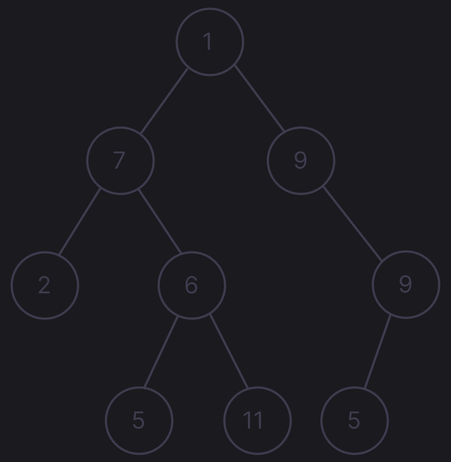
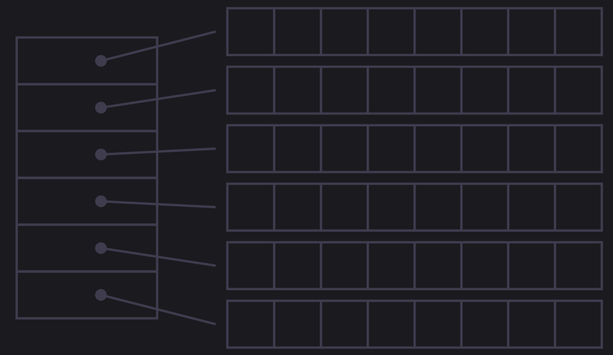
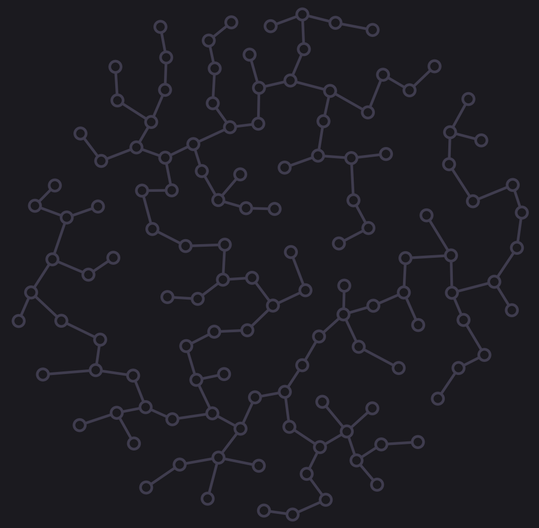
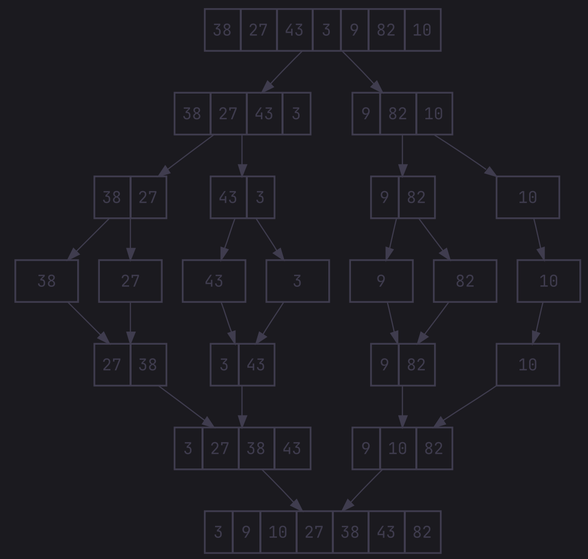




data structures & algorithms

Tutorial 4





Burning questions from last week?

This week's lab

Recursion

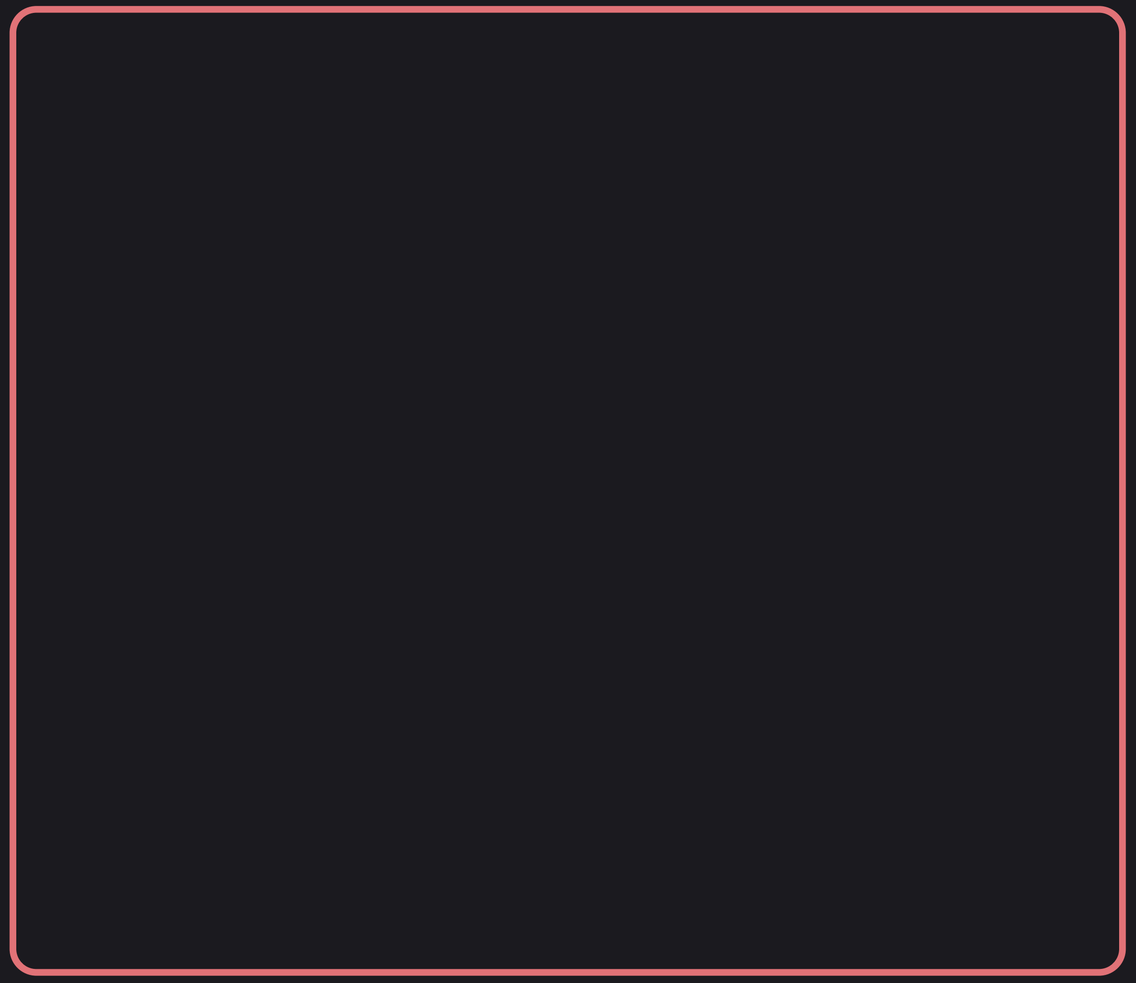
Learning a powerful new technique called **Recursion**!

- How do functions work?
- Recursion
  - Factorial
  - Fibonacci
- Iterators (Finally)

How do functions work?

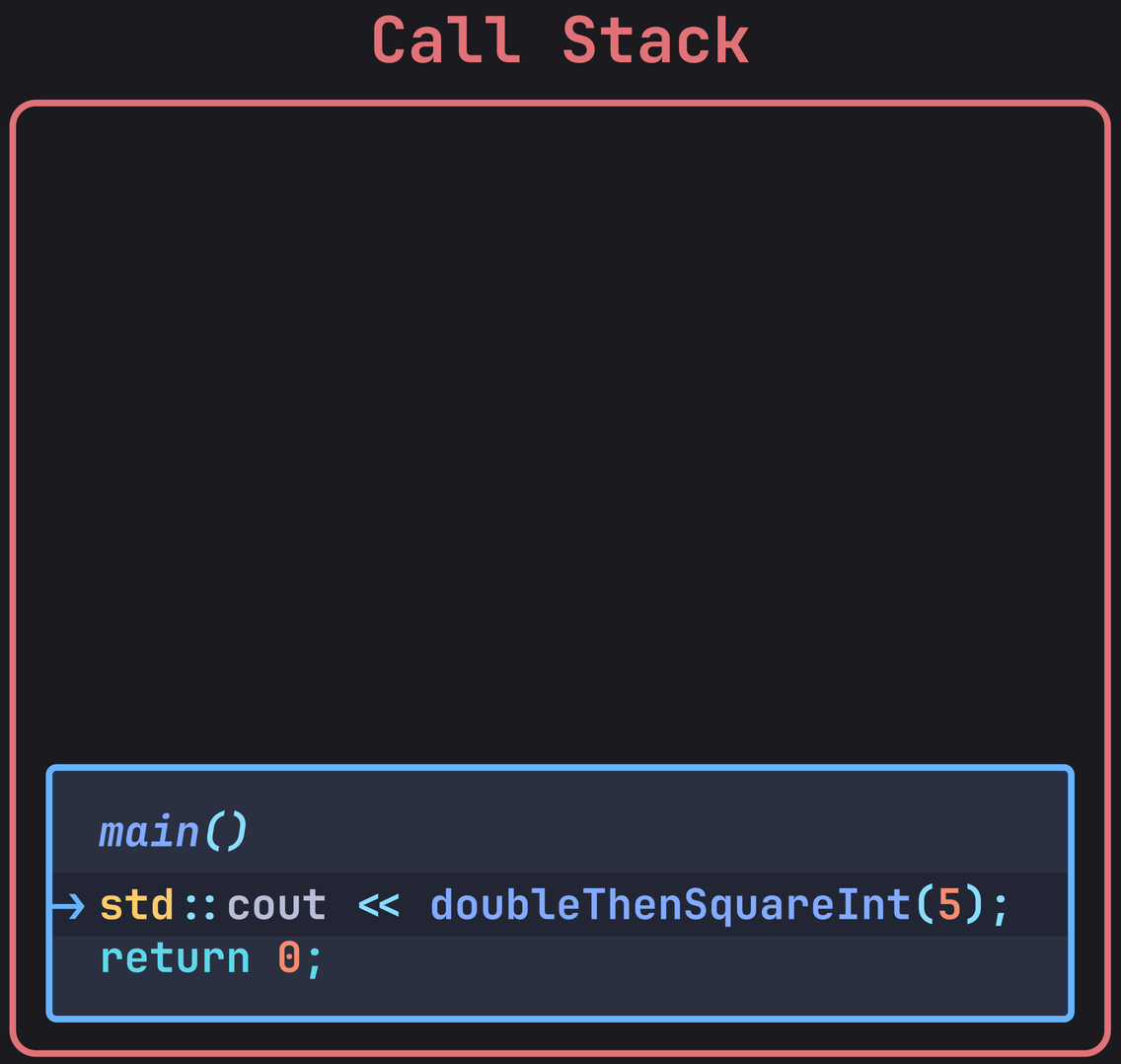
```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```

Call Stack



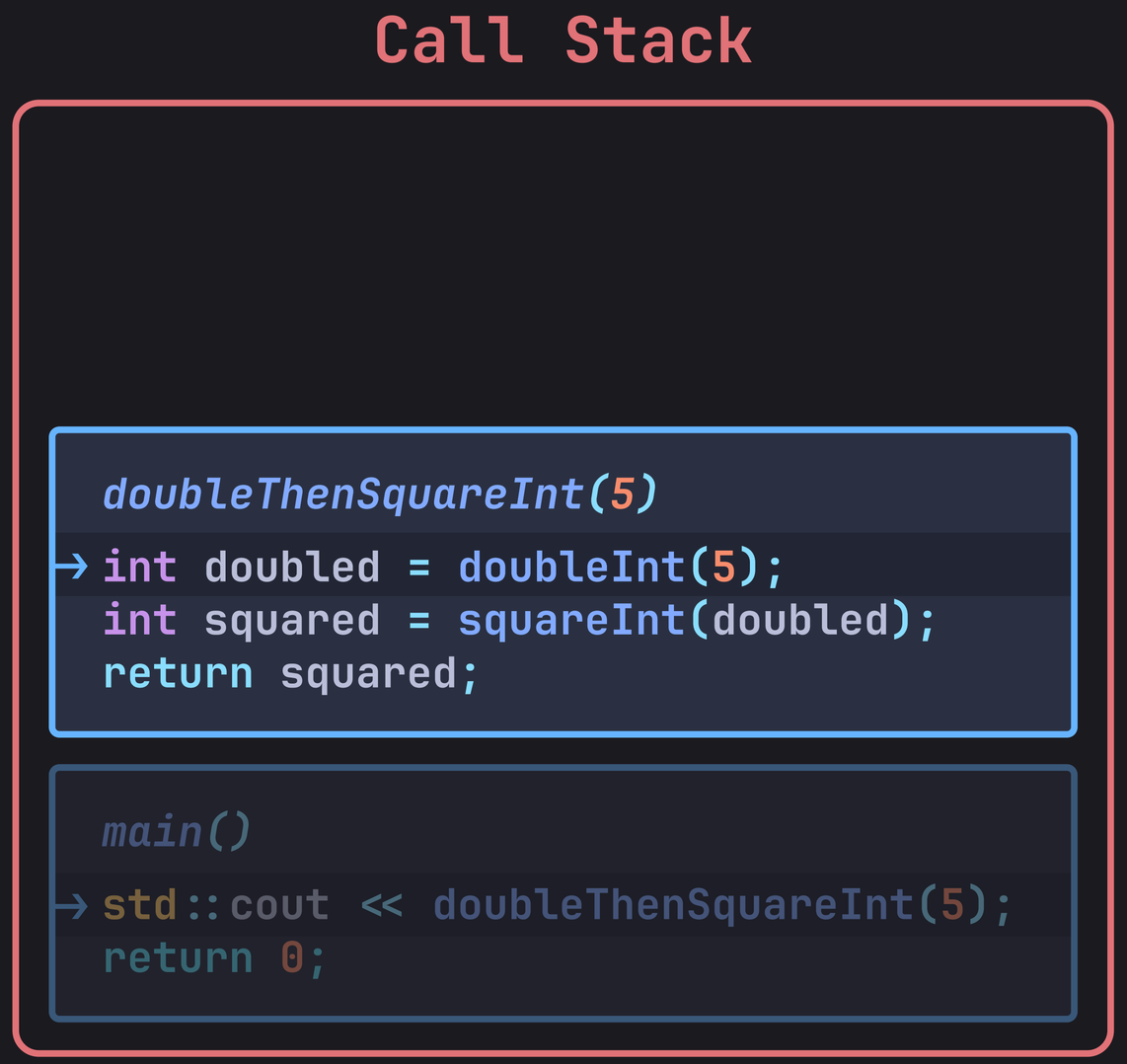
How do functions work?

```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



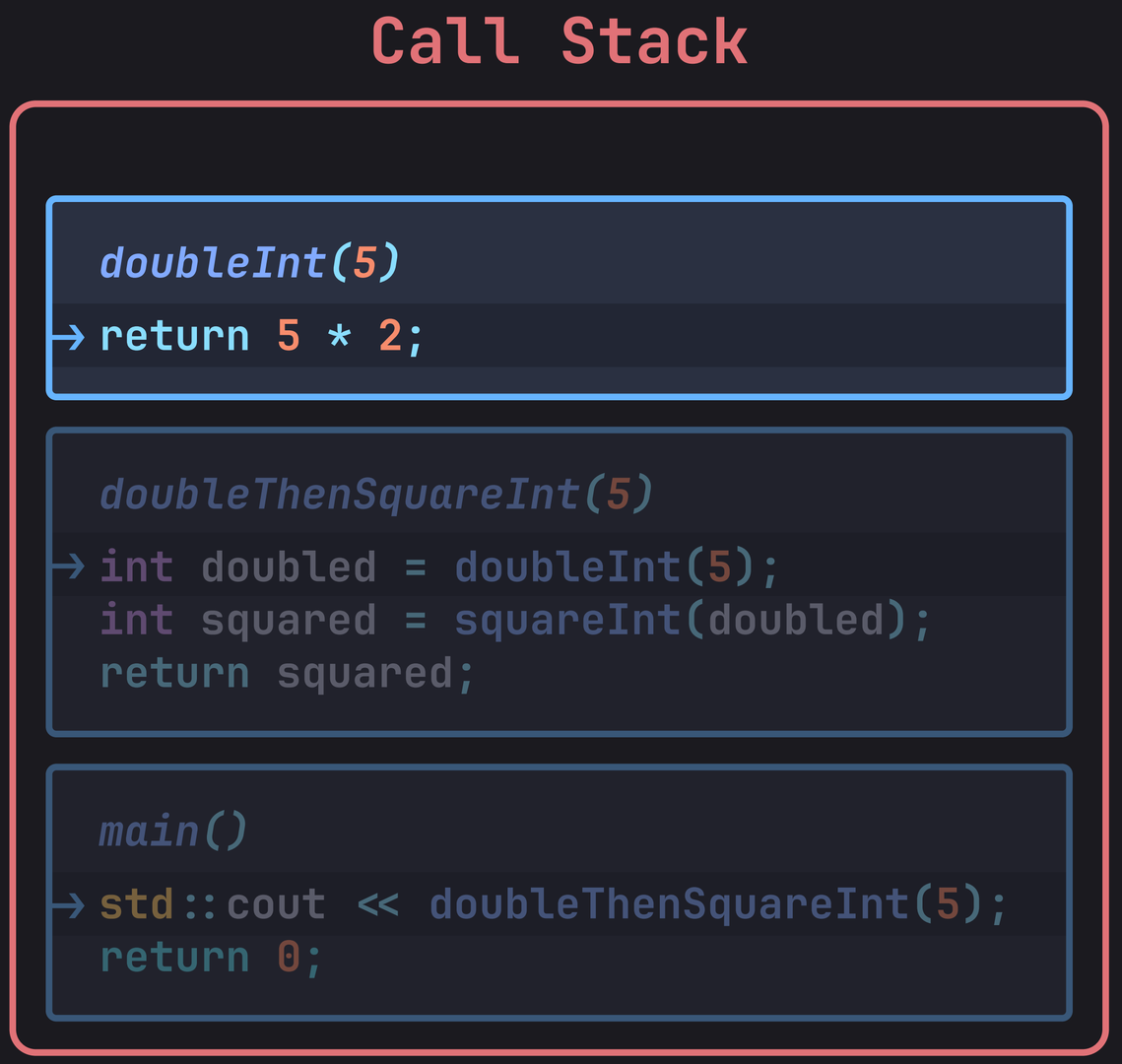
How do functions work?

```cpp
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



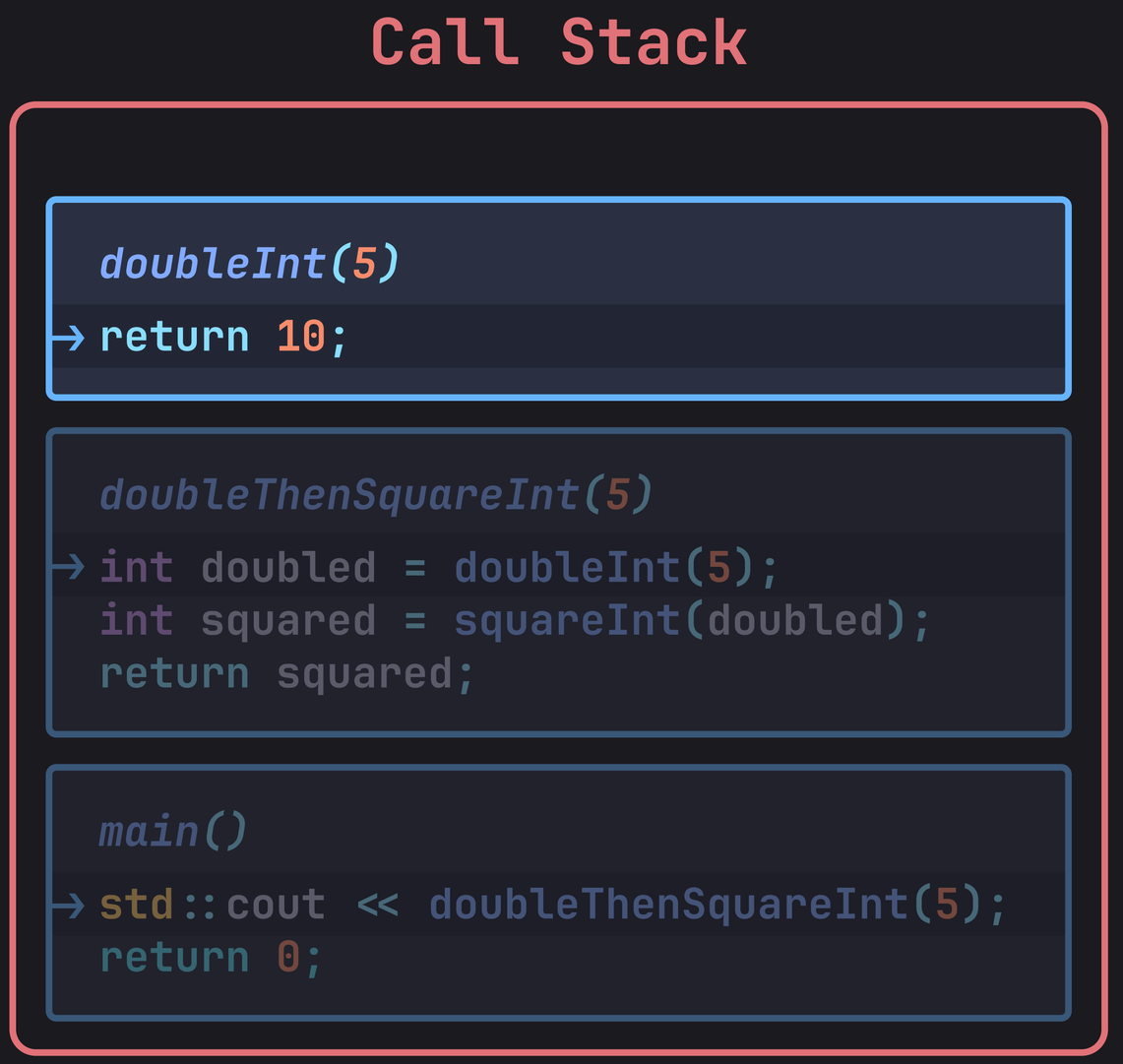
How do functions work?

