

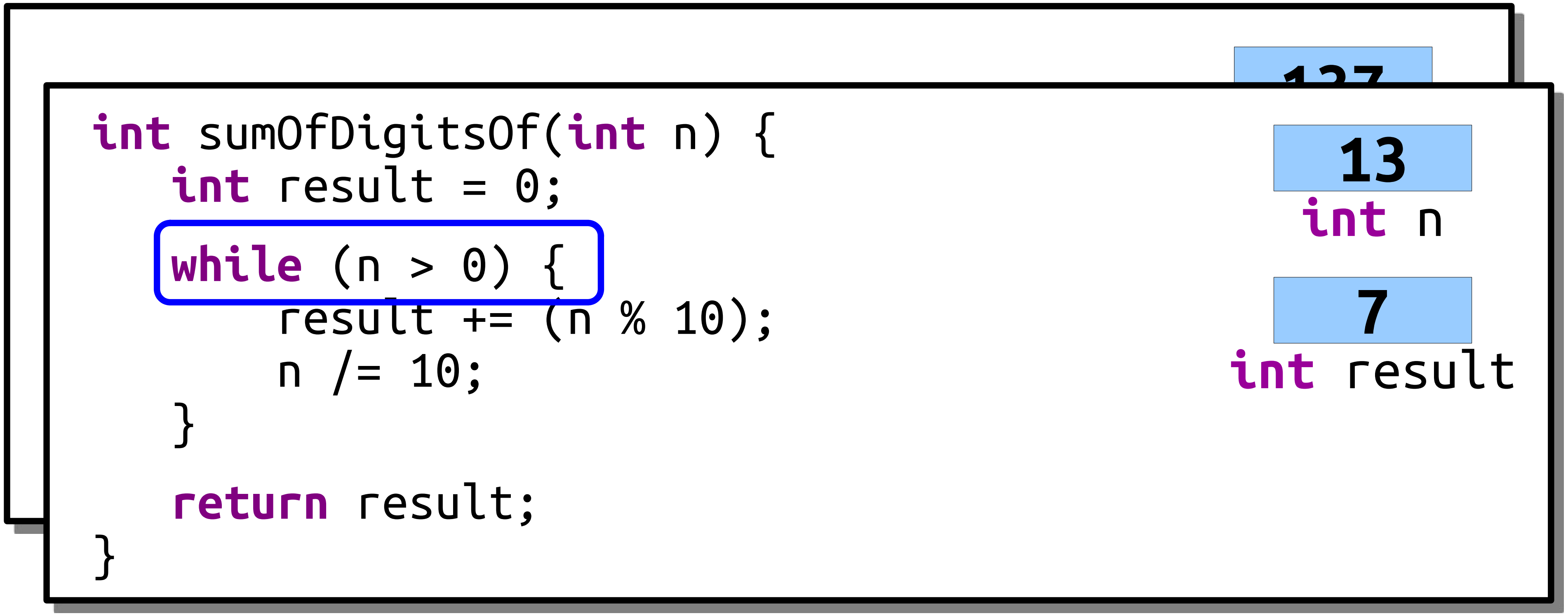


Functions in Action



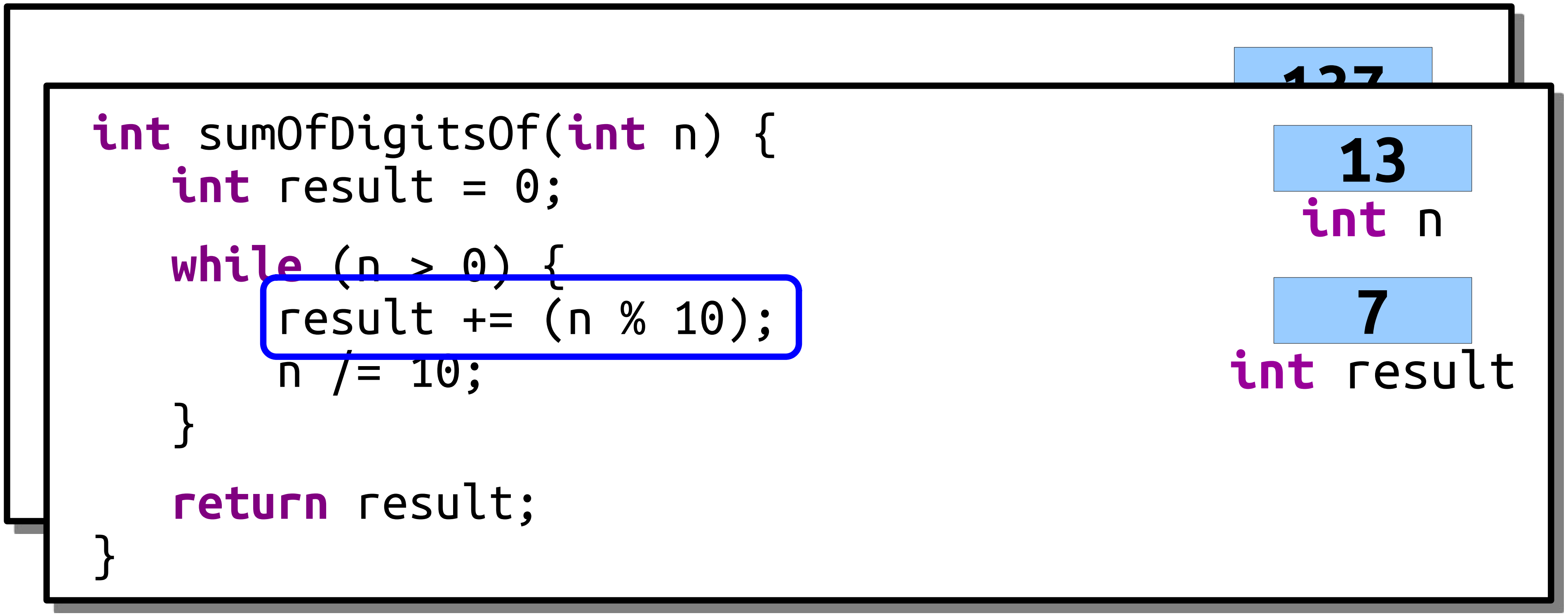
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



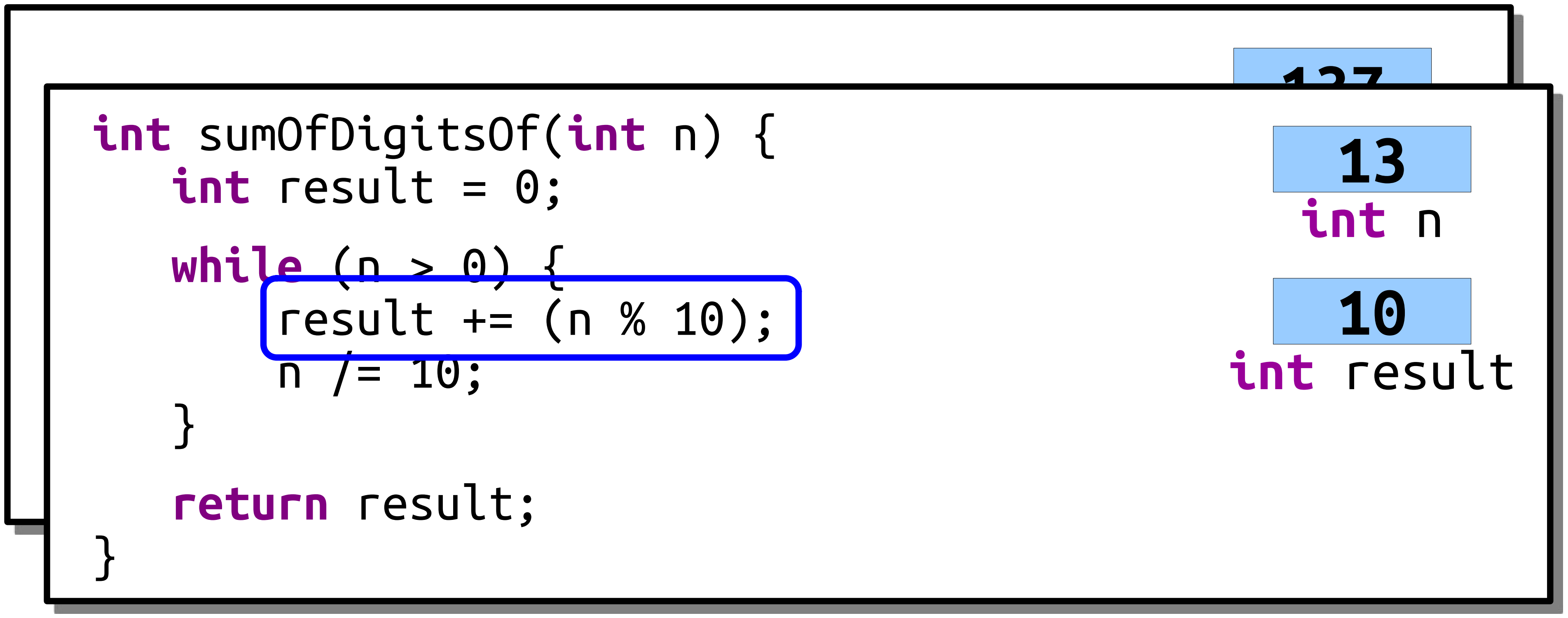
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



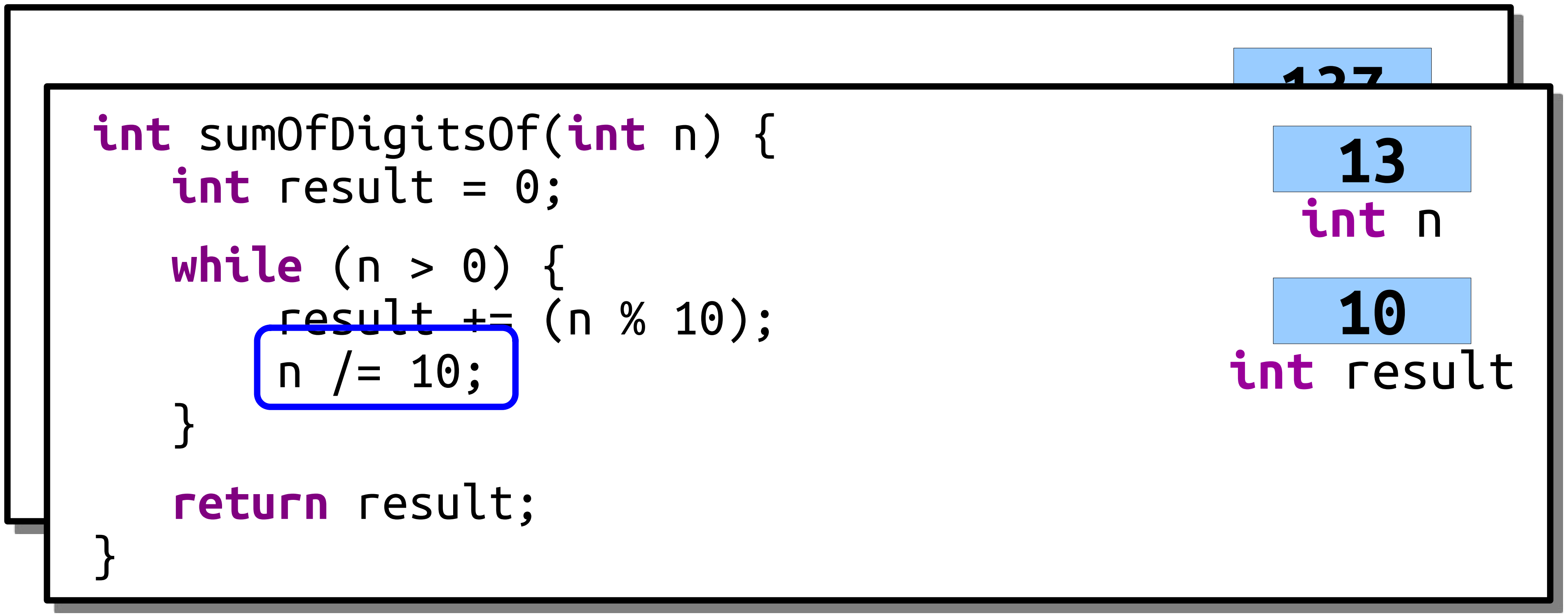
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



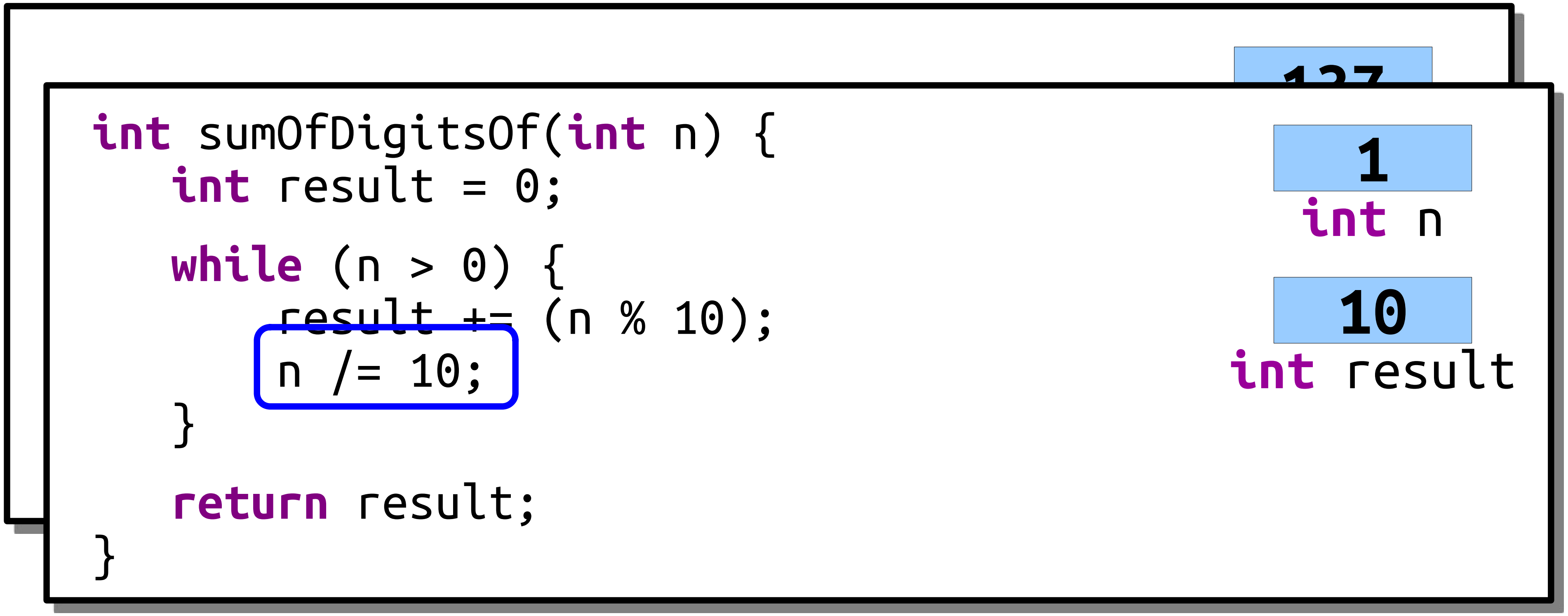
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

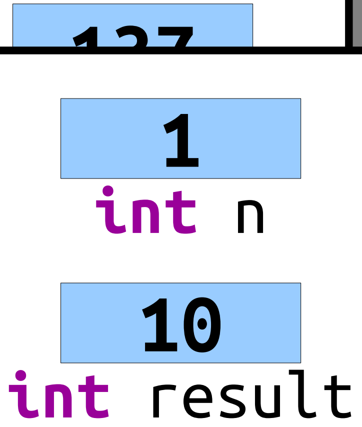
Functions in Action



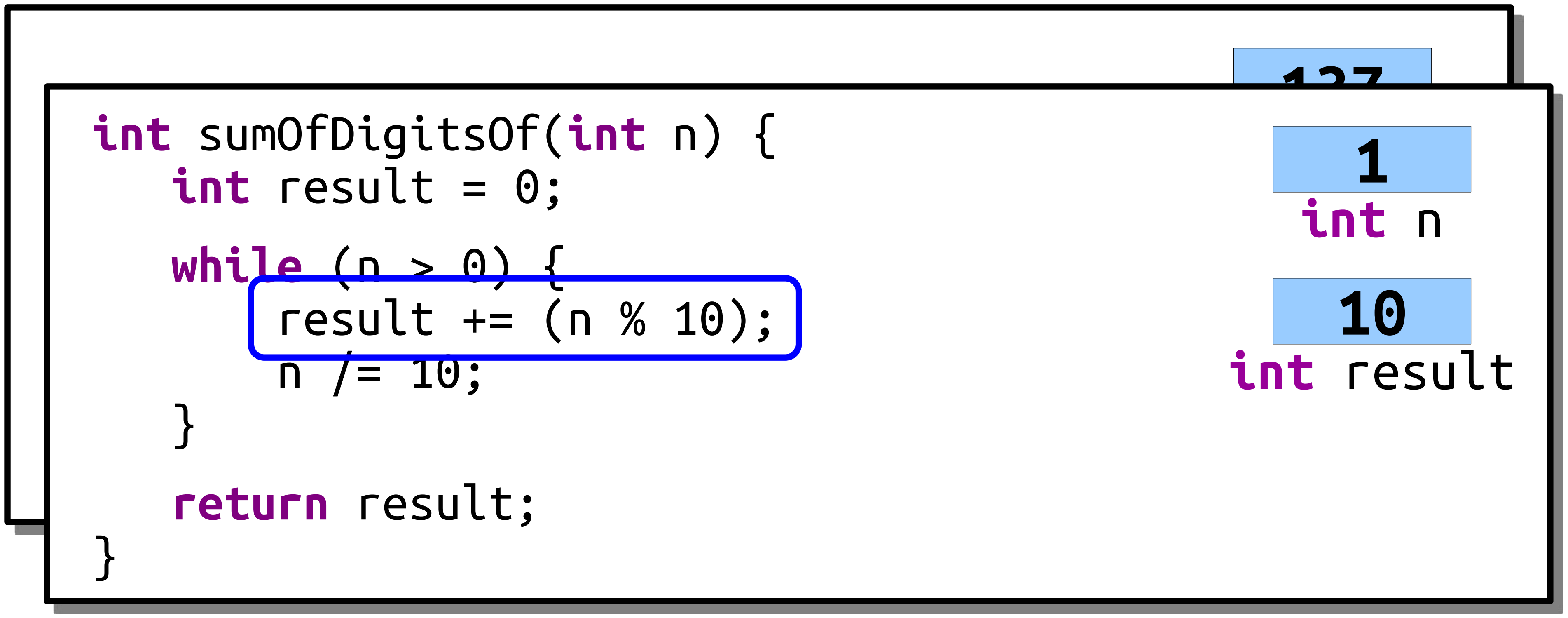
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action

```
int sumOfDigitsOf(int n) {
    int result = 0;
    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }
    return result;
}
```

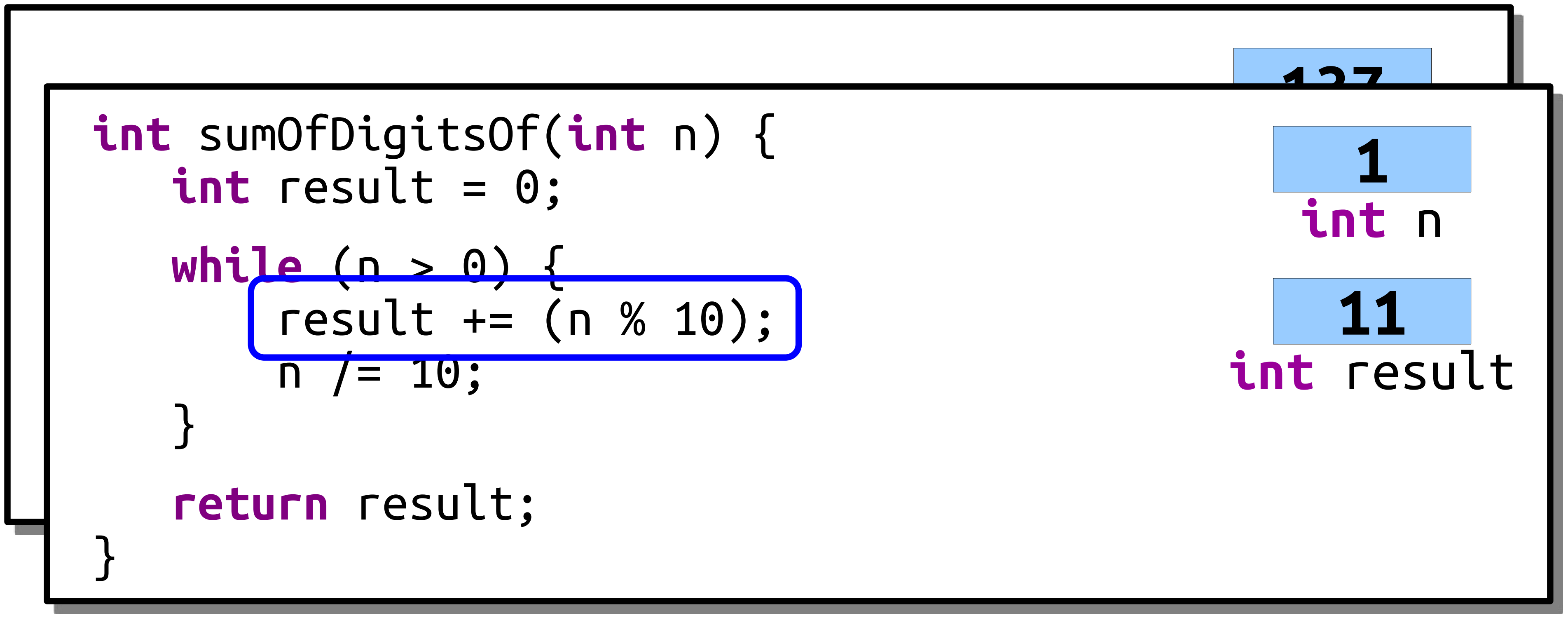


Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action

137

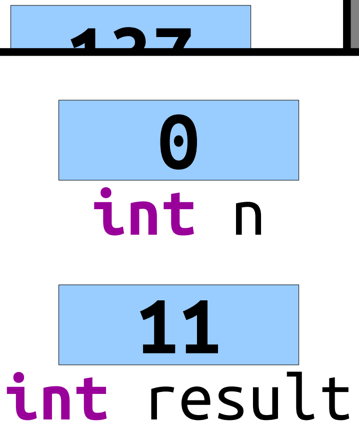
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

