Lesson 2

Introduction to C++

Assignment operator

Assignment operator is a binary operator: `a = b`
value of a variable can be changed with the help of an assignment operator: `x = 4`
Precedence: 16.
Associativity: right-to-left

Note: Don’t confuse assignment operator with variable initialization!

Assignment operator:

```
int x;
x = 4;
```

Variable initialization:

```
int x = 3;
```

Note: Assignment operator returns the assigned value.

```
int x = 0;
std::cout << (x = 3) << std::endl;
```

Will print out 3, because `x = 3` returns new value of `x` which is 3.

This idea can be used to assign multiple variables to the same value:

```
x = y = z = 3;
```

Arithmetic operators

Binary operators:
Addition: `a + b`
Subtraction: `a - b`
Multiplication: `a * b`
Division: `a / b`
Remainder: `a % b`

Presedence:
`*, /, %` operators: 5
`+, –` operators: 6

Associativy: left-to-right

Unary operators:
Unary plus: `+ a`
Unary minus: `- a`

Use case of unary –: `int x = - y`;
Use case of unary +: will be shown later.

Presedence: 3

Associativy: right-to-left

Note 1: In C++, division of two integers will also return integer: `7 / 3 = 2`
Note 2: Remainder operator should be applied only to integer types: `7 % 3 = 1`

Compound assignment arithmetic operators

Special binary operators `+=, -=, *=, /=, %=` are named compound assignment arithmetic operators.
Presedence: 16 (same as assignment operator)
Associativity: right-to-left (same as assignment operator)

Theese operators should be used to increase or decrease the value of the variable (let `x` be 5):
`x *= 3`;
Now `x` is 15.

Just like the assignment operator, left operand of compound assignment arithmetic operator should be lvalue (for example, this is prohibited: `5 += 4`, because 5 is an rvalue and not an lvalue)

Just like the assignment operator, compound assignment arithmetic operators return the new value of the assigned variable (let x be 2 by default):

```
std::cout << (x -= 3) << std::endl;
```

Will print out -1, because `x -= 3` returns new value of `x`, which is -1

This idea can be used to update multiple variables. For example (let x, y and z be 5 by default):

```
z /= x += y *=  z %= x -= 3;
```

Increment and decrement operators

Unary operators:
Prefix increment: `++ a`
Prefix decrement: `-- a`
Postfix increment: `a ++`
Postfix decrement: `a --`

Presedence:
Postfix operators: 2
Prefix operators: 3 (Same as unary + and –)

Associativy:
Postfix operators: left-to-right
Prefix operators: right-to-left

Example (let x be 0 by default):
```
std::cout << x++ << std::endl;
std::cout << ++x << std::endl;
```
Will print out 0 and then 2.

Note: Just like the compound assignment operators, increment and decrement operators should be provided with an lvalue as an operand.

Note: both prefix and postfix increment operators **in**crease the value of the variable by 1. Both prefix and postfix decrement operators **de**crease the value of the variable by 1.

Note: **prefix** increment or decrement operators return the **new** value of the varible, when **postfix** increment and decrement operators return the **previous** value the variable

Comparison operators

Binary operators:
equal: `a == b`
not equal: `a != b`
less: `a < b`
less or equal: `a <= b`
greater: `a > b`
greater or equal: `a >= b`

Presedence:
Relational operators: 9
Equality operators: 10

Associativy: left-to-right

Note: theese operators are used to compare numbers, they return boolean value which indicates the result of the comparison.

Example:

```
int x = 4;
int y = 5;
std::cout << (x < y) << std::endl;
```

prints out 1 (true)

```
std::cout << (y == x) << std::endl;
```

prints out 0 (false)
etc…

Logical operators

Binary operators:
Logical not: `!a`
Logical and: `a && b`
Logical or: `a || b`

Presedence:
Logical not: 3
Logical and: 14
Logical or: 15

Associativy: left-to-right

Note: theese operators are used to calculate the values of logical expressions. Boolean values are passed to these operators, and the output is also a Boolean value.

