

Алгоритмизация и программирование

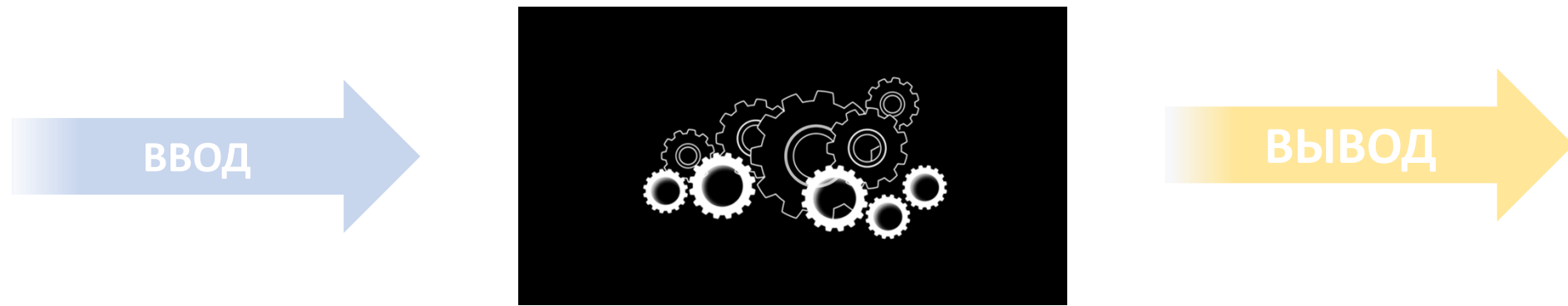
Лекция 2.2 (C++)

Введём ограничения

```
x = 42          # Так можно
```

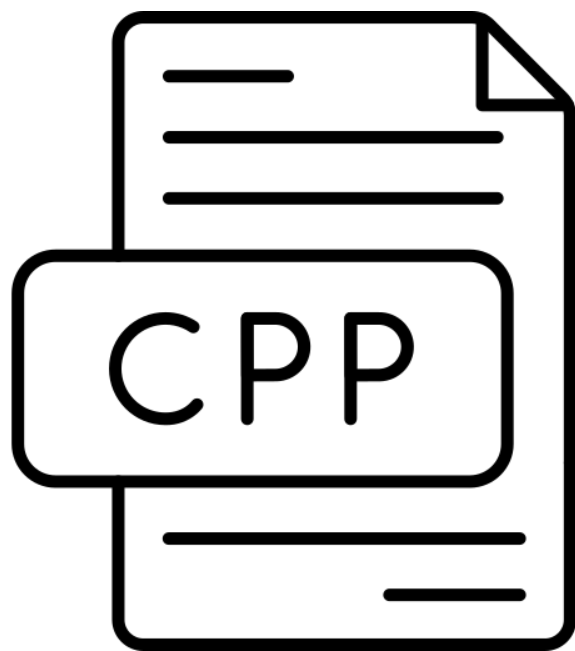
```
x:int = 42      # Так пишут
```

Модель программы



* МОНОЛИТ С ВВОДОМ И ВЫВОДОМ ТОЛЬКО В КОНСОЛЬ

Только один файл



Весь код программы
полностью в одном файле

Всего 2 секции

Файл делим на 2 секции, в пределах которых разрешено писать код.

```
1 #include <iostream>
```

В этой секции только подключаем библиотеки

```
2  
3  
4  
5 int main() {  
6     // your code goes here  
7  
8  
9  
10 }
```