```cpp
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



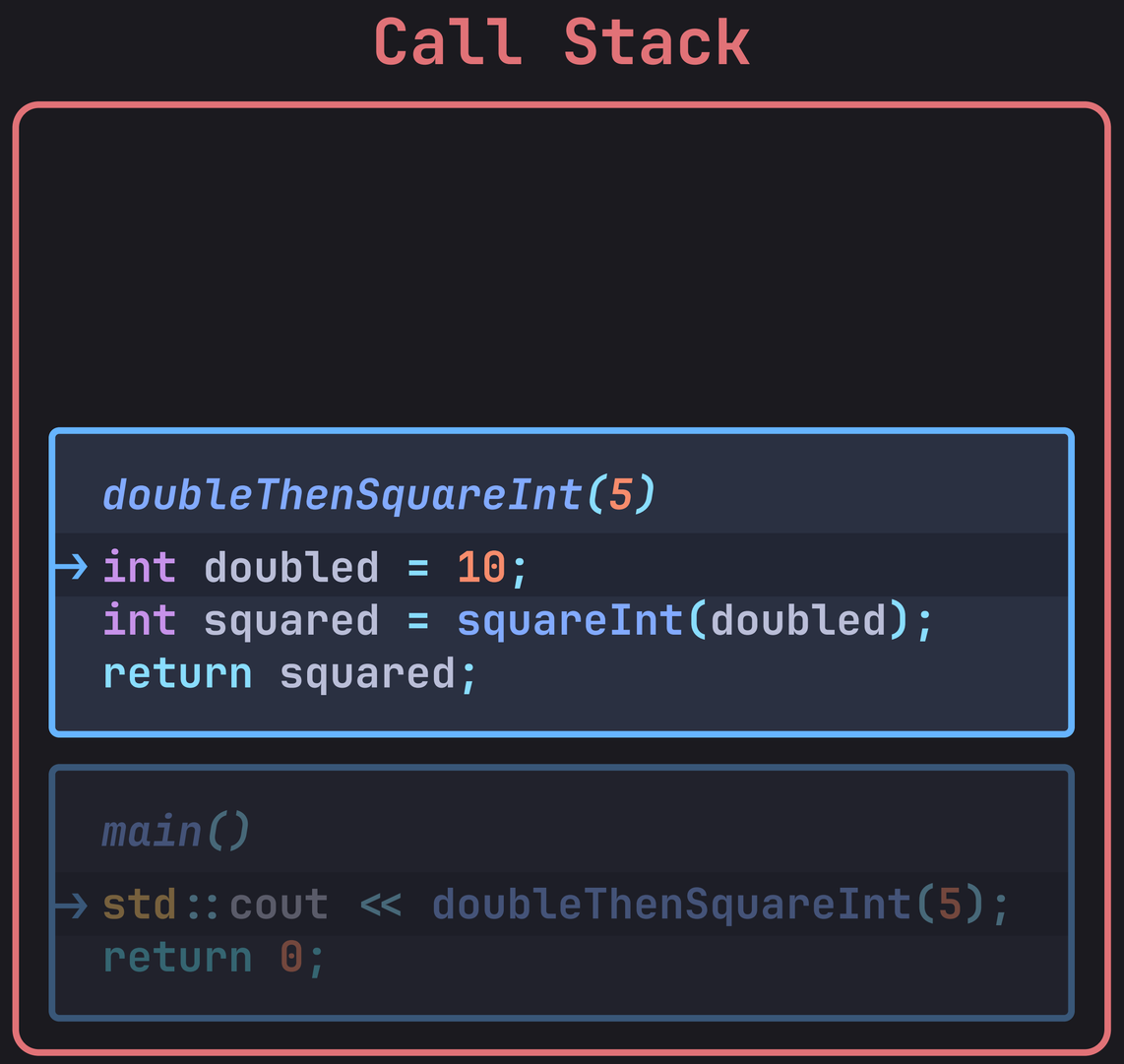
How do functions work?

```cpp
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



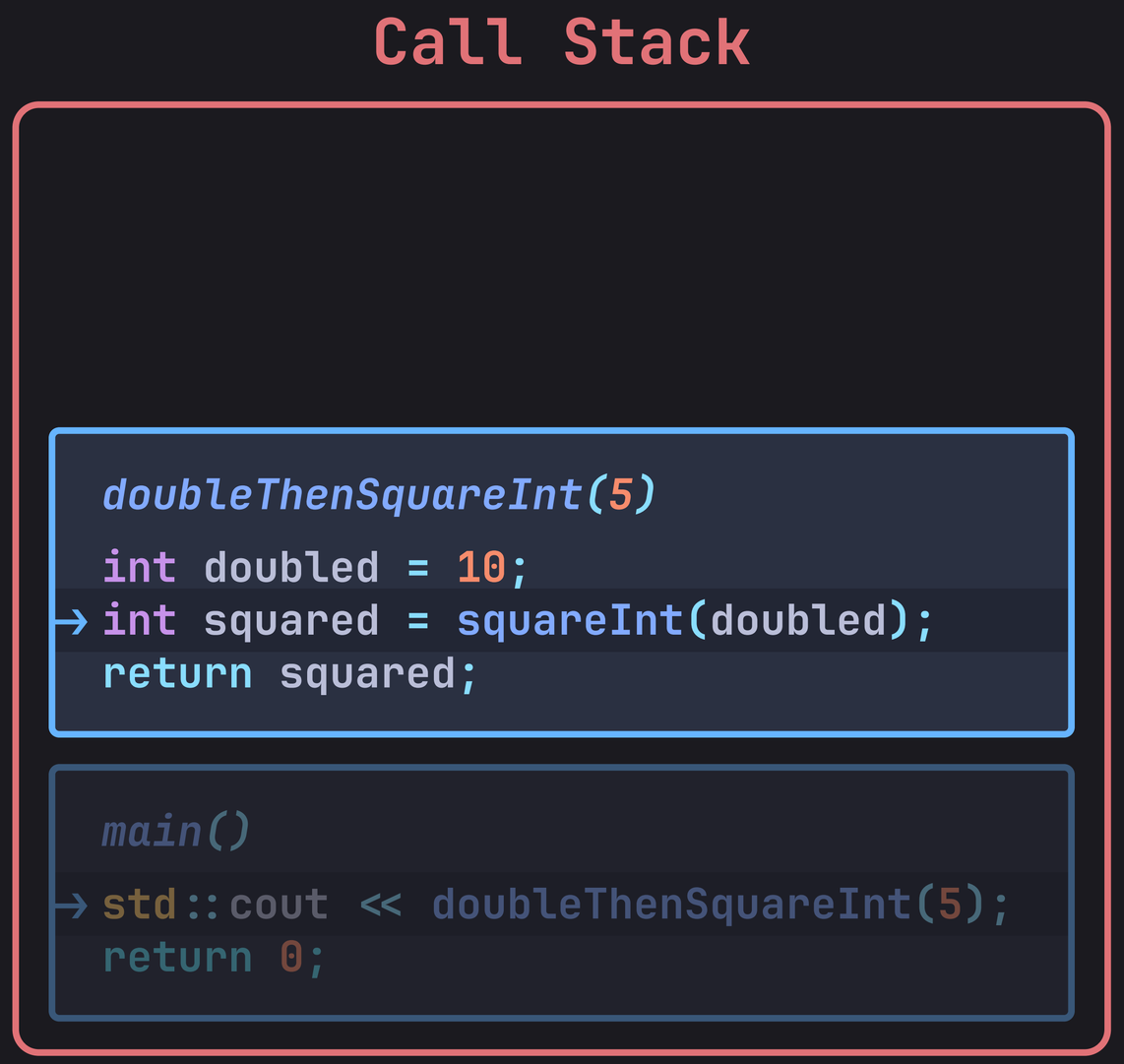
How do functions work?

```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



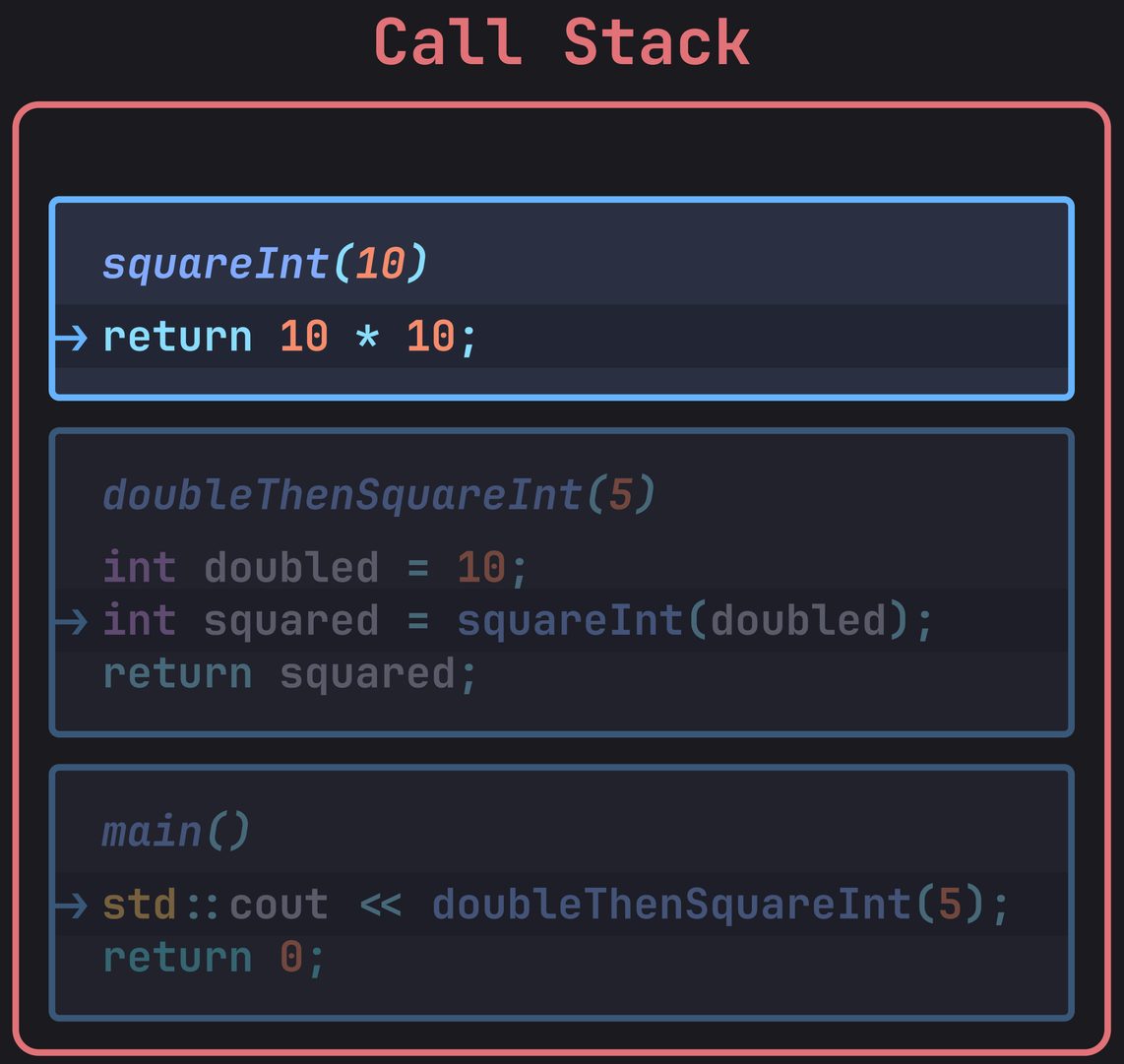
How do functions work?

```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



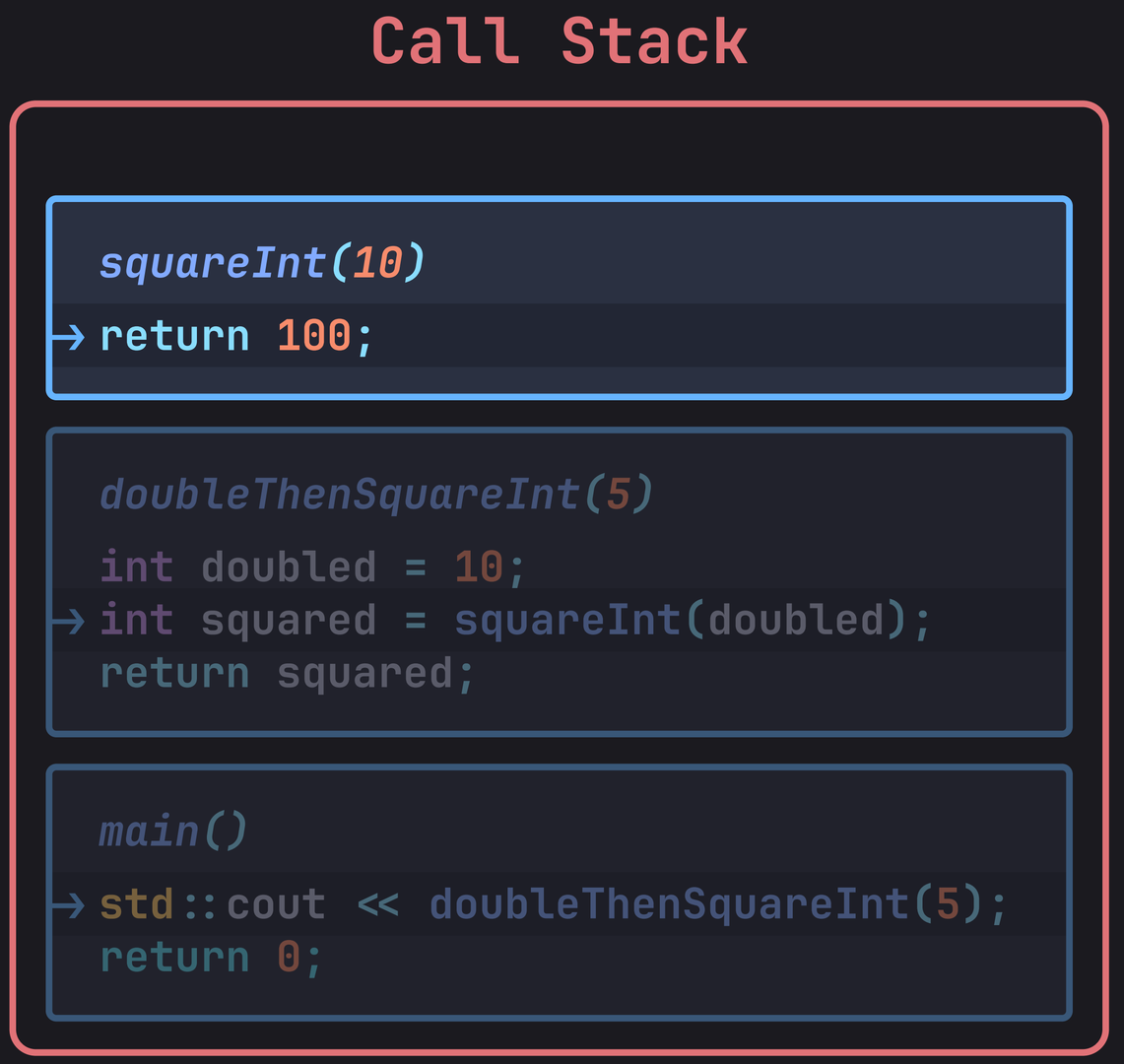
How do functions work?

```cpp
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



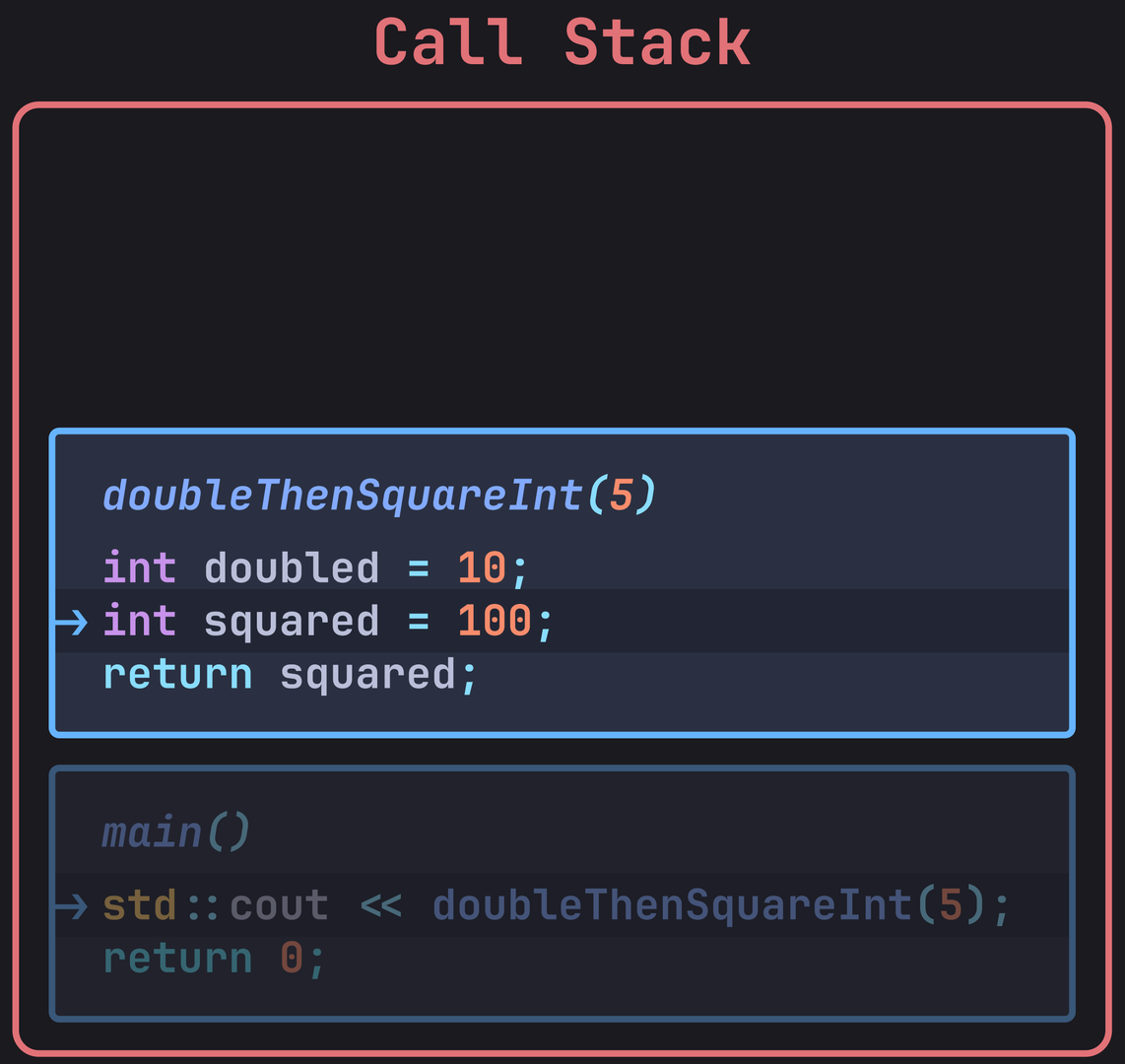
How do functions work?

```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



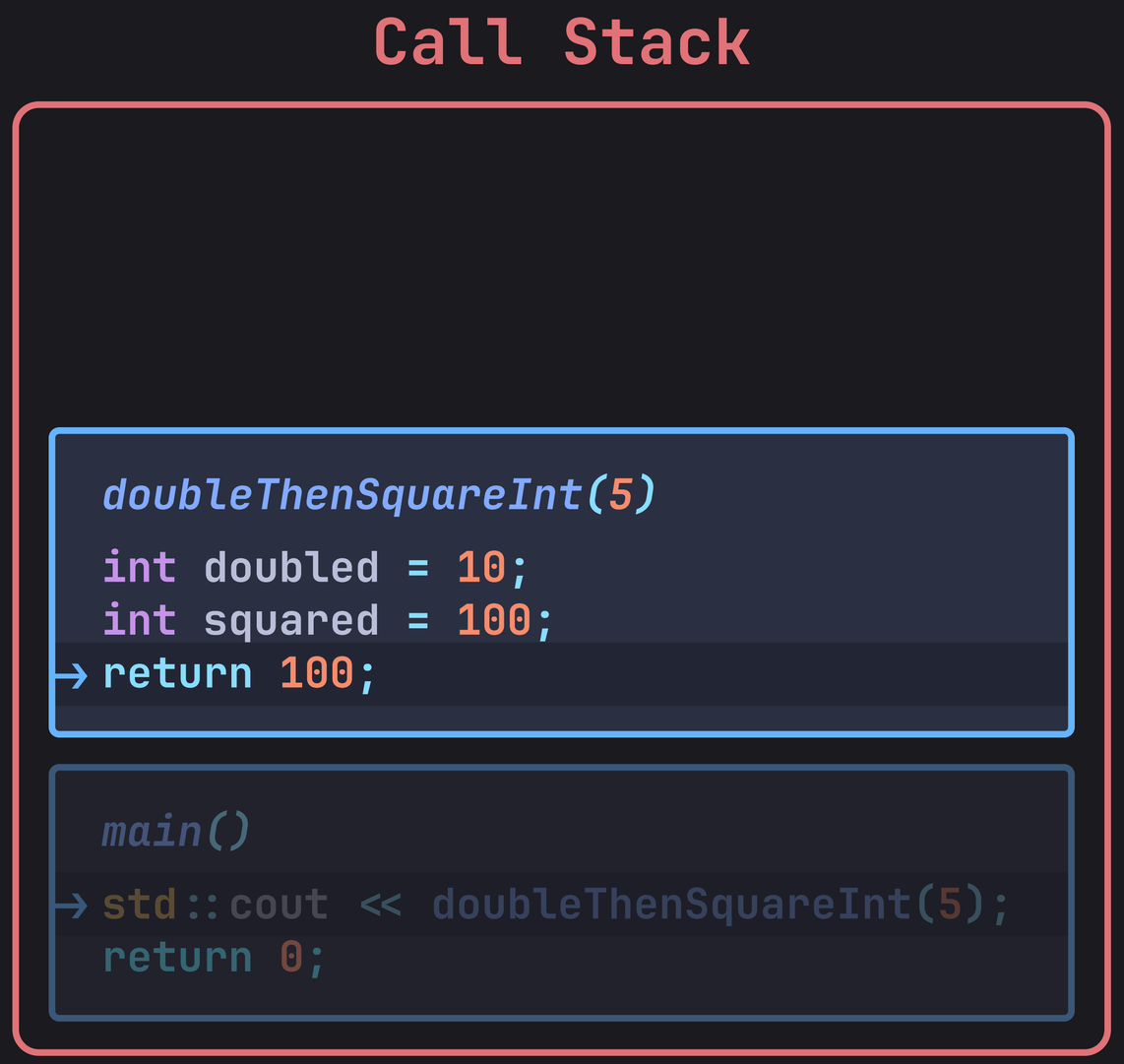
How do functions work?