Example:
```
bool x = true;
bool y = false;
std::cout << (x && y) << std::endl;
```
prints out 0 (false)
```
std::cout << (x || y) << std::endl;
```
prints out 1 (true)
```
y = !y
std::cout << (x && y) << std::endl;
```
prints out 1 (true)

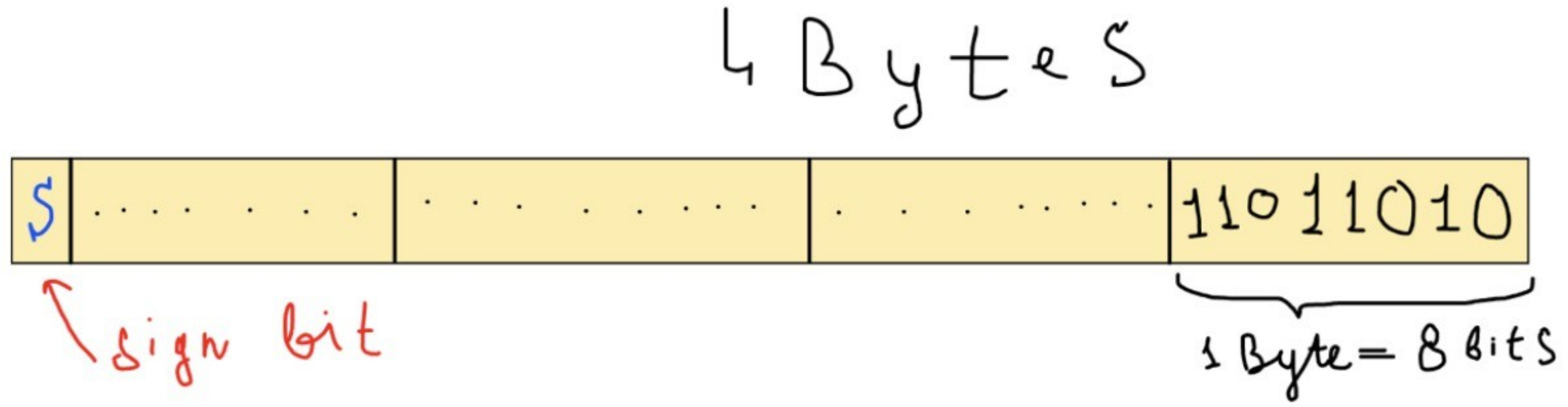
Operator sizeof

Operator sizeof:
Receives name of the type, literal, or identifier and returns an amount of **bytes**, needed for provided value to be stored in memory.

Examples (true for `apple clang-1205.0.22.9` compiler and the majority of other compilers):
`sizeof(int)` returns 4
`sizeof(54)` also returns 4 (because `54` is **int** literal)
`sizeof(123ull)` returns 8 (because `123ull` is an **unsigned long long** literal)
`sizeof('a')` returns 1 (because `'a'` is a **char** literal)
etc…

Storing integers in memory

Integer variables are stored as a sequence of bits (zeros or ones):



$S = 0 \Rightarrow$ num is positive

$S = 1 \Rightarrow$ num is negative

if num ≥ 0, than the binary form of it is stored inside the memory: num_2

if num < 0, than the binary form of the following value is stored inside the memory:

$$\left(2^{32} - |num|\right)_2$$

Storing other integer types in memory

The same mechanism is applied for other signed integer types (but size of the memory block will be different):

`short`: 2 Bytes
`int` : 4 Bytes
`long` : 4 Bytes
`long long` : 8 Bytes

Note: sizes of theese types are **not** standardized, i.e. they can be different, but the following rule must be preserved:
`sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`

Storing unsigned integer types in memory

For unsigned integers, the formula is simpler: the binary form of the variable is just stored inside the memory, i.e. there is no sign bit for unsigned variables.

`unsigned short`: 2 Bytes
`unsigned int` (or just `unsigned`): 4 Bytes
`unsigned long`: 4 Bytes
`unsigned long long` : 8 Bytes

Bitwise operators

Unary operator:
Bitwise not: `~a`

Binary operators:
Bitwise and: `a & b`
Bitwise xor: `a ^ b`
Bitwise or: `a | b`
Bitwise left shift: `a << b`
Bitwise right shift: `a >> b`

Presedence:
Bitwise not: 3
Bitwise shifts: 7
Bitwise and: 11
Bitwise xor: 12
Bitwise or: 13

Associativy: left-to-right

Note: theese operators are used to operate with bits of the variables.

Semantics:
- `~a` returns bitwise invertion of the number.
- `a & b, a | b, a ^ b` applies and, or and xor operations to the variables a and b respectively.
- `a << b, a >> b` shifts the binary representation of the number a by b bites to the left and right respectively.