Весь код пишем только в этой секции

Минимальная программа

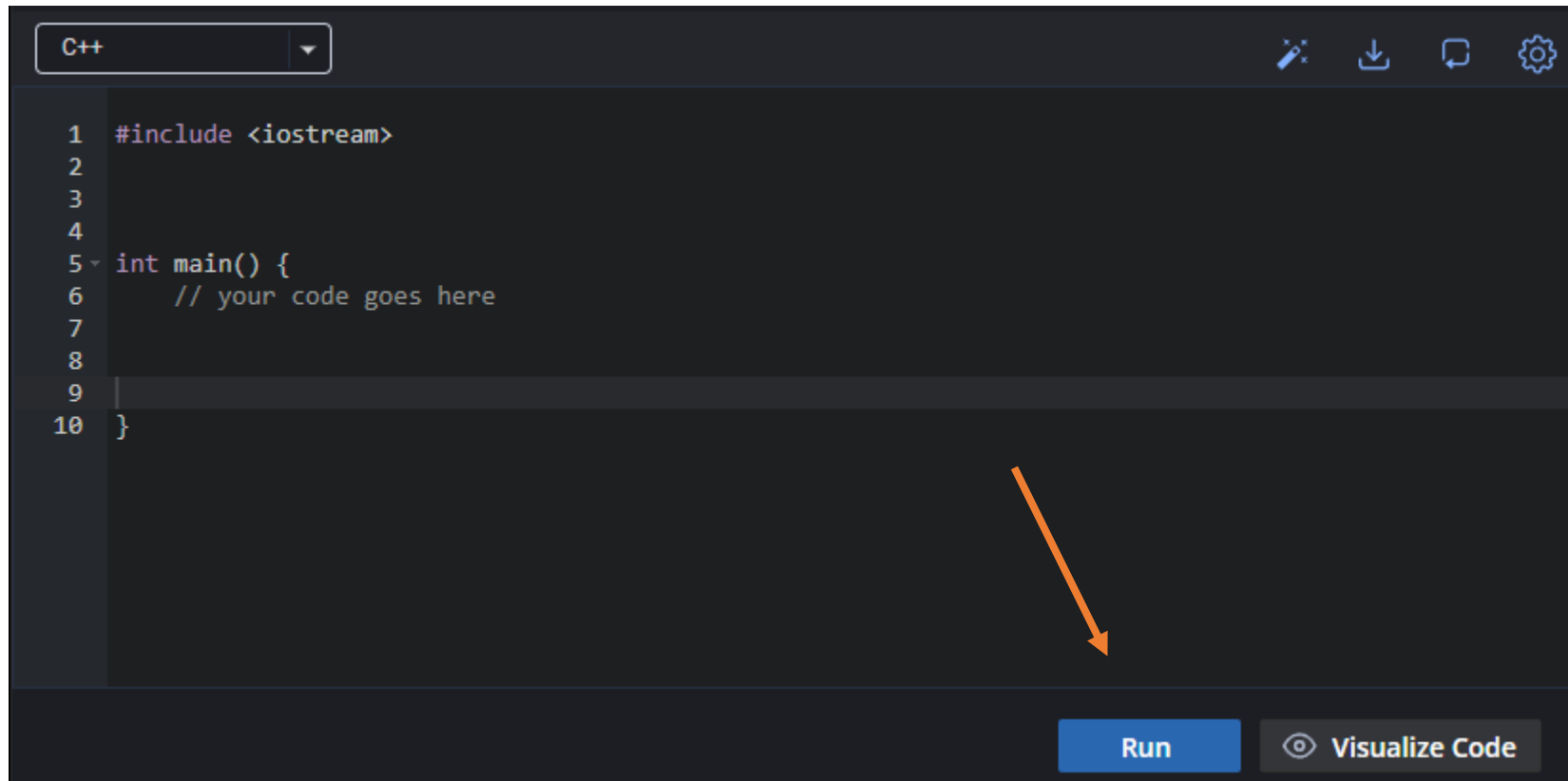
```
int main(){}  

```

```
int main() {  
    return 0; // по стандарту можно не писать  
}
```

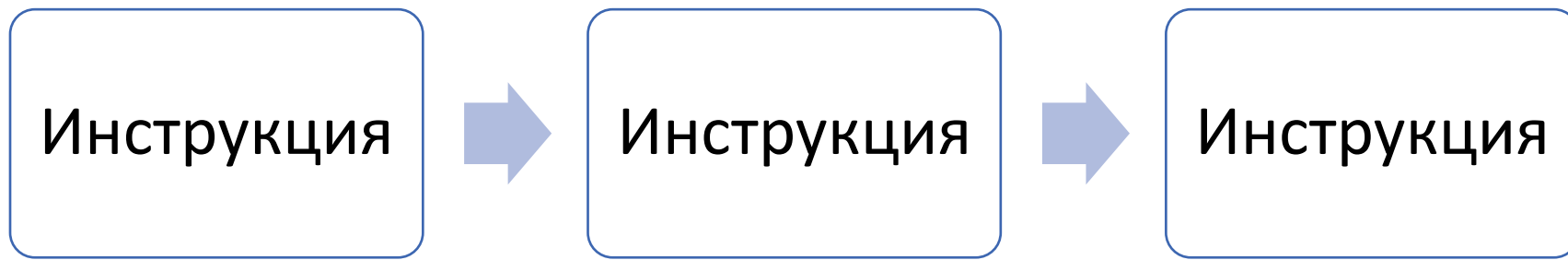
Программа просто запускается

Мы не управляем процессом сборки программы и не думаем о нём. По нажатию кнопки происходит магия и программа просто запускается.



Один поток

Процессор исполняет инструкции в одном потоке.



Программа

```
int main(){
```

Стейтмент

Стейтмент

Стейтмент

Стейтмент

Стейтмент

Стейтмент

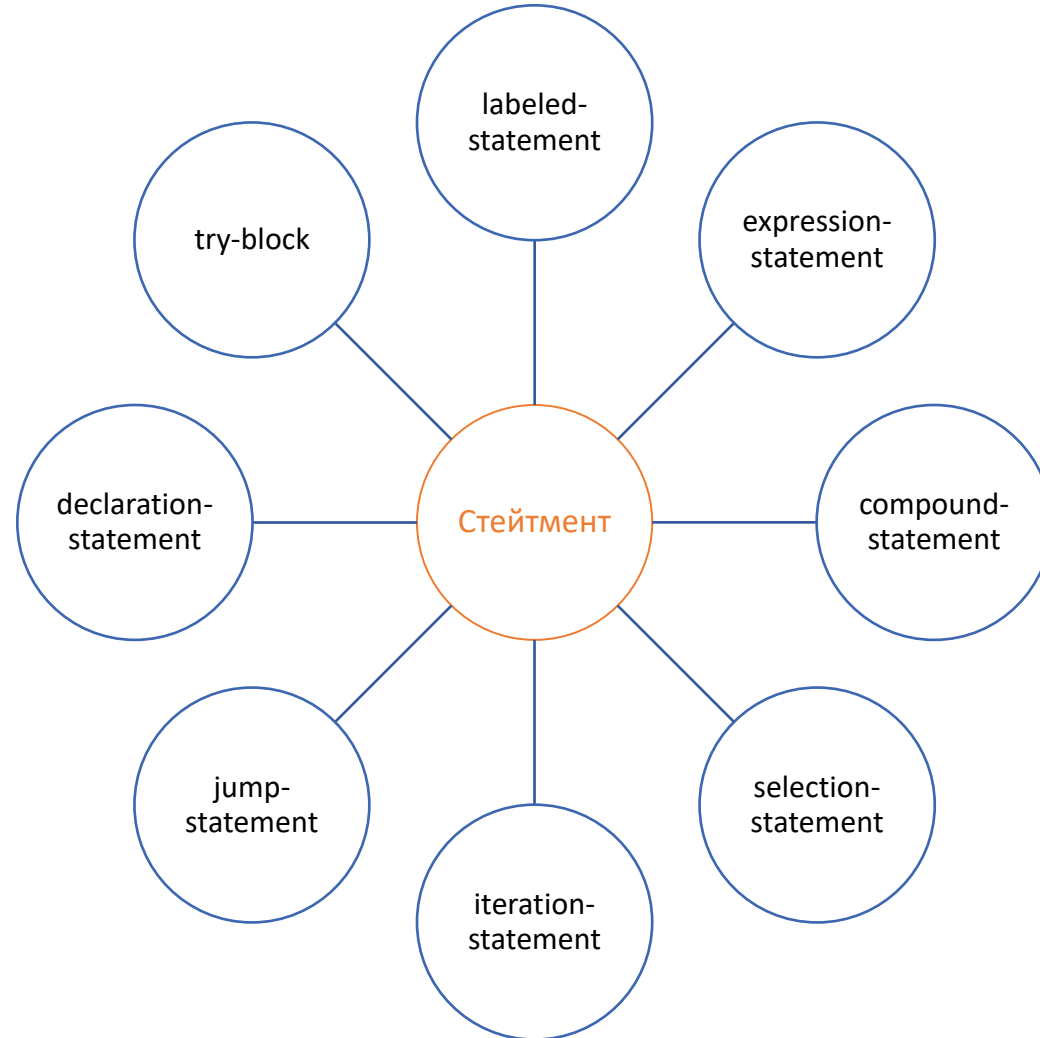
Стейтмент

```
}
```

исполняется и НЕ
возвращает значение

Виды стейтментов

Виды стейтментов



Expression Statement (Стейтмент-выражение)