1
int n

11
int result

Functions in Action

```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```



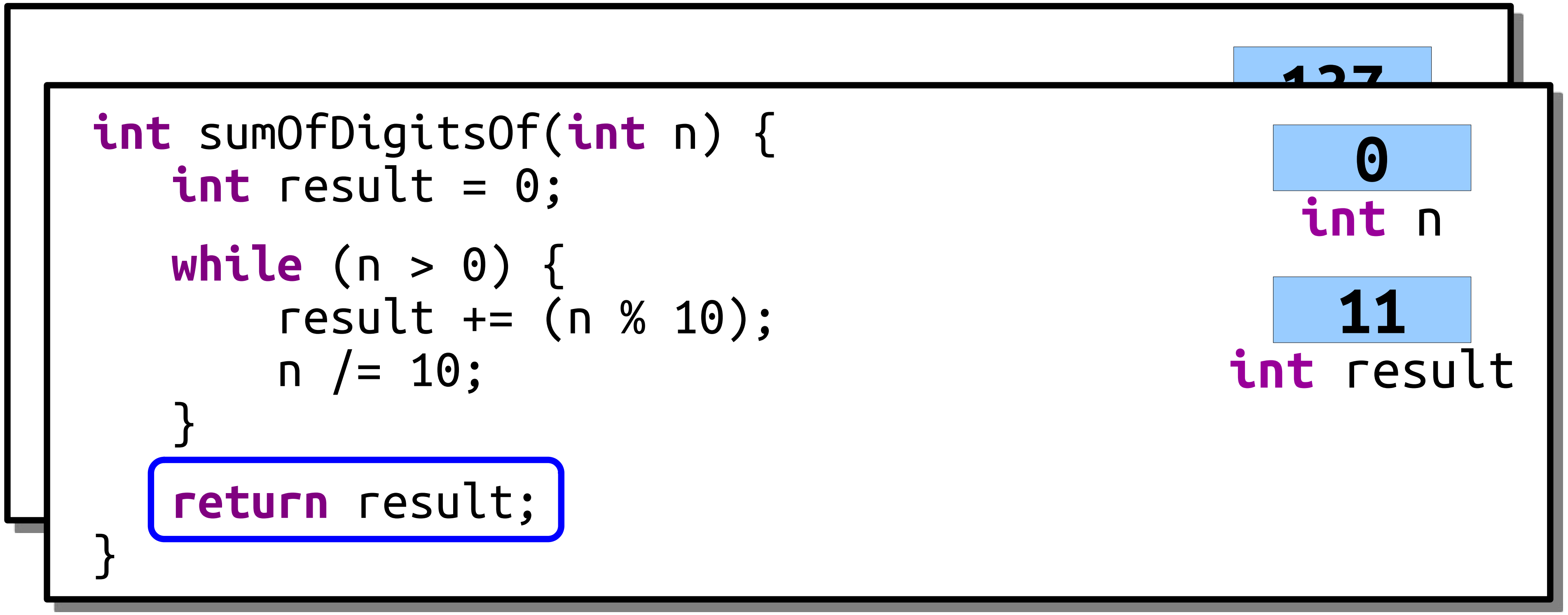
Functions in Action

```
int sumOfDigitsOf(int n) {
    int result = 0;
    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }
    return result;
}
```

0
int n

11
int result

Functions in Action



```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;
}
```

Functions in Action

```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```

137

int n

11

Functions in Action

```
int main() {
   int n = getInteger("Enter an integer: ");
   int digitSum = sumOfDigitsOf(n);
   cout << n << " sums to " << digitSum << endl;

   return 0;
}
```

11

137

int n

11

int digitSum

Functions in Action

```
int main() {
   int n = getInteger("Enter an integer: ");
   int digitSum = sumOfDigitsOf(n);
   cout << n << " sums to " << digitSum << endl;

   return 0;
}
```

137
int n

11
int digitSum

Functions in Action

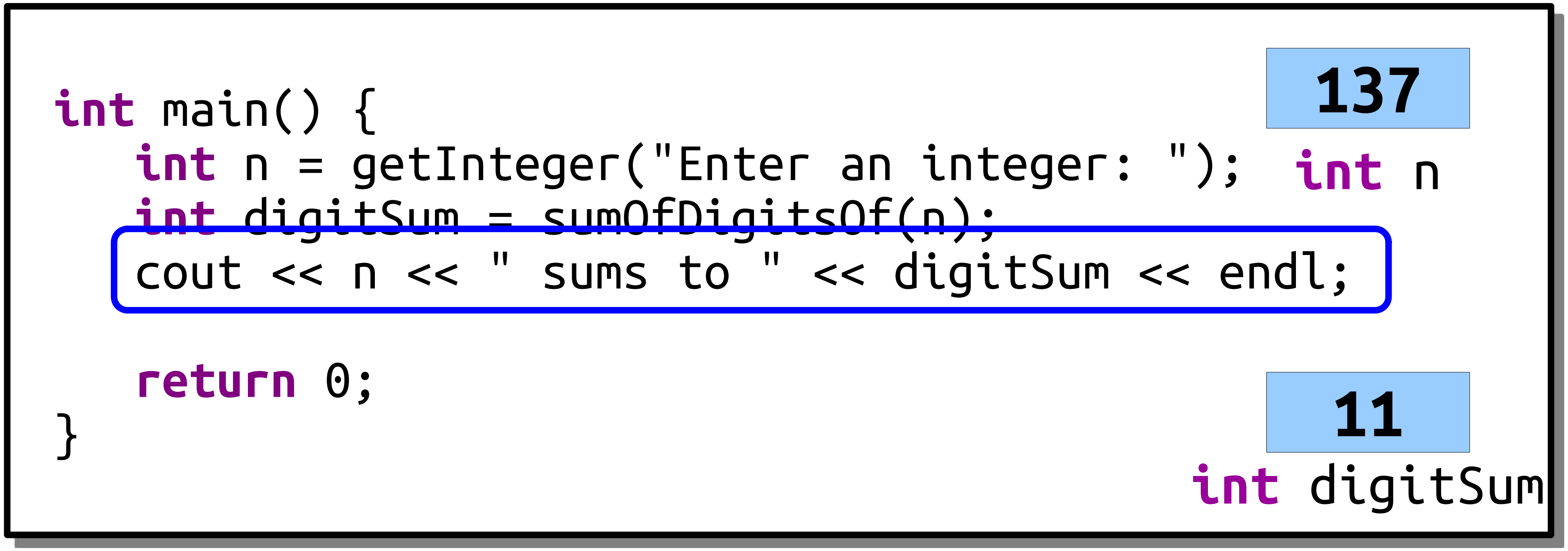
```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```

137
`int n`

11
`int digitSum`

Note that the value of `n` in `main` is unchanged, because `sumOfDigitsOf` got its own copy of `n` that only coincidentally has the same name as the copy in `main`.

Functions in Action



```
int main() {
    int n = getInteger("Enter an integer: ");
    int digitSum = sumOfDigitsOf(n);
    cout << n << " sums to " << digitSum << endl;

    return 0;
}
```

Time-Out for Announcements!

Section Signups

- Section signups go live tomorrow at 5:00PM and are open until Sunday at 5:00PM.
- Sign up using this link:

  https://cs198.stanford.edu/cs198/auth/default.aspx
- You need to sign up here even if you're already enrolled on Axess; *we don't use Axess for sections in this class*.

Qt Creator Help Session

- Having trouble getting Qt Creator set up? Chase is running a Qt Creator help session this Thursday from 2:00PM – 4:00PM over Zoom.
- Check EdStem for info on how to call in.
- A request: Before showing up, use the troubleshooting guide and make sure you followed the directions precisely. It’s easy to get this wrong, but easy to correct once you identify where you went off-script.

Back to CS106B!

Thinking Recursively

Factorials!

- The number ***n factorial***, denoted ***n!***, is

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

- Here's some examples!
  - $3! = 3 \times 2 \times 1 = 6$.
  - $4! = 4 \times 3 \times 2 \times 1 = 24$.
  - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.
  - $0! = 1$. (by definition!)
- Factorials show up in unexpected places! We'll see one later this quarter when we talk about sorting algorithms!
- Let's implement a function to compute factorials!

Computing Factorials

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!}$$

Computing Factorials

$$5! = 5 \times 4!$$

Computing Factorials

$$5! = 5 \times 4!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times \underbrace{3 \times 2 \times 1}_{3!}$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3!
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3!
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2 \times 1
\end{aligned}
$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times \underbrace{2 \times 1}_{2!}
\end{aligned}
$$

Computing Factorials

$$\begin{aligned} 5! &= 5 \times 4! \\ 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \end{aligned}$$

Computing Factorials

$$\begin{aligned} 5! &= 5 \times 4! \\ 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \end{aligned}$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1!
\end{aligned}
$$

Computing Factorials

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \times 0!
\end{aligned}$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \times 0! \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \times \mathbf{1} \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= \mathbf{1} \\
0! &= 1
\end{aligned}$$

Computing Factorials

$$\begin{aligned} 5! &= 5 \times 4! \\ 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \\ 2! &= 2 \times 1! \\ 1! &= 1 \\ 0! &= 1 \end{aligned}$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times \mathbf{1} \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= \mathbf{2} \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times \mathbf{2} \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= \mathbf{6} \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 6 \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times \mathbf{6} \\
3! &= 6 \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= \mathbf{24} \\
3! &= 6 \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 24 \\
3! &= 6 \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$
\begin{aligned}
5! &= 5 \times \mathbf{24} \\
4! &= 24 \\
3! &= 6 \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$5! = \mathbf{120}$

$4! = 24$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 1$

Computing Factorials

$$
\begin{aligned}
5! &= 120 \\
4! &= 24 \\
3! &= 6 \\
2! &= 2 \\
1! &= 1 \\
0! &= 1
\end{aligned}
$$

Computing Factorials

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \times 0! \\
0! &= 1
\end{aligned}$$

Another View of Factorials

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

Recursion in Action

```
int main() {
   int nFact = factorial(5);
   cout << "5! = " << nFact << endl;

   return 0;
}
```

Recursion in Action

```
int main() {
    int nFact = factorial(5);
    cout << "5! = " << nFact << endl;

    return 0;
}
```

Recursion in Action

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

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

5

`int n`

Recursion in Action

```
int main() {

}
```

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

5
int n

Recursion in Action

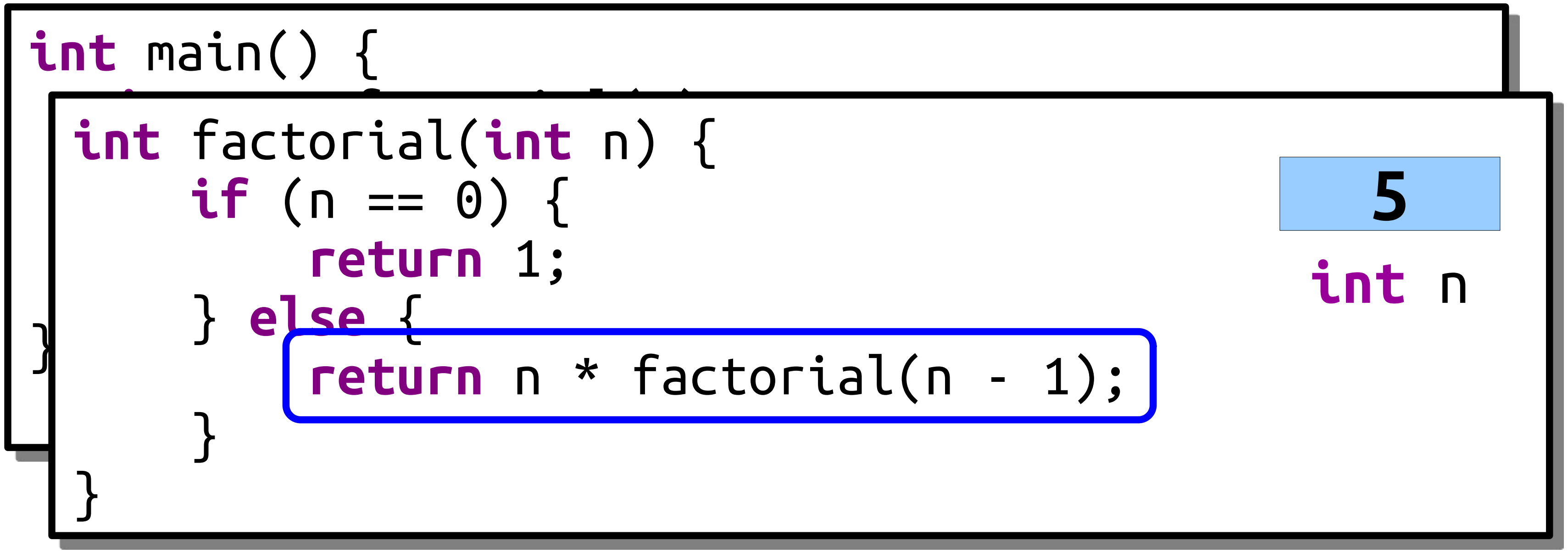
```
int main() {

}
```

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

5

int n

Recursion in Action



```
int main() {
```

```
}
```

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

Recursion in Action

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

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

5

int n

Recursion in Action

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

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

5

int n

Recursion in Action

```
int main() {
```

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

5

5

int n

Recursion in Action

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

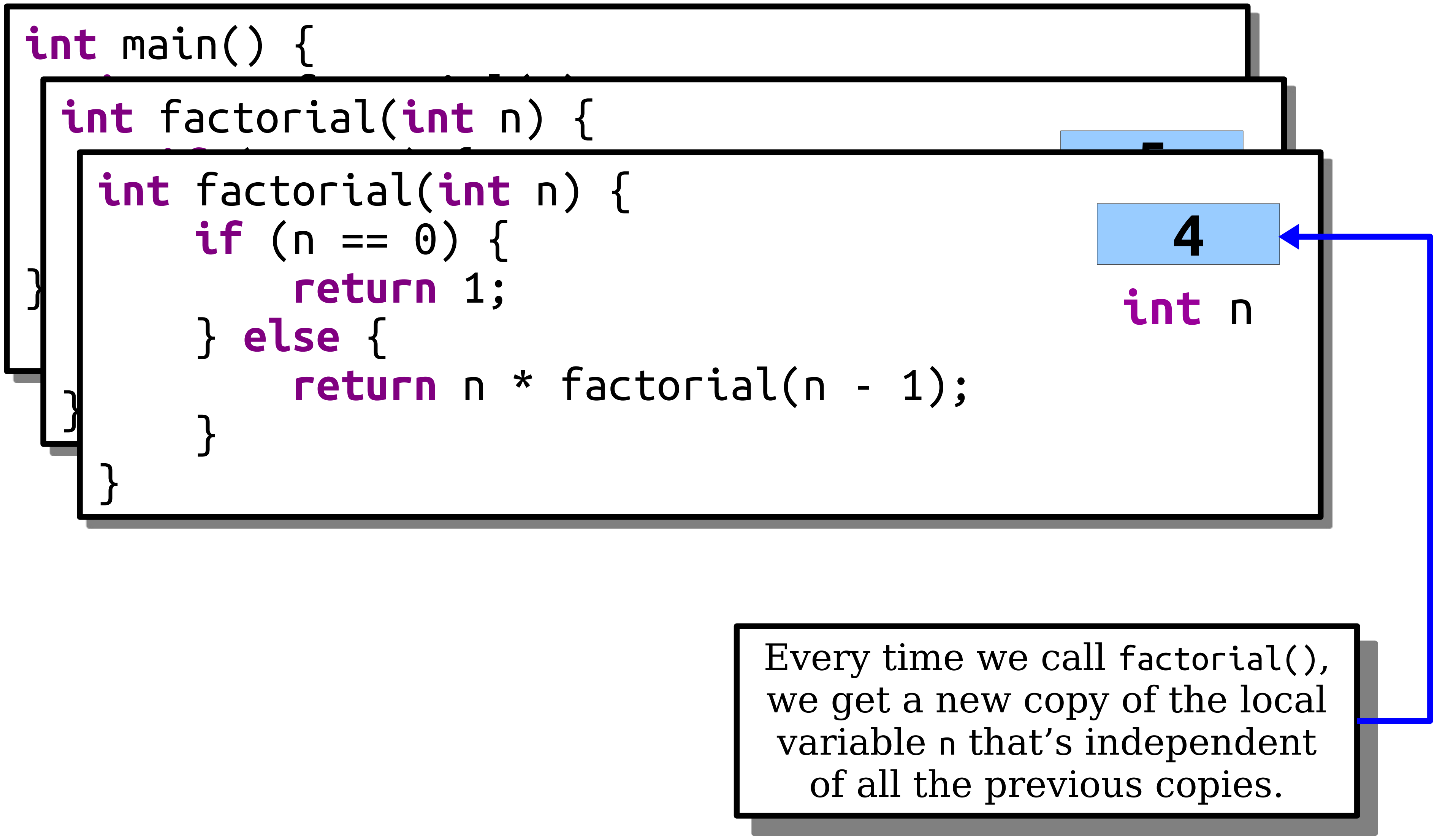
```
int factorial(int n) {
}
```

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

4

int n

Recursion in Action



```
int main() {
```

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

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

Every time we call `factorial()`, we get a new copy of the local variable `n` that's independent of all the previous copies.

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {

}
```