```cpp
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



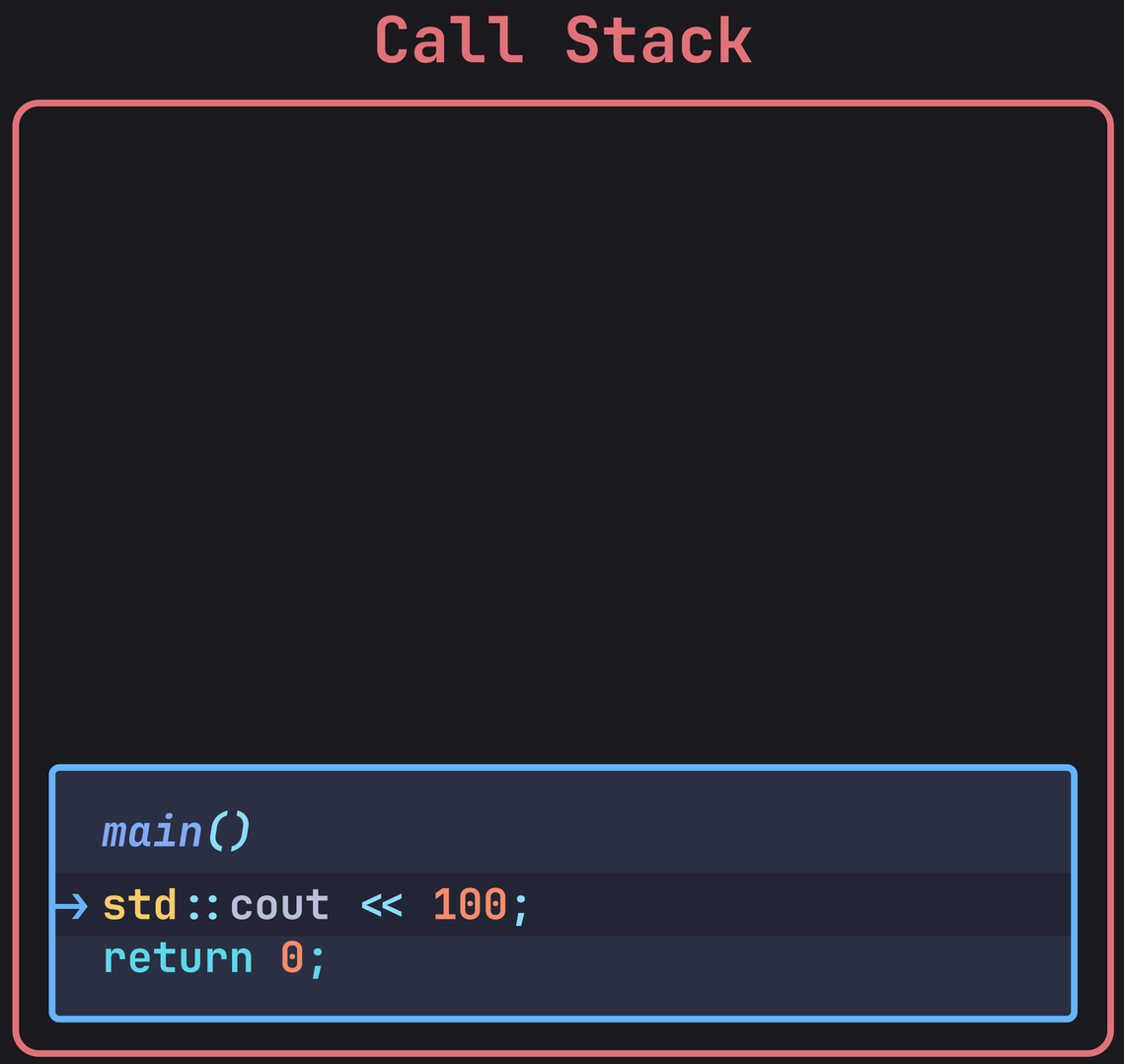
How do functions work?

```cpp
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



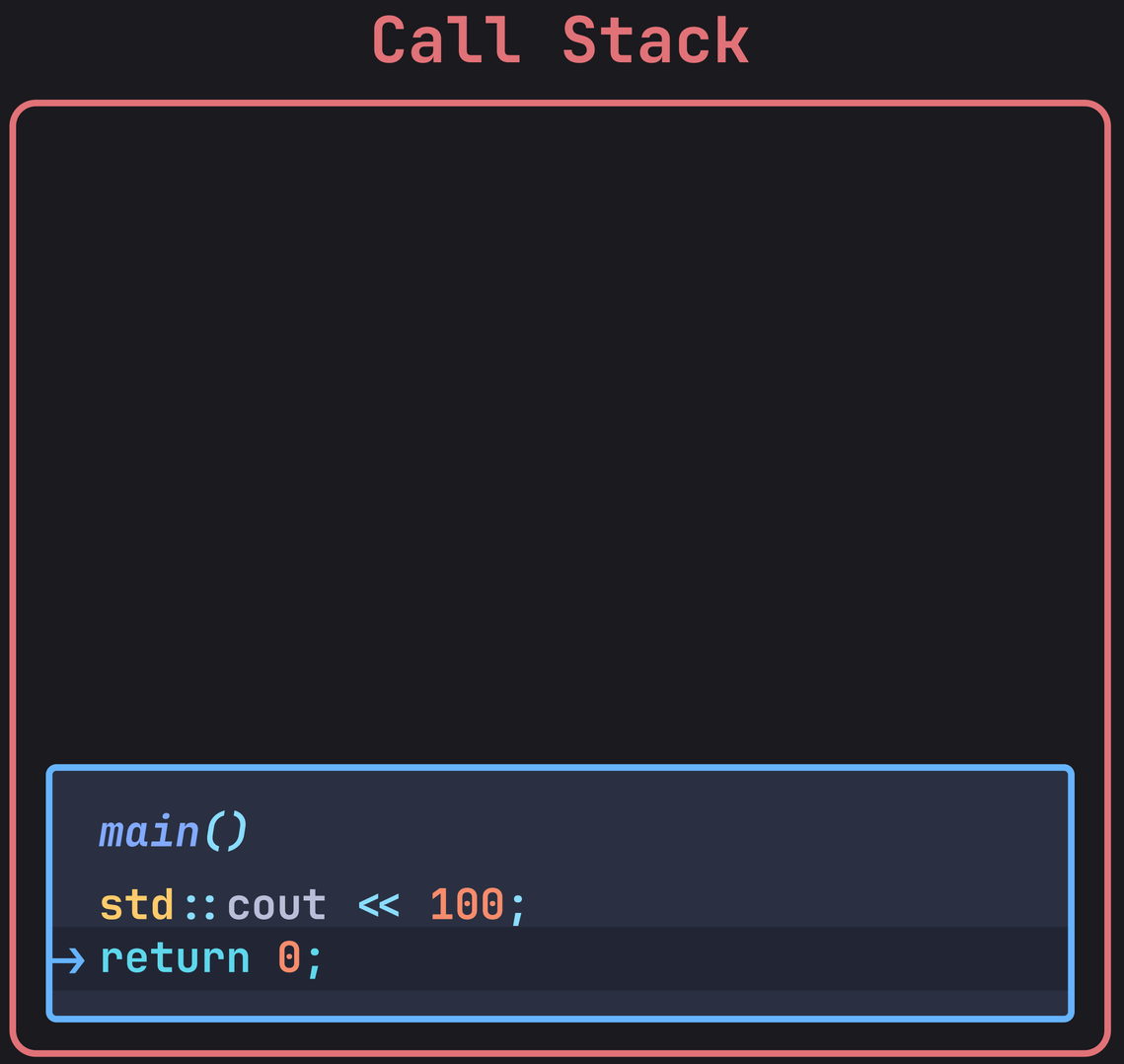
How do functions work?

```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



How do functions work?

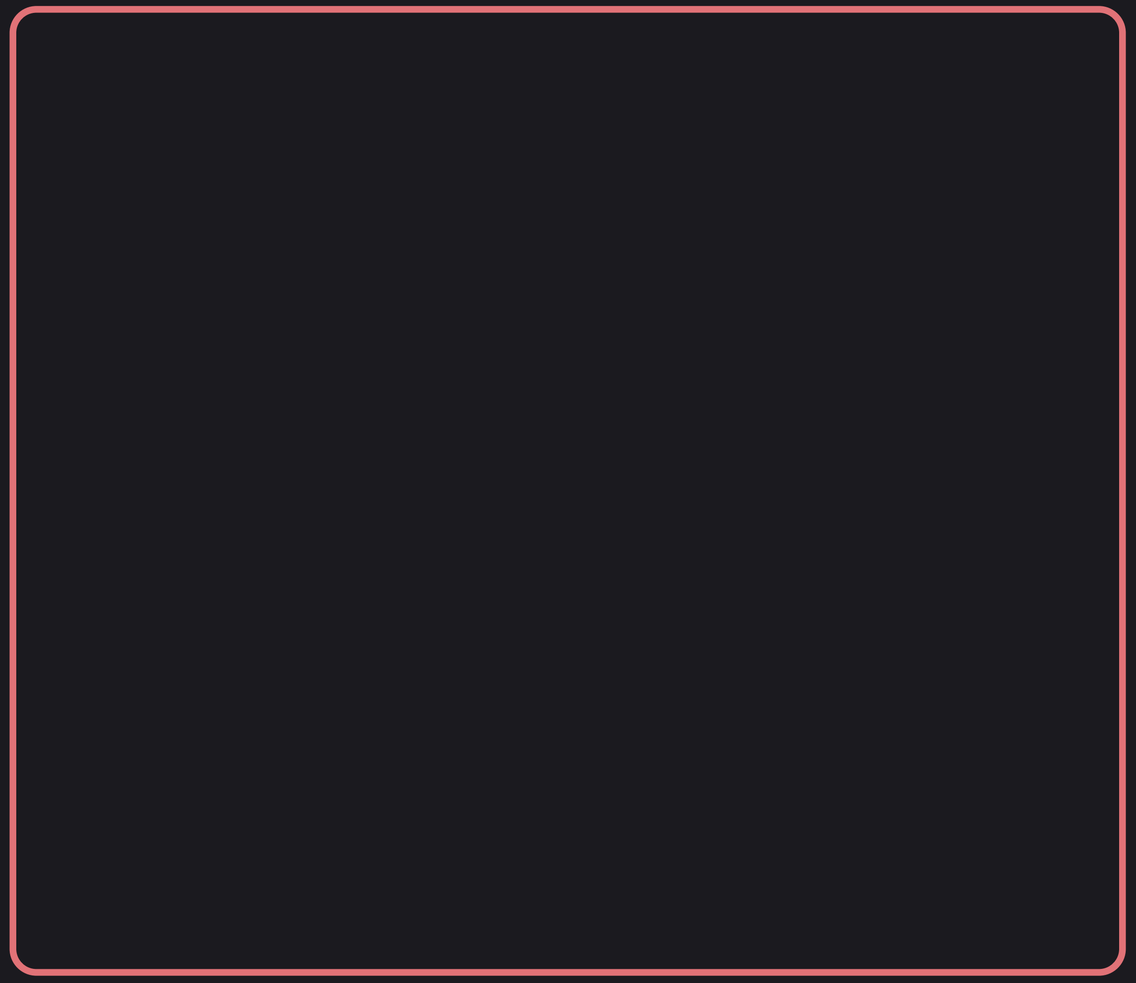
```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```



How do functions work?

```
int squareInt(int num) {
  return num * num;
}

int doubleInt(int num) {
  return num * 2;
}

int doubleThenSquareInt(int num) {
  int doubled = doubleInt(num);
  int squared = squareInt(doubled);
  return squared;
}

int main() {
  std::cout << doubleThenSquareInt(5);
  return 0;
}
```

Call Stack



Recursion


PROBLEMS WITH RECURSION ?
↓Please take one↓
WHAT THE-
PROBLEMS WITH RECURSION ?
↓Please take one↓
smbc-comics.com


Memes that don't recurse
Memes that don't recurse
Memes that don't recurse

Recursion (Motivation)

Recursion **simplifies complex problems** by breaking them down into simpler, **identical subproblems**

Recursion

is a programming concept where a function **calls itself** in order to solve a problem


```
void dream() {
  dream()
}
```

Recursive Factorial

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

A factorial is the multiplication of all positive integers from 1 up to a given number.

Recursive Factorial

$$3! = 3 \times 2 \times 1 = 6$$

```
factorial(3) == 3 * 2 * 1 == 6
```

A factorial is the multiplication of all positive integers from 1 up to a given number.

Recursive Factorial

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

```
factorial(5) == 5 * 4 * 3 * 2 * 1 == 120
```

A factorial is the multiplication of all positive integers from 1 up to a given number.

Recursive Factorial

So notice that 5! can be calculated by first calculating 4!

$$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!} = 120$$

Recursive Factorial

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

```
factorial(5) == 5 * factorial(4)
```

Recursive Factorial

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

```
factorial(4) == 4 * factorial(3)
```

Recursive Factorial

So what is the general pattern?

$$n! = ?$$