Любое **выражение**, завершающееся точкой с запятой, становится стейтментом.

выражение ;

Empty Statement (Пустой стейтмент)

Если выражение отсутствует, то получается пустой стейтмент, который не делает ничего. Это не отдельный вид, а частный случай **Expression Statement**.

;

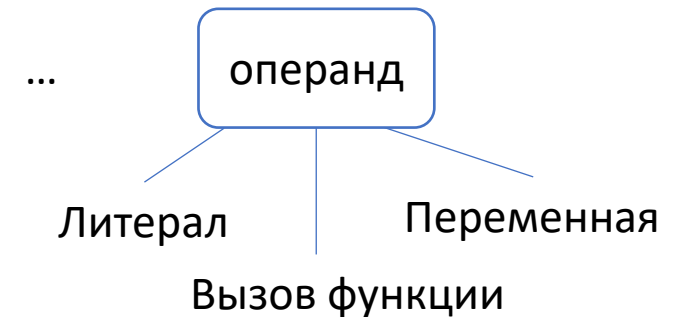
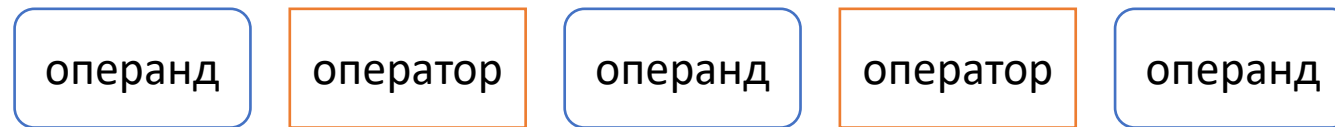
Empty Statement (Пустой стейтмент)

```
int main(){
```

}

Выражение (expression)

Выражение (expression) - последовательность операторов, переменных, литералов и вызовов функций, которая **вычисляется и возвращает значение**.



Любое из этого является выражением:

- Отдельная переменная: x (возвращает своё значение);
- Литерал: 42 (возвращает 42);
- Вызов функции: $\text{sqrt}(25)$ (возвращает 5.0);
- Комбинация всего вышеперечисленного с операторами: $(a + b) * 2 - \text{func}()$.

Литерал

Литерал в C++ — это явное представление фиксированного значения в исходном коде. Это константы, которые непосредственно записываются в программе и не могут быть изменены во время выполнения.

```
42;           // целочисленный литерал (integer literal)
3.14159;      // литерал с плавающей точкой (floating-point literal)
'A';         // символьный литерал (character literal)
"Hello";      // строковый литерал (string literal)
true;         // булев литерал (boolean literal)
nullptr;      // указатель литерал (pointer literal)
1.2_w;        // Пользовательский литерал (user-defined literal)
```


Операторы

Оператор — это заранее определённый символ или ключевое слово в языке, которое выполняет **конкретную операцию** над одним или несколькими операндами и возвращает значение. Порядок выполнения операторов определяется приоритетом и ассоциативностью.

Операторы бывают:

- **Унарные** (работают с одним операндом): $-x$, $++i$, $!flag$, $*ptr$ (разыменование).
- **Бинарные** (работают с двумя операндами): $a + b$, $c = 10$, $x > y$, $n \% 2$.
- **Тернарный** (работает с тремя операндами): единственный такой оператор — условный $a ? b : c$.

Примеры операторов:

$=$, $+$, $-$, $*$, $/$, $\%$, $==$, $!=$, $<$, $>$, $<=$, $>=$, $\&\&$, $||$, $!$, $++$, $--$, $\&$ (адрес или побитовое И), $->$ и т.д.

Приоритет операторов

Приоритет определяет порядок выполнения операторов в выражении. Если выражение содержит несколько операторов, то операторы с более высоким приоритетом выполняются раньше, чем операторы с более низким приоритетом.

Operator Precedence	
$X = 10 + 20 * 30$	Multiply will happen first as * has <i>higher</i> precedence
$X = 10 + 600$	Now addition will happen as + has <i>lower</i> precedence
$X = 610$	We have got the <i>solution</i> for the equation