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

4

int n

Recursion in Action

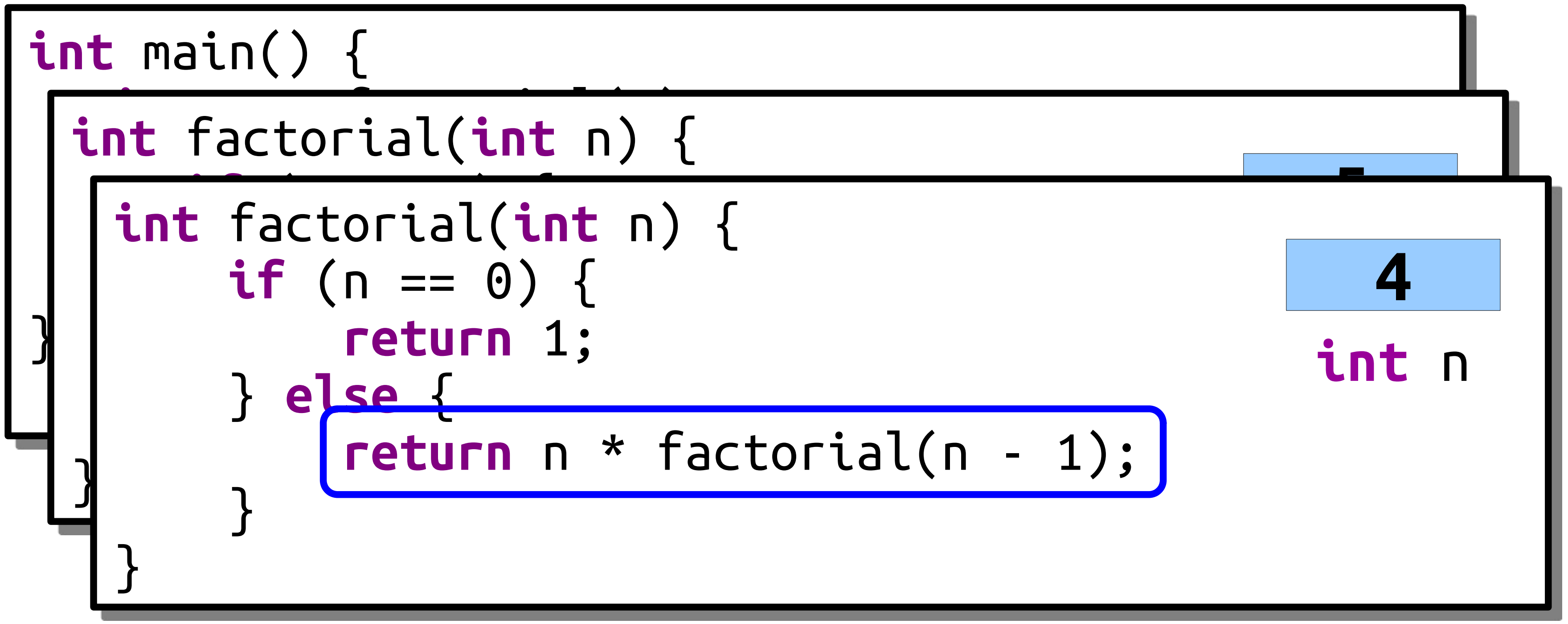
```
int main() {
}
```

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

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

4
int n

Recursion in Action



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

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

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

Recursion in Action

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

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

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

4

int n

Recursion in Action

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

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

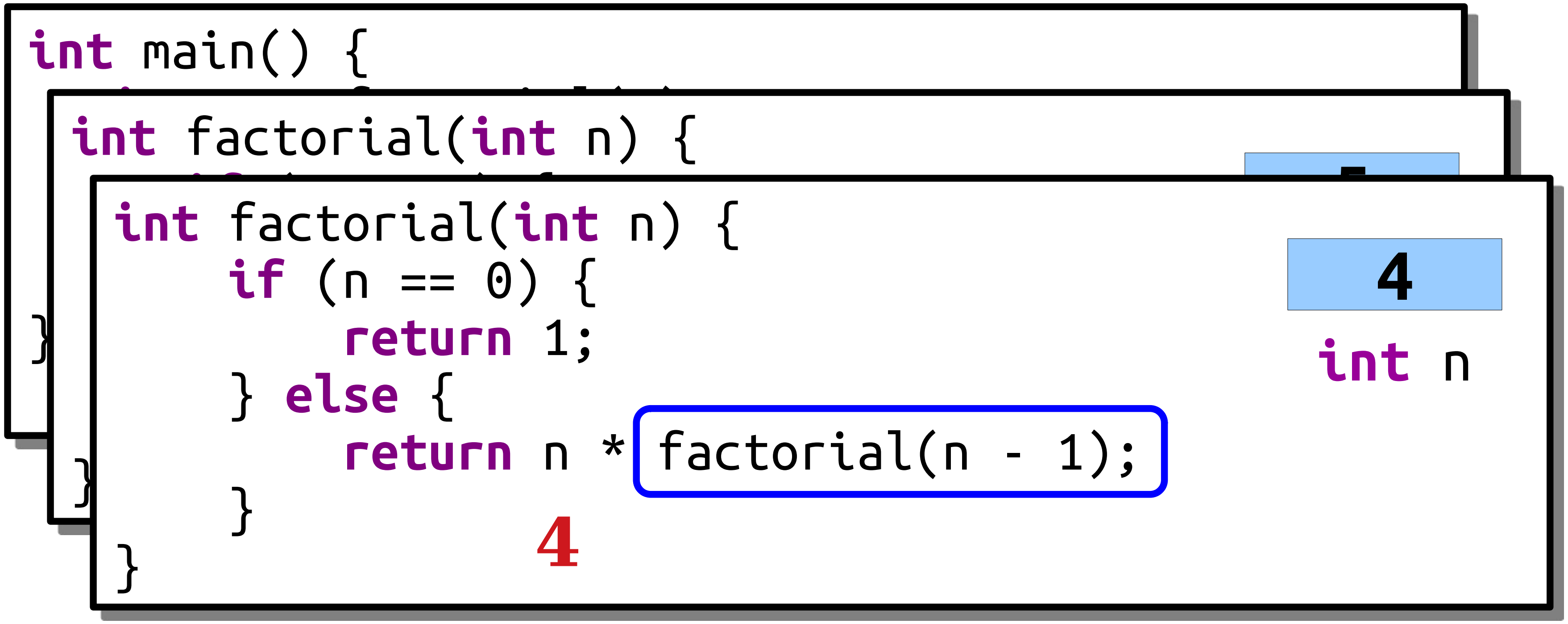
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

4

int n

4

Recursion in Action



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

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

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

Recursion in Action

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

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

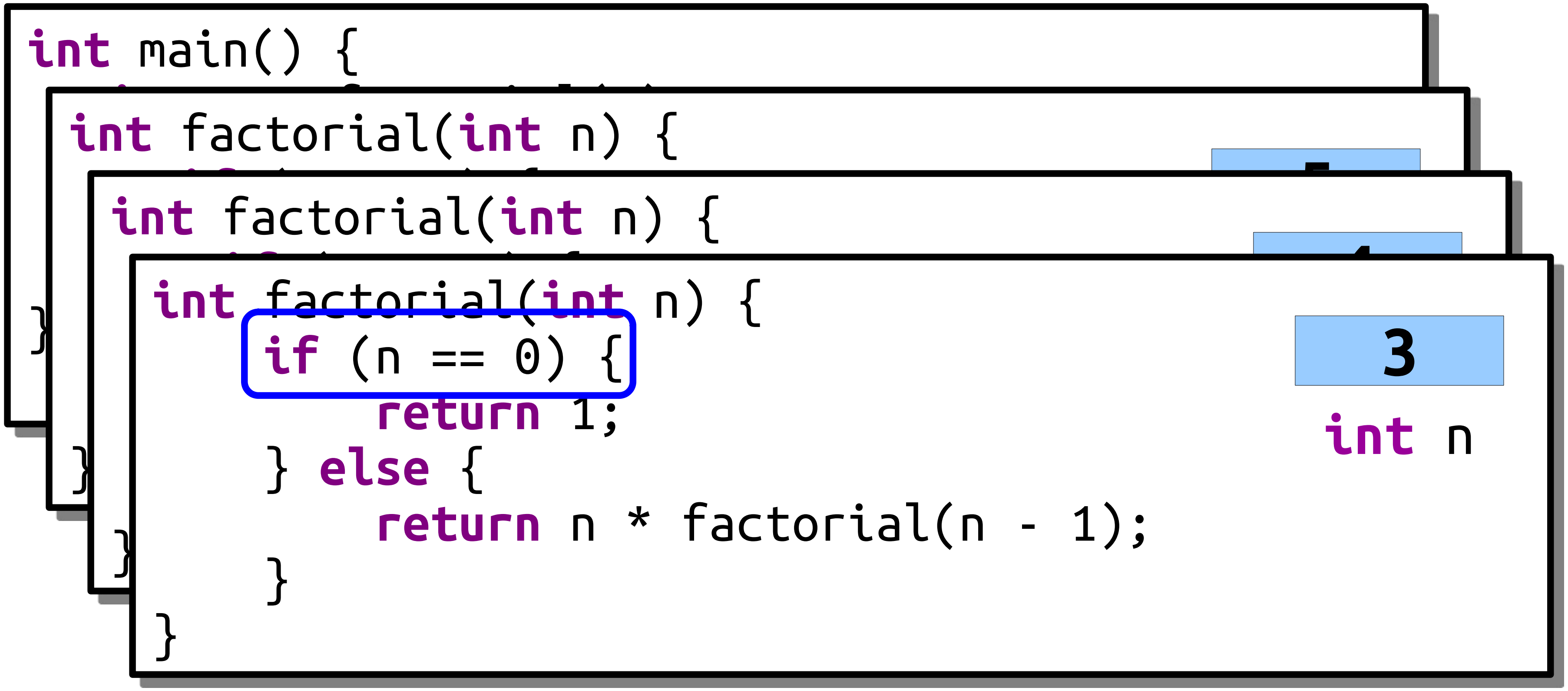
```
int factorial(int n) {
}
```

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

3

int n

Recursion in Action



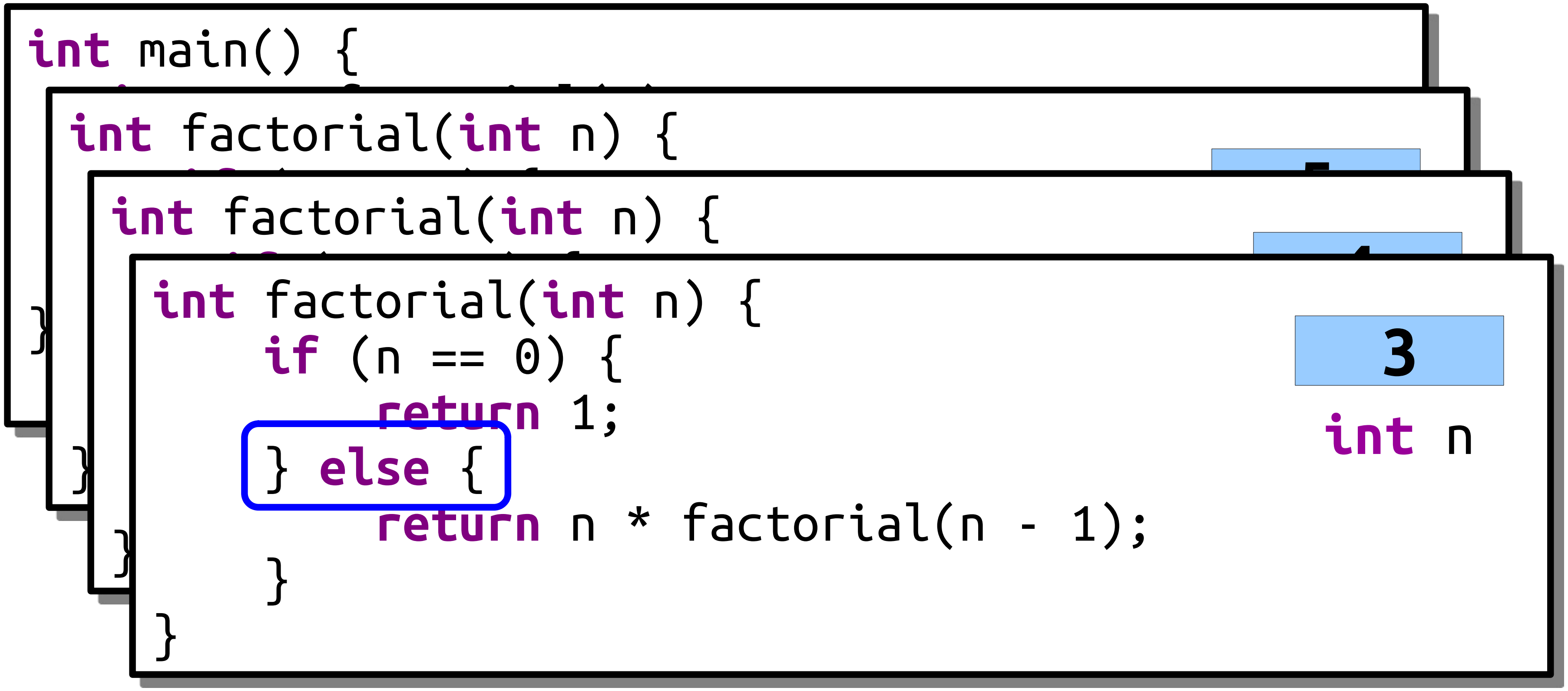
```
int main() {
```

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

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

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

Recursion in Action



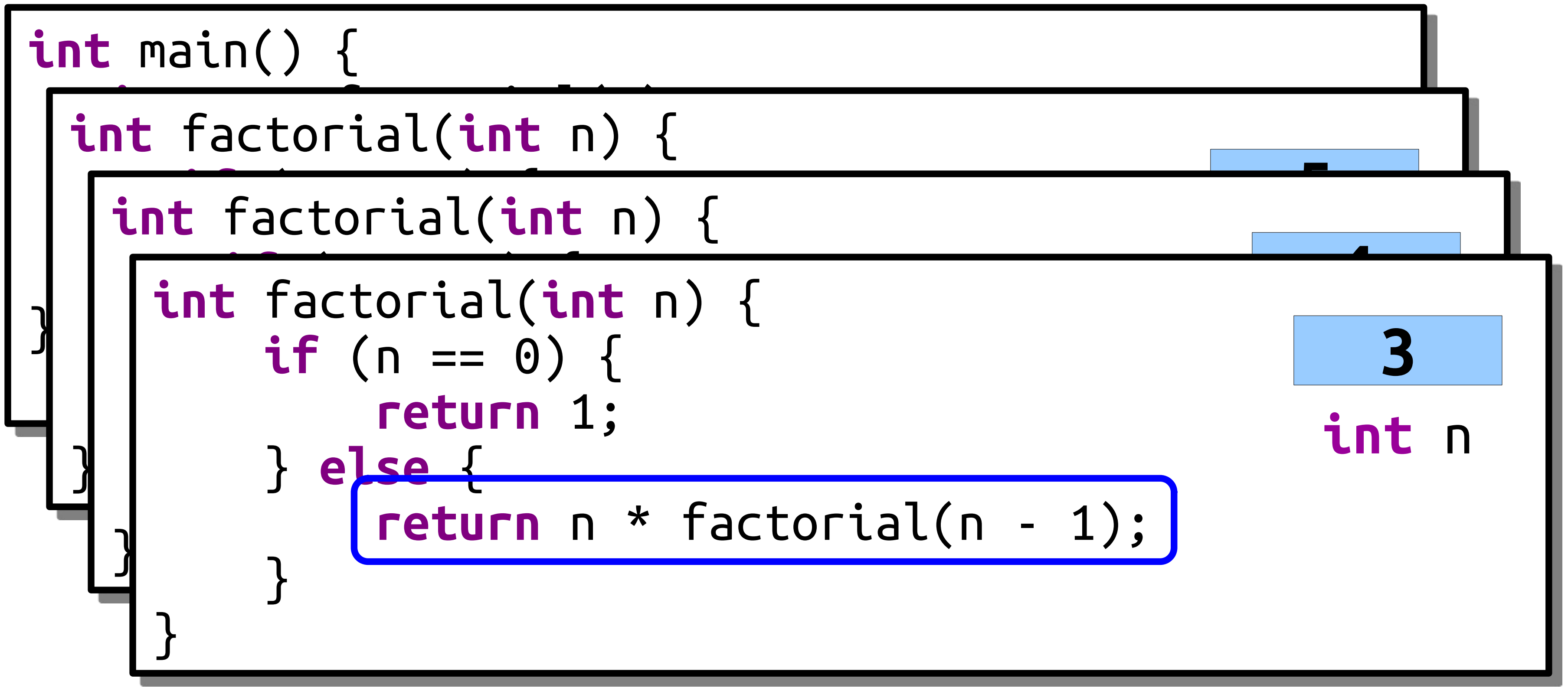
```
int main() {
}
```

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

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

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

Recursion in Action



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

Recursion in Action

```
int main() {
```

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

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

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

3

int n

Recursion in Action

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

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

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

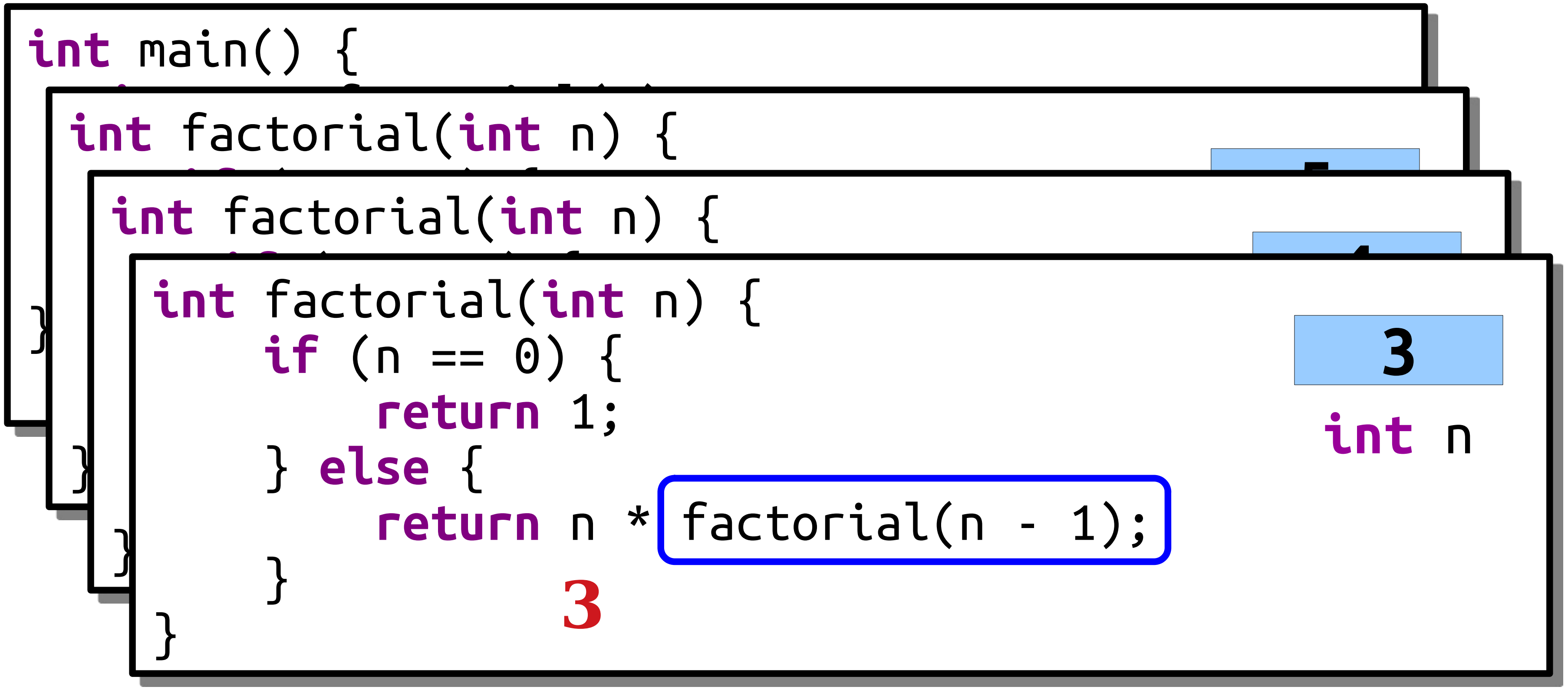
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