Note: there is a compound assignment version of each binary bitwise operator:
`&=, ^=, |=, <<=, >>=`
Presedence: 16
Associativity: right-to-left (same as every other assignement operator)

Ternary operator

Ternary operator:
`a ? b : c`

Presedence: 16 (same as assignement operators)
Associativy: left-to-right

Semantics:
- `a` – boolean expression
- `b` and `c` should be of the same type (or should be convertible to the same type)
- Ternary operator returns `b` if a is true, otherwise `c`.

Example:
```
int a = 5;
int b = 3;
int max_two = (a > b) ? a : b;
```

Other operators

C-style cast:
```
(type)
```

Presedence: 3 (same as assignement operators)
Associativy: right-to-left

Semantics:
- casts expression to the given type

Example:
```
unsigned short a = (unsigned short)5 + (unsigned
short)true;
```

Scope resolution operator

Scope resolution (aka *four dots* operators**)**:

```
::
```

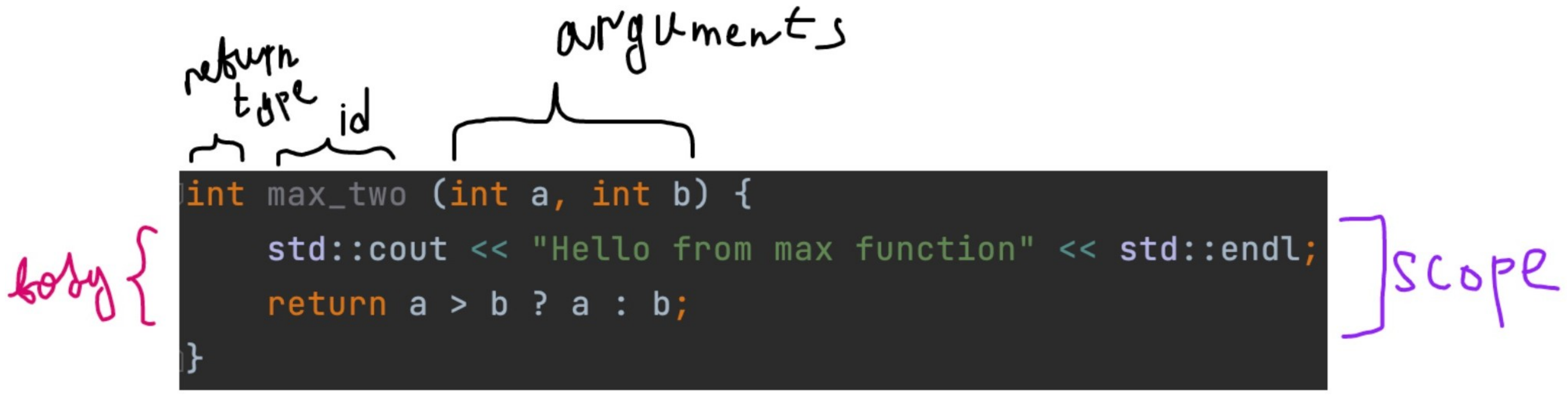
Presedence: 1 (highest possible)
Associativy: left-to-right

Example:

```
std::cout << … << std::endl;
```

Functions

In C++, you can create functions in order to avoid copypaste and combine specific logic.
For example, calculation of the n-th Fibbonacci number, or calculating max of two values.



```
int max_two (int a, int b) {
    std::cout << "Hello from max function" << std::endl;
    return a > b ? a : b;
}
```

Operator function call

In order to call your function, you should use operator function call:

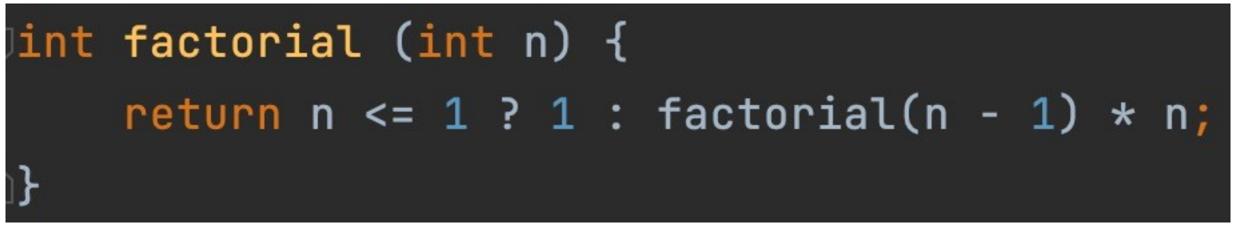
```
func_id(arg1, arg2, …, argn);
```

Presedence: 2
Associativy: left-to-right

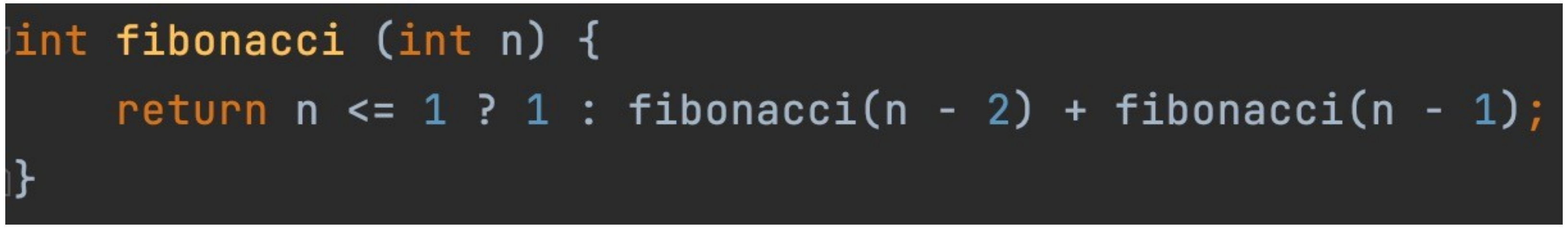
Note:
arguments are not guaranteed to be calculated sequentially from left to right!

Recursion

In C++, you can call a function from within itself:



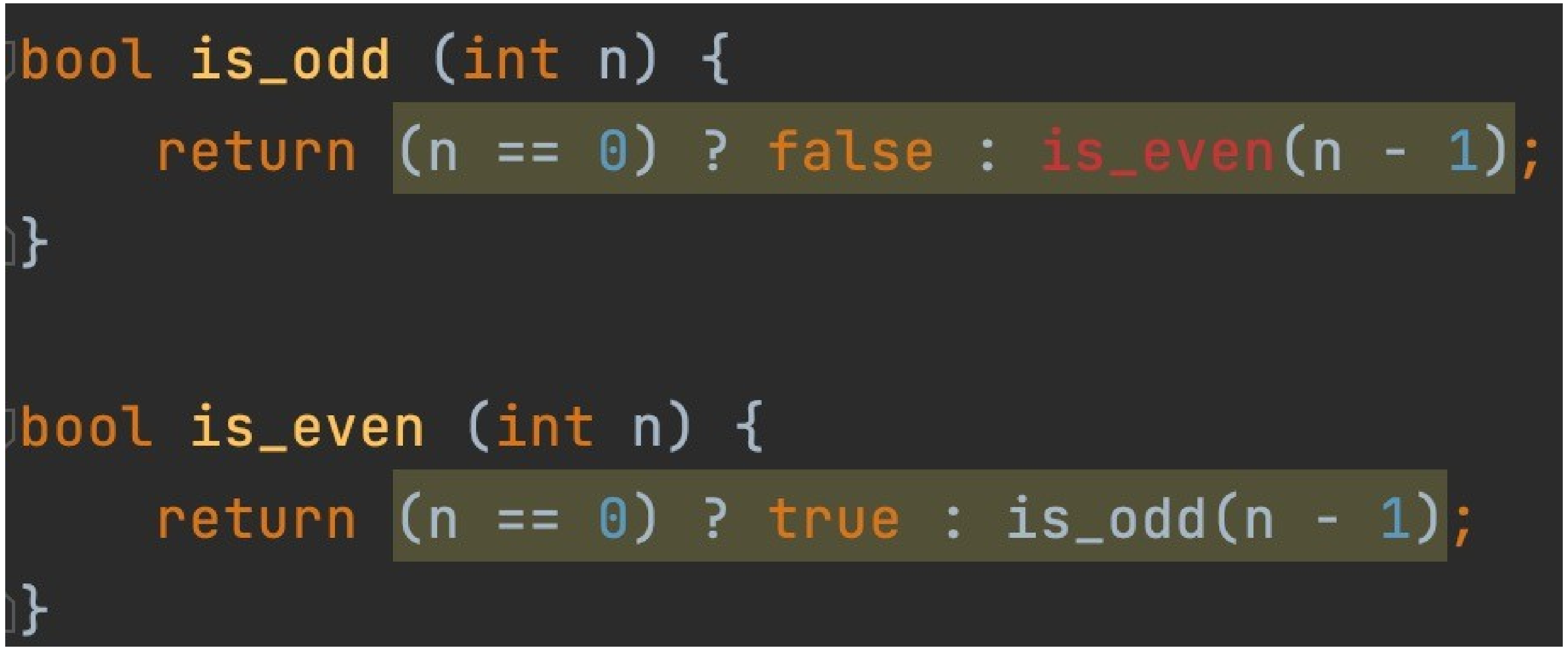
```cpp
int factorial (int n) {
    return n <= 1 ? 1 : factorial(n - 1) * n;
}
```



```cpp
int fibonacci (int n) {
    return n <= 1 ? 1 : fibonacci(n - 2) + fibonacci(n - 1);
}
```