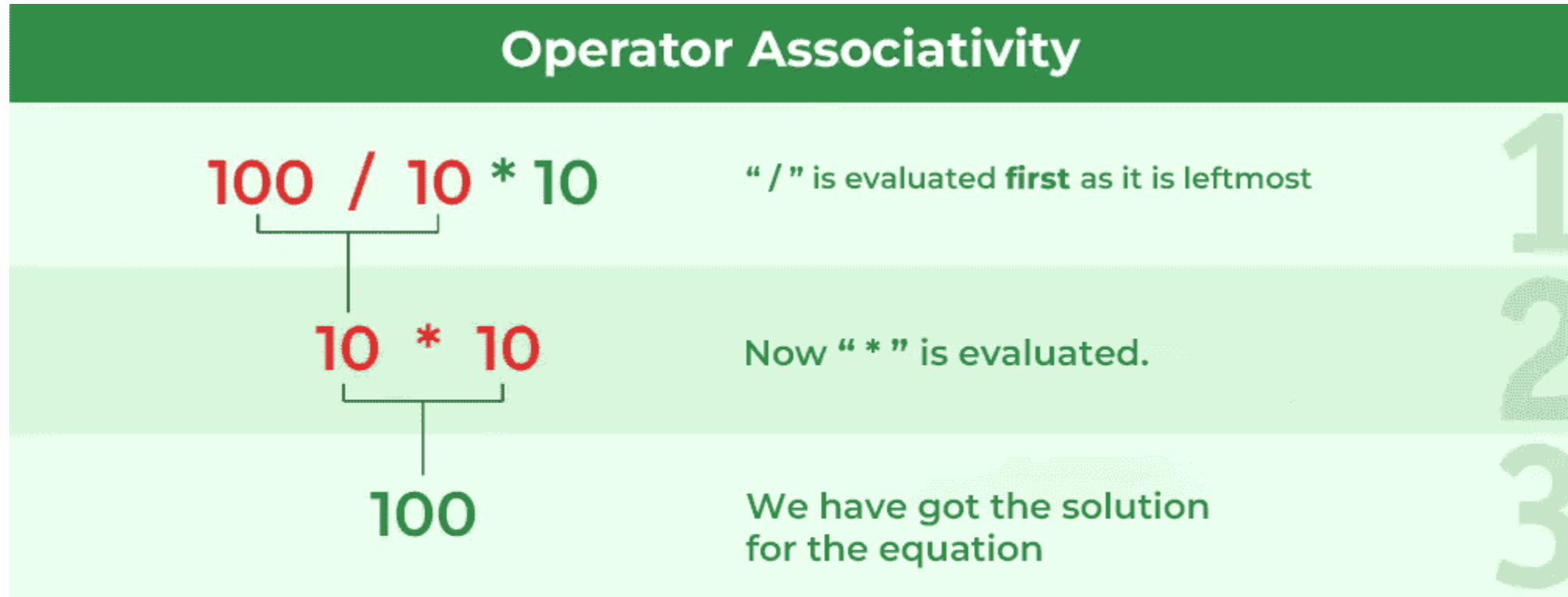
610

Ассоциативность операторов

Ассоциативность операторов определяет порядок, в котором операнды группируются, когда несколько операторов имеют одинаковый приоритет. Существует два типа ассоциативности:

- Ассоциативность **слева направо** означает, что при наличии в выражении нескольких операторов с одинаковым приоритетом они вычисляются слева направо. Например, в выражении $a + b - c$ сложение и вычитание имеют одинаковый приоритет и являются левоассоциативными, поэтому выражение вычисляется как $(a + b) - c$.
- Ассоциативность **справа налево** означает, что операторы вычисляются справа налево. Например, оператор присваивания $=$ является правоассоциативным. Таким образом, выражение $a = b = 4$ вычисляется как $a = (b = 4)$. Т.е. значение 4 сначала присваивается переменной b , а затем результат этого присваивания (b , которое теперь равно 4) присваивается переменной a .

Ассоциативность операторов



/ & * both have the same precedence but *left to right (LTR) associativity*

Комплексный пример

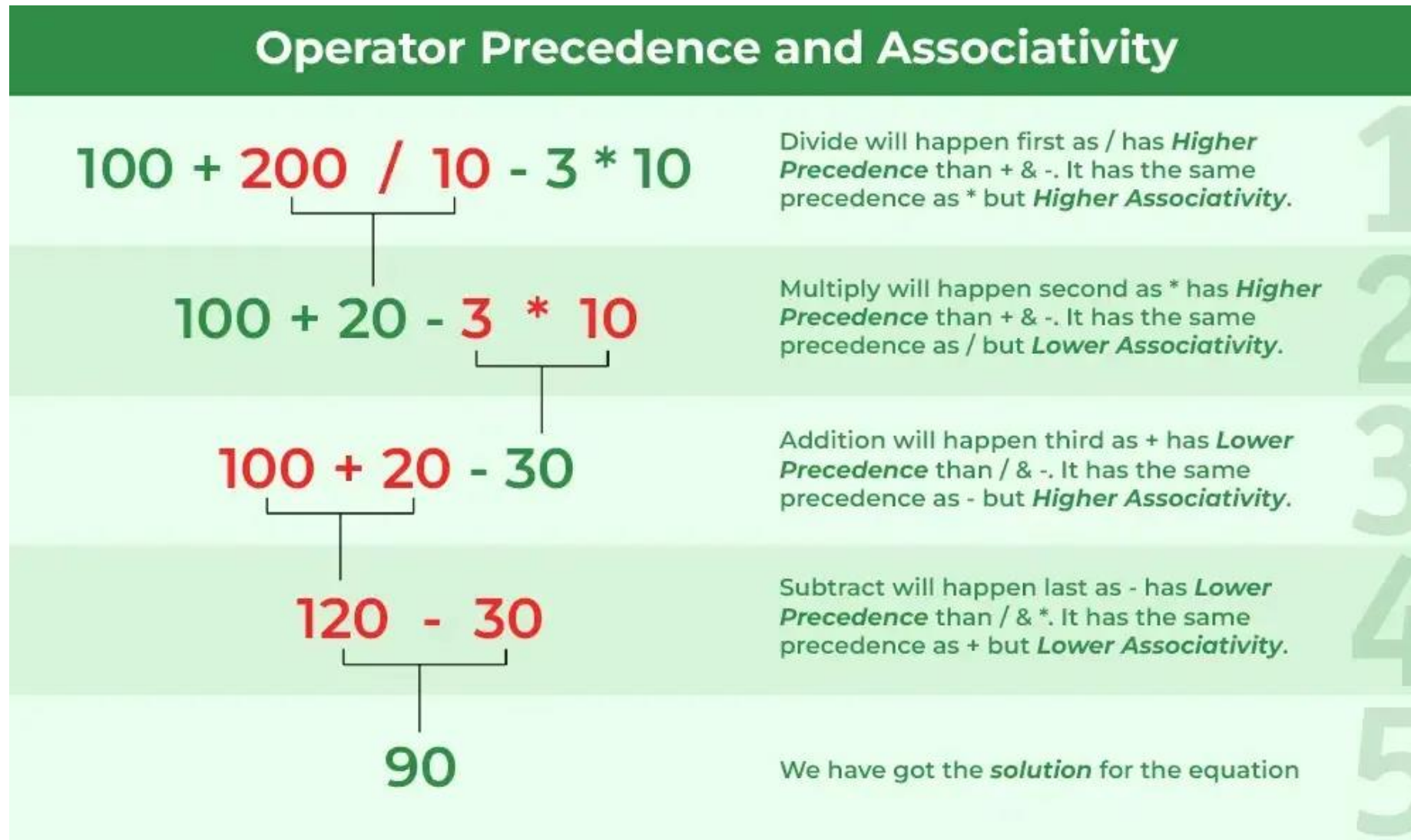


Таблица приоритетов

Полный список операторов, их приоритеты и ассоциативность можно найти в таблице, например: https://en.cppreference.com/w/cpp/language/operator_precedence.html