$\boxed{n} = 3$

3

int n

Recursion in Action



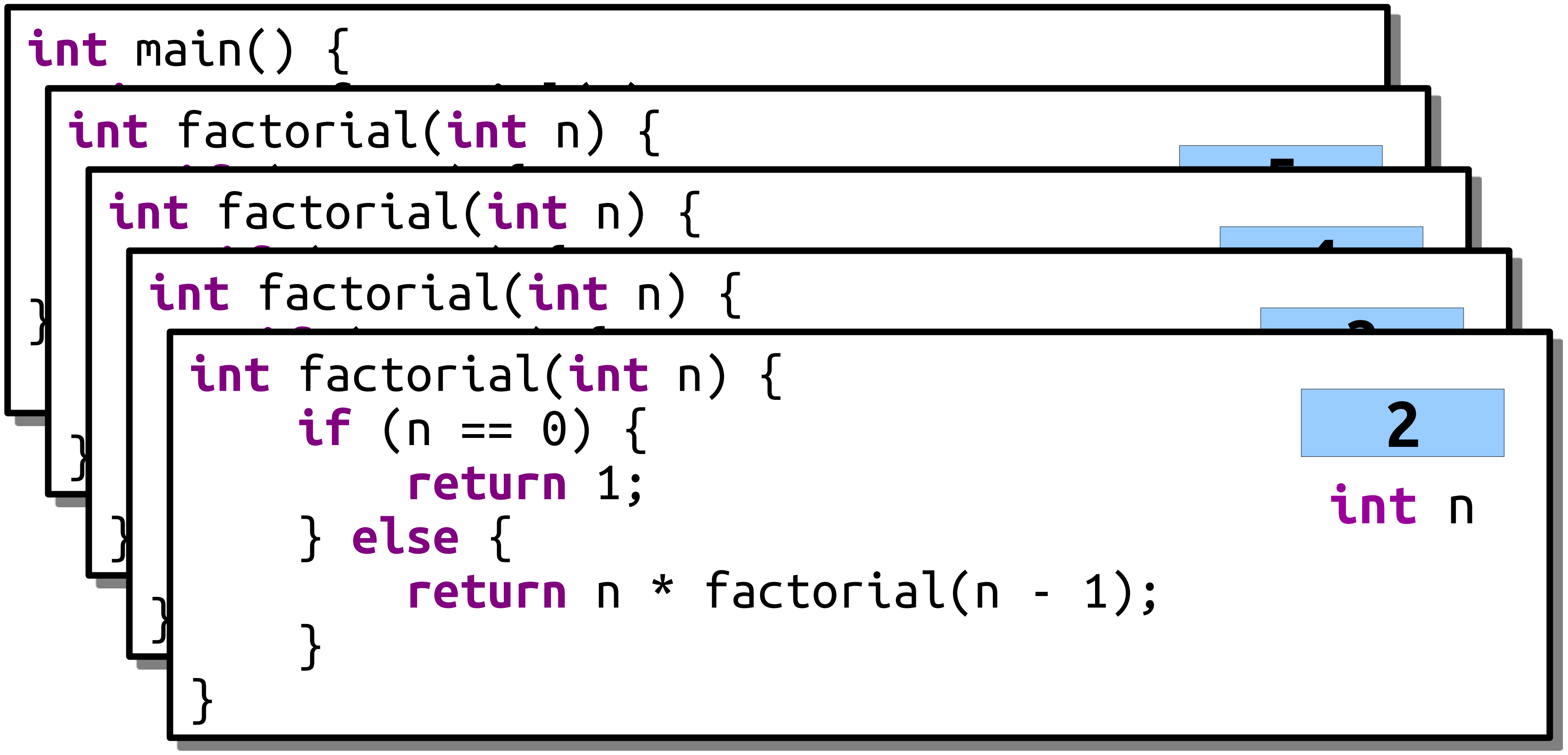
```
int main() {
```

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

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

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

Recursion in Action



```
int main() {
```

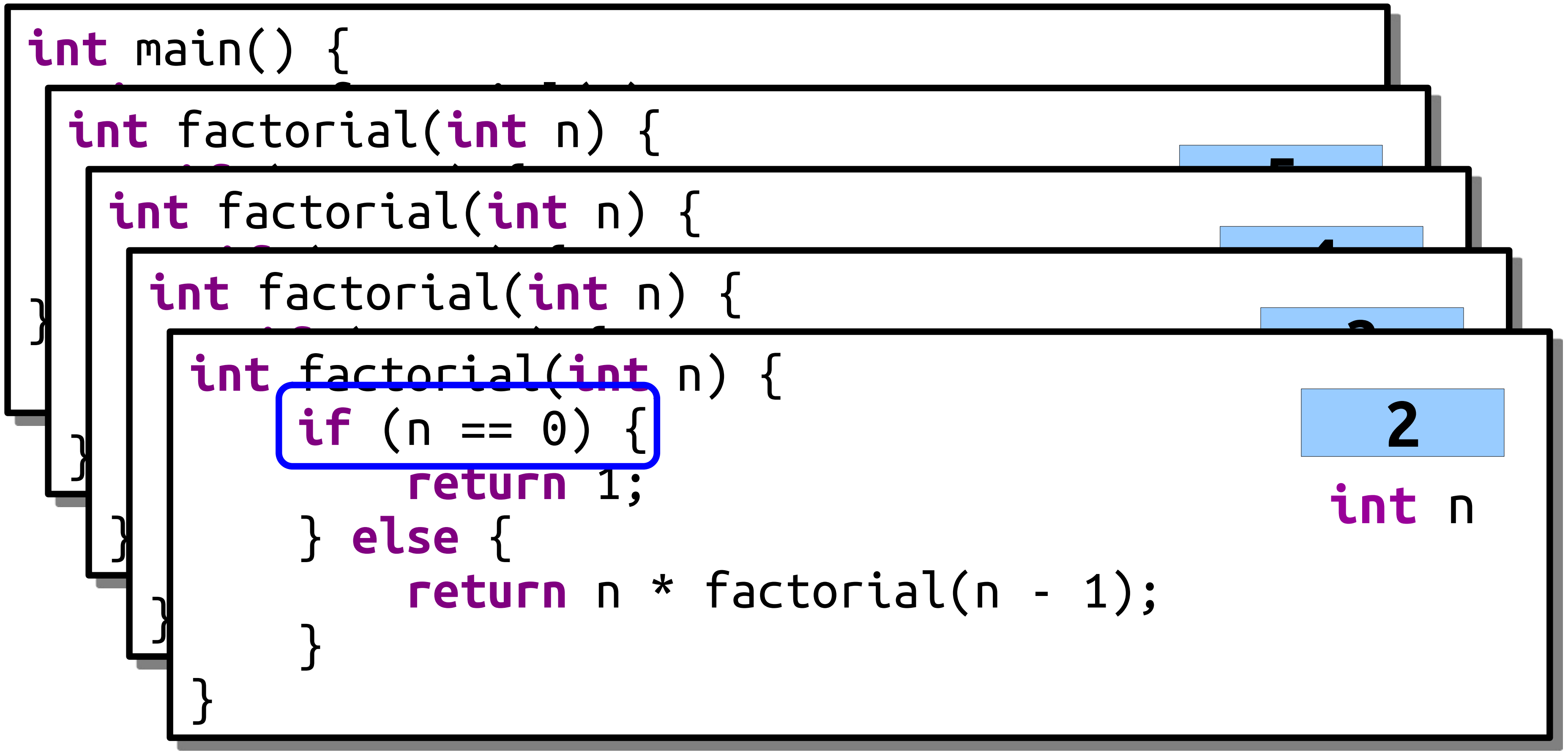
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

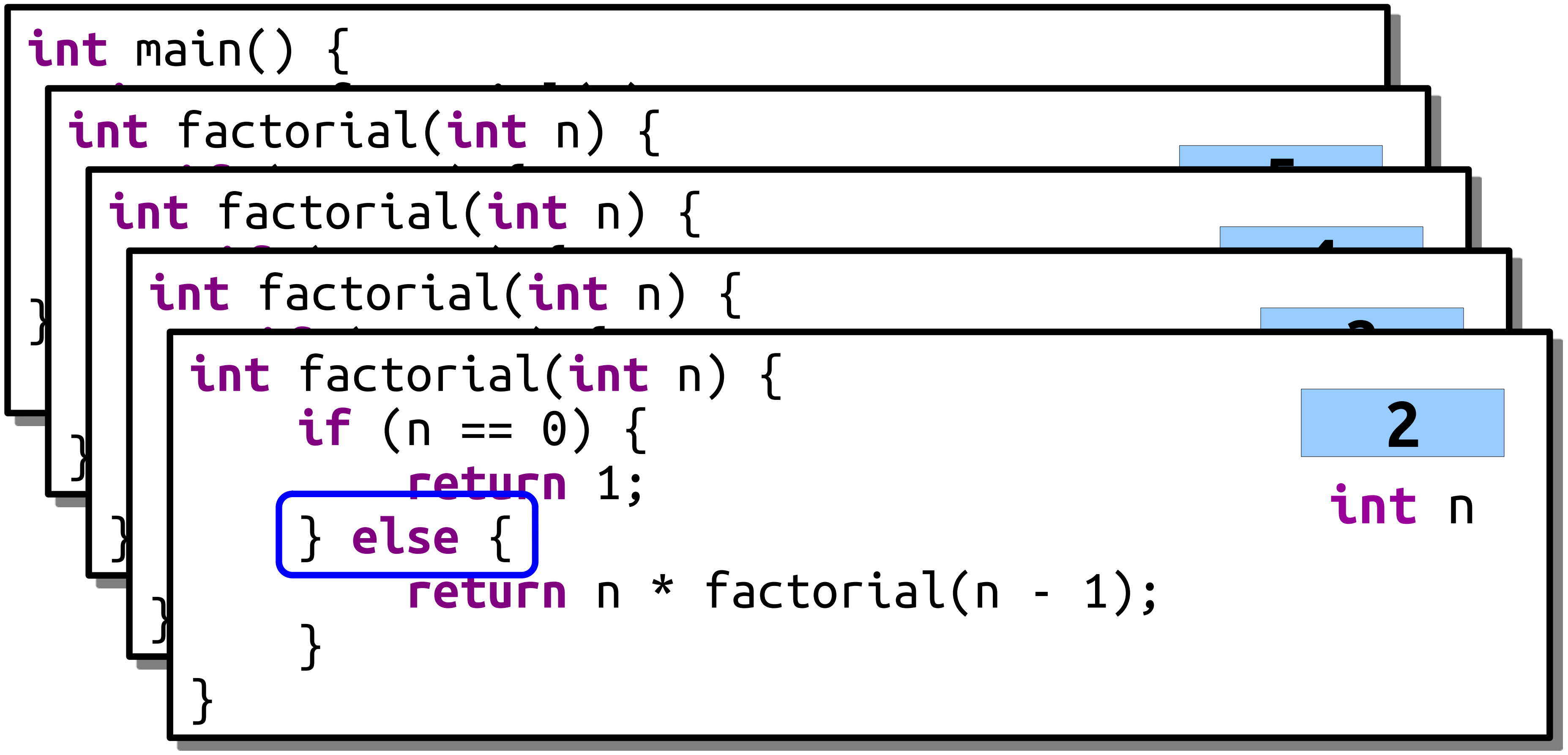
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

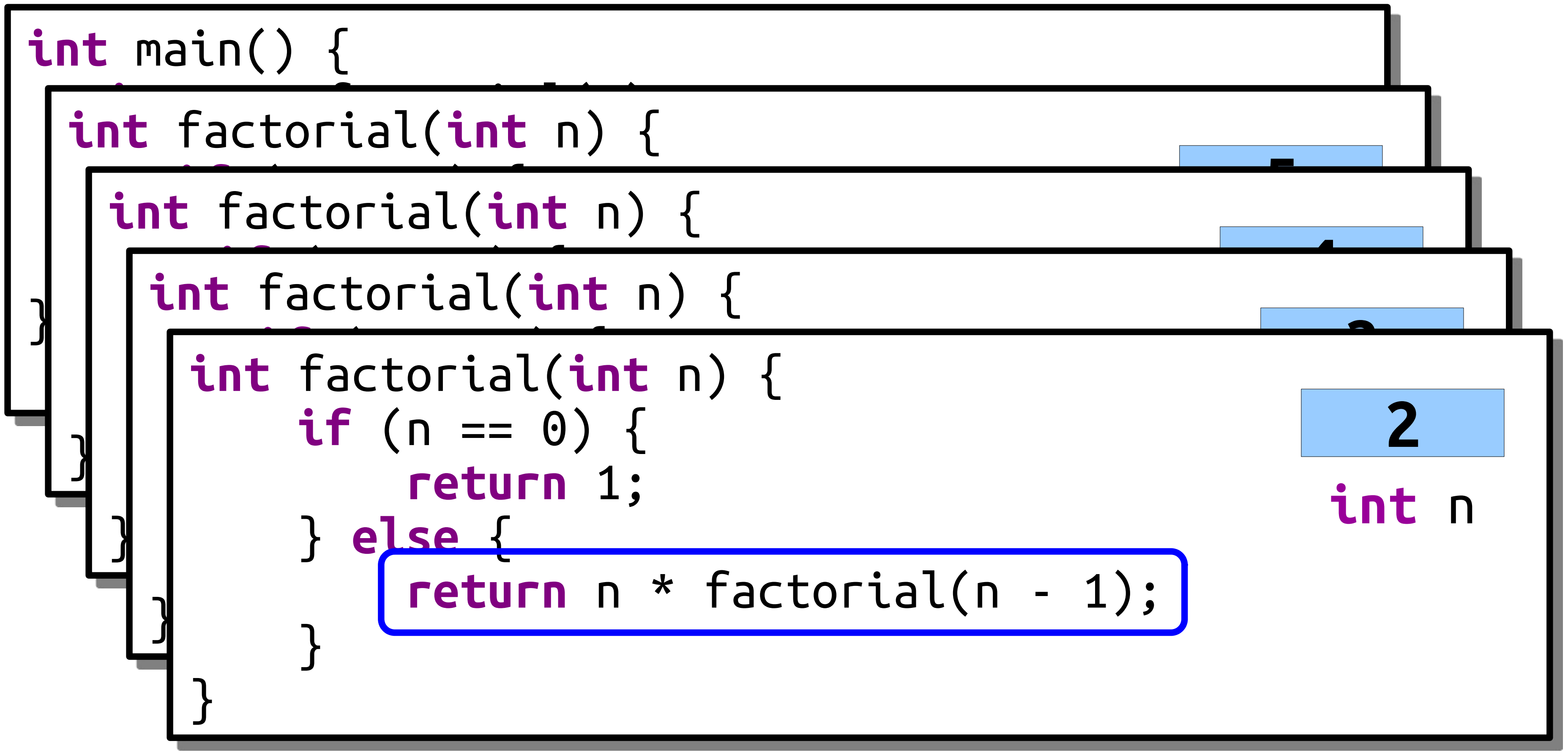
```
int factorial(int n) {
```

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

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

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

2

int n

Recursion in Action

```
int main() {
```

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

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

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

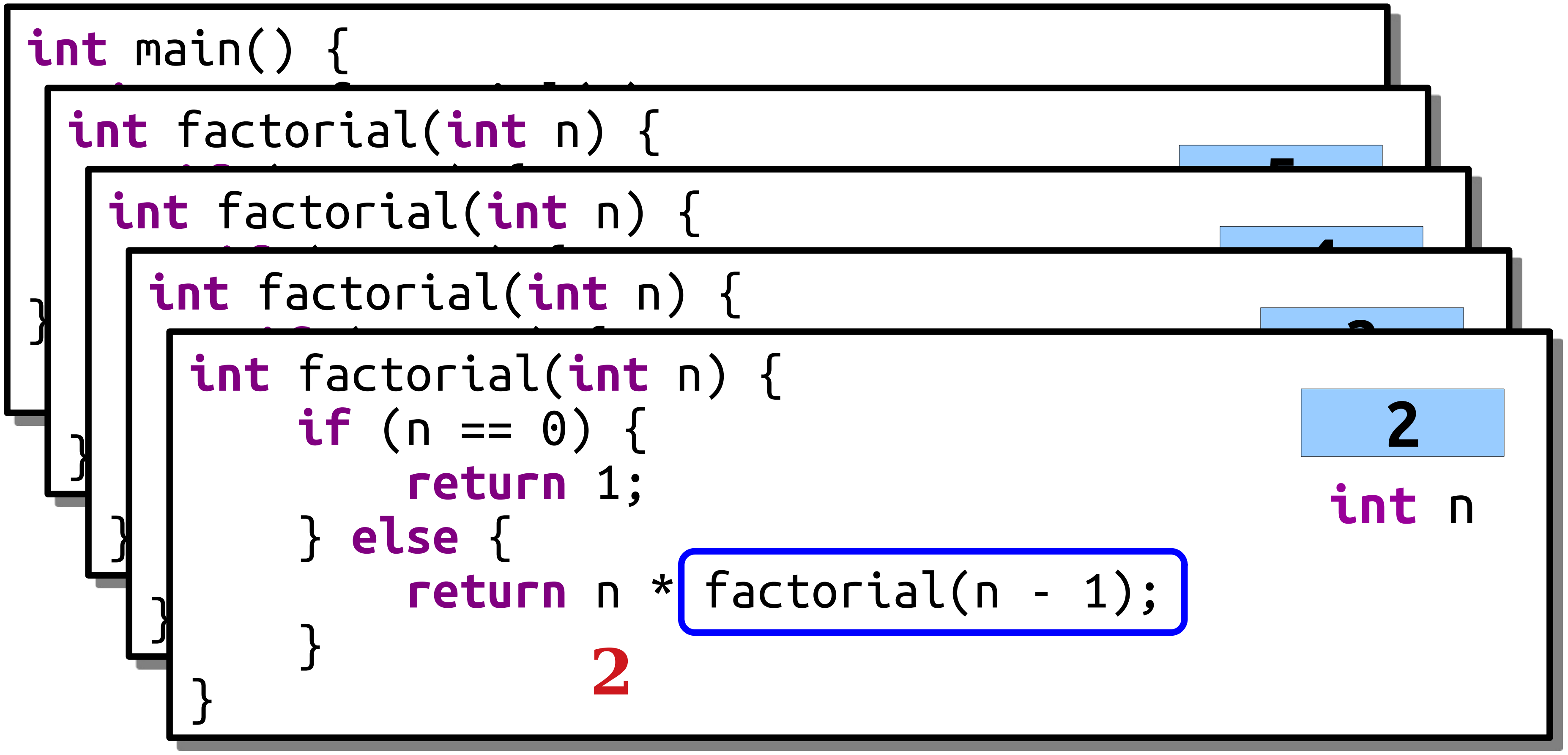
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

2

int n

$\boxed{n}$ = 2

Recursion in Action



```
int main() {
```

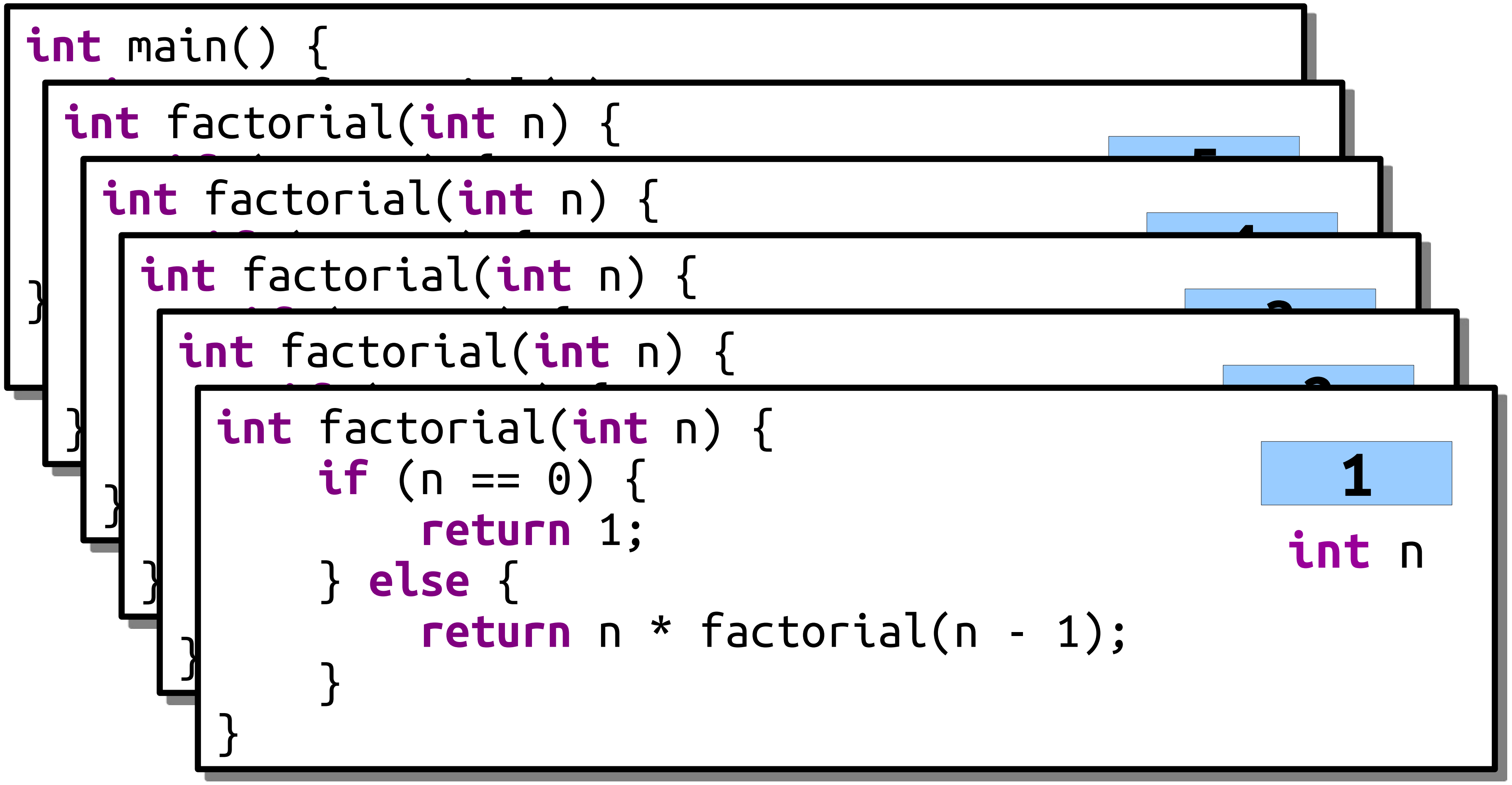
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