Function declaration

How to you compile this code =(?

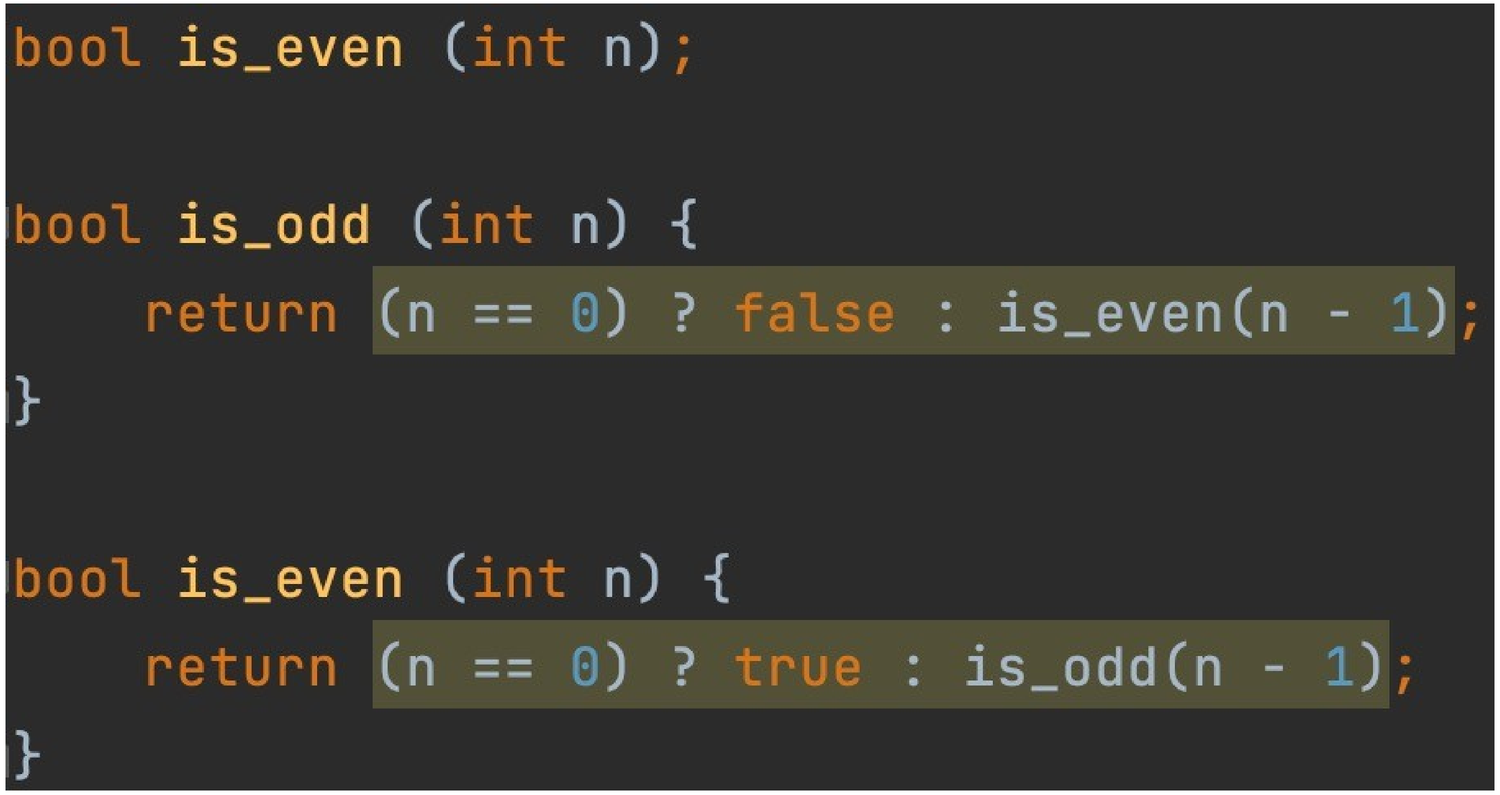


```
bool is_odd (int n) {
    return (n == 0) ? false : is_even(n - 1);
}

bool is_even (int n) {
    return (n == 0) ? true : is_odd(n - 1);
}
```