Precedence	Operator	Description	Associativity
1	<code>a::b</code>	Scope resolution	Left-to-right →
2	<code>a++</code> <code>a--</code> <code>type(a)</code> <code>type{a}</code> <code>a()</code> <code>a[]</code> <code>a.b</code> <code>a->b</code>	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	<code>++a</code> <code>--a</code> <code>+a</code> <code>-a</code> <code>!a</code> <code>~a</code> <code>(type)a</code> <code>*a</code> <code>&a</code> <code>sizeof</code> <code>co_await</code> <code>new - new[]</code> <code>delete - delete[]</code>	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of ^[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left ←
4	<code>a.*b</code> <code>a->*b</code>	Pointer-to-member	Left-to-right →
5	<code>a * b</code> <code>a / b</code> <code>a % b</code>	Multiplication, division, and remainder	
6	<code>a + b</code> <code>a - b</code>	Addition and subtraction	

Expression Statement

```
int main() {  
    ;  
    x;  
    42;  
    x + 42;  
    y * pow(x + 42);  
    z = 1 + y * pow(x + 42);  
    x + y  
}
```

Ошибка. Нет ';' ;
Компилятор будет искать
конец стейтмента и не найдёт

Expression Statement

```
#include <iostream>

int main() {
    std::cout << 2 + 2;
}
```


Compound Statement (Составной стейтмент / Блок)

Составной стейтмент начинается с открывающей фигурной скобки, затем указывается 0 или более стейтментов. Составной стейтмент заканчивается закрывающей фигурной скобкой.

```
{ } // пустой блок. 0 стейтментов внутри
```

```
{ // начало блока  
    x + 42;  
    y * pow(x + 42);  
    { // начало вложенного блока  
    } // конец вложенного блока  
} // конец блока
```

Compound Statement (Составной стейтмент / Блок)

Зачем вкладывать стейтменты в блок, если они итак прекрасно исполняются?

У блока есть 2 фишки:

- Внутри блока создаётся отдельная блочная область видимости (Block scope);
- Блок (вместе со всеми вложенными стейтментами считается одним стейтментом), поэтому в конструкция, в которых разрешено указывать один стейтмент можно использовать блок;

`if (условие)` стейтмент

слот под один любой стейтмент
можно использовать блок

Compound Statement

```
#include <iostream>
```

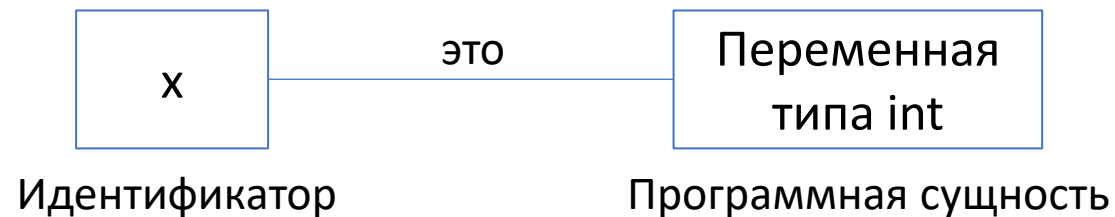
```
int main() {  
    { std::cout << 2 + 2; }  
  
    {  
        std::cout << 2 + 2;  
    }  
}
```

Declaration Statement (Стейтмент-объявления)

Declaration Statement **объявляет** в текущем блоке идентификатор и указывает, что за программная сущность обозначена этим идентификатором: переменная, функция, пространство имён и т.д.

Объявление - это способ, с помощью которого **имена** вводятся (или повторно вводятся) в программу на C++. Каждый тип объектов объявляется по-разному. После объявления имя можно использовать в программе.

```
int x;  // Теперь имя x доступно в программе
```



Объявление переменной

Переменная — это именованный блок данных определённого типа. Чтобы объявить переменную, нужно указать её тип и имя. В общем виде это выглядит так:

```
Type name;
```

```
Type name1, name2, name3;
```

```
Type name1 = value1, name2, name3 = value3;
```

```
auto name = value; // value инициализация
```

```
Type name = value; // Копирующая инициализация
```

```
Type name(value); // Прямая инициализация
```

```
Type name{value}; // Uniform инициализация
```