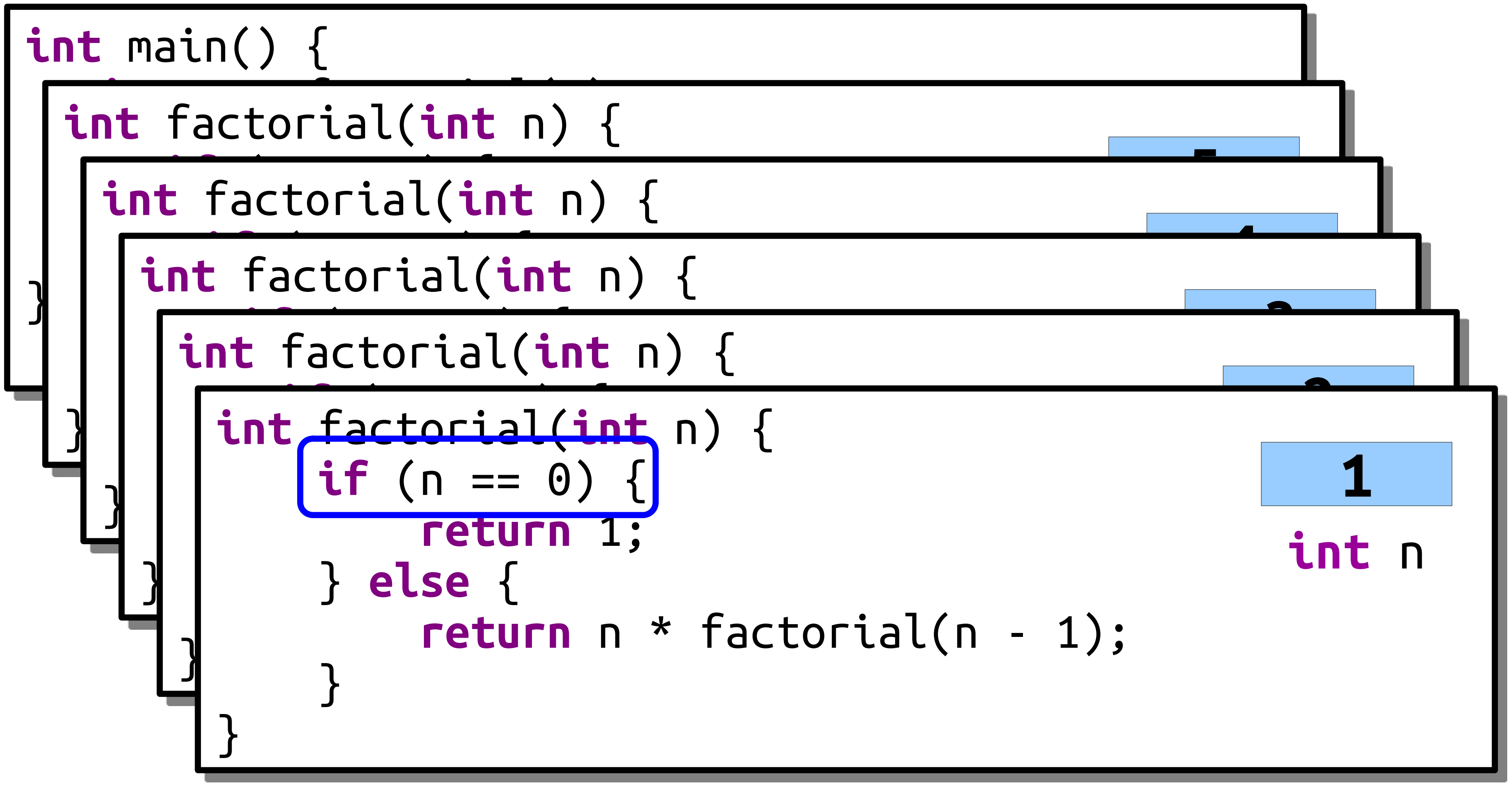
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

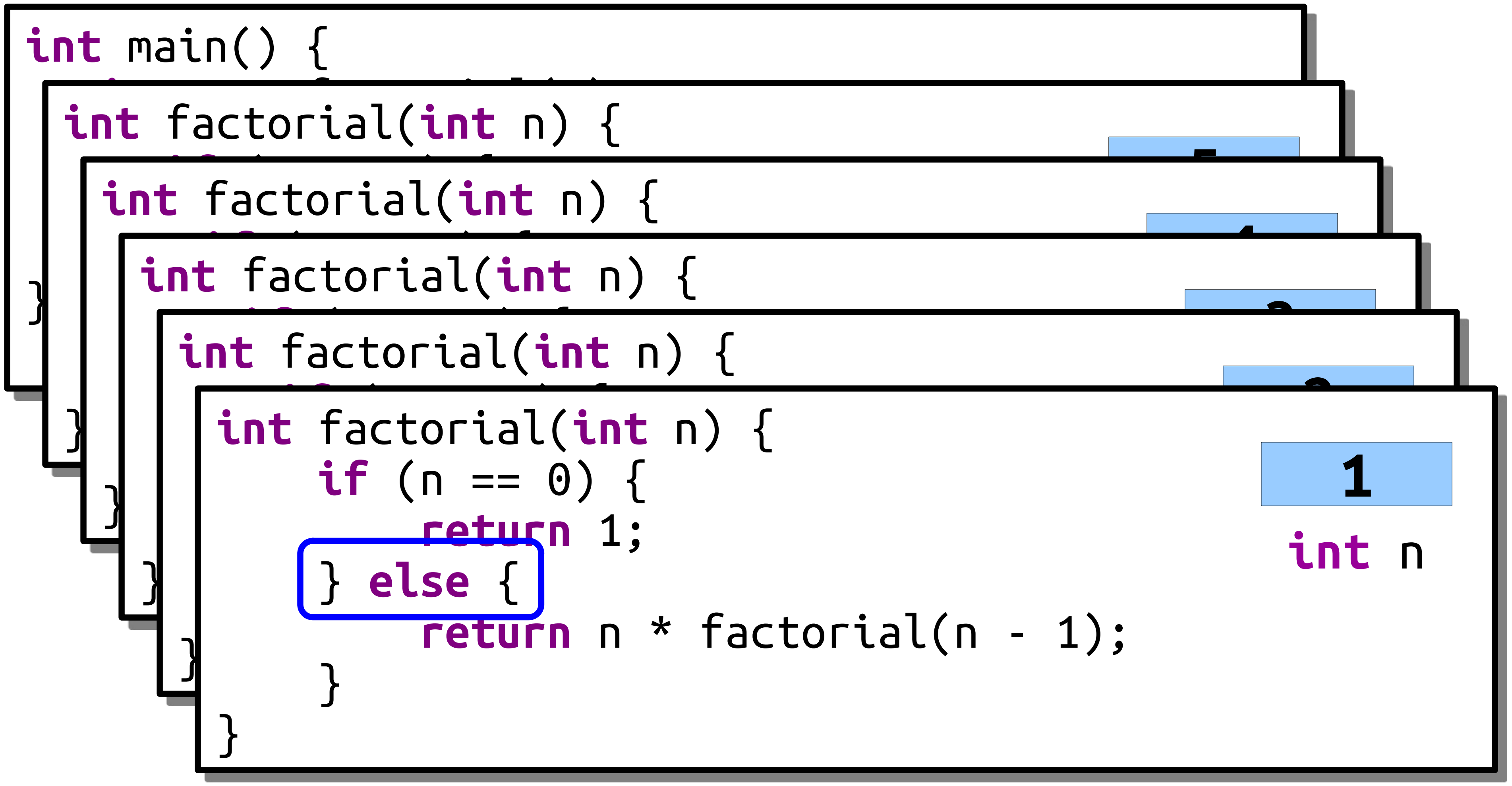
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

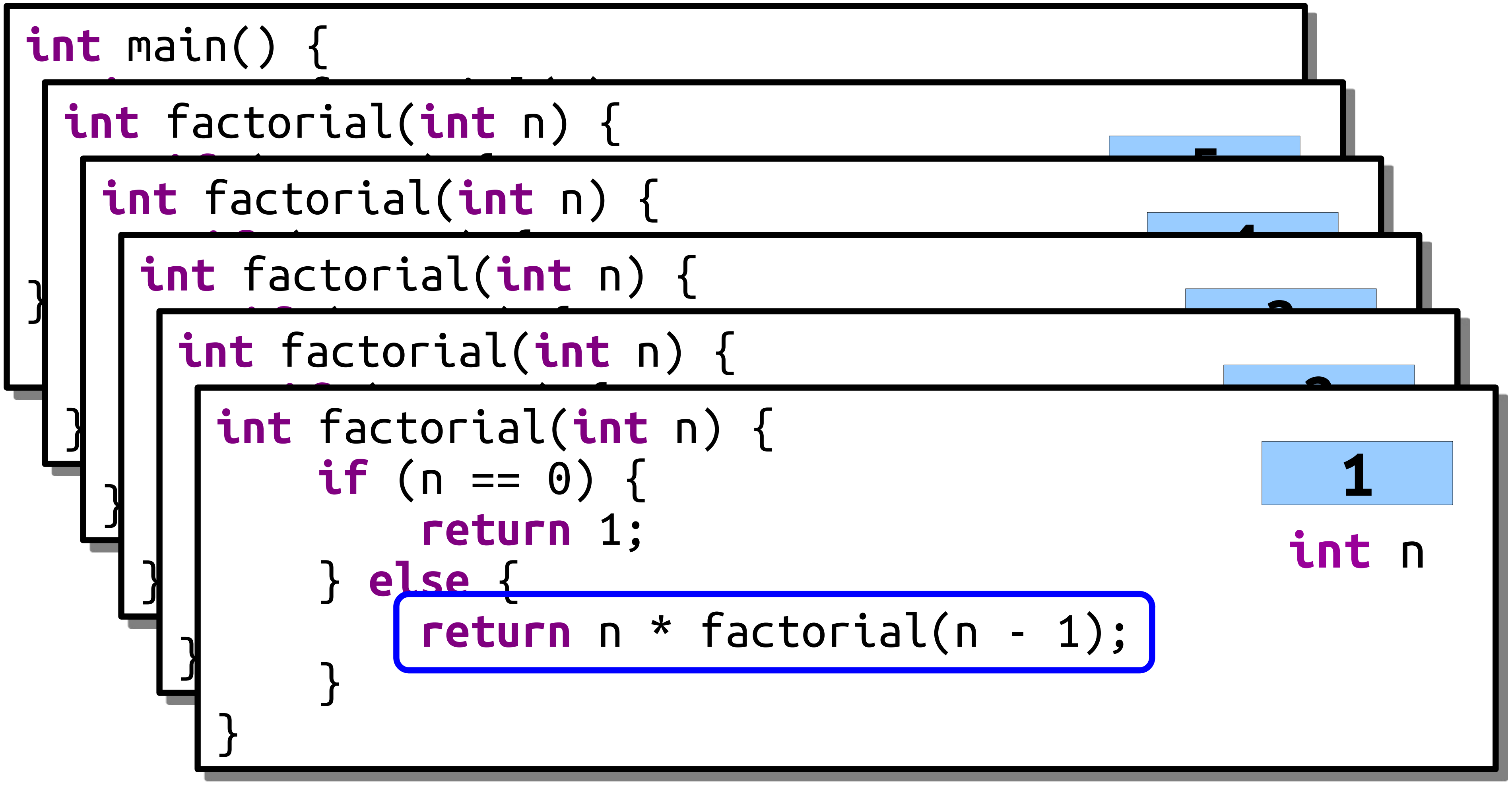
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

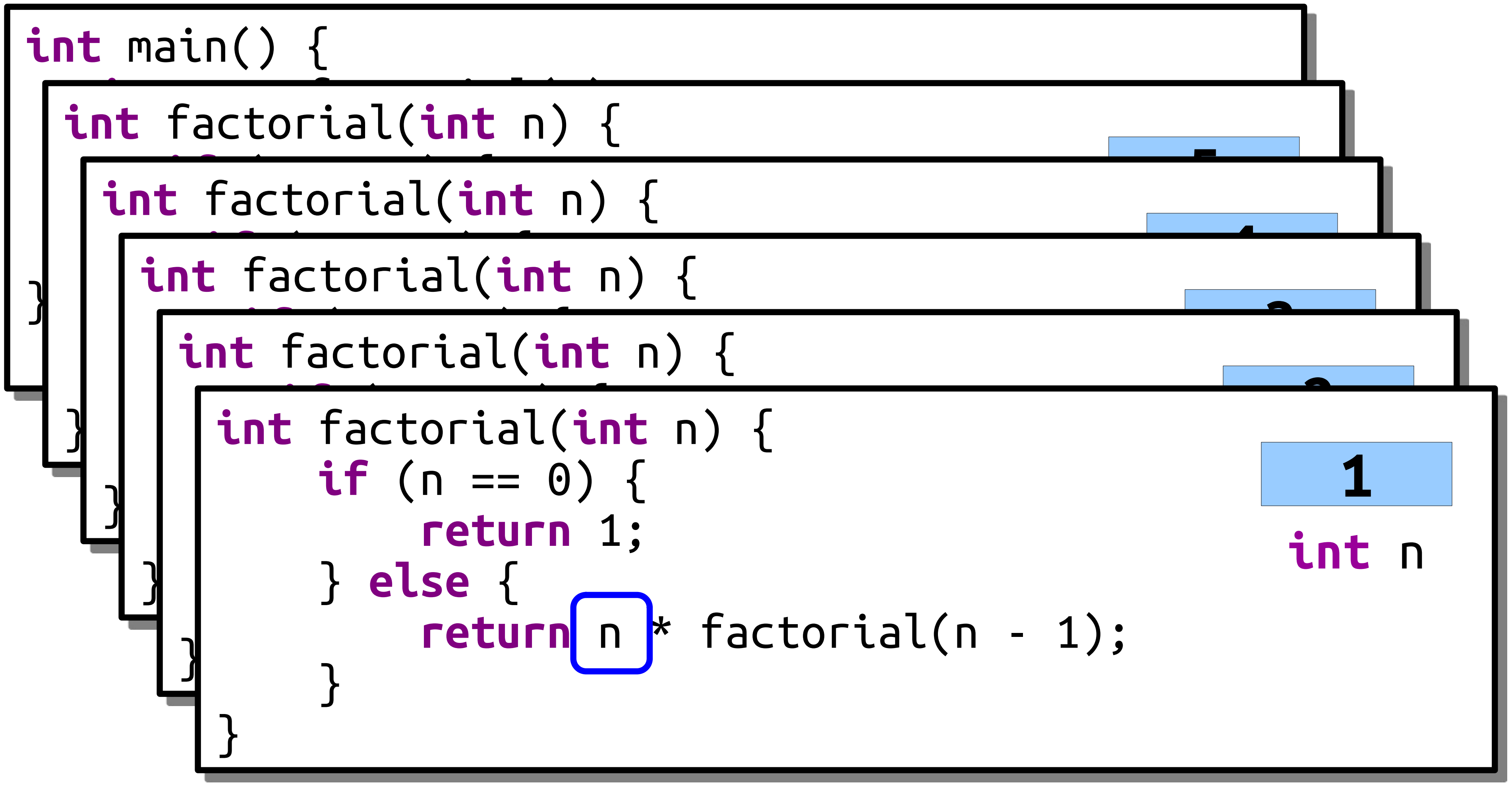
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

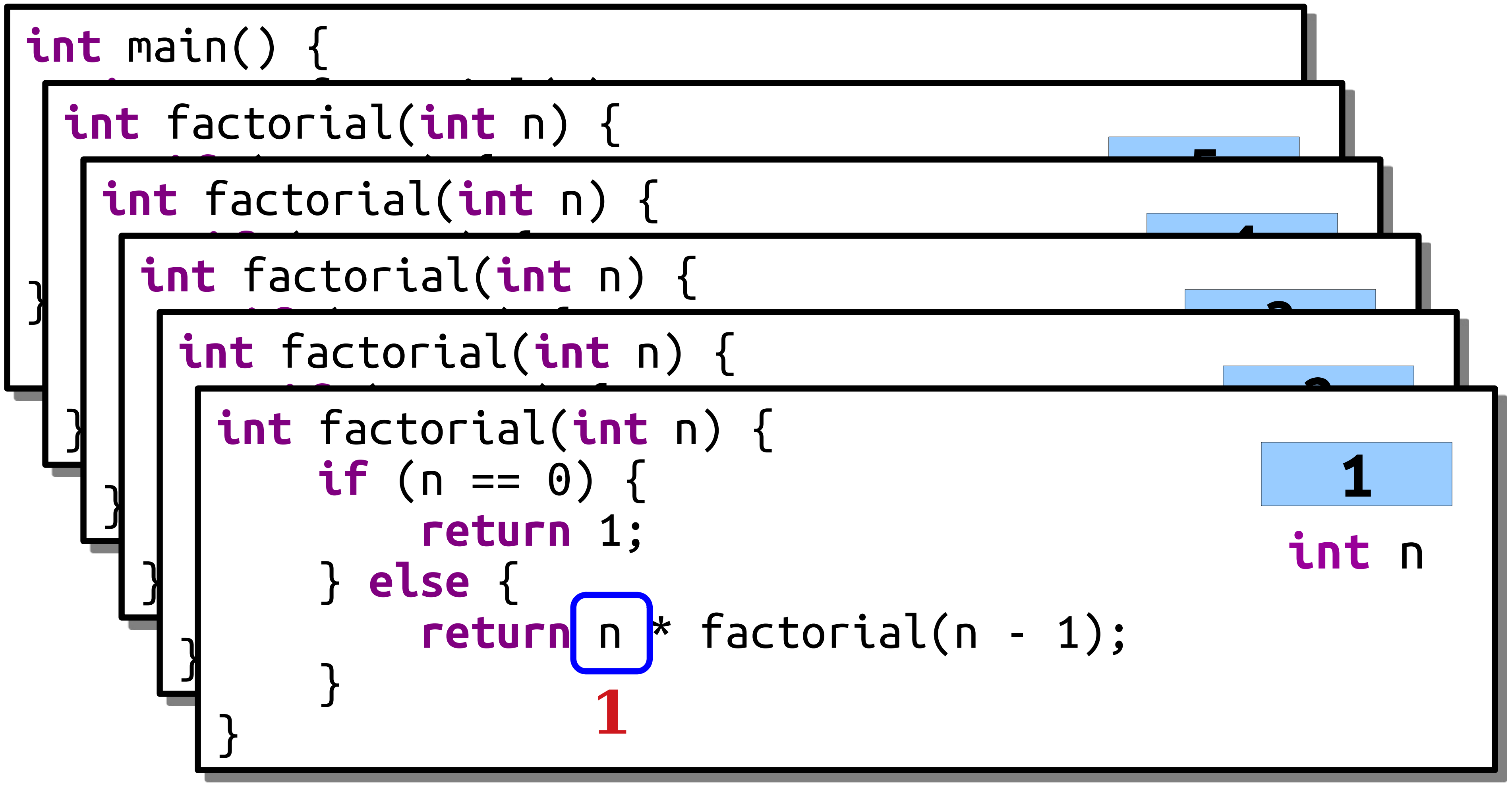
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