```
factorial(n) = ?
```

Recursive Factorial

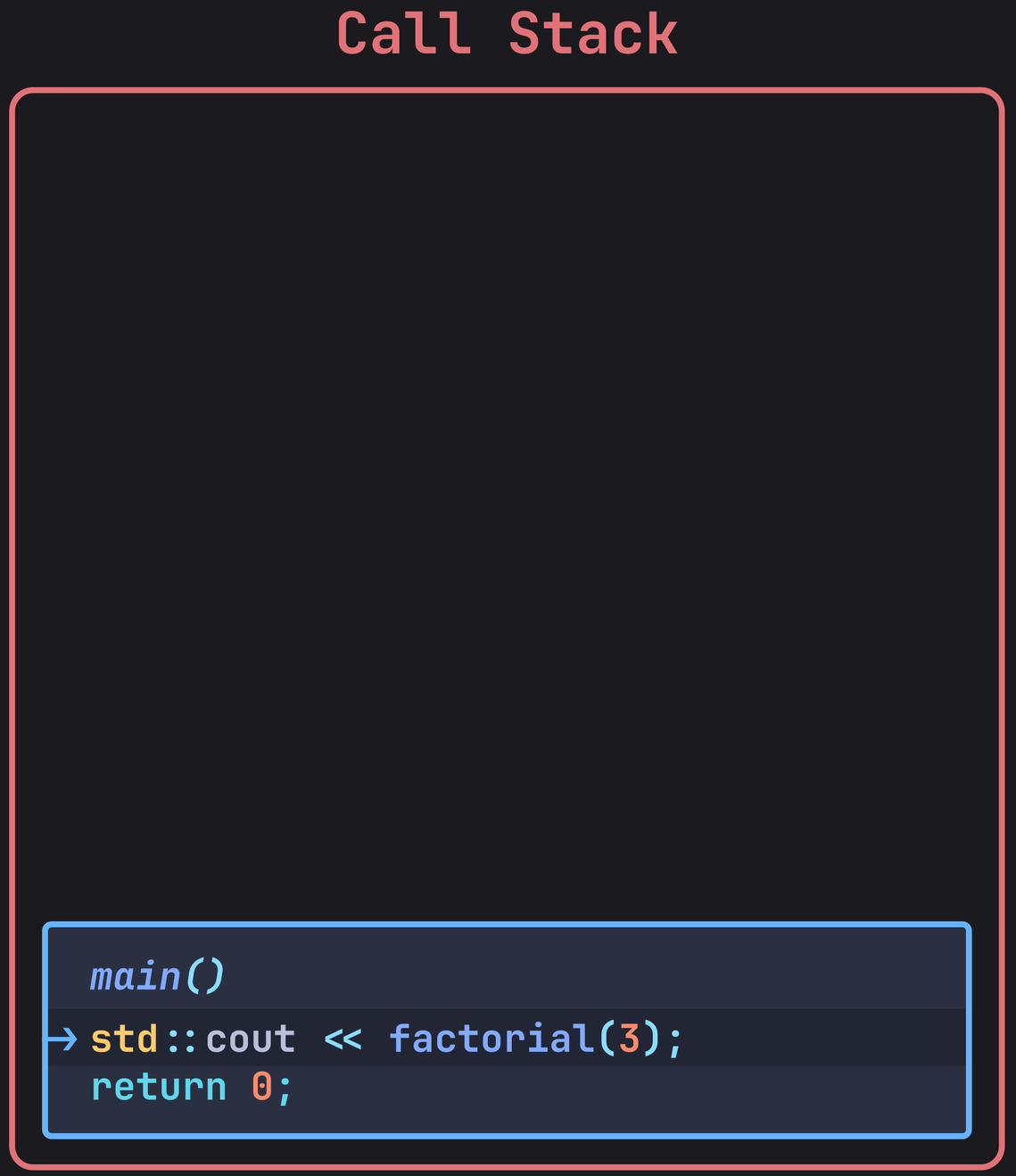
So what is the general pattern?

$$n! = n \times (n-1)!$$

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

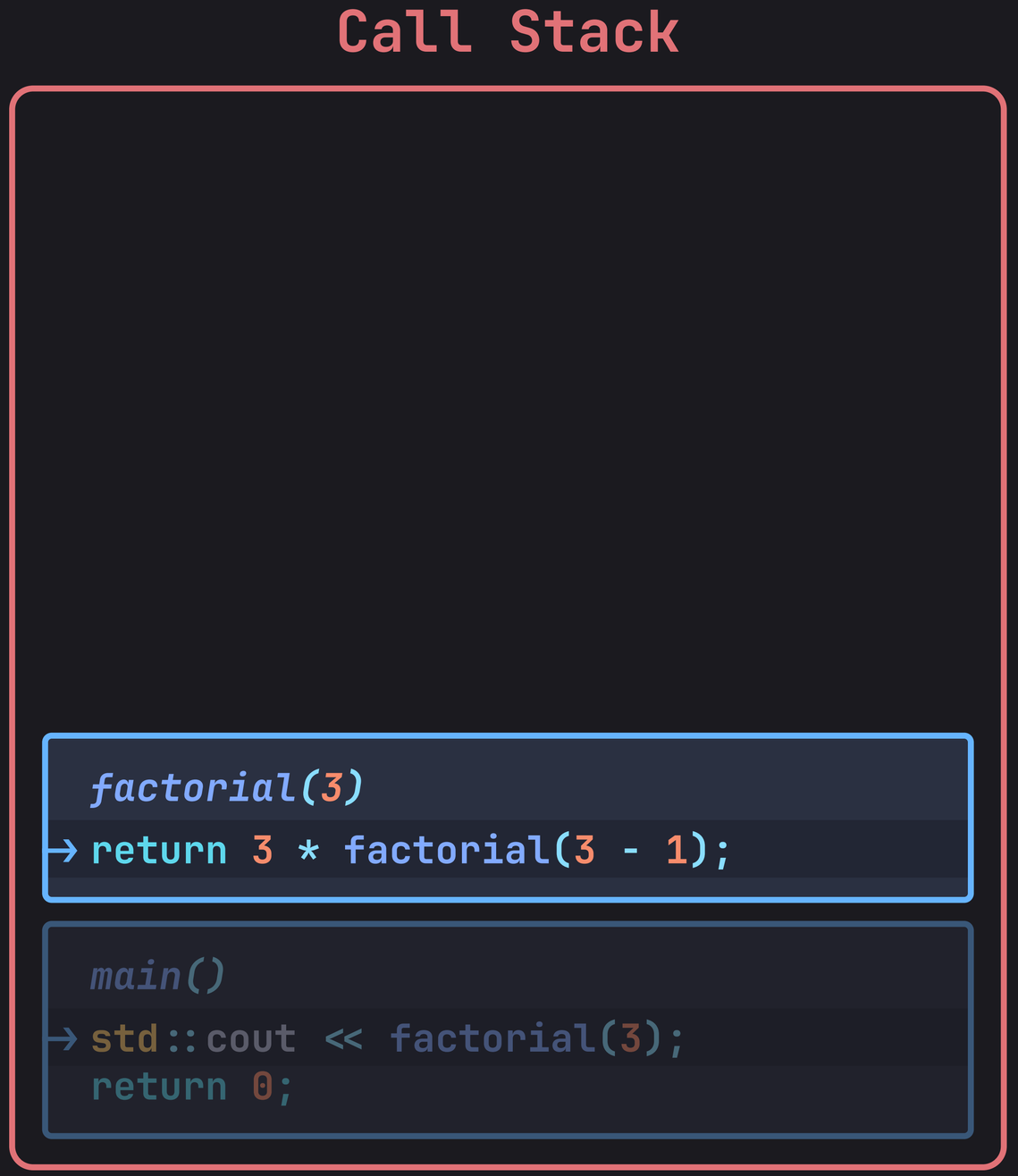
Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



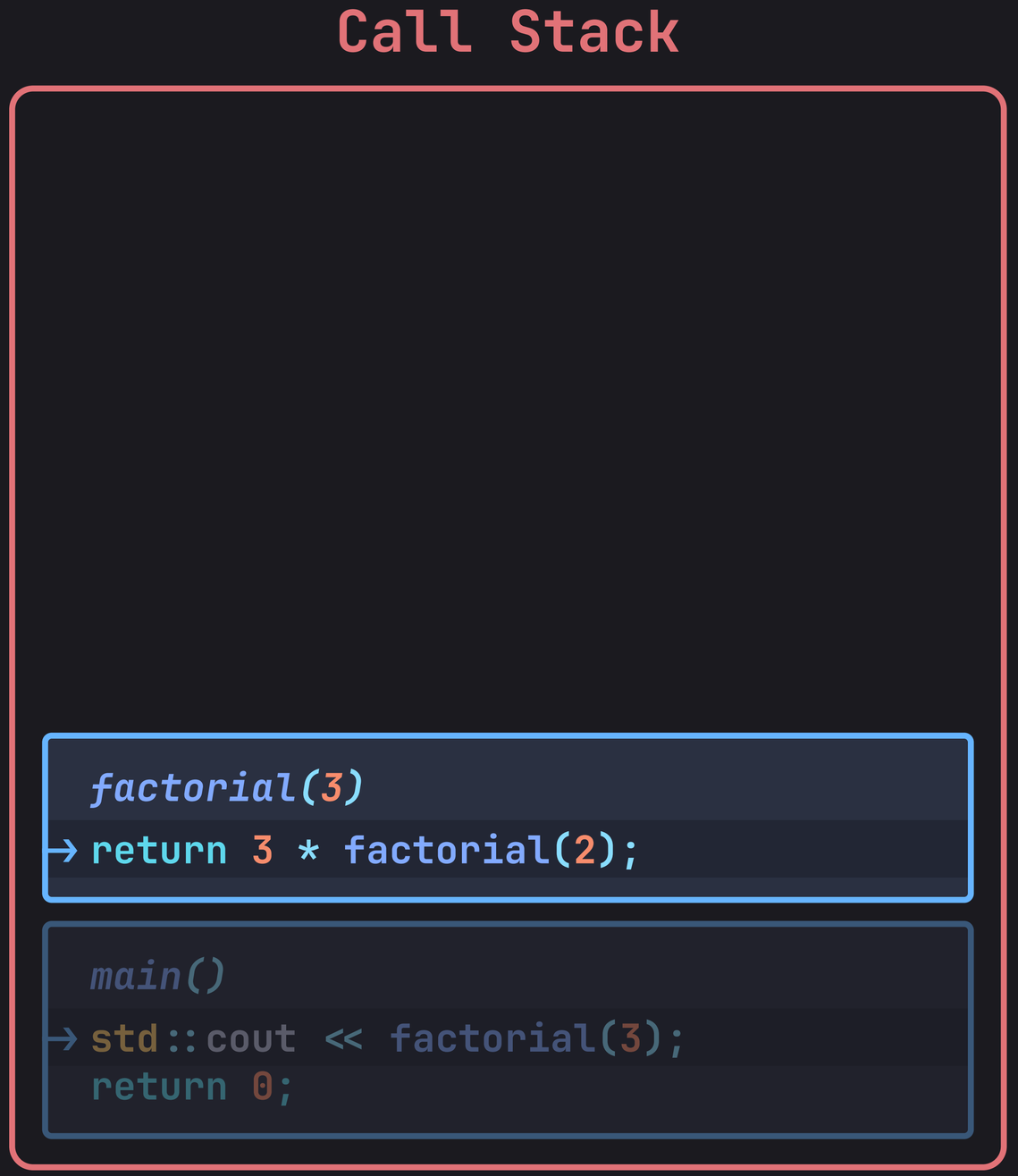
Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



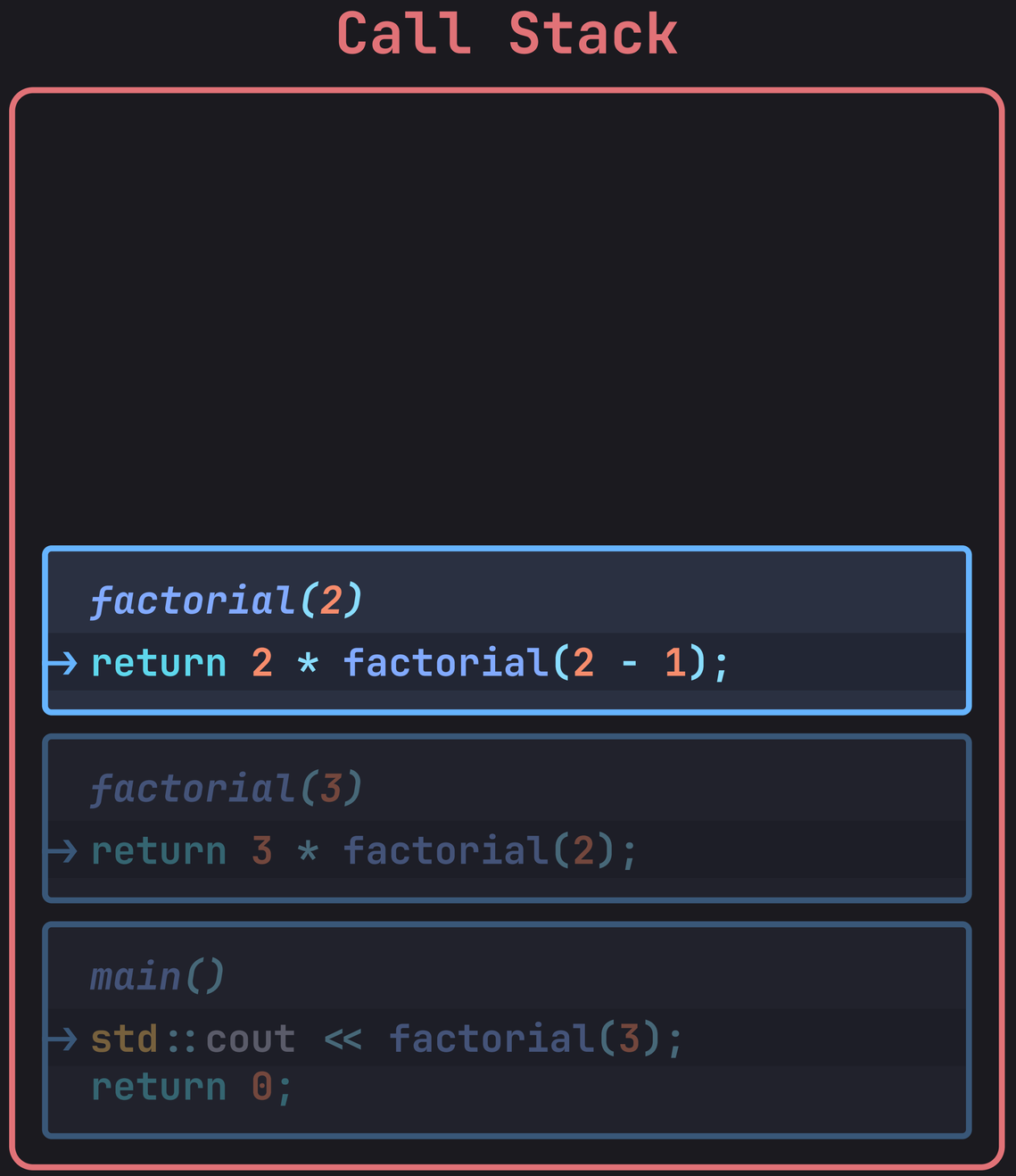
Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



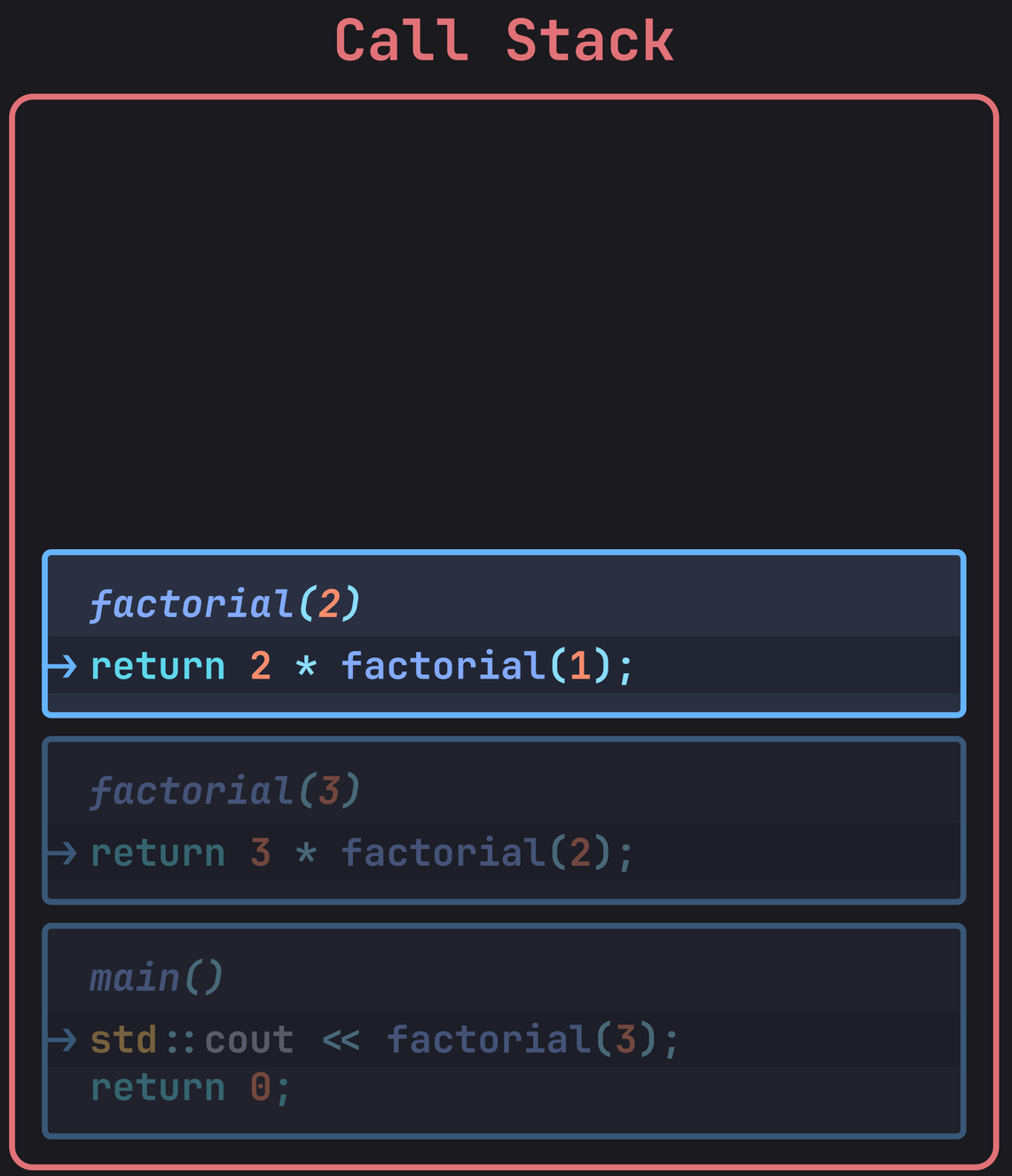
Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



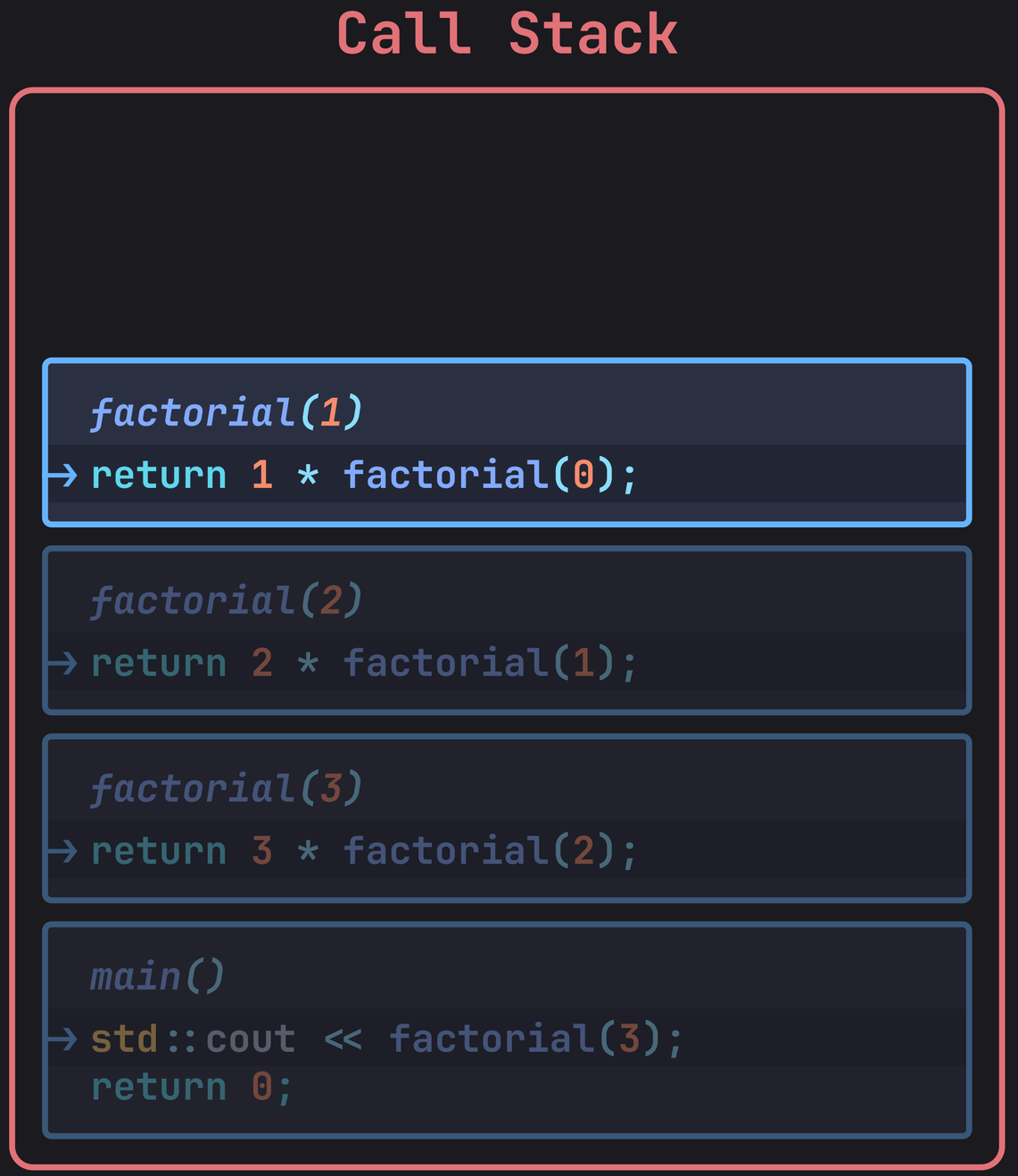
Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



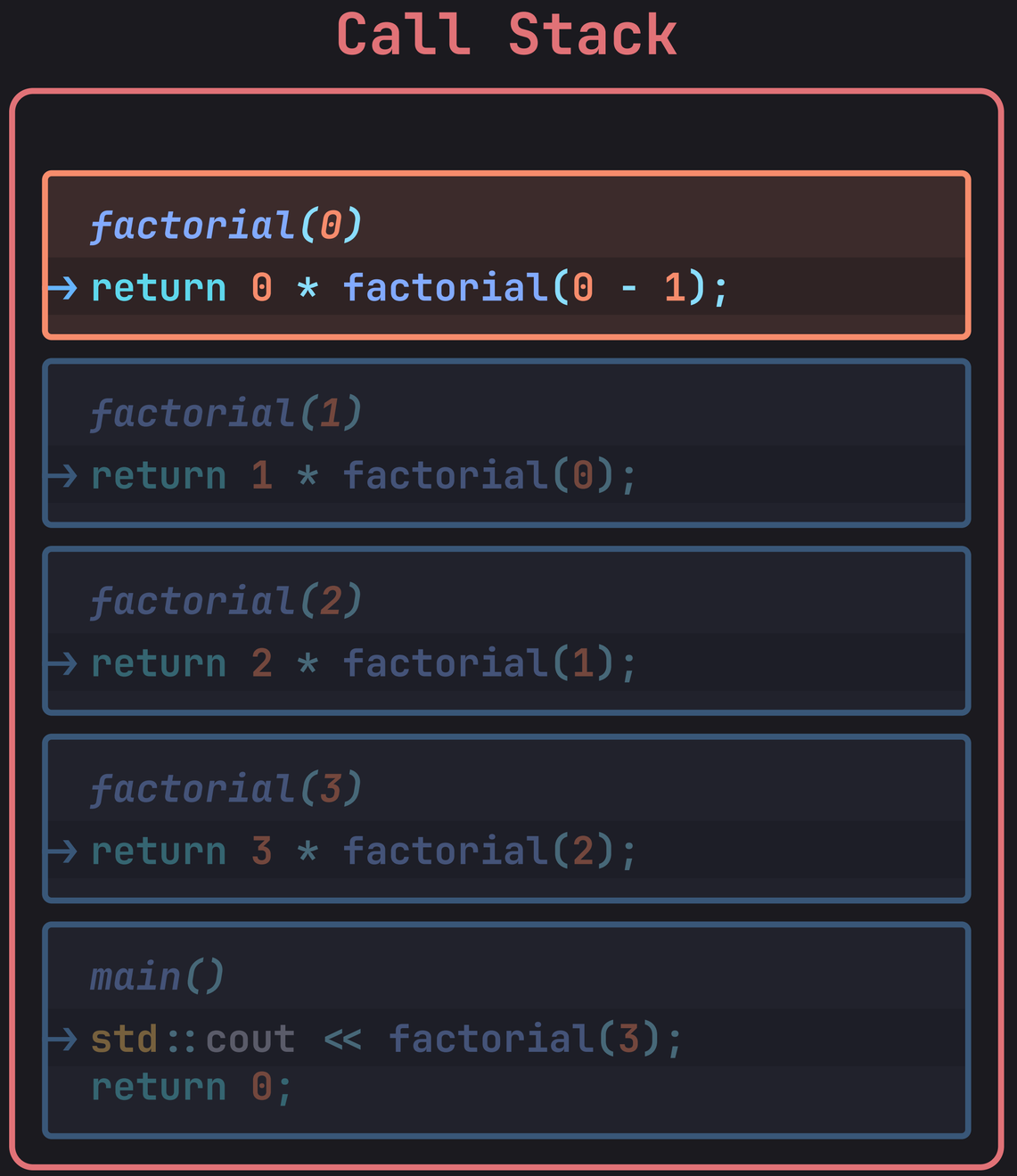
Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



Factorial

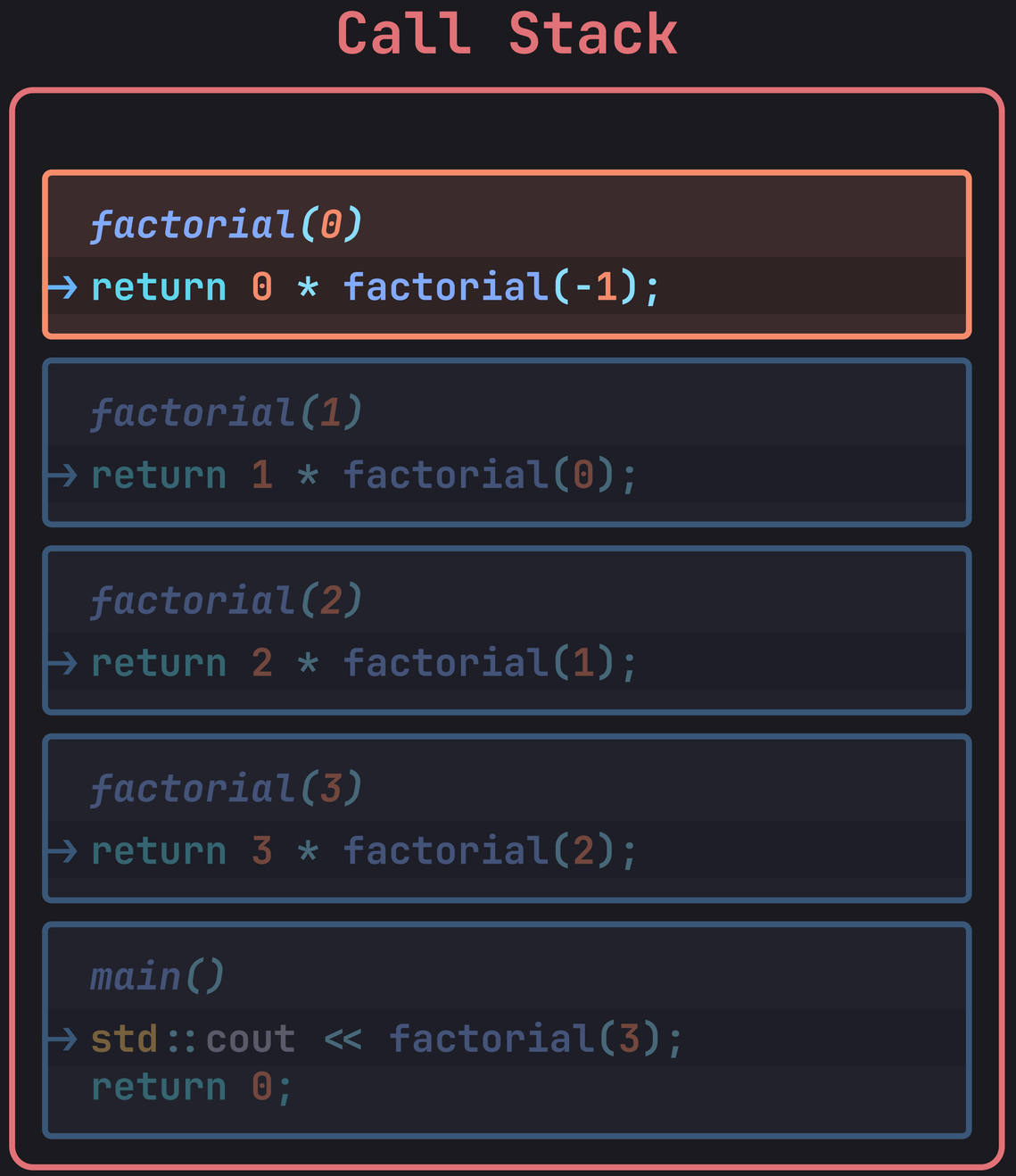
```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```



Factorial

```cpp
int factorial(int n) {
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(3);
  return 0;
}
```

We are stuck in an endless loop 😵‍💫



Base Case (Very Important)

Base Case (Very Important)

Whenever we are writing a recursive algorithm
we want to identity a **base base**

This is usually when the input is as
small as possible

Base Case (Very Important)

$$0! = 1$$

```
factorial(0) == 1
```

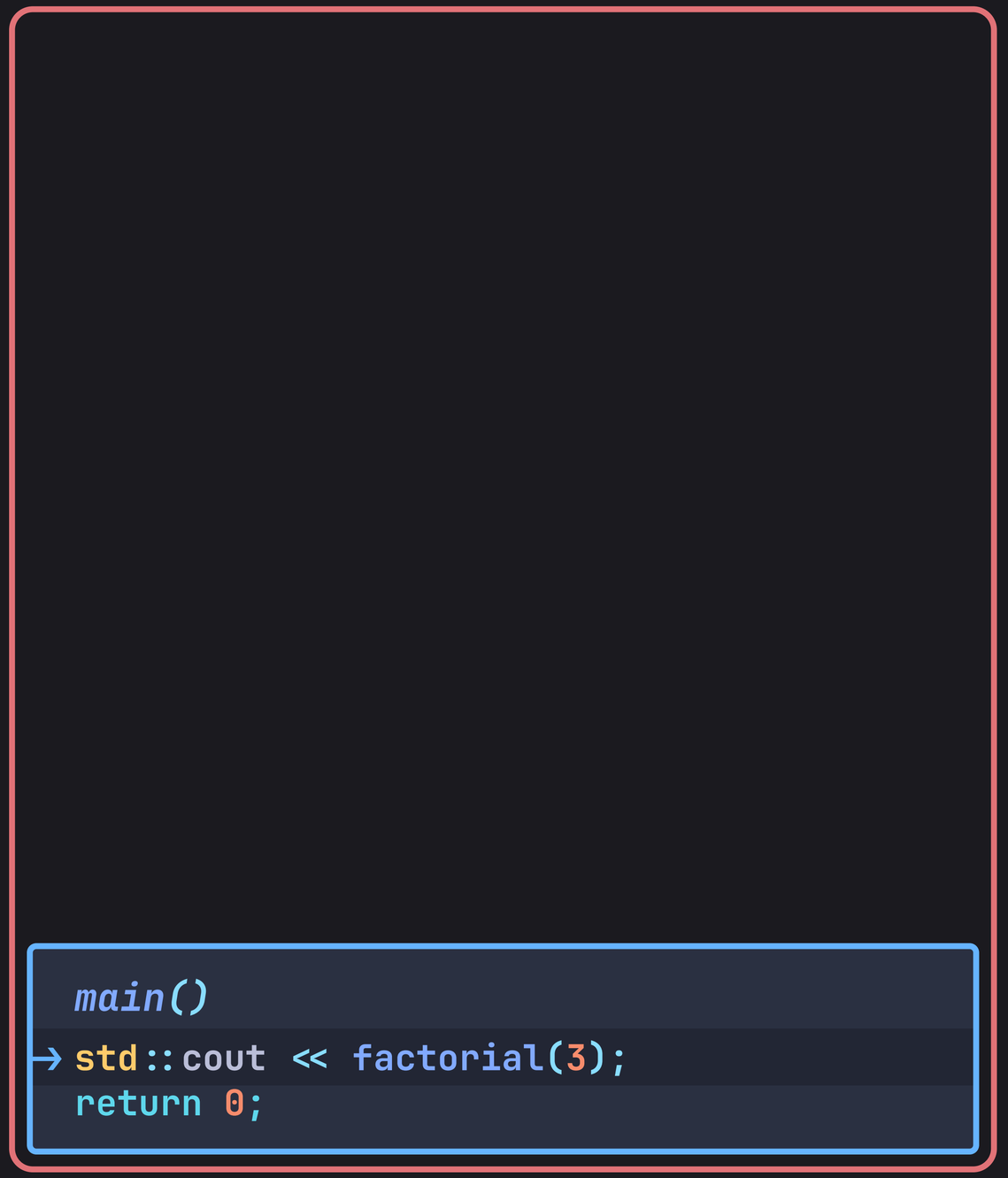
(Maybe somewhat counter intuitively...)