Declaration Statement

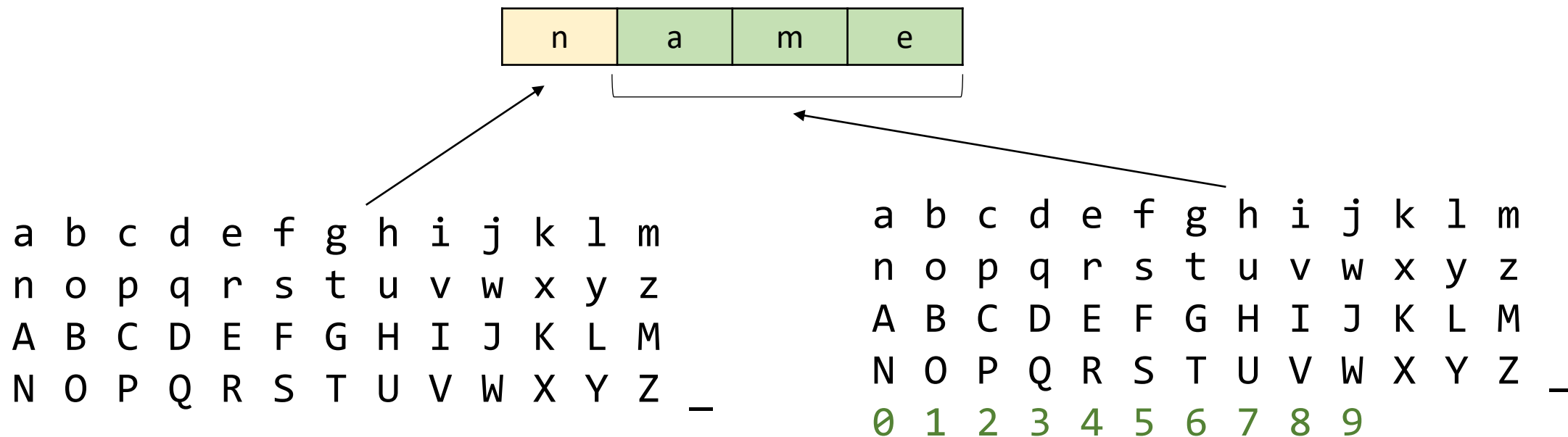
```
int main() {  
    int x;  
    double y;  
    bool flag = true;  
    auto pi = 3.14159;  
}
```

↑
double

инициализация

Идентификатор (Identifier)

Идентификатор (имя) - это последовательность букв и цифр произвольной длины.



Идентификатор

```
int main() {  
    int _;          // OK  
    int _42;        // OK  
    int 4cheeses;   // не OK  
  
    int myname;  
    int myName;     // camel case  
    int MyName;     // pascal case  
    int my_name;    // snake case  
  
    int Число😊 = 5; // OK  
}
```


Declaration Statement (Стейтмент-объявления)

До момента объявления компилятор ничего не знает об идентификаторе и любое его упоминание в коде считается ошибкой.

Можно представить это так: компилятор читает код программы сверху вниз и как только встретит объявление идентификатора, добавляет его в таблицу. Если компилятор встречает упоминание идентификатора в коде (не объявление), то он ищет его в своей таблице. Если идентификатор не найден, то компилятор сообщает об этом при помощи ошибки **имя was not declared** (идентификатор не был объявлен).

```
int x;
```

```
double y;
```

Идентификатор	Тип
x	int

Идентификатор	Тип
y	double
x	int

Declaration Statement (Стейтмент-объявления)

```
int main() {  
    x + 42;    // Ошибка. Имя x не объявлено  
    int x = 1;  
    x + 42;    // Ок. Имя x объявлено  
}
```

Scope (Область видимости)

Вышесказанное означает, что компилятор "видит" идентификатор начиная с определённой точки кода (Point of declaration).

Часть кода, в которой компилятор связывает идентификатор с конкретной программной сущностью называется **областью видимости** этого идентификатора.

```
int main() {  
    int x;  
    x + 42;  
}
```

point of declaration

Область видимости идентификатора x

Идентификатор x не виден

Scope

```
int main() {  
    int x;  
    x + 42;  
}
```

```
int main() {  
    int x = 1;  
    x + 42;  
}
```

```
int main() {  
    int x = x;  
    int y(y);  
    int z{z};  
    int k[k];  
}
```

 Область видимости

Объявление и Определение

Объявление:

```
extern int x;  
extern double y;
```

Идентификатор	Тип
y	double
x	int

Определение:

```
int x;  
double y;
```

Идентификатор	Тип	Адрес
y	double	0x7ffd99418a88
x	int	0x7ffd99418a84

Повторное объявление

```
int main() {  
    extern int a;    // Объявляем a  
    extern int a;    // ОК  
  
    extern double a; // Не ОК  
}
```

Повторное определение

```
int main() {  
    int a;  // Объявляем и определяем a  
    int a;  // Не OK нарушение ODR  
  
    double a; // Не OK нарушение ODR  
}
```

Пространство имён C

Если два идентификатора с одинаковым именем и разным типом лежат в области видимости друг друга, то это НЕ является ошибкой, в случае если они лежат в разных ограниченных пространствах имён C.

В языке C существует несколько отдельных пространств имён для разных типов идентификаторов:

- Структуры, классы, объединения и перечисления (struct, class, union, enum);
- Обычные идентификаторы (переменные, функции, typedef-имена);
- Метки (labels для goto).

Пространство имён C

```
#include <iostream>

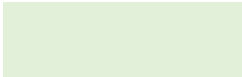
int main() {
    int x;           // 1. Обычная переменная типа int
    class x{         // 2. Объявление класса с именем x
        double data;

    };

    x:               // 3. Метка (label) с именем x

    std::cout << x;  // 4. Использование переменной x
    class x variable; // 5. Объявление переменной типа class x
    goto x;          // 6. Переход к метке x
}
```

Вложенные блоки

```
int main() {  
    int x;   
    {  
        int y = x;  
    }  
}
```

 Область видимости идентификатора x

Вложенные блоки

```
#include <iostream>
```

Потенциальная область
видимости идентификатора

```
int main() {  
    int x = 1;  
    {  
        int x = 2;  
        std::cout << x; // Вывод: 2  
    }  
}
```

Область видимости идентификатора **x**

Область видимости **вложенного** идентификатора **x**

Вложенные блоки

```
int x;  
int y;  
{  
    int x;  
}
```

Идентификатор	Тип	Адрес
x	int	0x7ffe6625f9d4

Идентификатор	Тип	Адрес
y	int	0x7ffe6625f9d8
x	int	0x7ffe6625f9d4

Идентификатор	Тип	Адрес
x	int	0x7ffe6625f9dc
y	int	0x7ffe6625f9d8
x	int	0x7ffe6625f9d4