Function declaration

Solution: predeclare the second function:



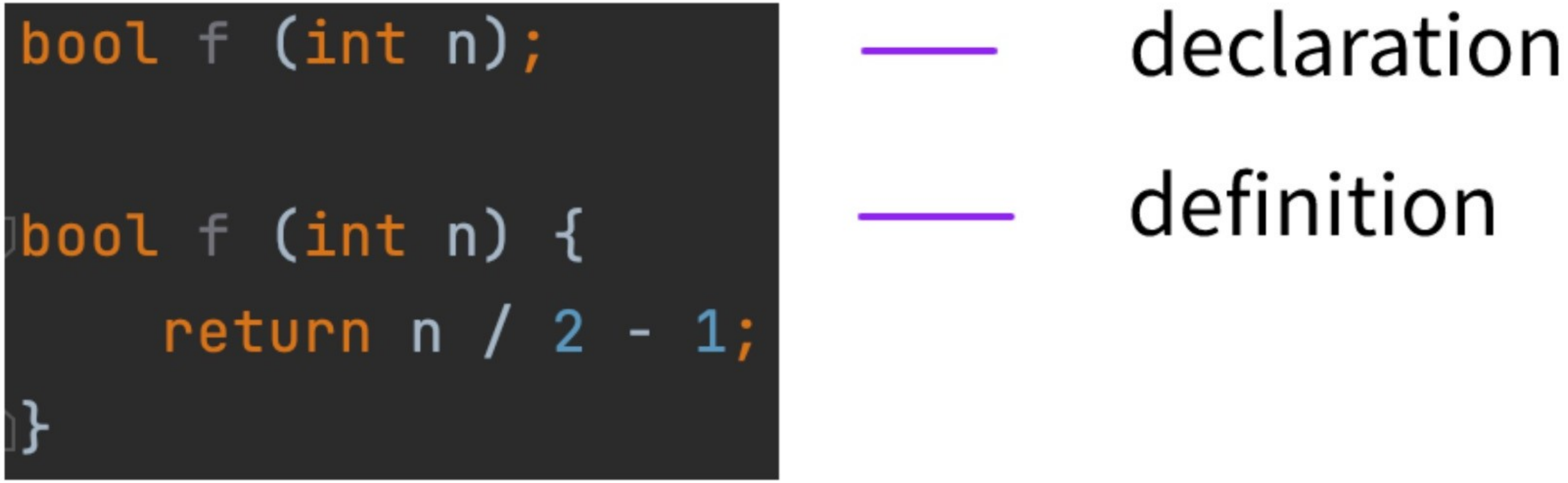
```
bool is_even (int n);

bool is_odd (int n) {
    return (n == 0) ? false : is_even(n - 1);
}

bool is_even (int n) {
    return (n == 0) ? true : is_odd(n - 1);
}
```

Declaration vs definition

Function can be declared multiple times, but should be defined only once!



```
bool f (int n);

bool f (int n) {
    return n / 2 - 1;
}
```

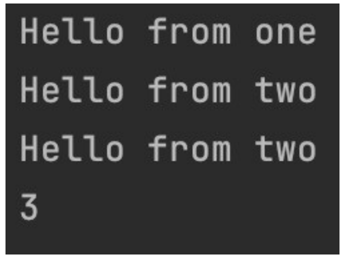
Comma operator

Comma operator:

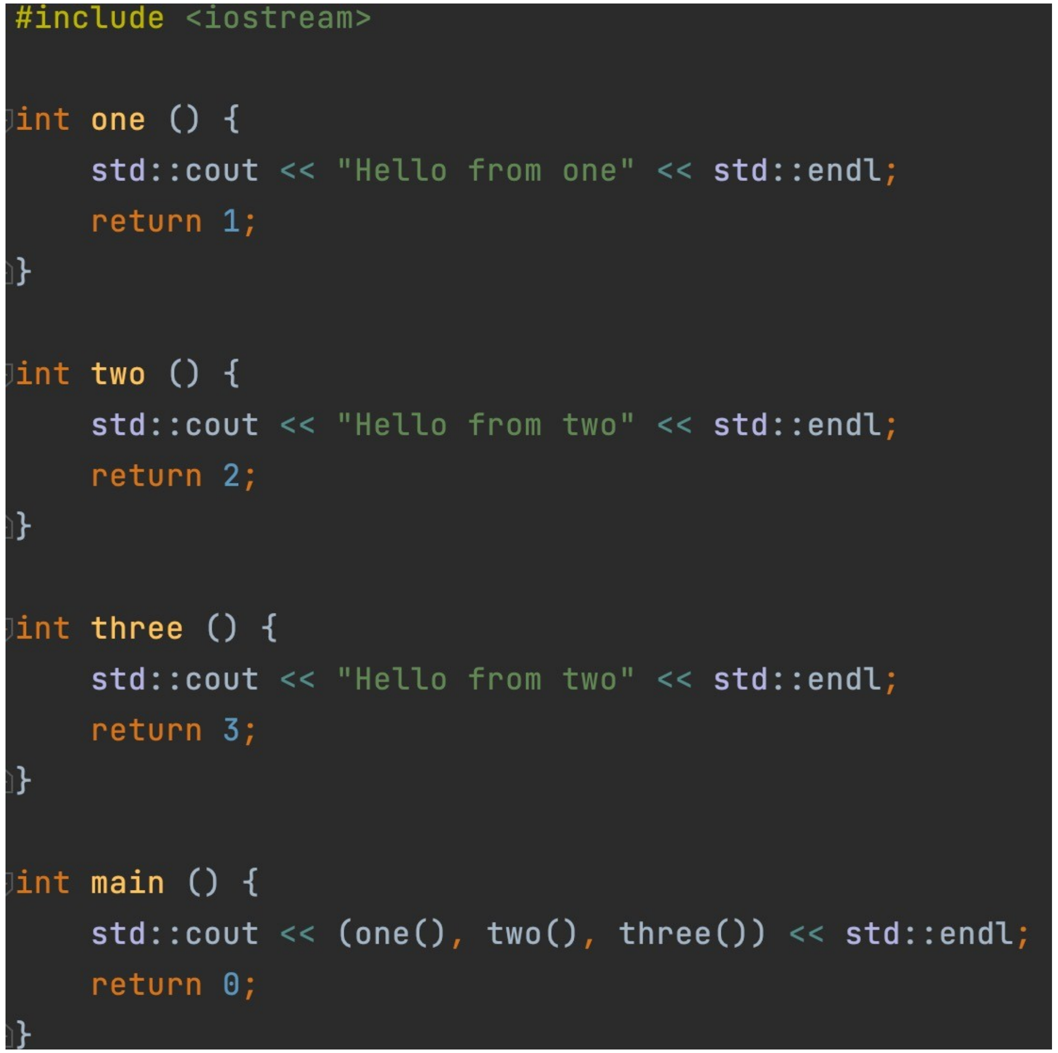
,

Presedence: 17 (lowest possible)
Associativy: left-to-right

Usage:
Calculates the left argument, then the right argument and returns the result of right argument



```
Hello from one
Hello from two
Hello from two
3
```



```cpp
#include <iostream>

int one () {
    std::cout << "Hello from one" << std::endl;
    return 1;
}

int two () {
    std::cout << "Hello from two" << std::endl;
    return 2;
}

int three () {
    std::cout << "Hello from two" << std::endl;
    return 3;
}

int main () {
    std::cout << (one(), two(), three()) << std::endl;
    return 0;
}
```

Logical operators vs Bitwise operators

Optimisations:

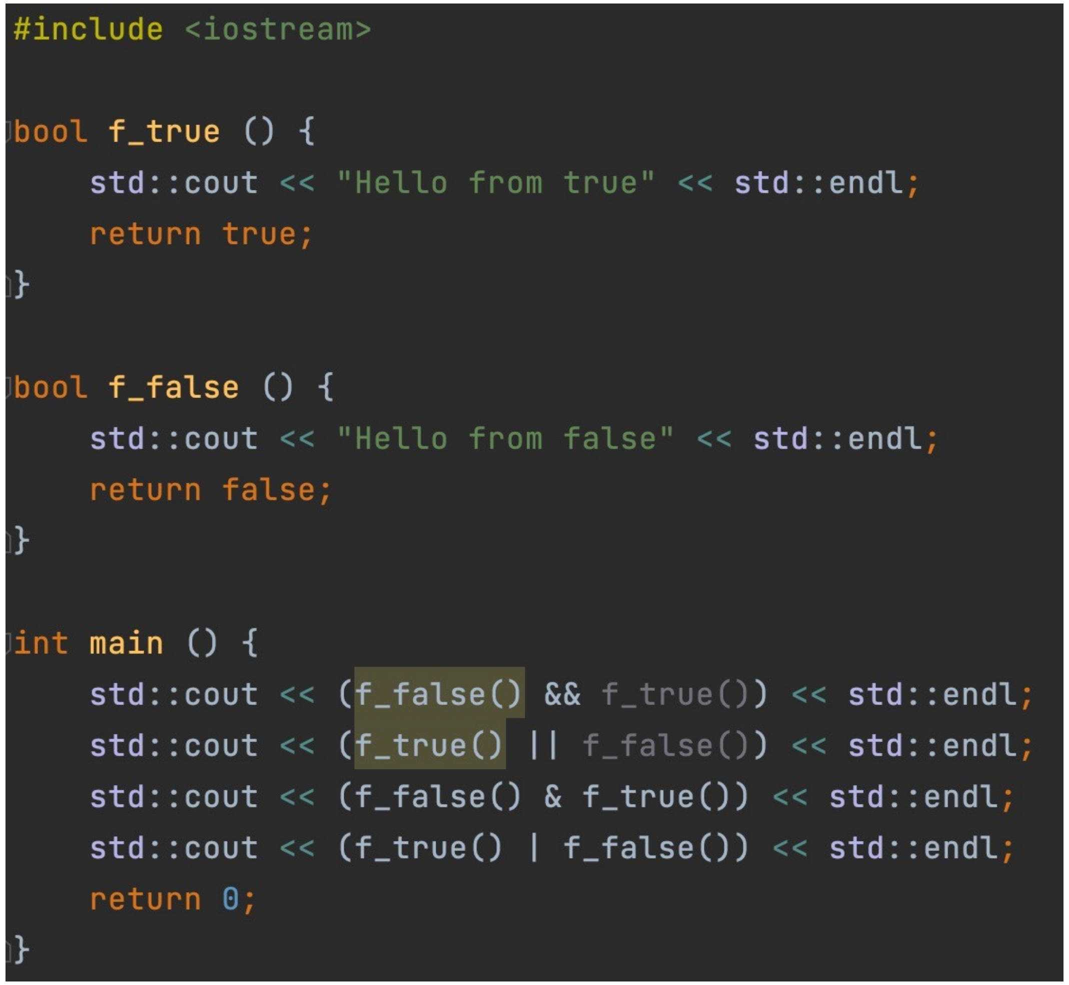
In the case of operator `&&`, if the first argument evaluates to false, then, the second argument isn’t getting calculated, because the result of the expression can already be predicted.

The same rule is followed in the case of operator `||` if the first argument evaluates to true.

But there is no such rule for bitwise operators `|` and `&`

So, operators `|` and `&` are also applicable for boolean expressions too!
(If you want to force the calculation of both arguments)

Logical operators vs Bitwise operators

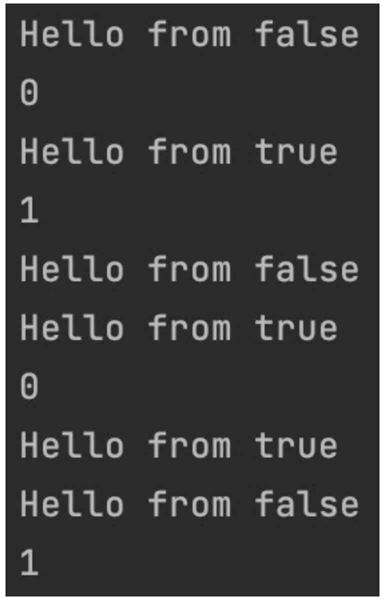


```cpp
#include <iostream>

bool f_true () {
    std::cout << "Hello from true" << std::endl;
    return true;
}

bool f_false () {
    std::cout << "Hello from false" << std::endl;
    return false;
}

int main () {
    std::cout << (f_false() && f_true()) << std::endl;
    std::cout << (f_true() || f_false()) << std::endl;
    std::cout << (f_false() & f_true()) << std::endl;
    std::cout << (f_true() | f_false()) << std::endl;
    return 0;
}
```



```
Hello from false
0
Hello from true
1
Hello from false
Hello from true
0
Hello from true
Hello from false
1
```