Base Case (Very Important)

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

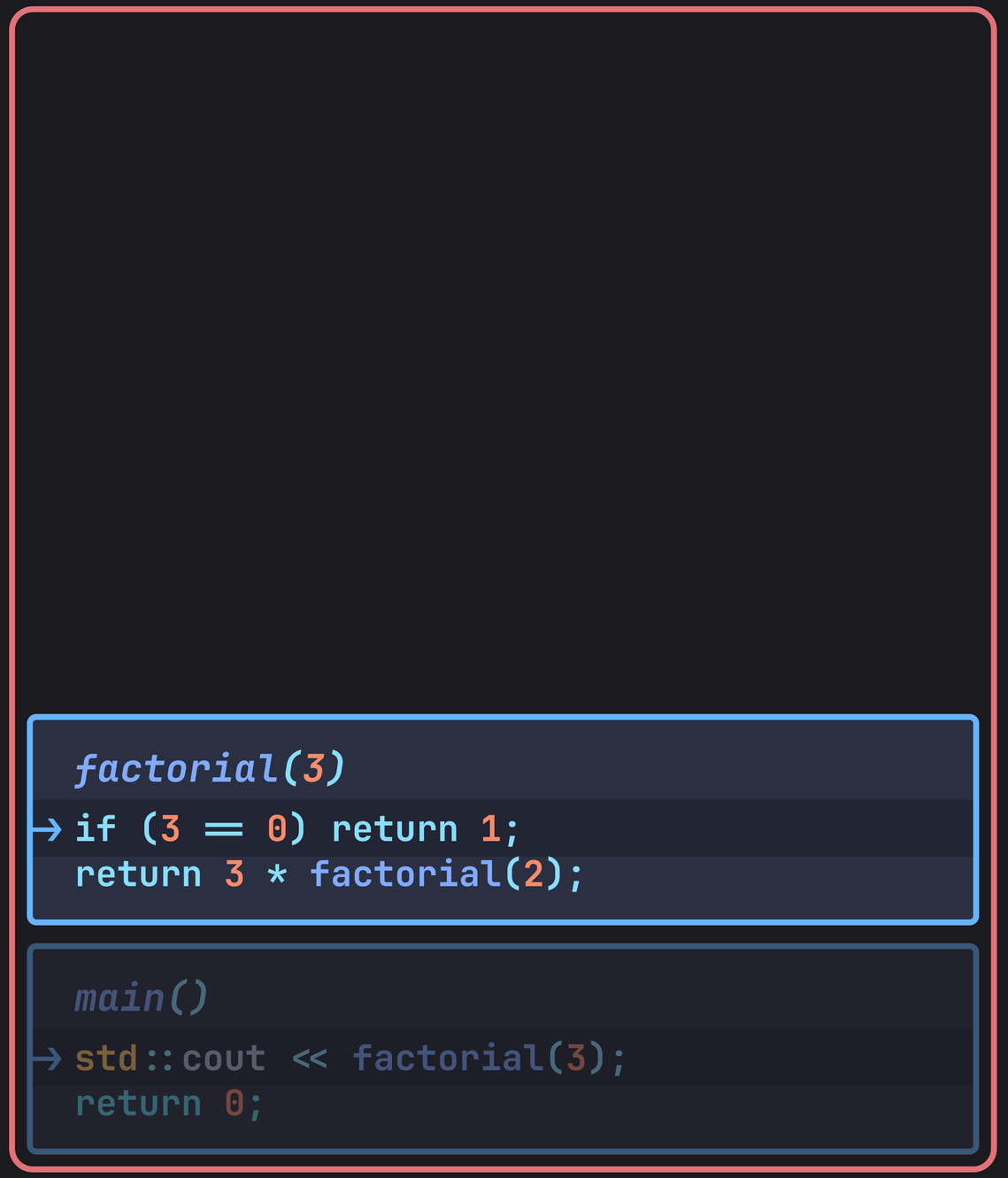
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



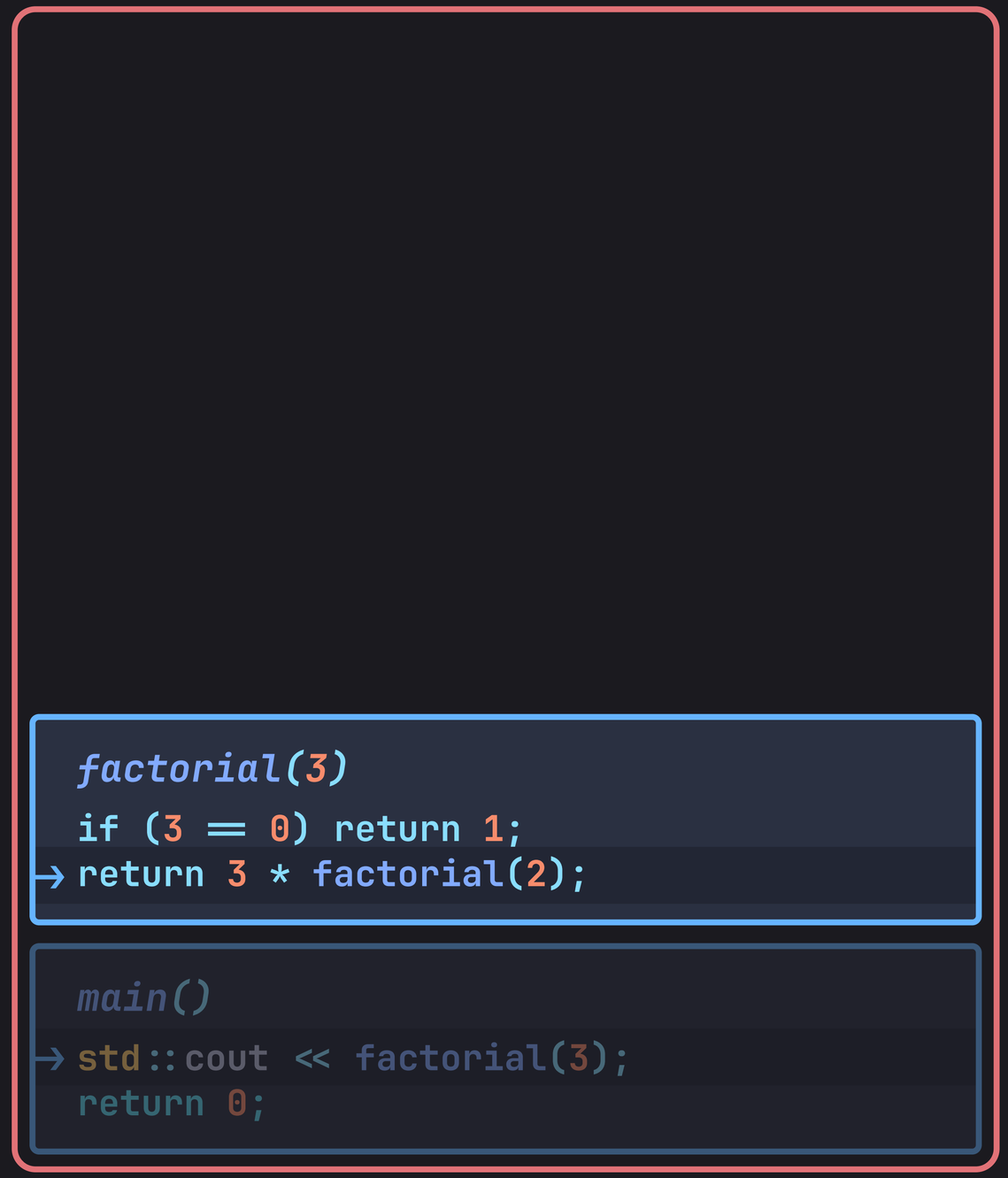
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



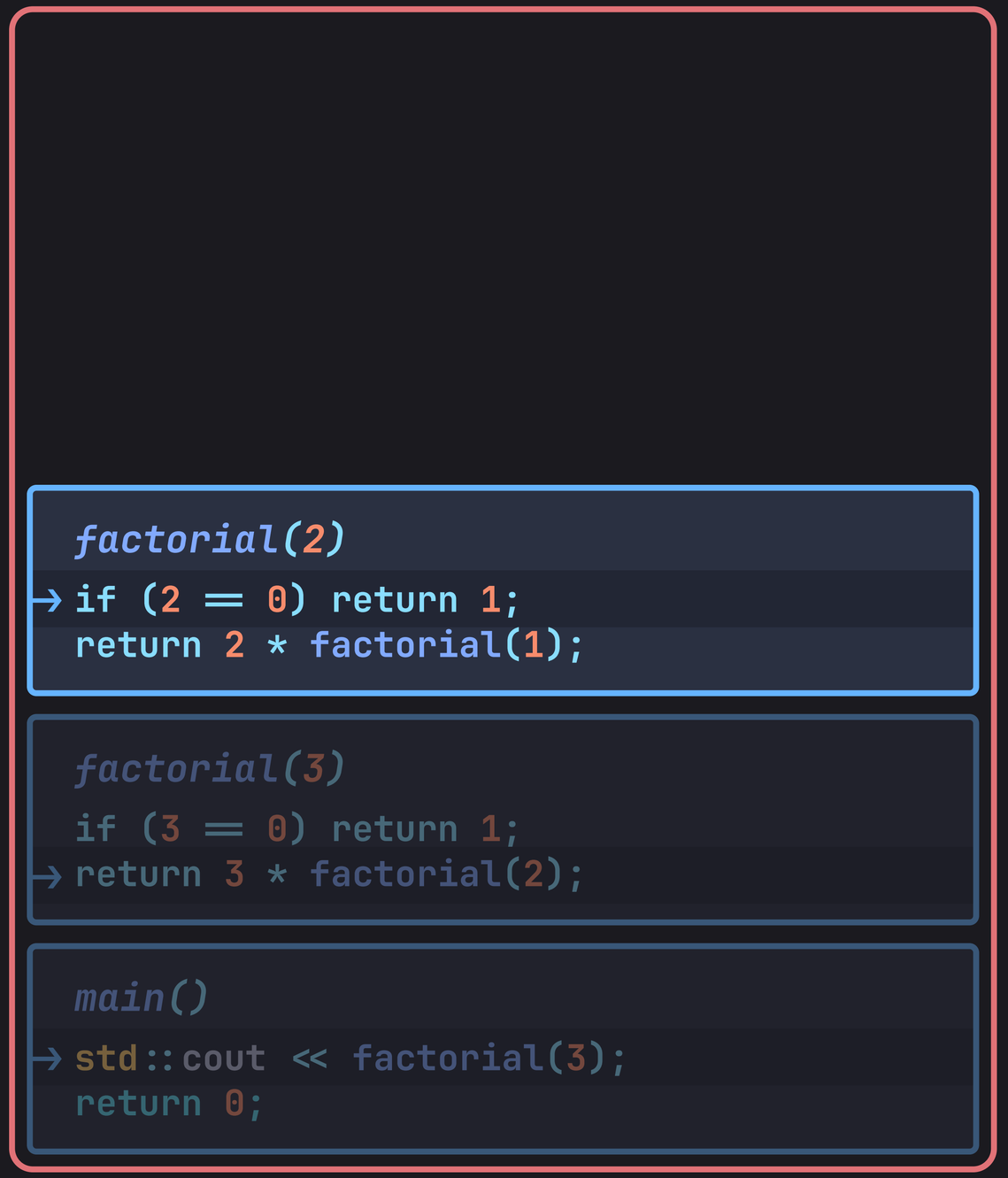
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



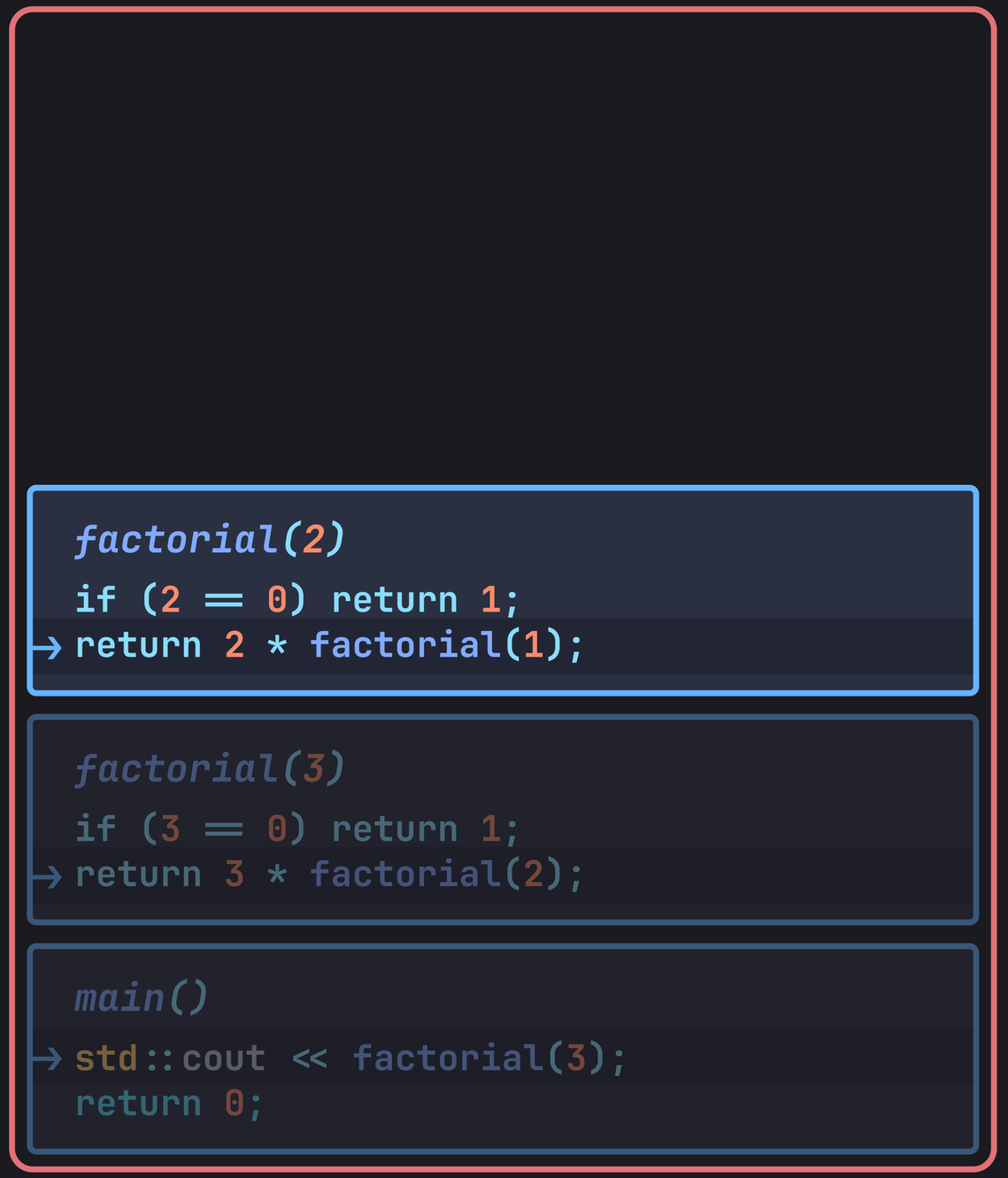
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



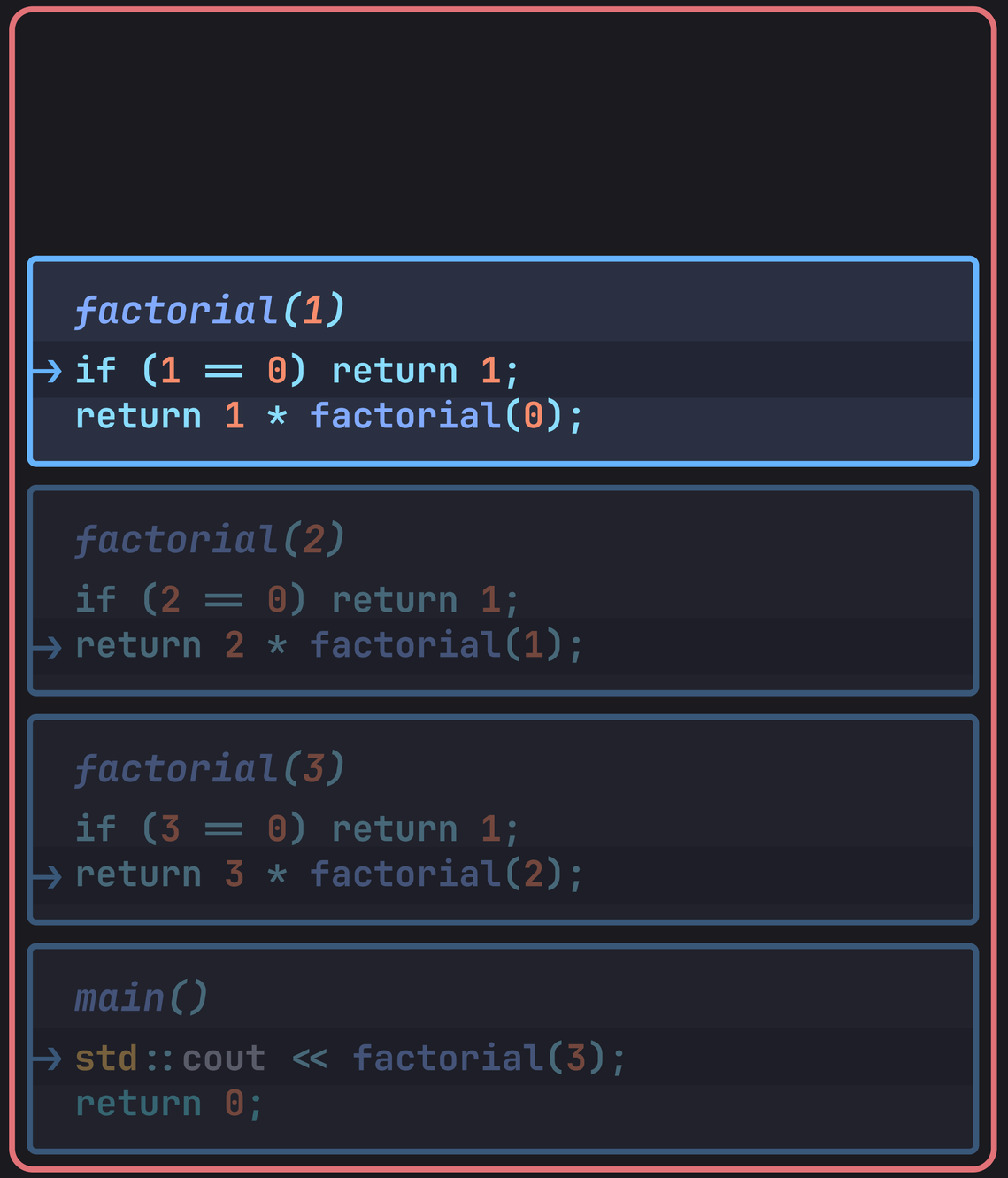
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



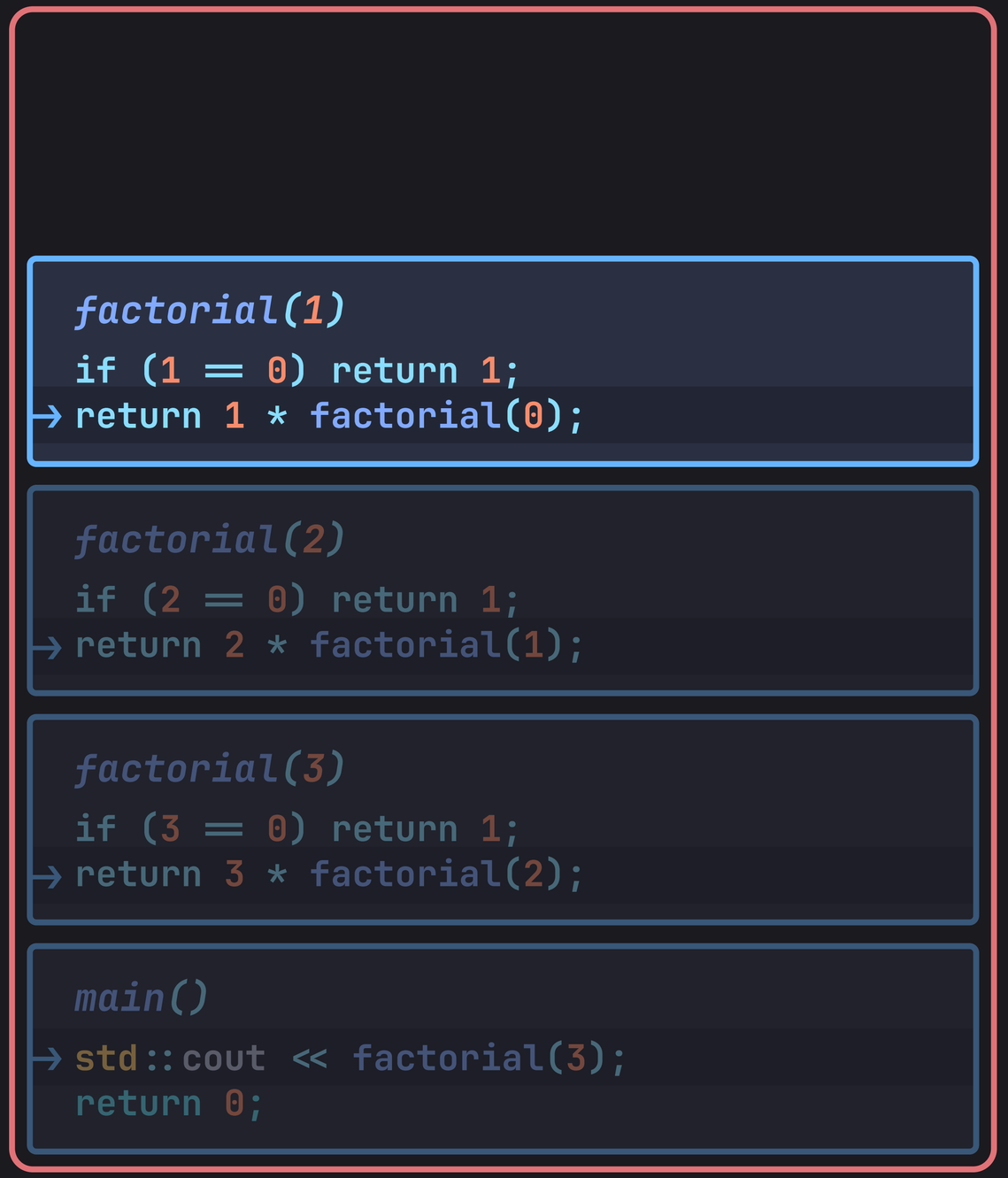
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