Идентификатор	Тип	Адрес
y	int	0x7ffe6625f9d8
x	int	0x7ffe6625f9d4

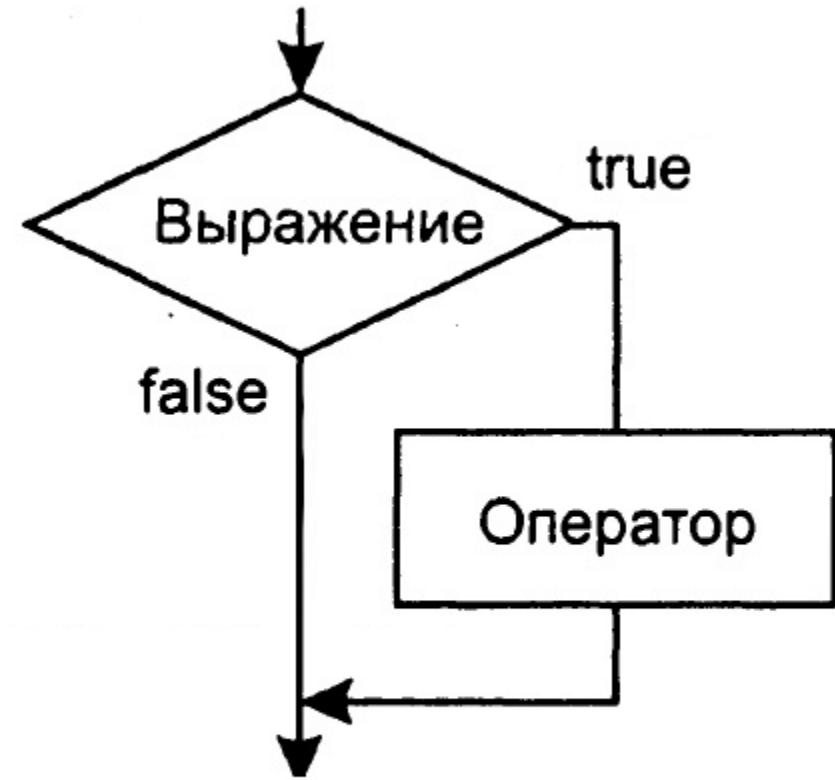
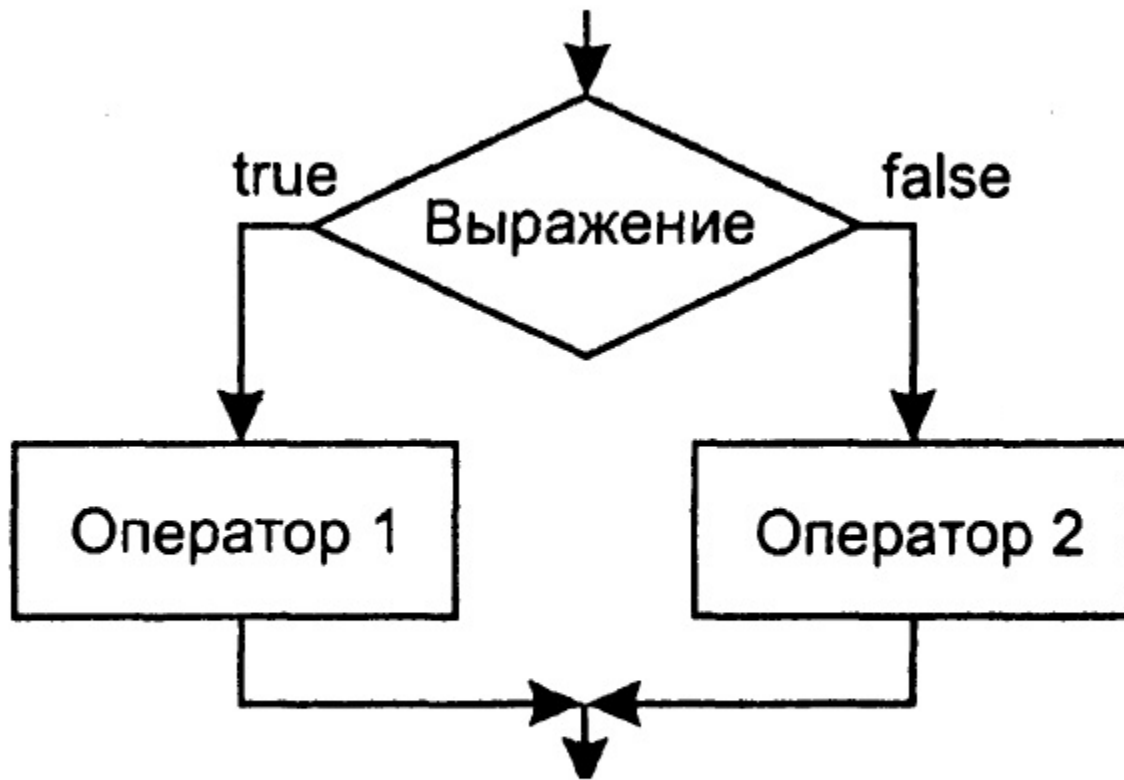
Selection Statements (Стейтменты выбора)

В стандарте указано 2 вида:

- if statement – условный оператор if;
- switch statement – оператор множественного выбора switch.

Внимание! На самом деле if и switch – не операторы, а стейтменты. Некорректный перевод на русский настолько закрепился в языке, что стал почти каноном.

Условный оператор



Структурная схема условного оператора

Тип bool

```
bool a = true;    // Истина  
bool b = false;   // Ложь
```

* Маленькими буквами

Преобразование типа bool

Преобразование **bool** в числовые типы:

- Значение **true** преобразуется в **1** для типов **int**, **float**, **double** и других числовых типов;
- Значение **false** преобразуется в **0** для типов **int**, **float**, **double** и других числовых типов;

Преобразование числовых типов в **bool**:

- Любое НЕнулевое значение числового типа преобразуется в **true**;
- Значение **0** числового типа преобразуется в **false**;

* Преобразование происходит автоматически

Преобразование типа bool

```
#include <iostream>

int main() {
    int num = true; // Преобразование bool в int, значение num будет равно 1
    float f = false; // Преобразование bool в float, значение f будет равно 0

    int num2 = 0;
    bool b2 = num2; // Преобразование int в bool, значение b2 будет равно false

    int num3 = -70; // Преобразование int в bool, значение b2 будет равно true
    bool b3 = num3; // Преобразование int в bool, значение b2 будет равно false

    std::cout << "int: " << num << std::endl;
    std::cout << "float: " << f << std::endl;
    // По умолчанию тип bool выводится на экран как 1 и 0
    // std::boolalpha меняет способ вывода на true и false
    std::cout << "bool: " << std::boolalpha << b2 << std::endl;
    std::cout << "bool: " << std::boolalpha << b3 << std::endl;
}
```