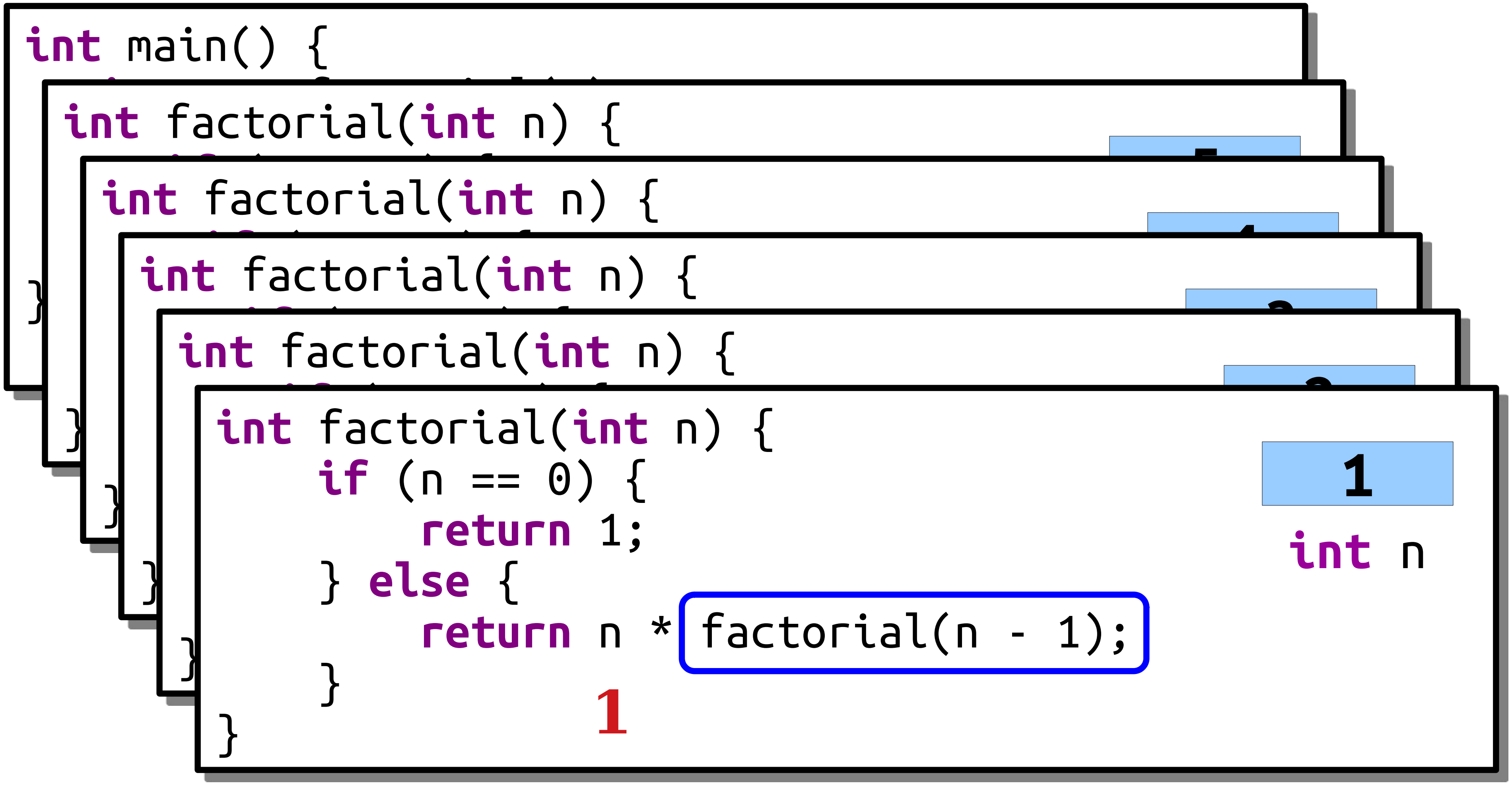
```
int factorial(int n) {
```

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

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

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

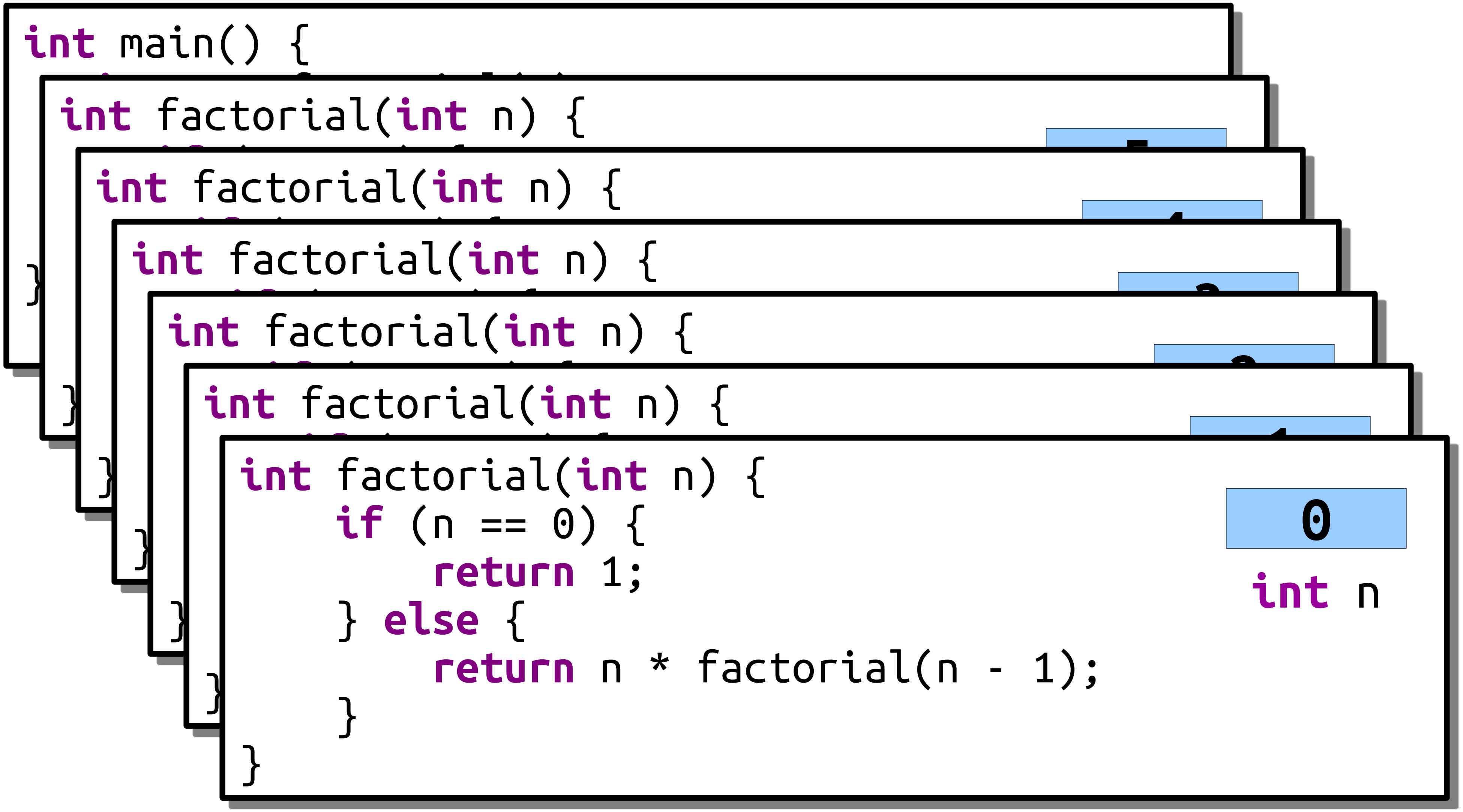
Recursion in Action



```
int main() {
```

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

Recursion in Action



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

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

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

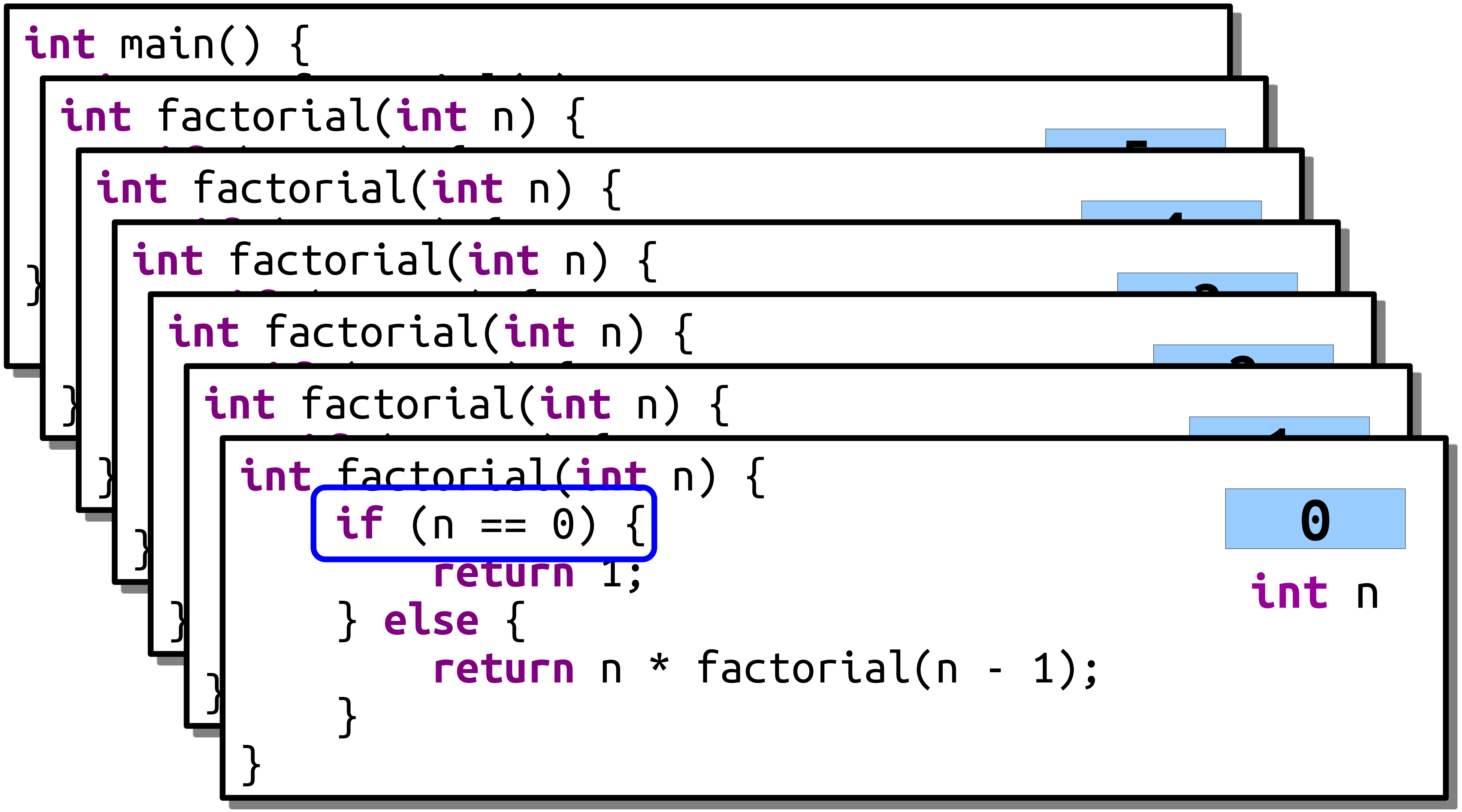
```
int factorial(int n) {
}
```

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

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

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

Recursion in Action



```
int main() {
```

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

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

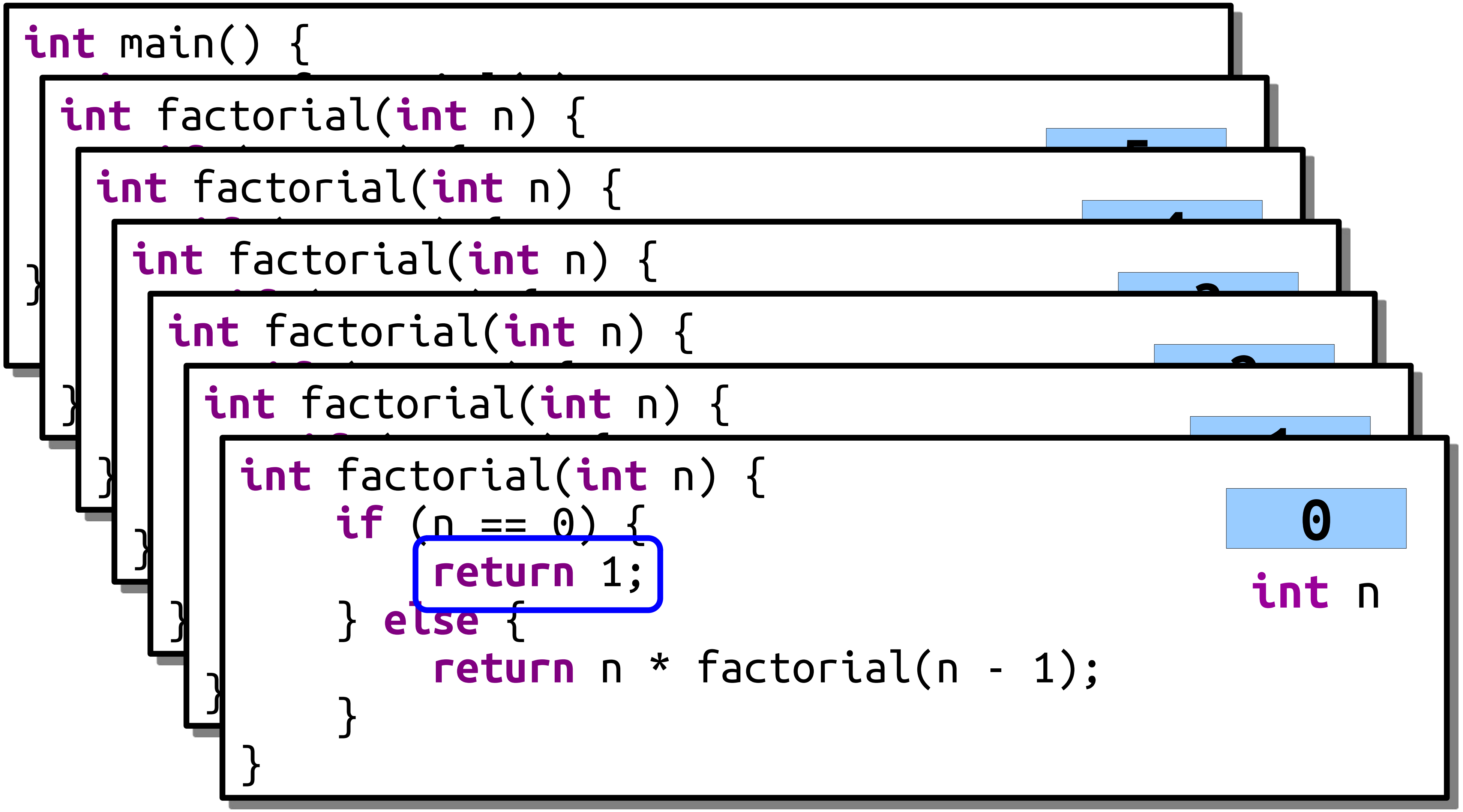
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

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

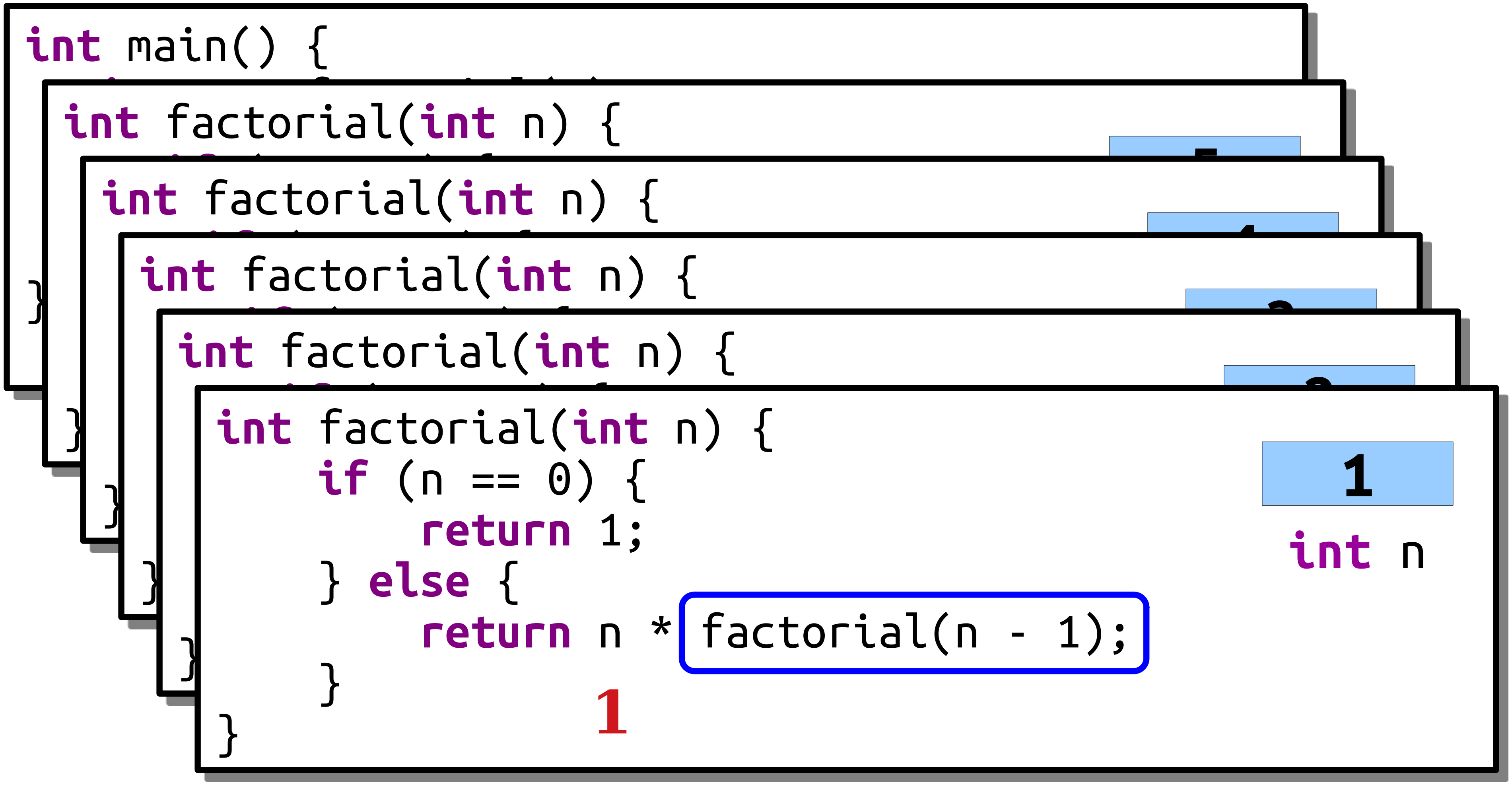
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