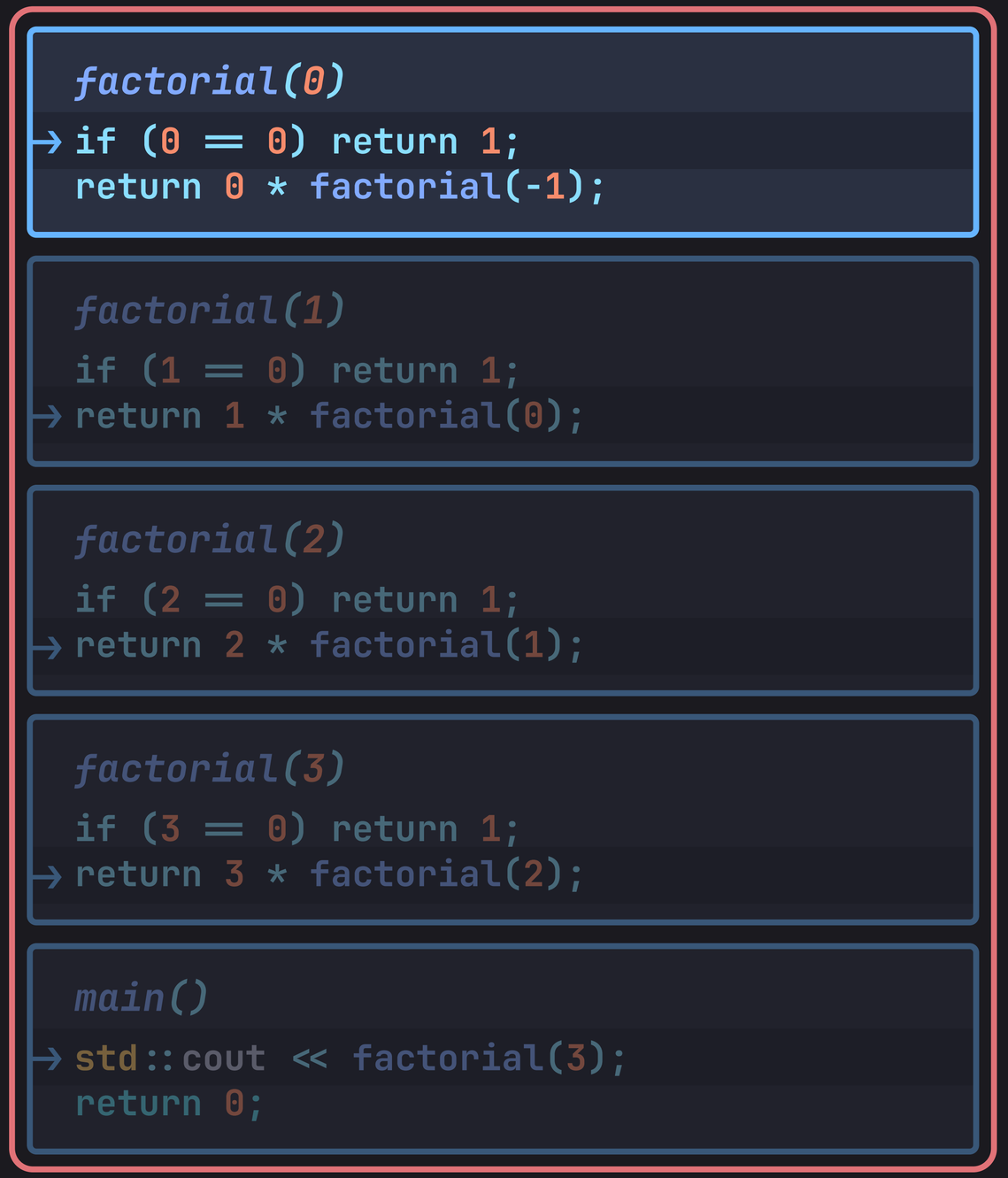
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



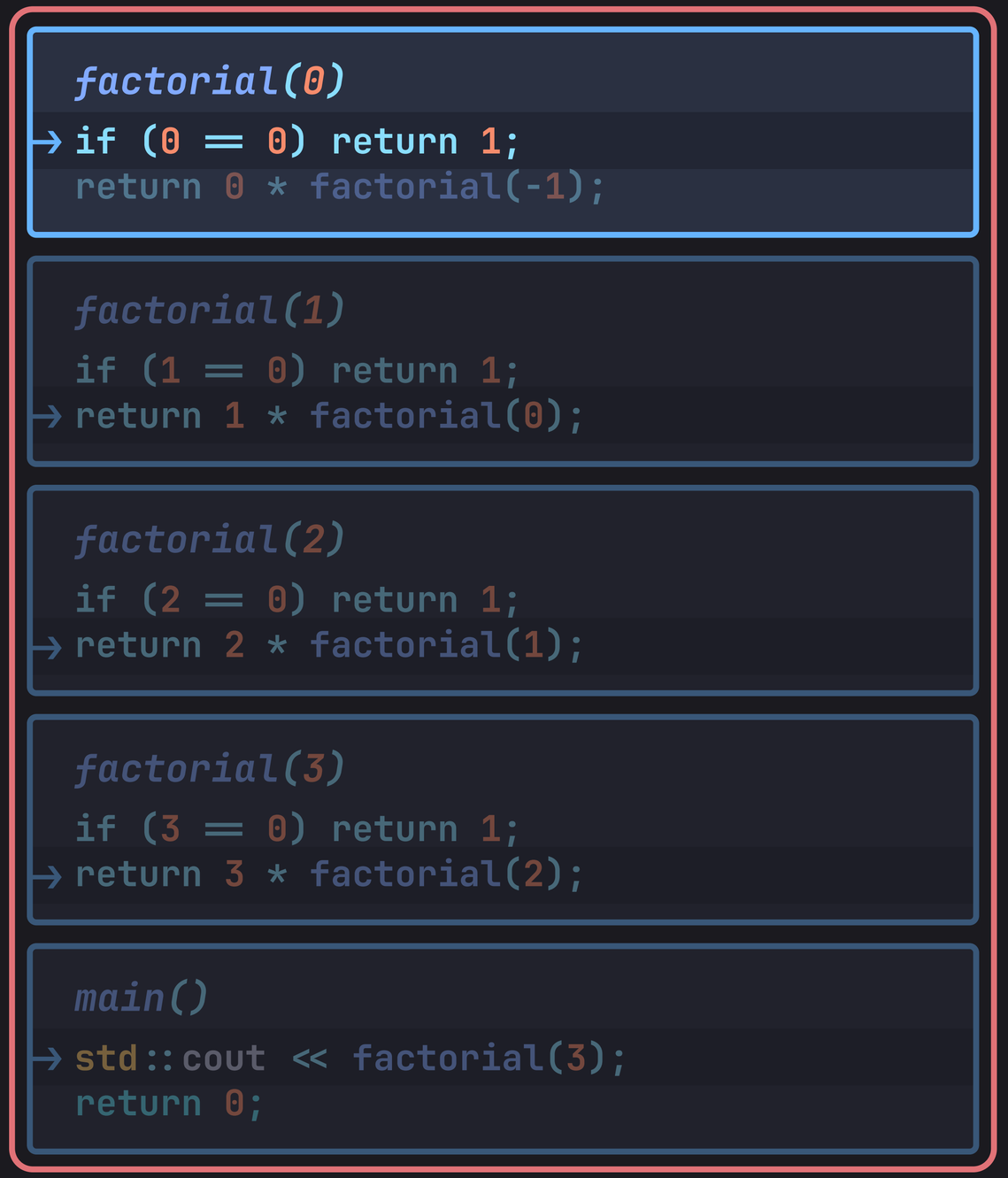
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



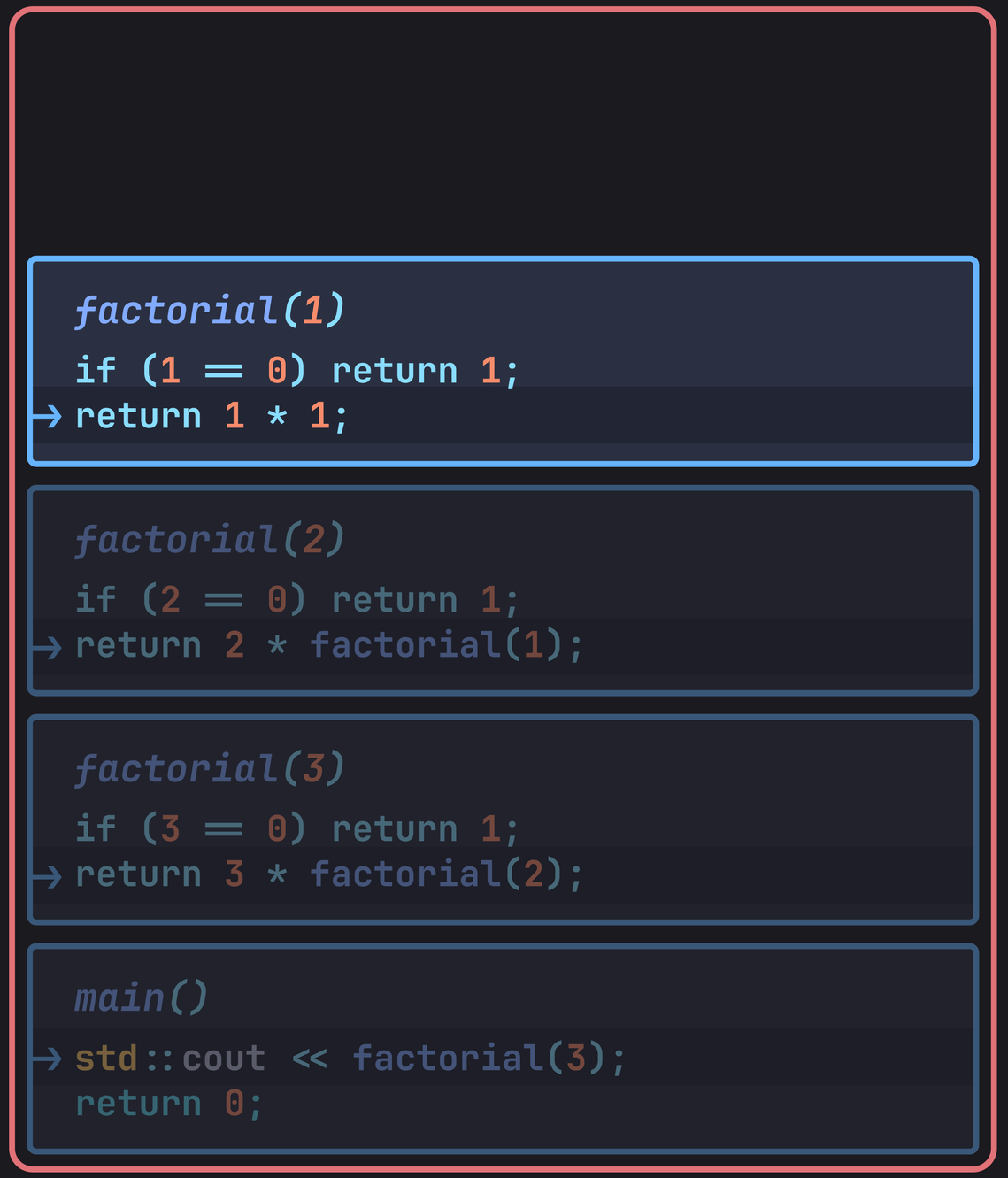
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



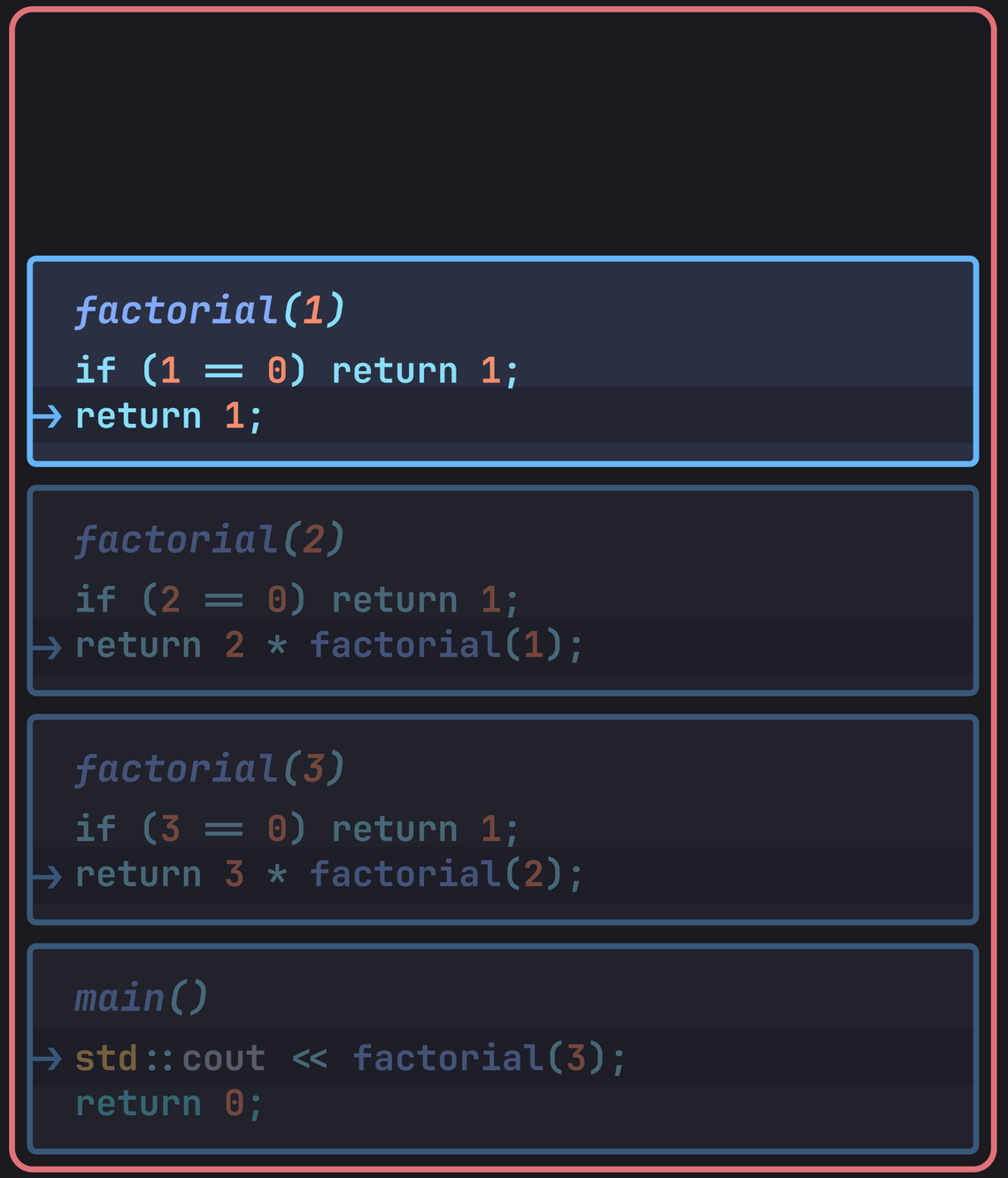
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



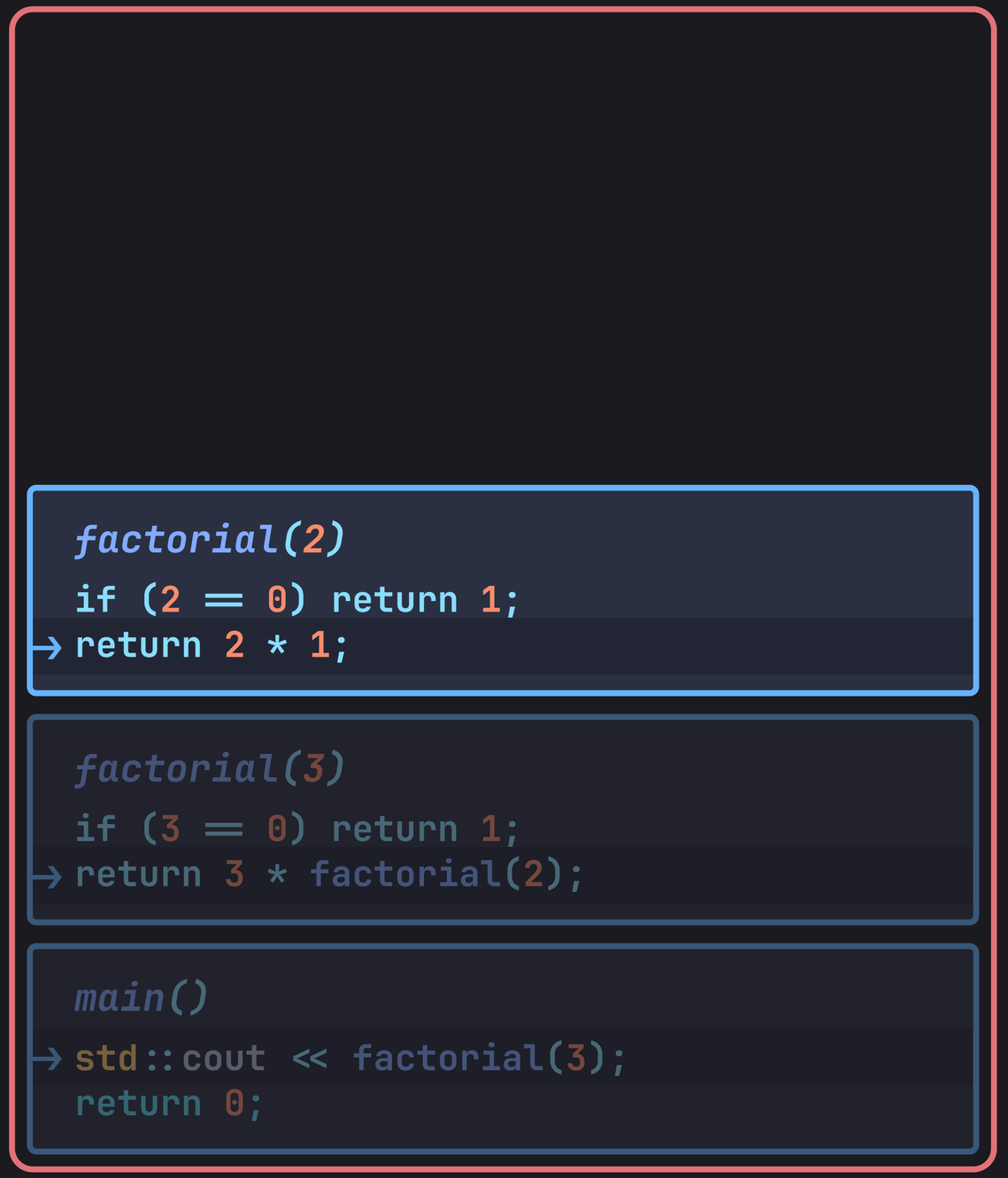
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



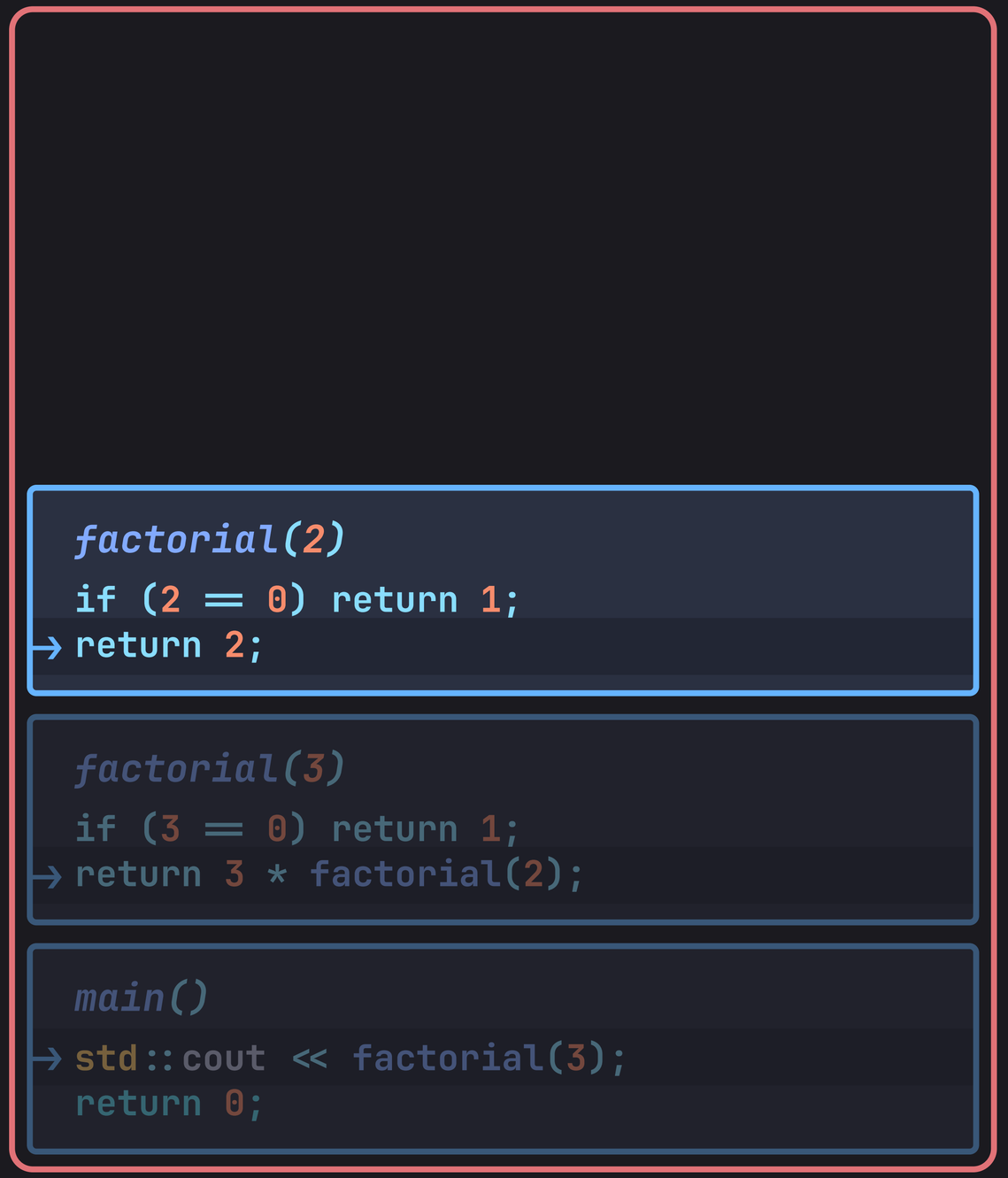
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



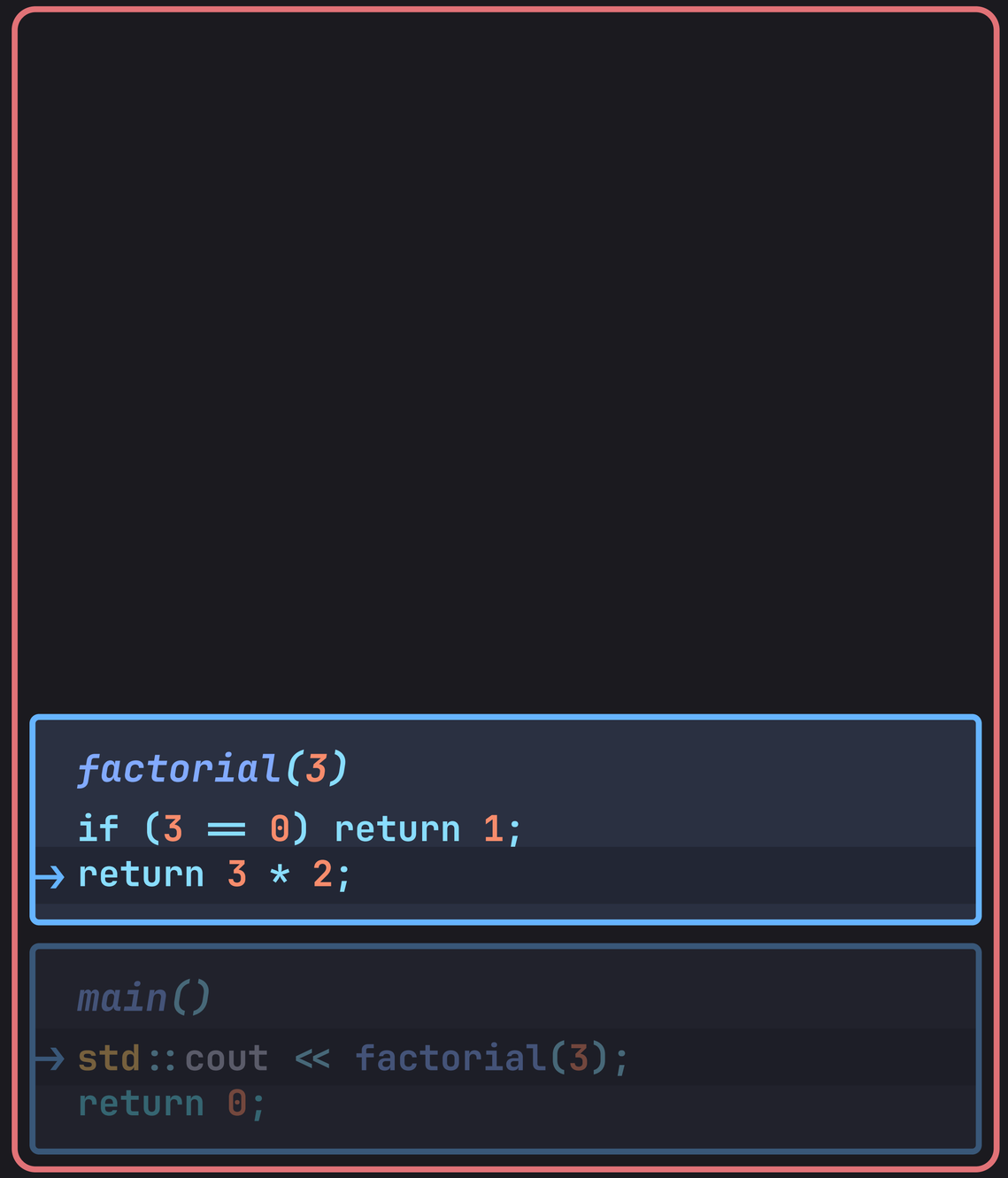
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



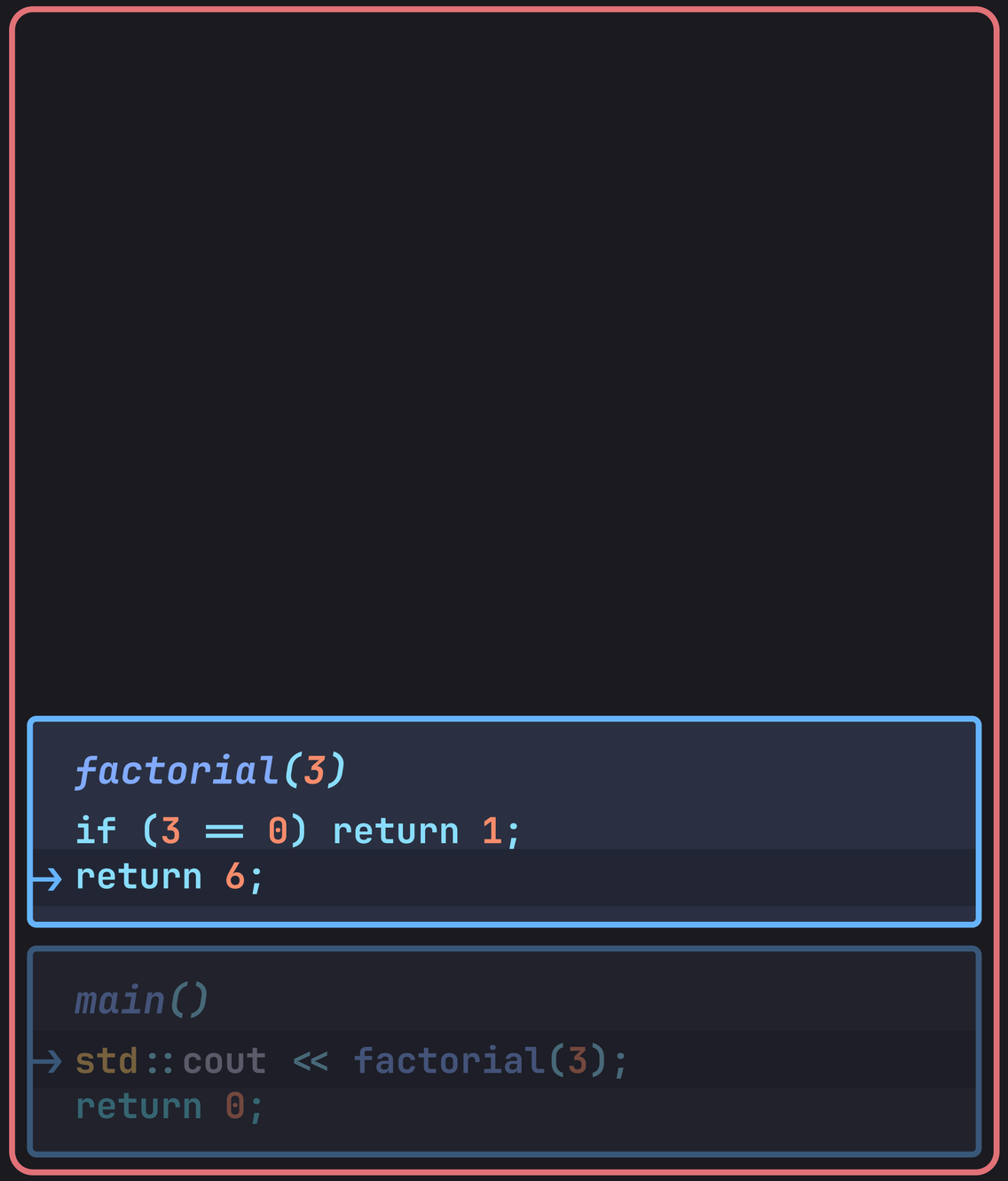
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



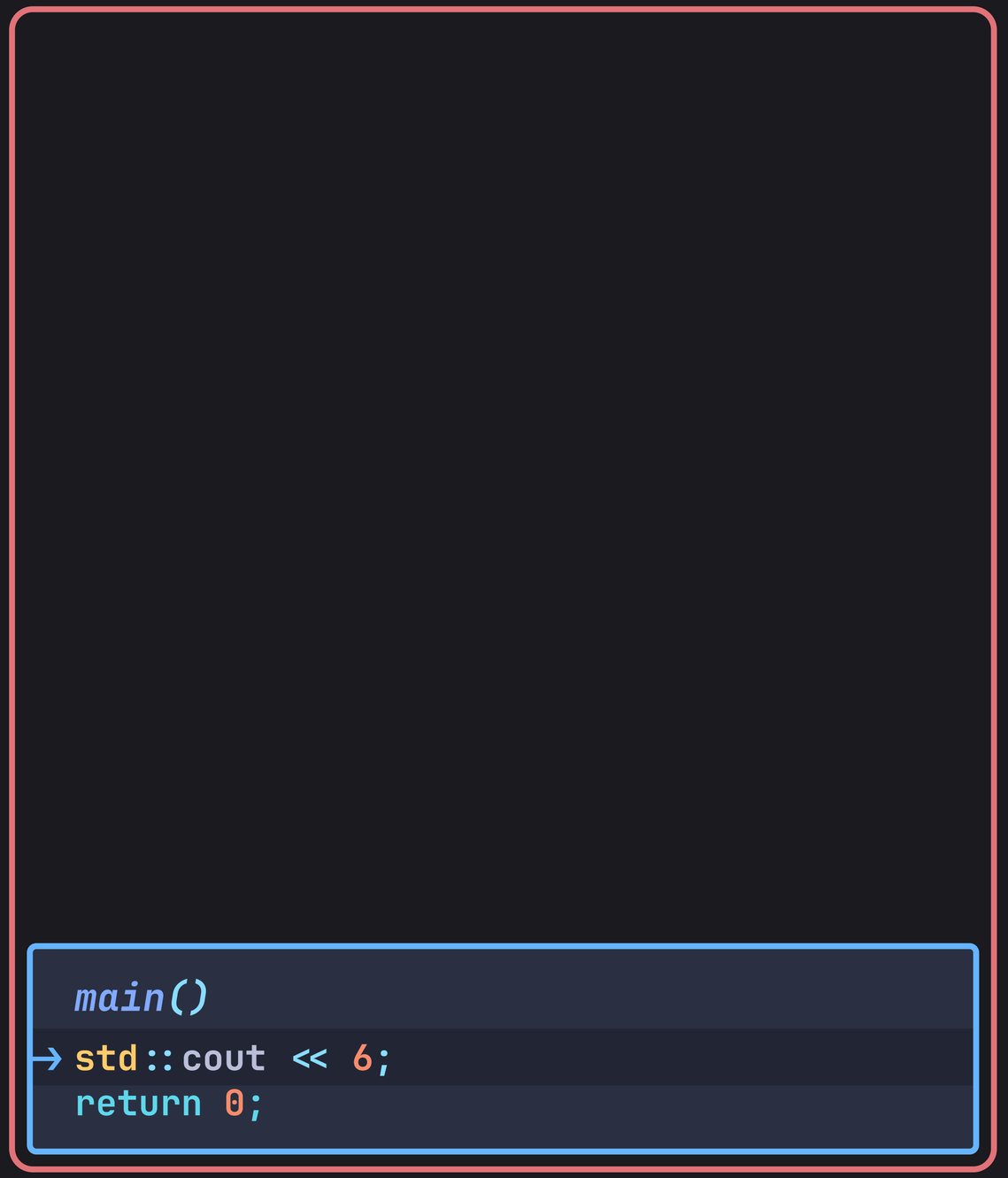
Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



Factorial

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

int main() {
  std::cout << factorial(3);
  return 0;
}
```



Let’s have a go on Ed

Divide and Conquer

1. **Divide:** Break the problem into smaller subproblems that are similar to the original problem but smaller in size
2. **Conquer:** Solve the subproblems recursively until the subproblems become simple enough to solve directly
3. **Combine:** Combine the solutions to the subproblems into a solution to the original problem

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

2 + 3 = 5

each number in the sequence is the sum of the two previous numbers

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

3 + 5 = 8

each number in the sequence is the sum of the two previous numbers

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

5 + 8 = 13

each number in the sequence is the sum of the two previous numbers

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

0 1 2 3 4 5 6 7 8 9 10 11

fib(5) = 2 + 3 = 5

Now lets index the sequence so that we can talk about the nth fibonacci number

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

0 1 2 3 4 5 6 7 8 9 10 11

fib(6) = 3 + 5 = 8

Now lets index the sequence so that we can talk about the nth fibonacci number

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

0 1 2 3 4 5 6 7 8 9 10 11

```
fib(8) = fib(?) + fib(?) = 21
```

Now lets index the sequence so that we can talk about the nth fibonacci number

Fibonacci Sequence

0	1	1	2	3	5	8	13	21	34	55	89	...
0	1	2	3	4	5	6	7	8	9	10	11	

$$fib(8) = \underbrace{fib(6)}_{8} + \underbrace{fib(7)}_{13} = 21$$

Fibonacci Sequence

0	1	1	2	3	5	8	13	21	34	55	89	...
0	1	2	3	4	5	6	7	8	9	10	11	

```
fib(n) = fib(n - 1) + fib(n - 2)
```

Fibonacci Sequence

```
int fib(int n) {
  return fib(n - 1) + fib(n - 2);
}
```

Can anyone see the problem here?

Do not forget the base case!

Fibonacci Sequence

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...
0  1  2  3  4  5  6   7   8   9  10  11
```

```
fib(0) = 0
fib(1) = 1
```