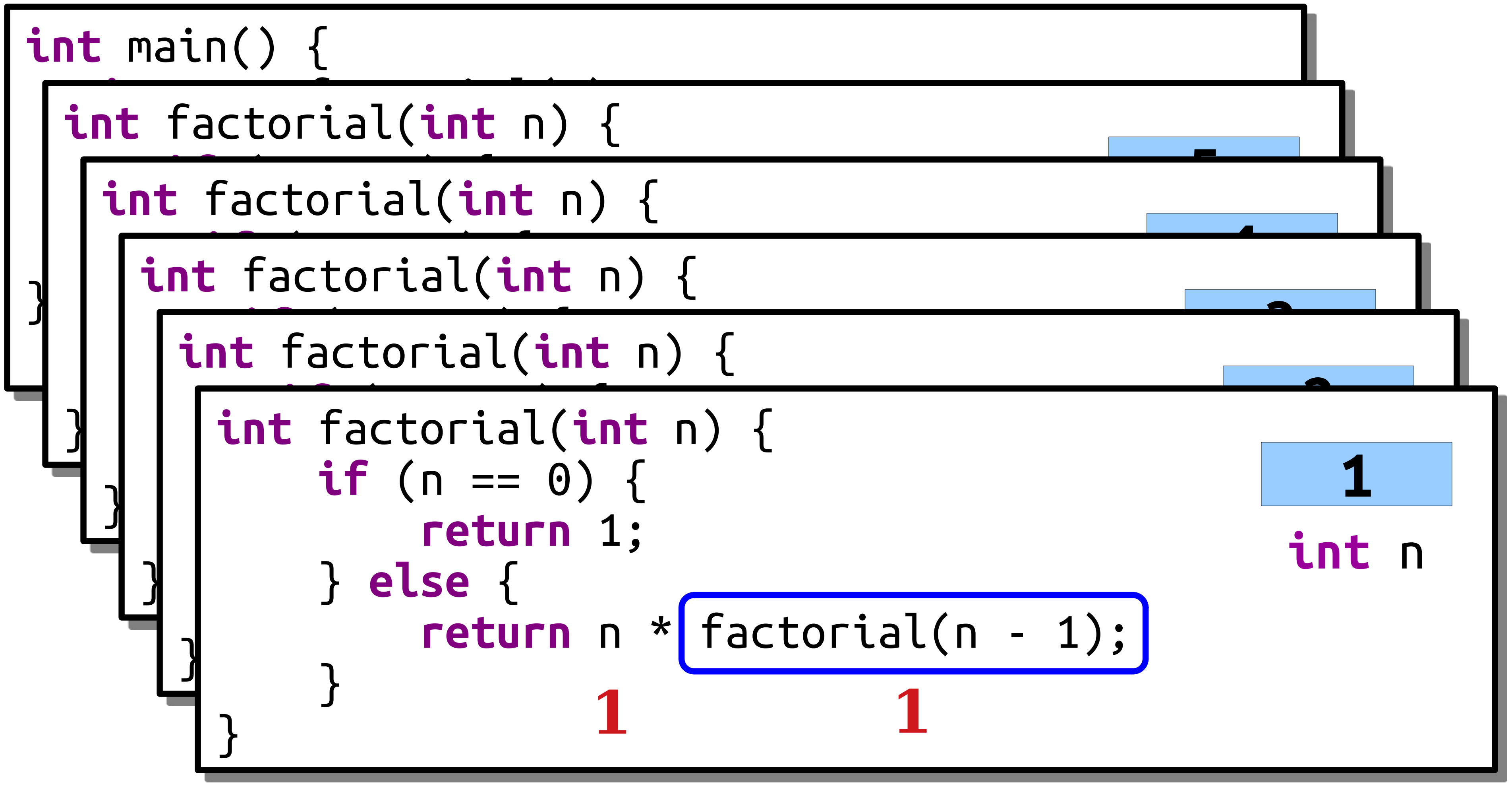
```
int factorial(int n) {
```

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

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

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

Recursion in Action



```
int main() {
```

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

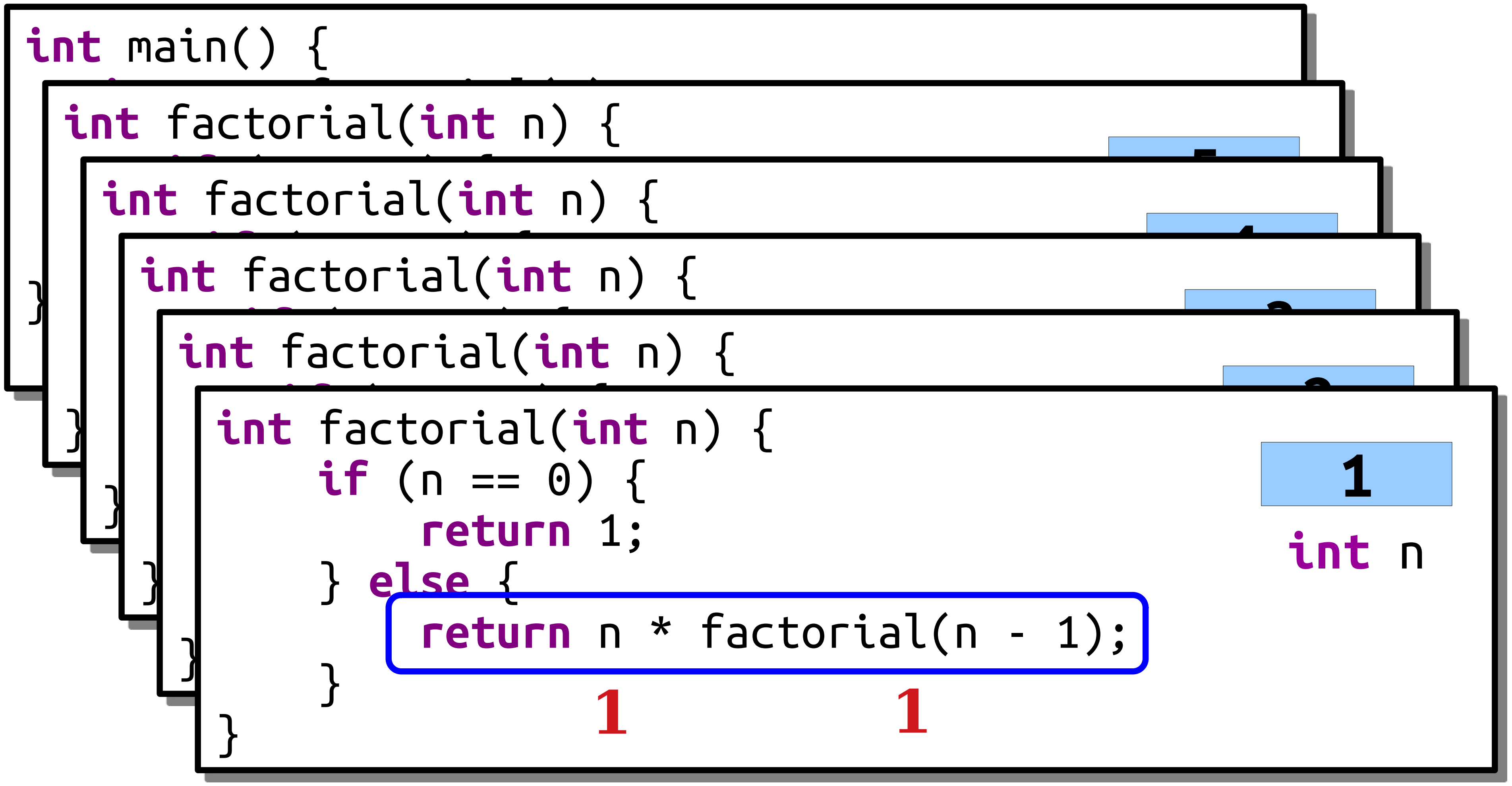
```
int factorial(int n) {
```

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

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

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

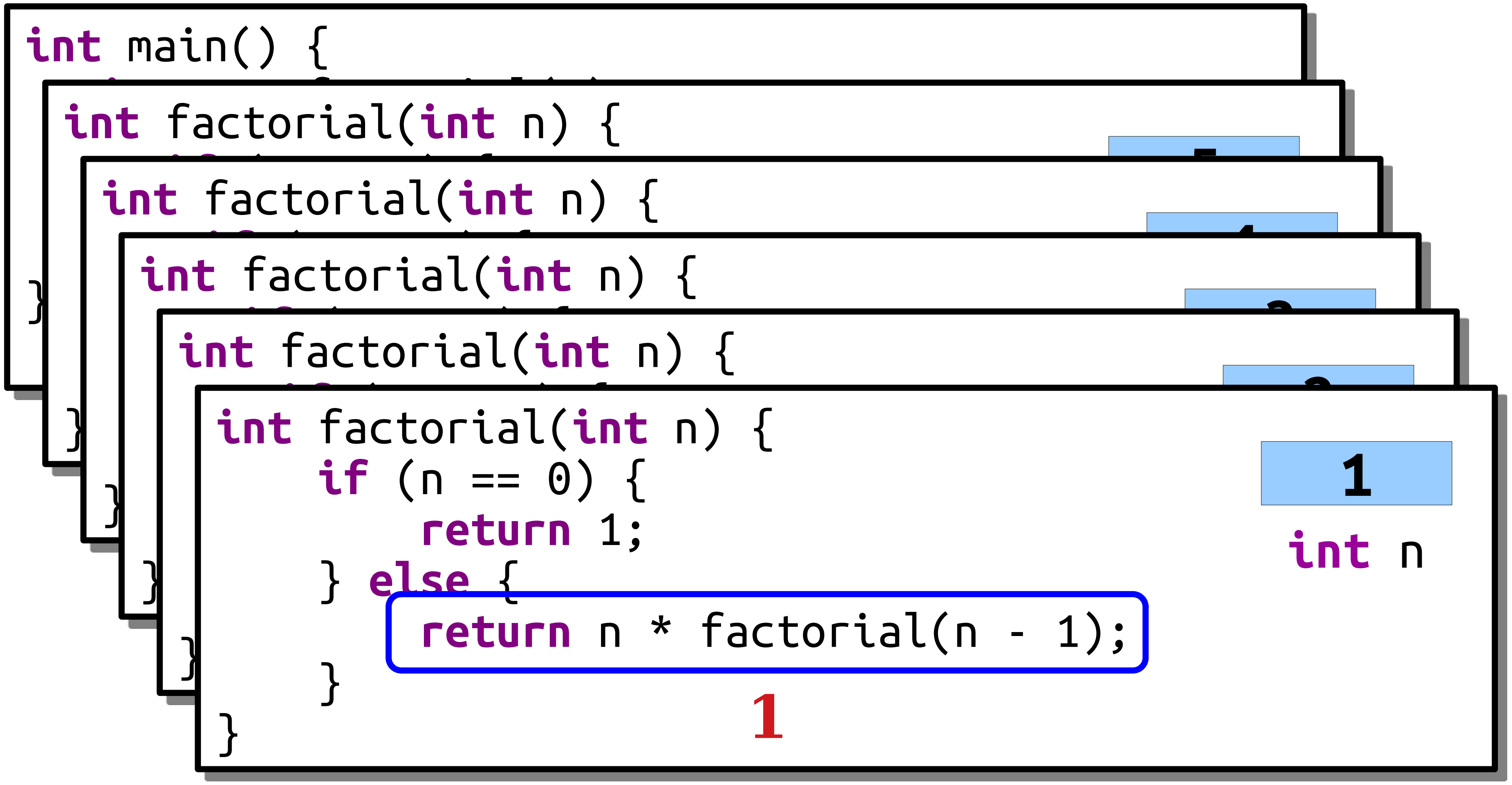
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

1

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

Recursion in Action

```
int main() {
```

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

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

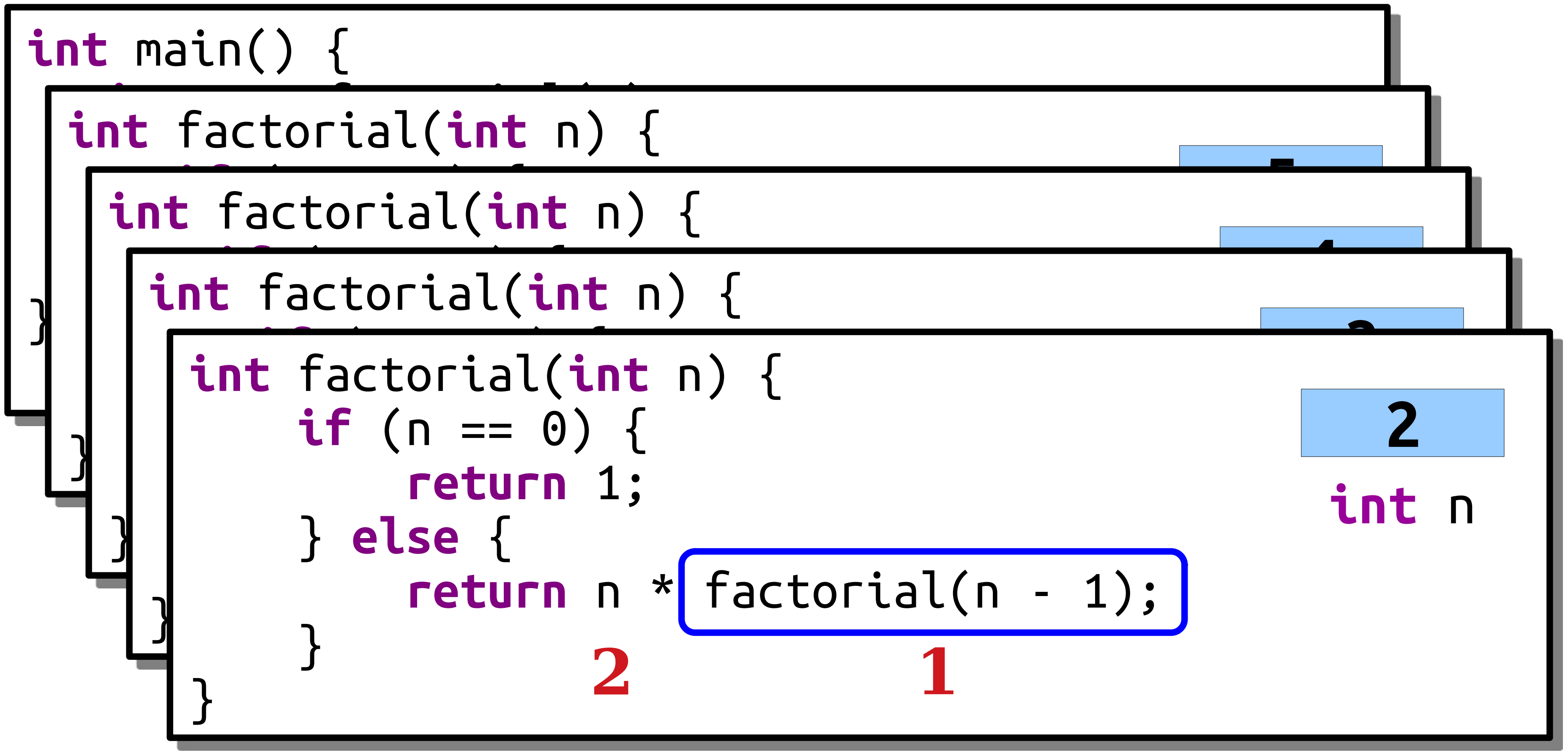
```
int factorial(int n) {
```

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

2

int n

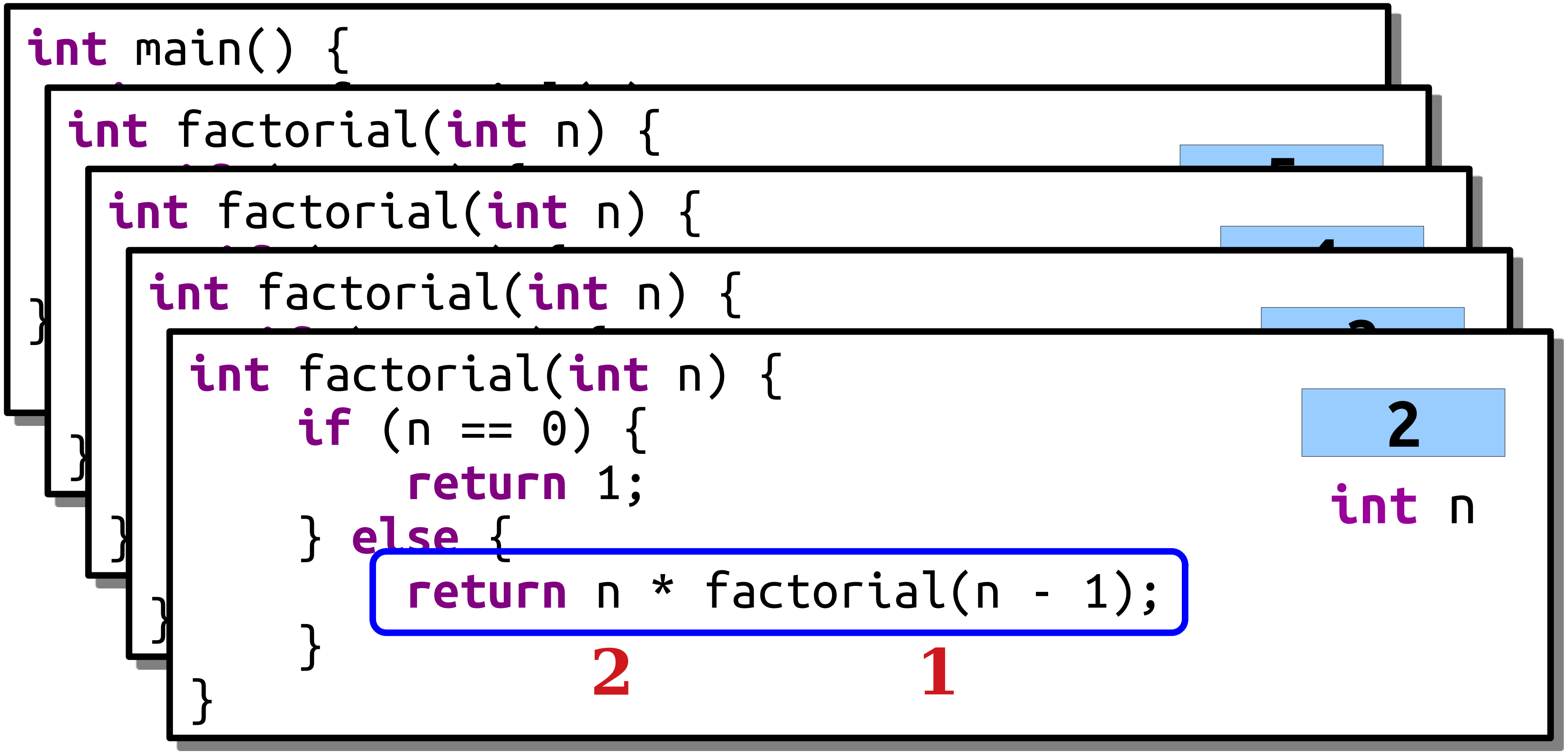
Recursion in Action



```
int main() {
```

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

Recursion in Action



```
int main() {
```

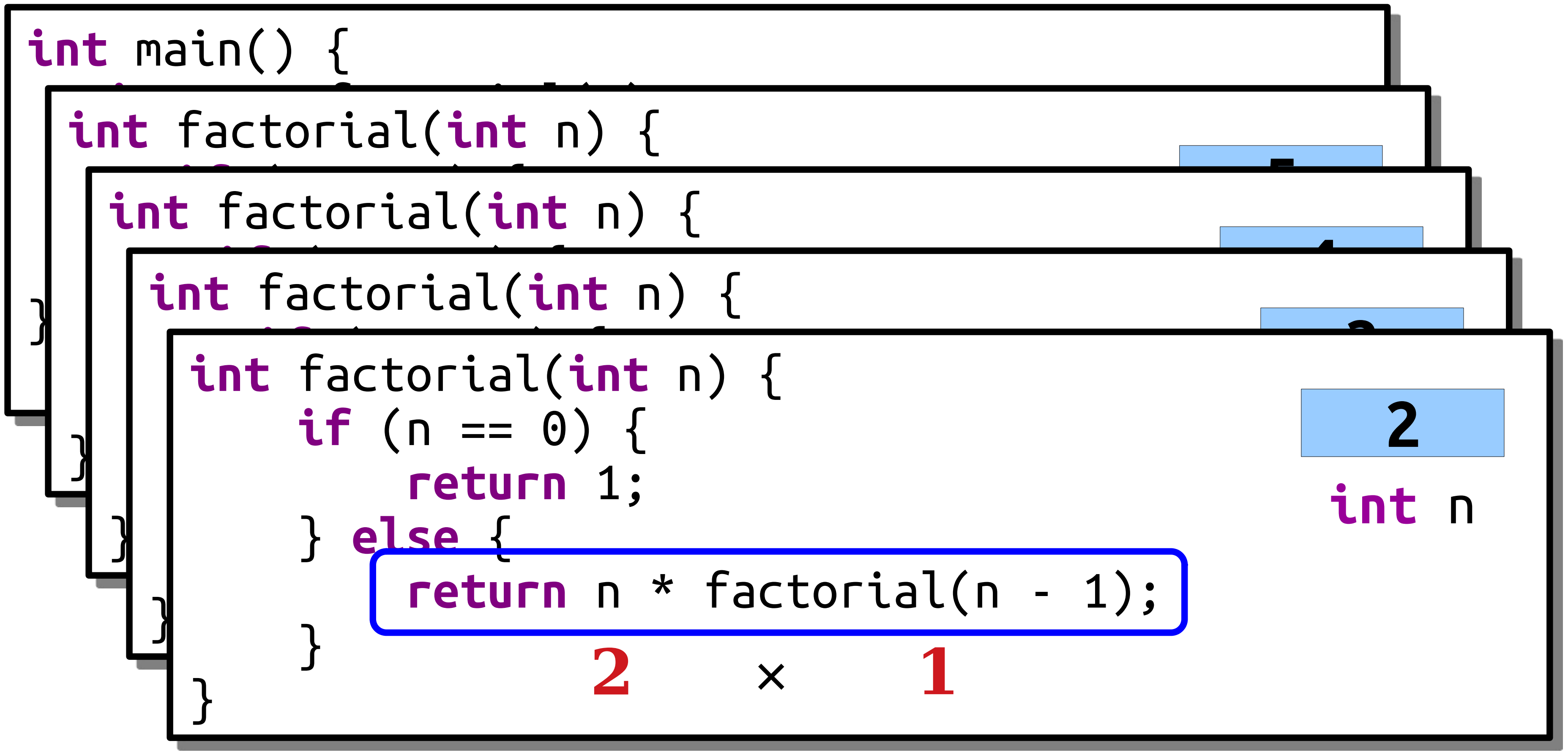
```
int factorial(int n) {
```

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

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

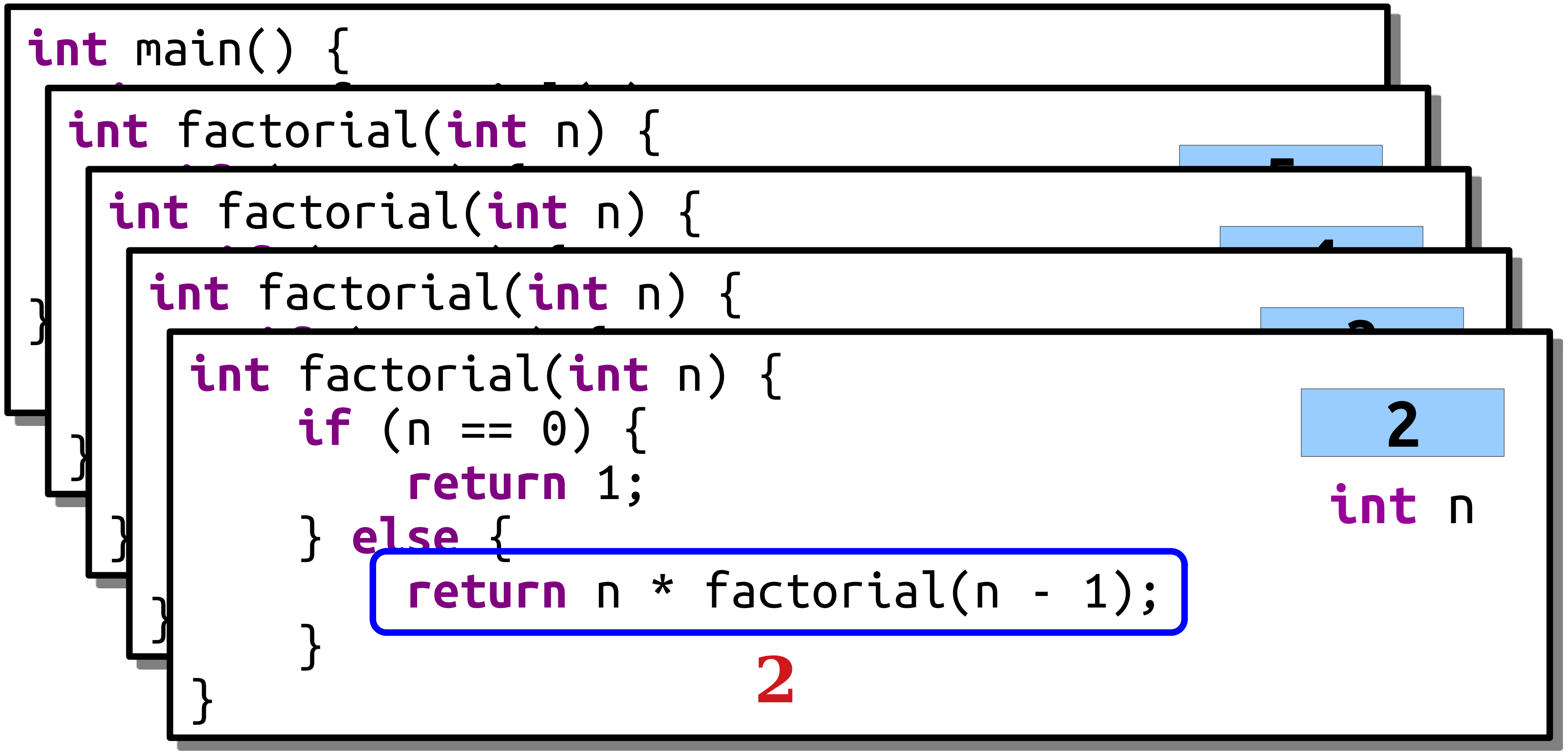
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



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

Recursion in Action



```
int main() {
```

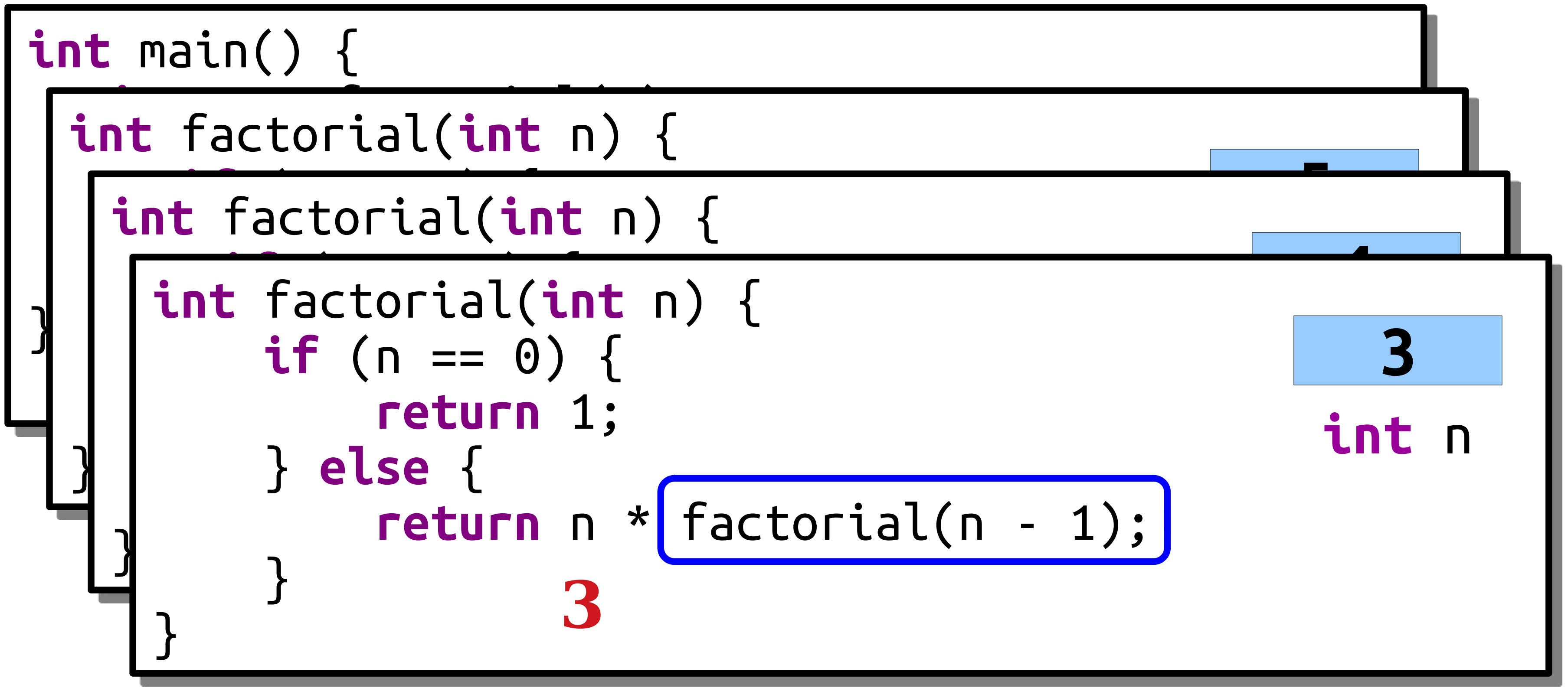
```
int factorial(int n) {
```

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

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

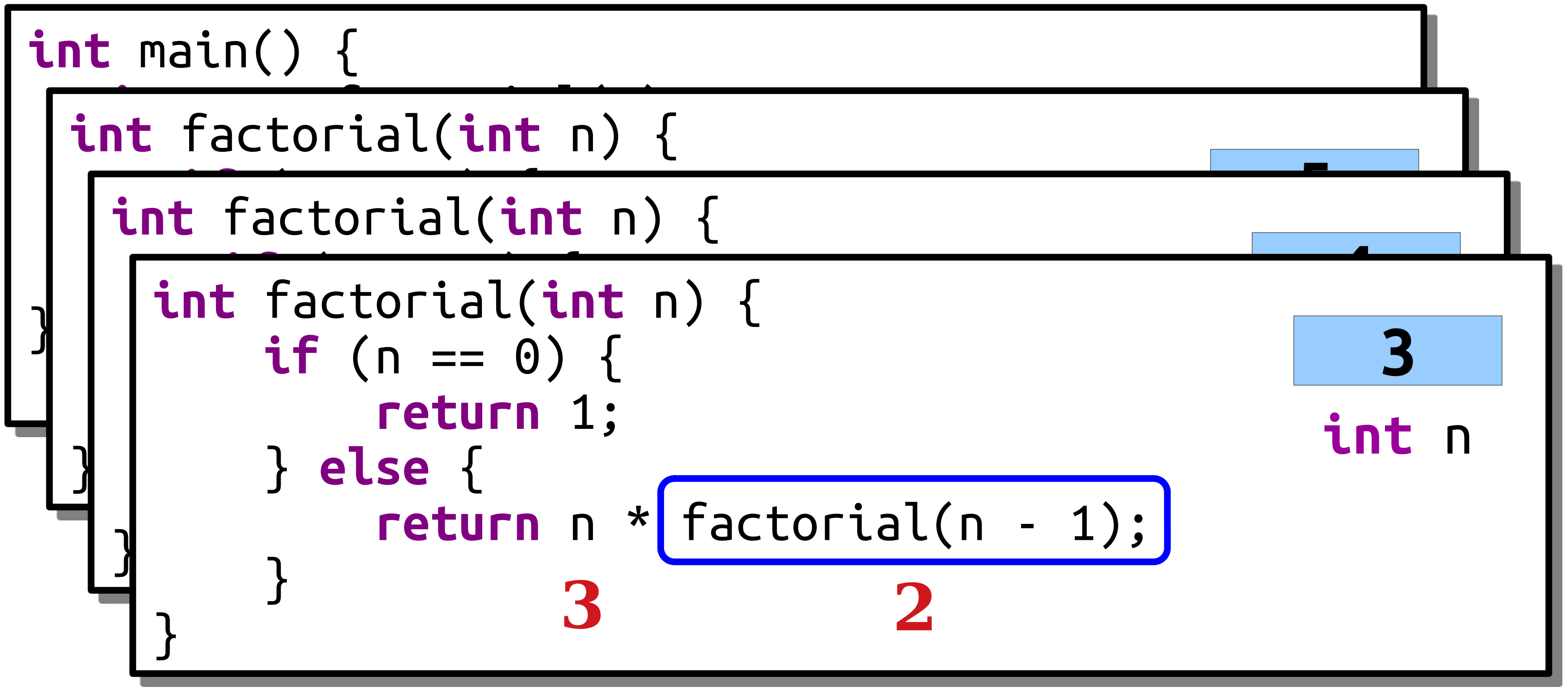
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



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

Recursion in Action



```
int main() {
```

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

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

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

Recursion in Action

```
int main() {
```

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

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

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

$\boxed{3}$

`int n`

3 2

Recursion in Action

```
int main() {
```

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

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

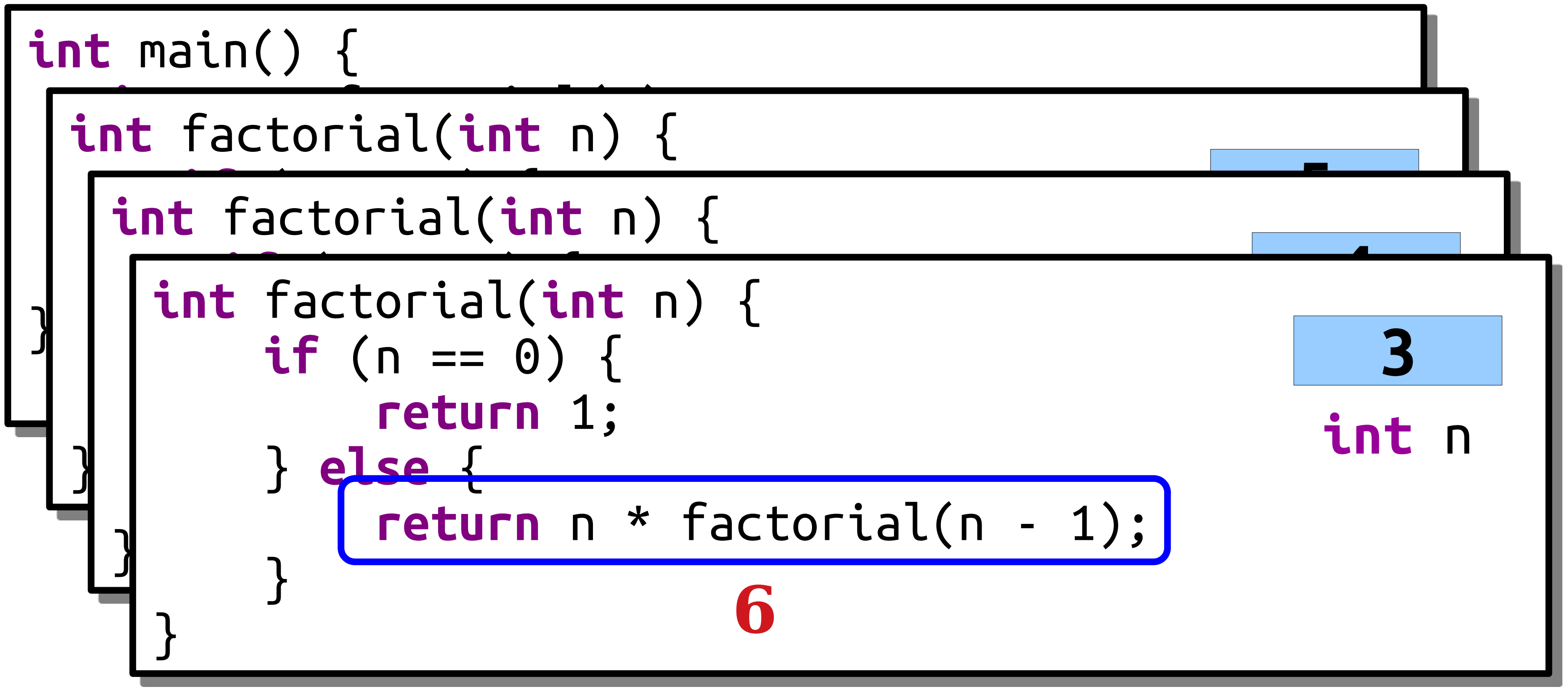
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

3 × 2

3

int n

Recursion in Action



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

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

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

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

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {

}
```

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

4

int n

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {

}
```

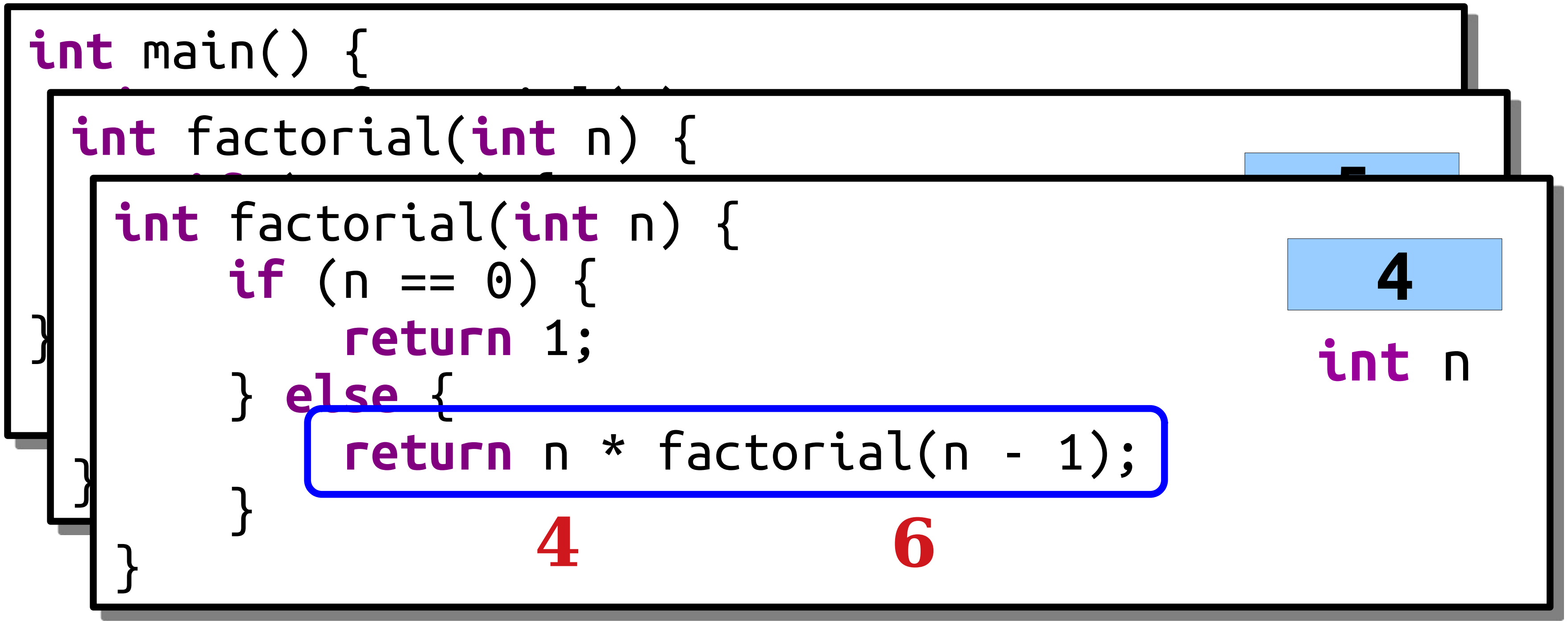
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

4

int n

4 **6**

Recursion in Action

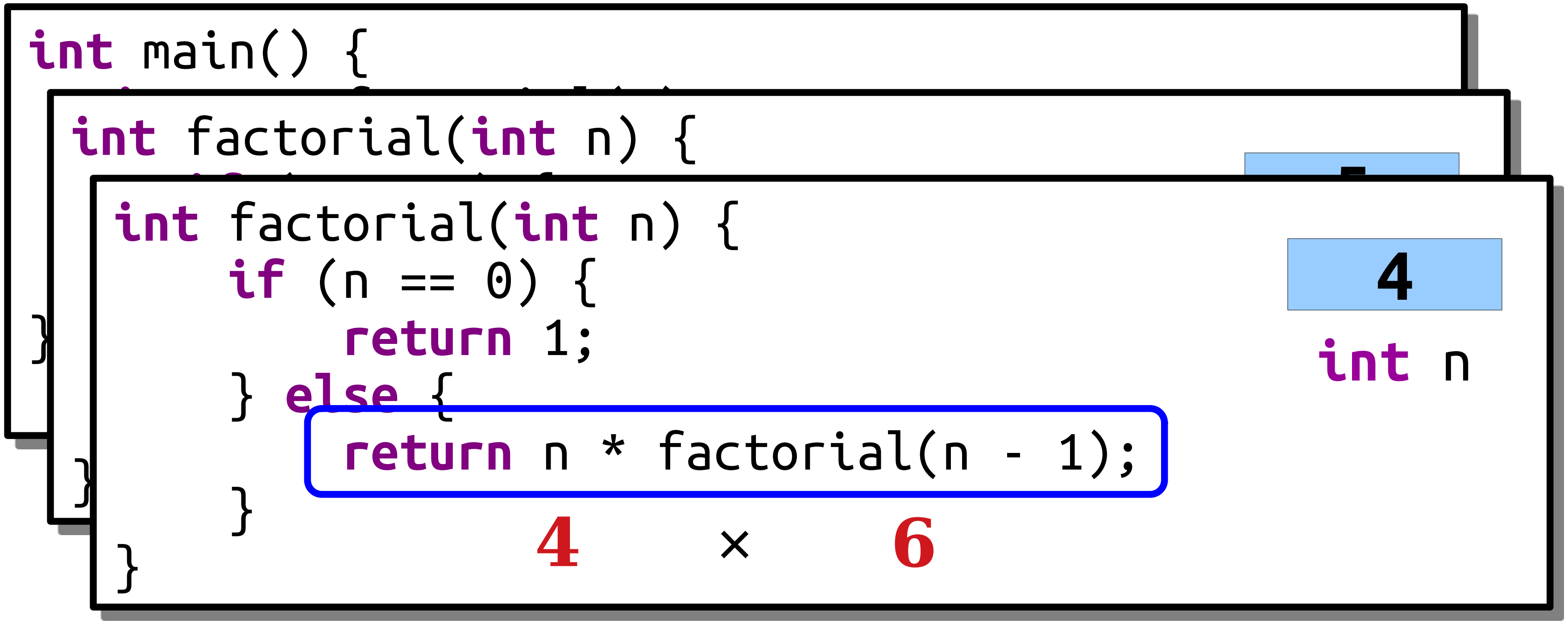


```
int main() {
}
```

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

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

Recursion in Action



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

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

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

Recursion in Action

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

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

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

24

4

int n

Recursion in Action

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

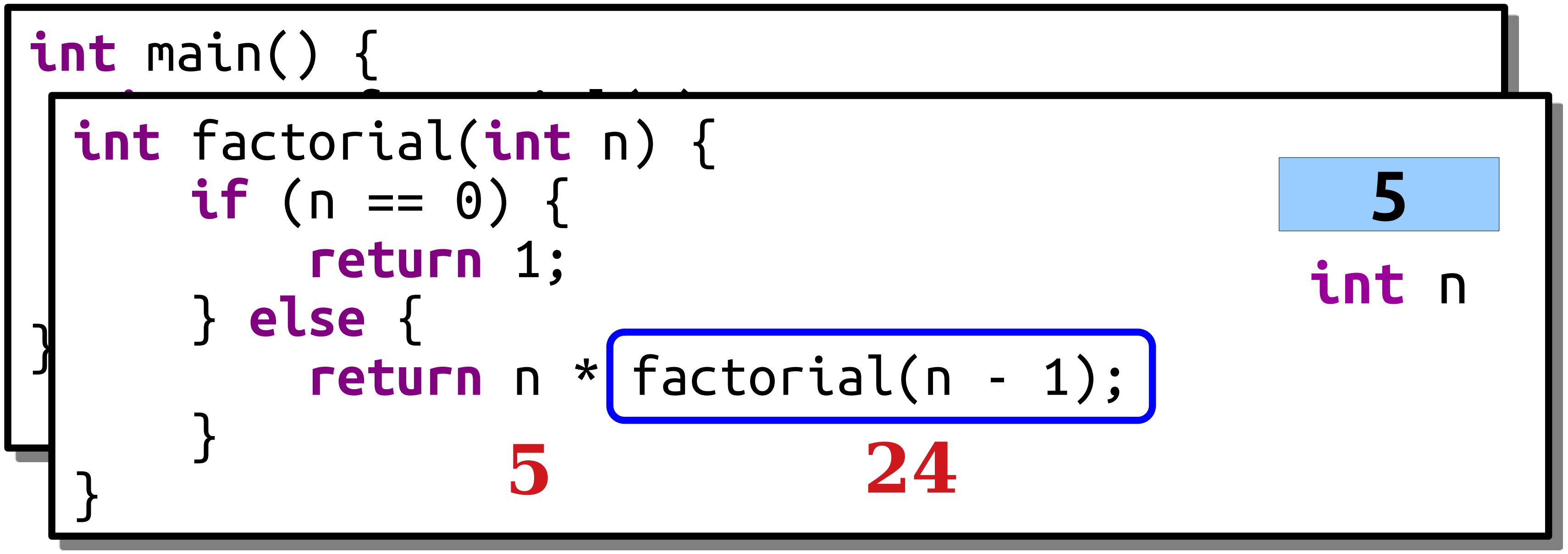
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

5

int n

5

Recursion in Action



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

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

Recursion in Action

```
int main() {

}
```

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

5 **24**

5
int n

Recursion in Action

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

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

5 × **24**

5

int n

Recursion in Action

```
int main() {

}
```

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

5

int n

120

Recursion in Action

```
int main() {
    int nFact = factorial(5);
    cout << "5! = " << nFact << endl;

    return 0;
}
```

Recursion in Action

```
int main() {
    int nFact = factorial(5);
    cout << "5! = " << nFact << endl;

    return 0;
}
```

```
120
int nFact
```

Thinking Recursively

- Solving a problem with recursion requires two steps.
- First, determine how to solve the problem for simple cases.
  - This is called the ***base case***.
- Second, determine how to break down larger cases into smaller instances.
  - This is called the ***recursive step***.

Recap from Today

- The C++ compiler reads from the top of the program to the bottom. You cannot call a function that hasn't either been prototyped or defined before the call site.
- Each time you call a function, C++ gives you a fresh copy of all the local variables in that function. Those variables are independent of any other variables with the same name found elsewhere.
- You can split a number into "everything but the last digit" and "the last digit" by dividing and modding by 10.
- A ***recursive function*** is one that calls itself. It has a ***base case*** to handle easy cases and a ***recursive step*** to turn bigger versions of the problem into smaller ones.
- Functions can be written both iteratively and recursively.

Your Action Items

- ***Read Chapter 1 and Chapter 2.***
  - We’re still easing into C++. These chapters talk about the basics and the mechanics of function call and return.
- ***Read Chapter 7.***
  - We’ve just started talking about recursion. There’s tons of goodies in that chapter.
- ***Sign up for a Discussion Section.***
  - The link goes out tomorrow afternoon.
- ***Work on Assignment 0.***
  - Just over a third of you are already done! Exciting!

Next Time

- ***Strings and Streams***
  - Representing and Manipulating Text.
  - Recursion on Text.
  - File I/O in C++.
- ***More Recursion***
  - Getting more comfortable with this strategy.