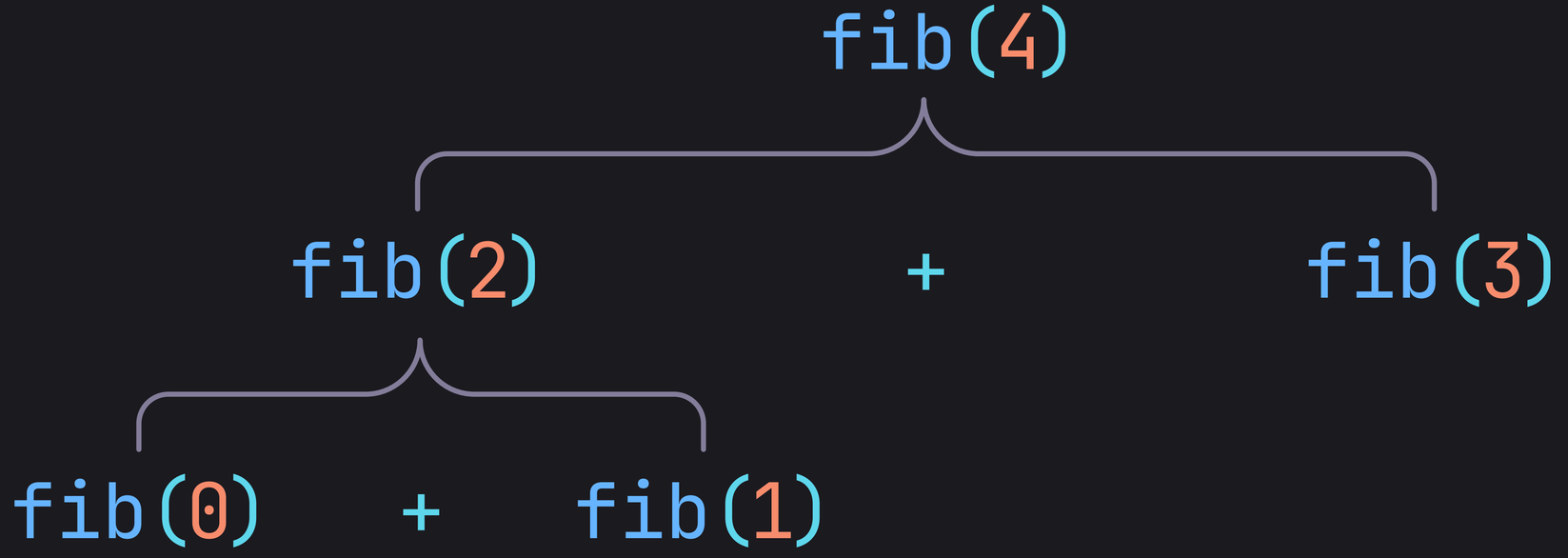
We can just hardcode these cases

Fibonacci Sequence

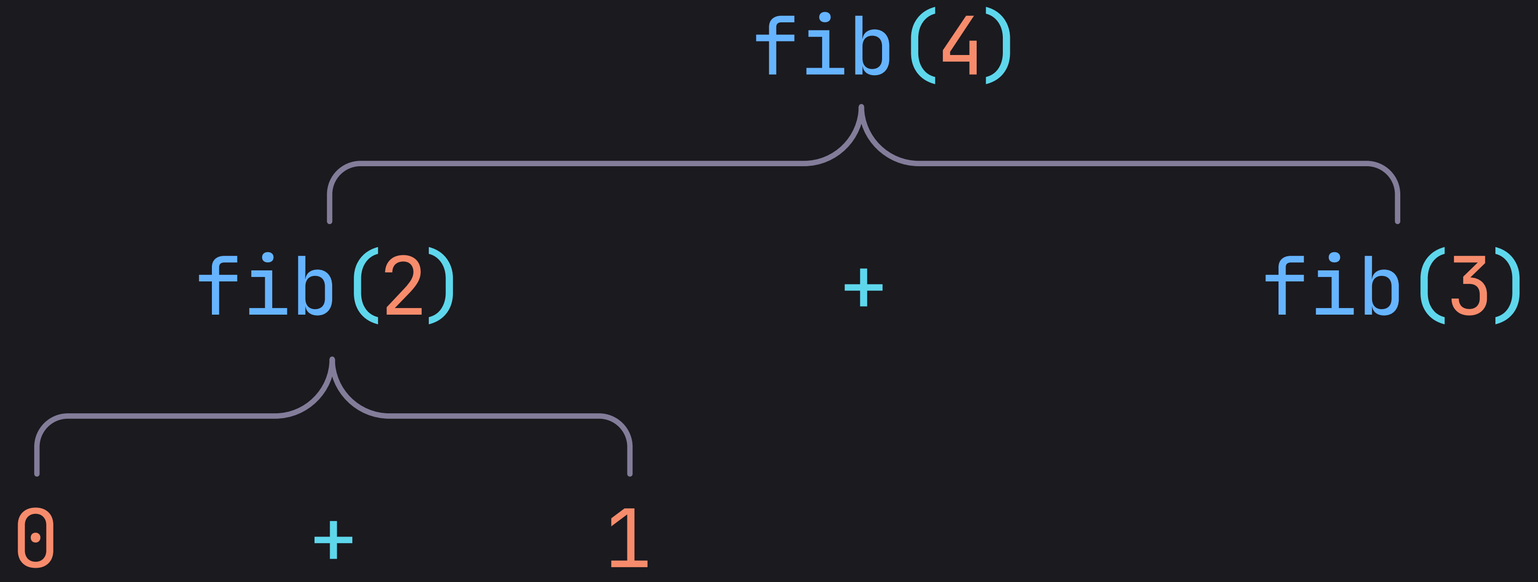
```
fib(4)
```

Fibonacci Sequence

Fibonacci Sequence

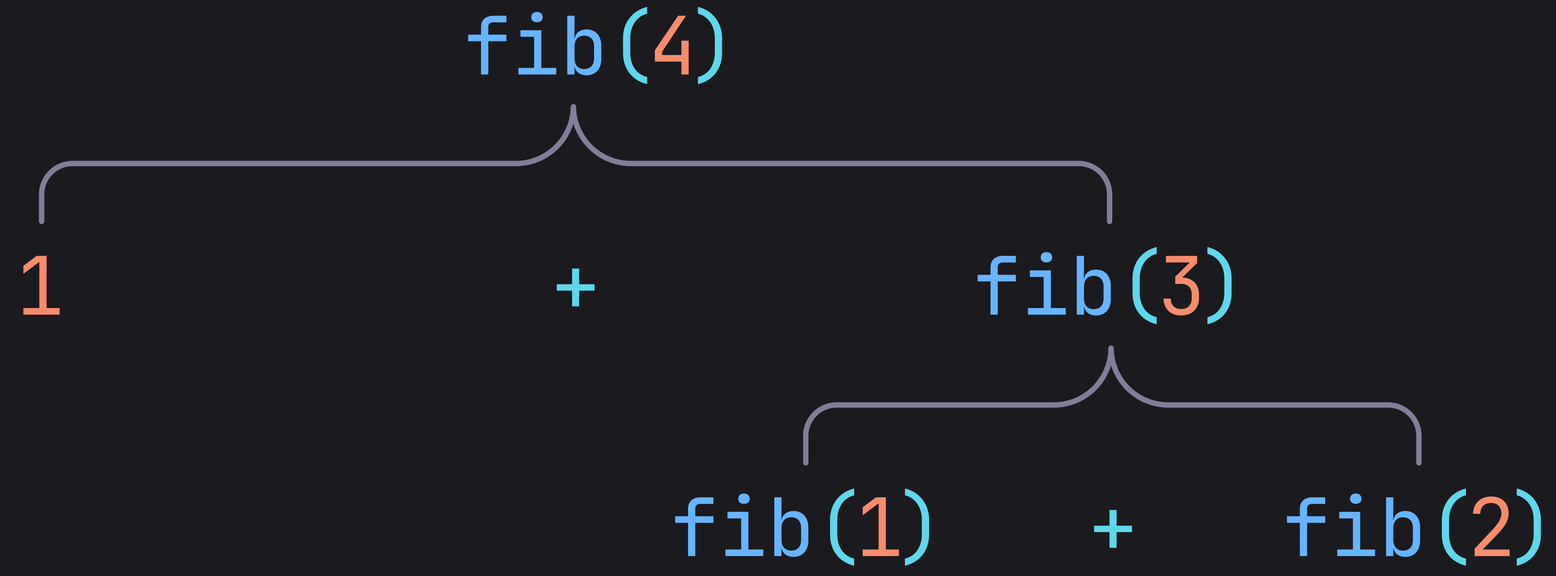


Fibonacci Sequence

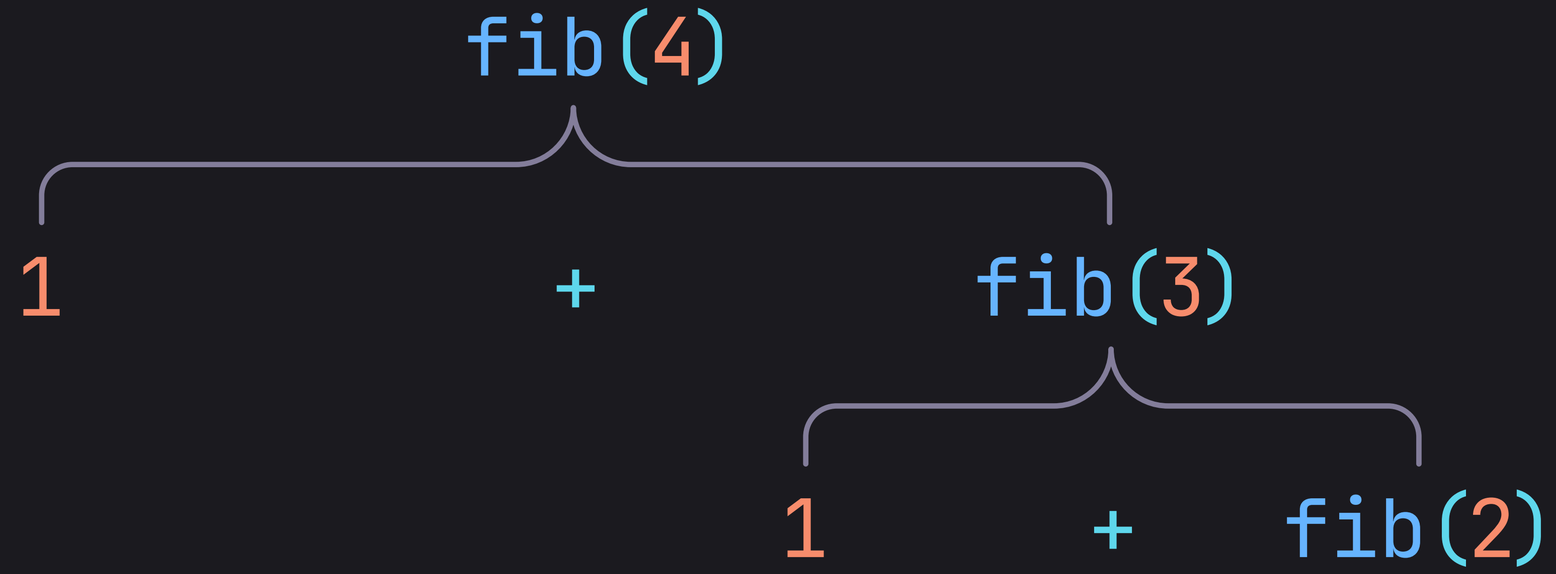


Fibonacci Sequence

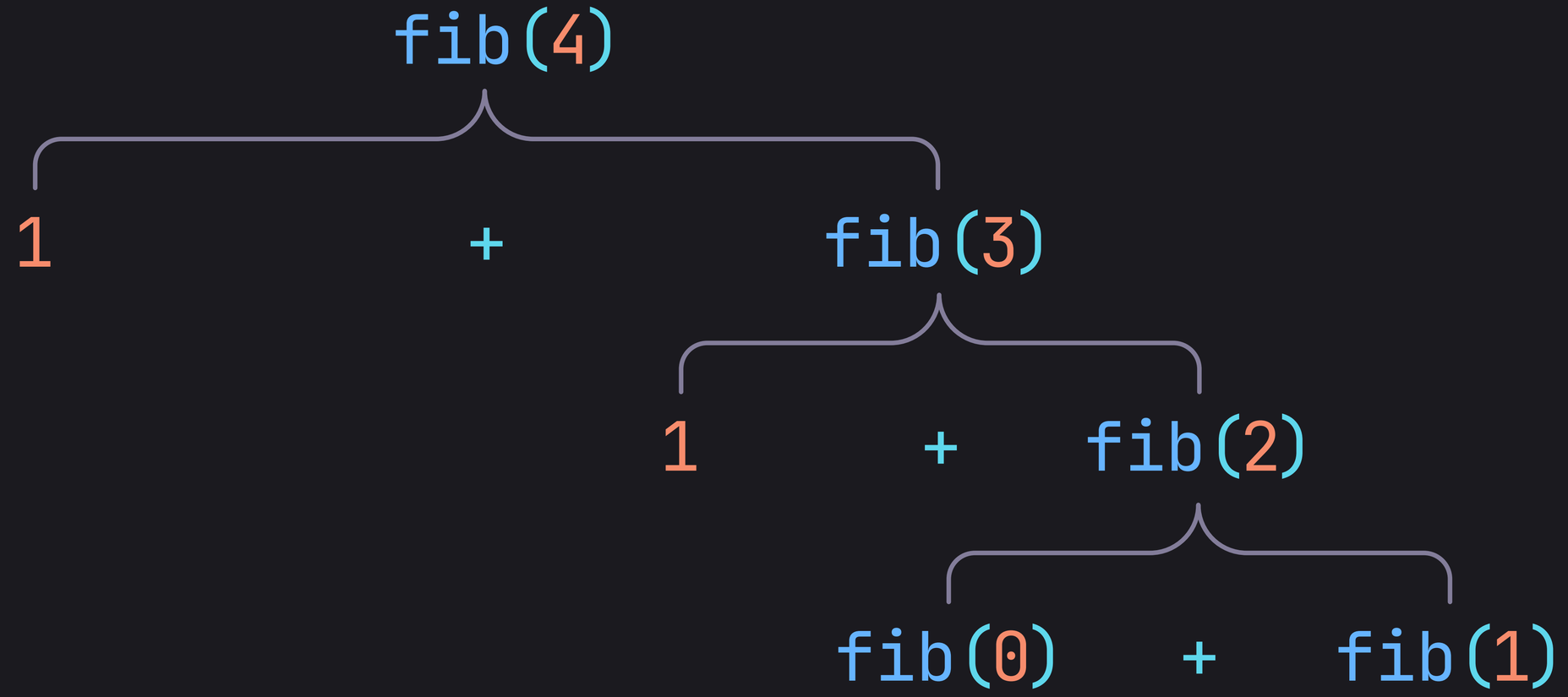
Fibonacci Sequence



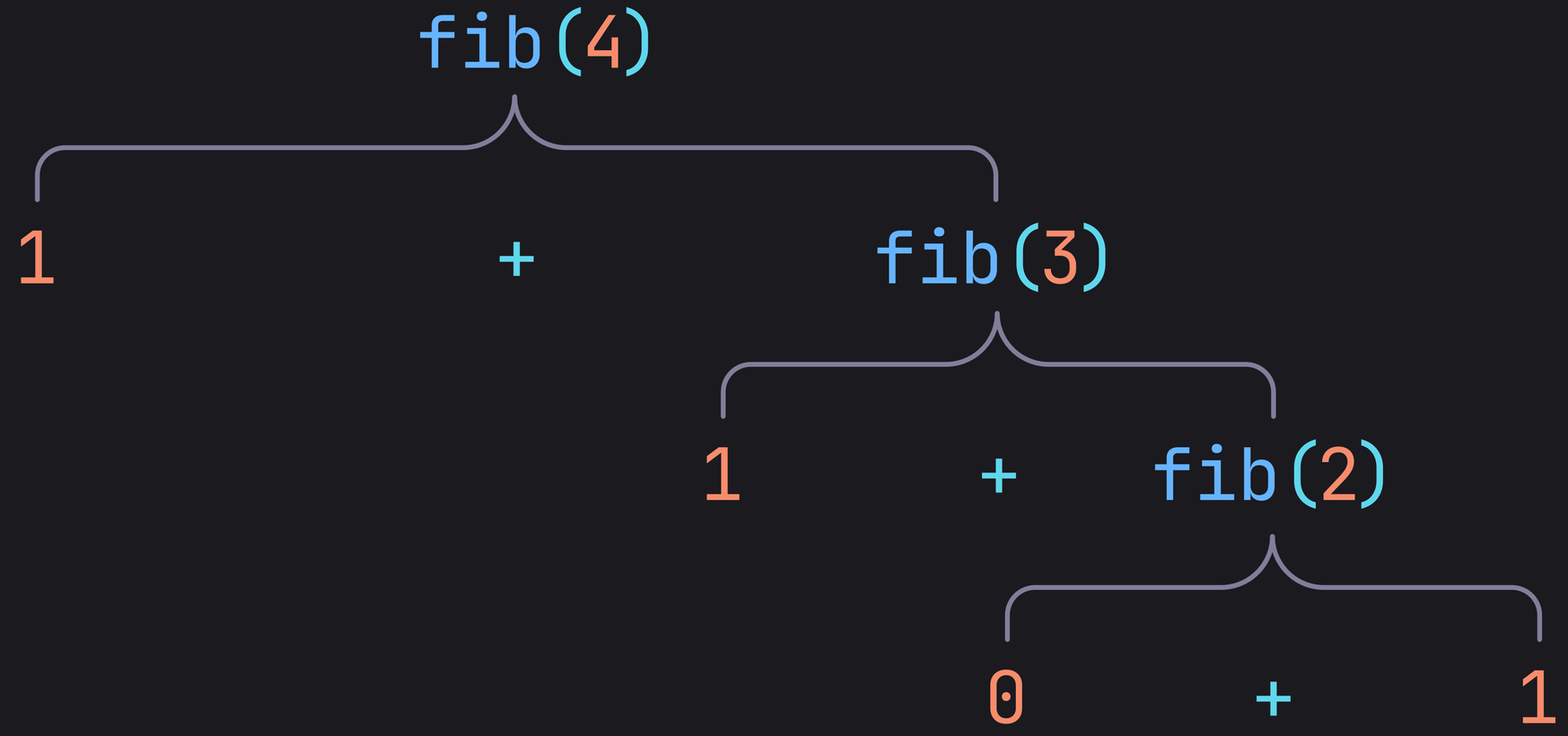
Fibonacci Sequence



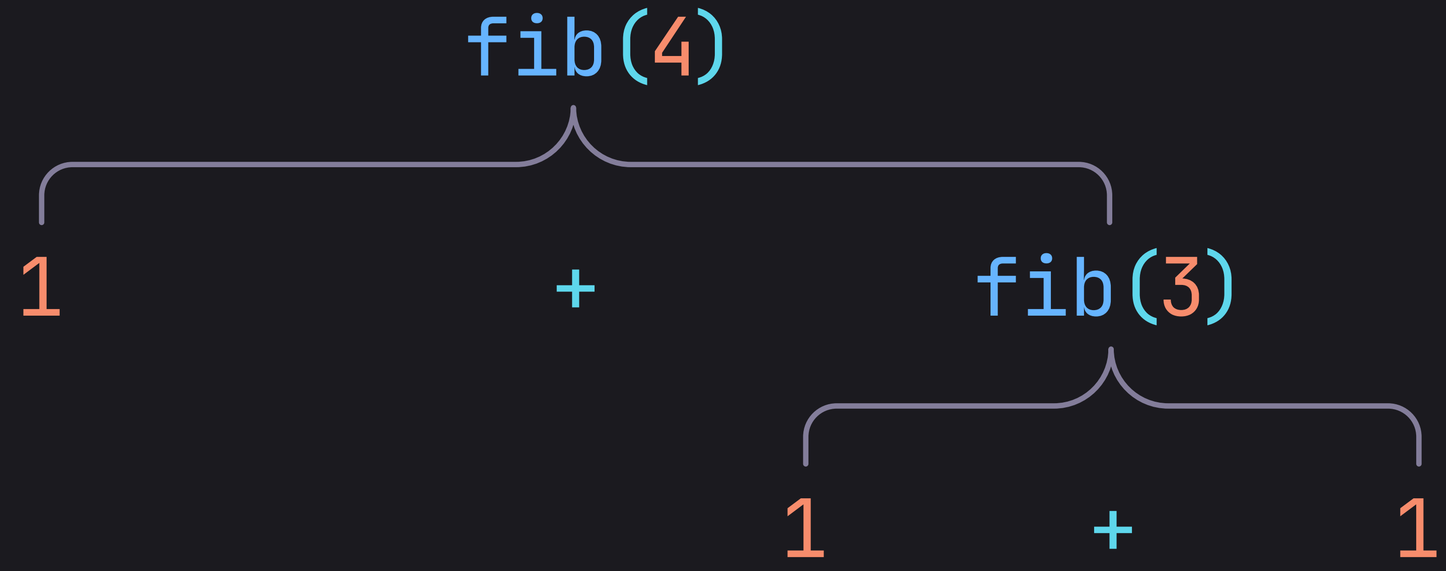
Fibonacci Sequence



Fibonacci Sequence



Fibonacci Sequence

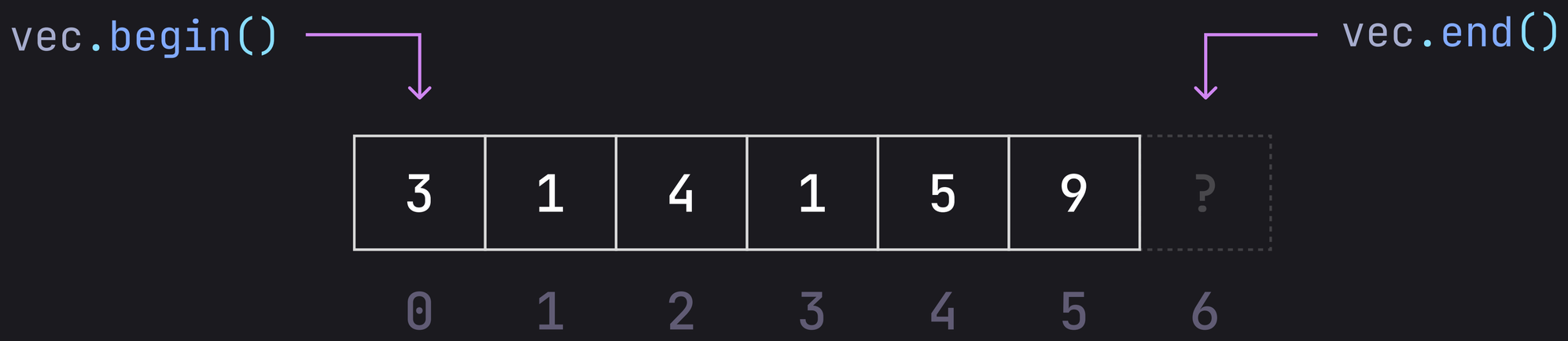


Fibonacci Sequence

Fibonacci Sequence

3

Iterators



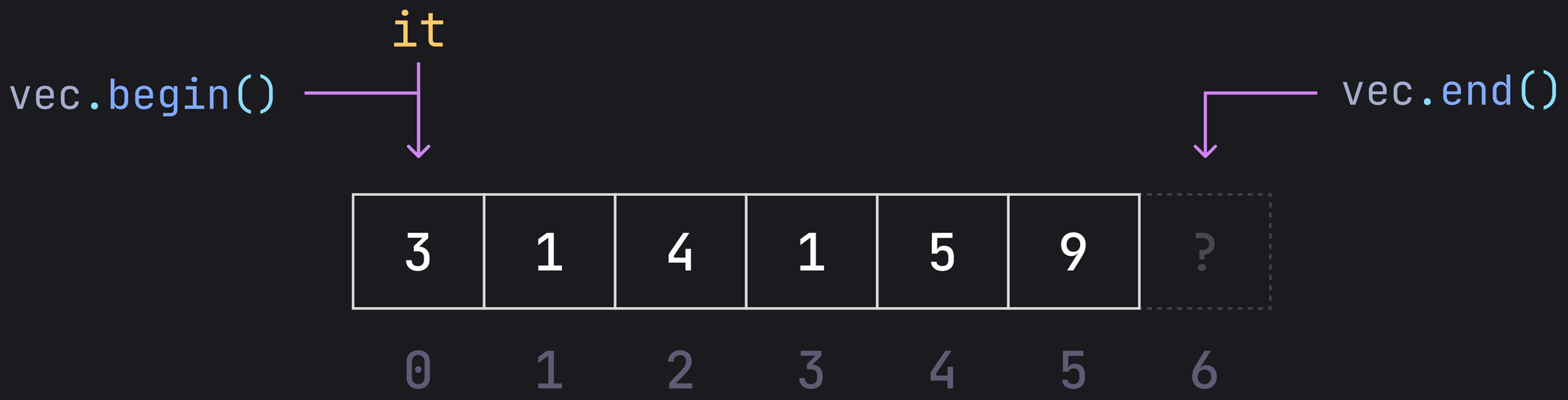
you can think of an iterator as a cursor that **points** to a specific element in the collection, and can be used to **move forward or backward** to access other elements.

Iterators

```cpp
std::vector<int> vec {3, 1, 4, 1, 5, 9};
auto it = vec.begin();
it++ // move the iterator forward
it-- //move the iterator backward
*it // access the value the iterator points to
```

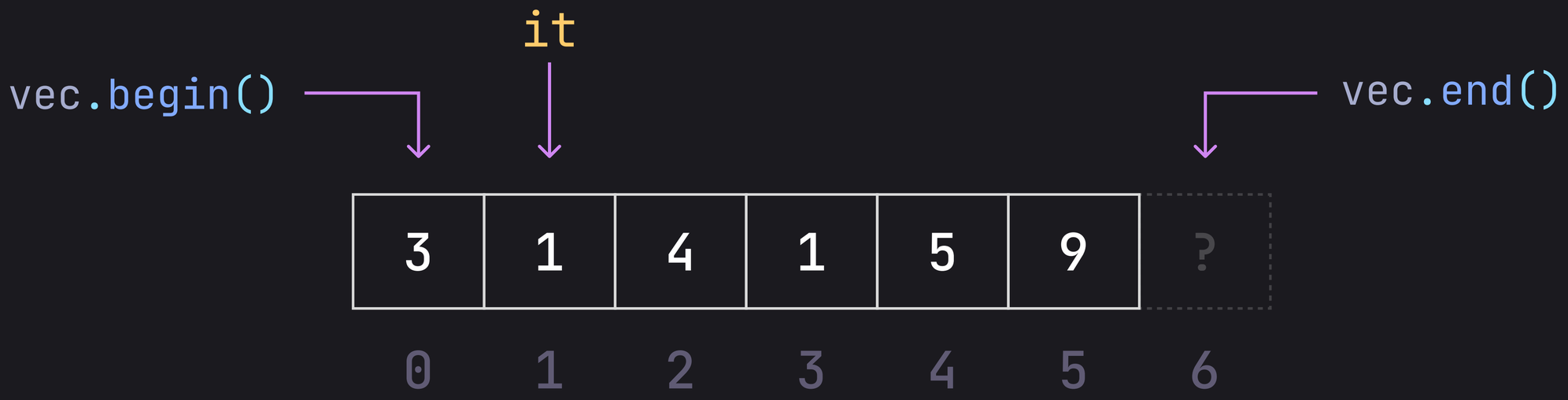
you can think of an iterator as a cursor that **points** to a specific element in the collection, and can be used to **move forward or backward** to access other elements.

Iterators




```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

Iterators




```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

Iterators


```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

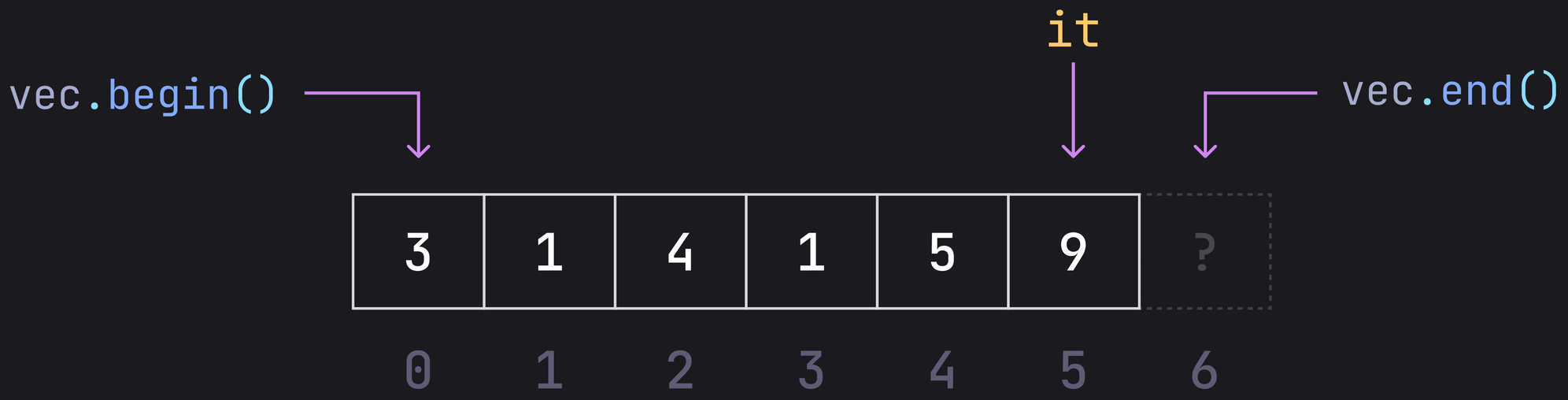
Iterators


```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

Iterators

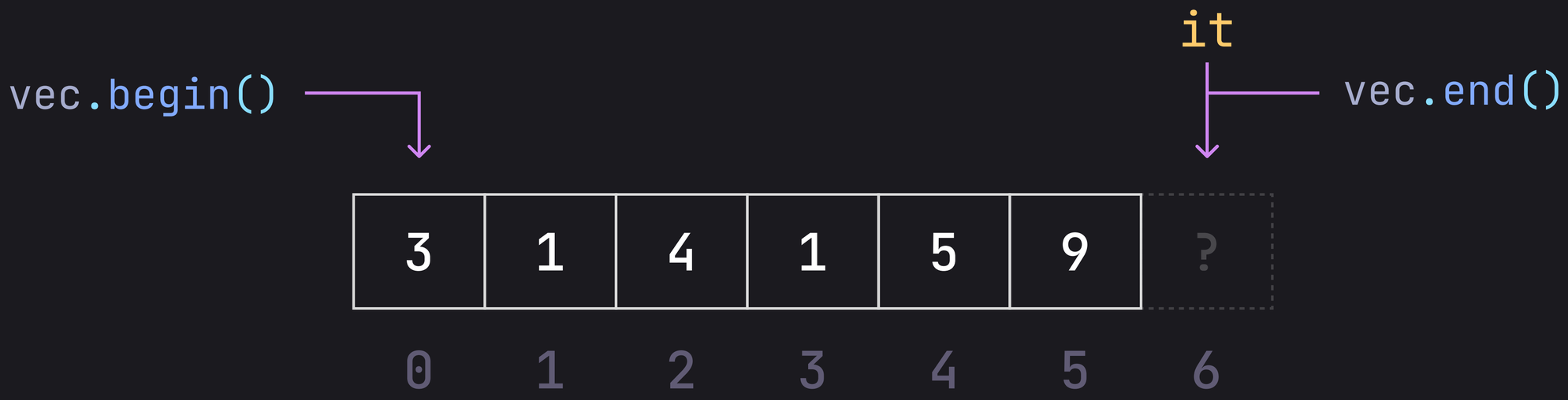

```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

Iterators



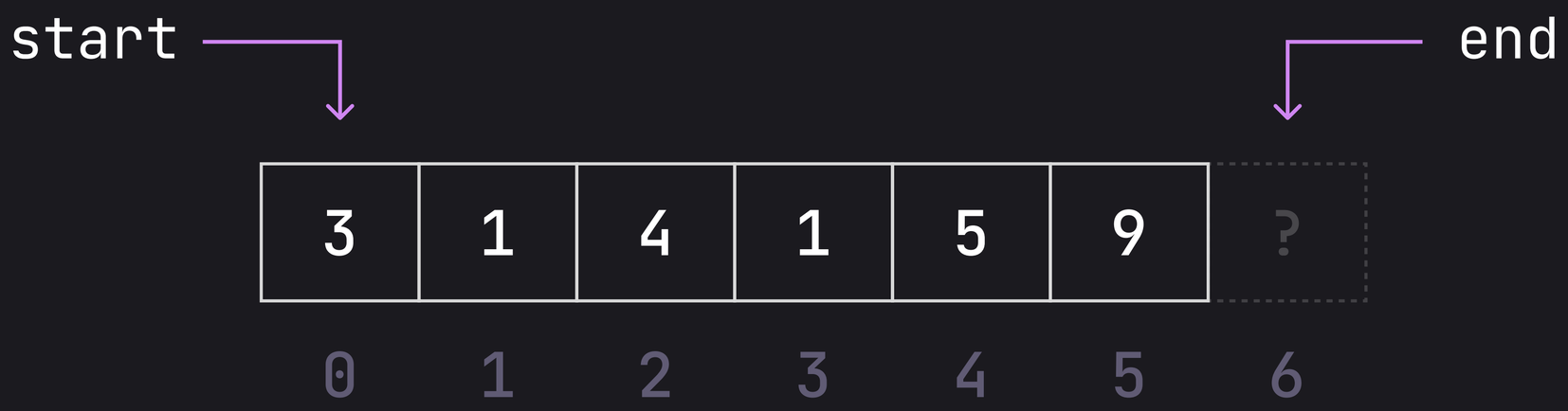

```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

Iterators




```
1 std::vector<int> vec {3, 1, 4, 1, 5, 9};
2 for (auto it = vec.begin(); it ≠ vec.end(); it++) {
3     std::cout << *it << " ";
4 }
```

Iterators




```
1  std::vector<int> vec {3, 1, 4, 1, 5, 9};
2  auto start = vec.begin();
3  auto end = vec.end();
```

Iterators


```
1  std::vector<int> vec {3, 1, 4, 1, 5, 9};
2  auto start = vec.begin();
3  auto end = vec.end();
4  auto mid = start + (end - start) / 2;
```

But what is an iterator?

it is an instance of a class that has the following methods and operators

```
class Iterator {
public:
    T& operator*();
    Iterator& operator++();
    Iterator& operator--();
    bool operator==(const Iterator& x, const Iterator& y)
    bool operator≠(const Iterator& x, const Iterator& y)
}
```

How do I add an iterator to the MyVector class?

implement the Iterator class and add the following methods to the MyVector class

```
Iterator MyVector<T>::begin()
Iterator MyVector<T>::end() const
```

Let’s have a go on Ed