Откуда берётся true и false

Операторы сравнения

Оператор	Проверяет на	Пример использования	Результат
==	Равенство	3 == 7	false
!=	Неравенство	3 != 7	true
<	Меньше	3 < 7	true
>	Больше	3 > 7	false
<=	Меньше или равно	7 <= 7	true
>=	Больше или равно	7 >= 7	true

Начиная с C++ 20 был добавлен оператор трёхстороннего сравнения <=> ("spaceship" космический корабль). Этот оператор возвращает не тип bool, поэтому здесь не будем его рассматривать.

Логические операторы

Название	Как выглядит	Как использовать
И	&& and	a && b; a and b;
ИЛИ	or	a b; a or b;
НЕ	! not	!a; not a;

a	b	a and b	a or b	not a
false	false	false	false	true
true	false	false	true	false
false	true	false	true	true
true	true	true	true	false

Логические операторы применяются только к операндам типа `bool`, поэтому перед их применением будет попытка преобразовать операнды в `bool`. Если это не возможно, то получаем ошибку.

Операторы И и ИЛИ вычисляются по сокращённым правилам, т.к. если результат можно получить вычислив первый аргумент, второй не вычисляется:

```
false && std::cout << 1; // пусто
```

```
true && std::cout << 2; // 2
```

Операторы сравнения. Возможные ошибки

```
int a = 10;  
-1 < a < 2    // true
```

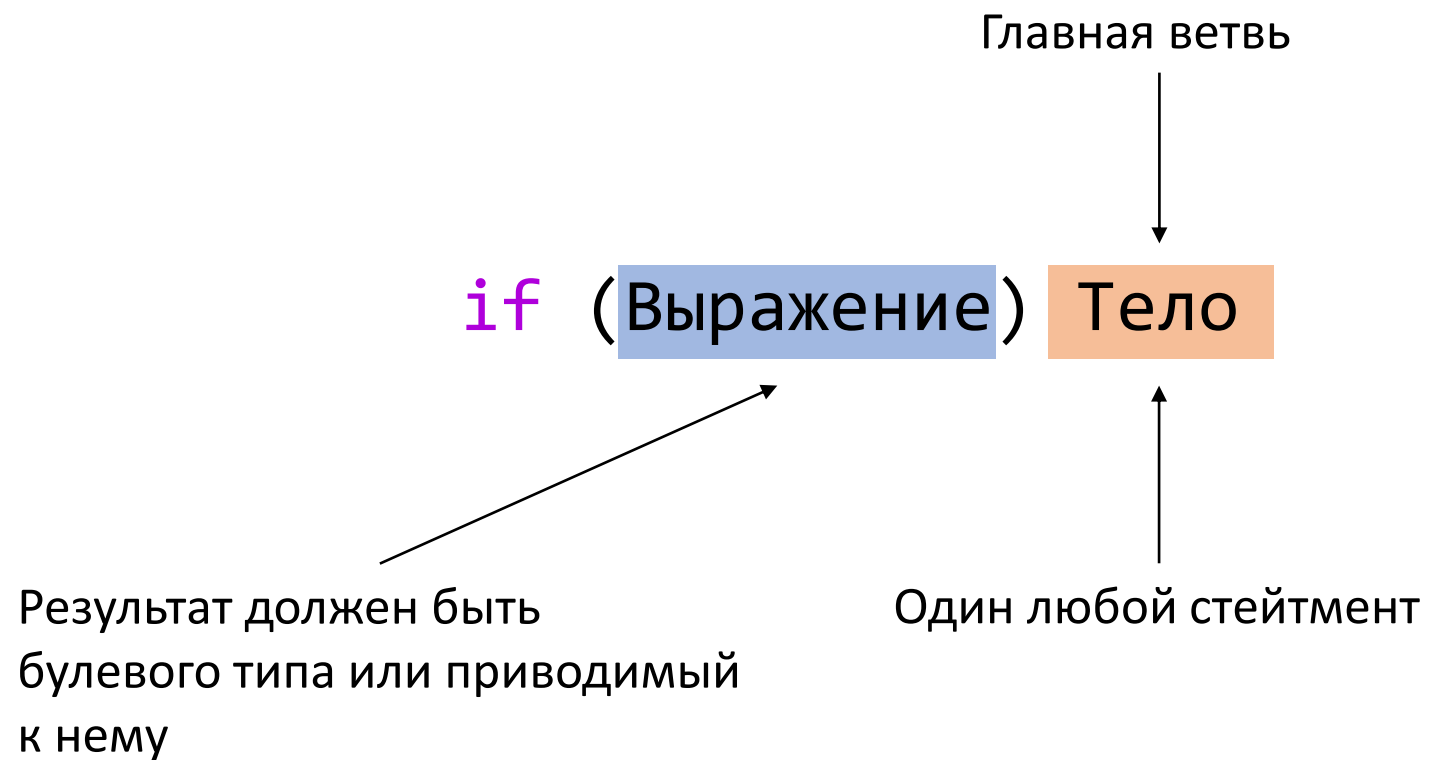
Последнее выражение вычисляется последовательно:

```
    -1 < a < 2  
(-1 < a) < 2  
    true < 2  
        true
```

Если хотите получить результат по математическим правилам пишите:

```
(-1 < a) and (a < 2)    // false
```

if



if

Ключевое слово **если**

Выражение

Тело

The diagram shows the components of a C++ if statement. An arrow points from the label 'Ключевое слово если' to the word 'if'. Another arrow points from 'Выражение' to the expression '(a > b)'. A third arrow points from 'Тело' to the statement 'std::cout << "Hello";'. The code is: `if (a > b) std::cout << "Hello";`

Ключевое слово **если**

Выражение

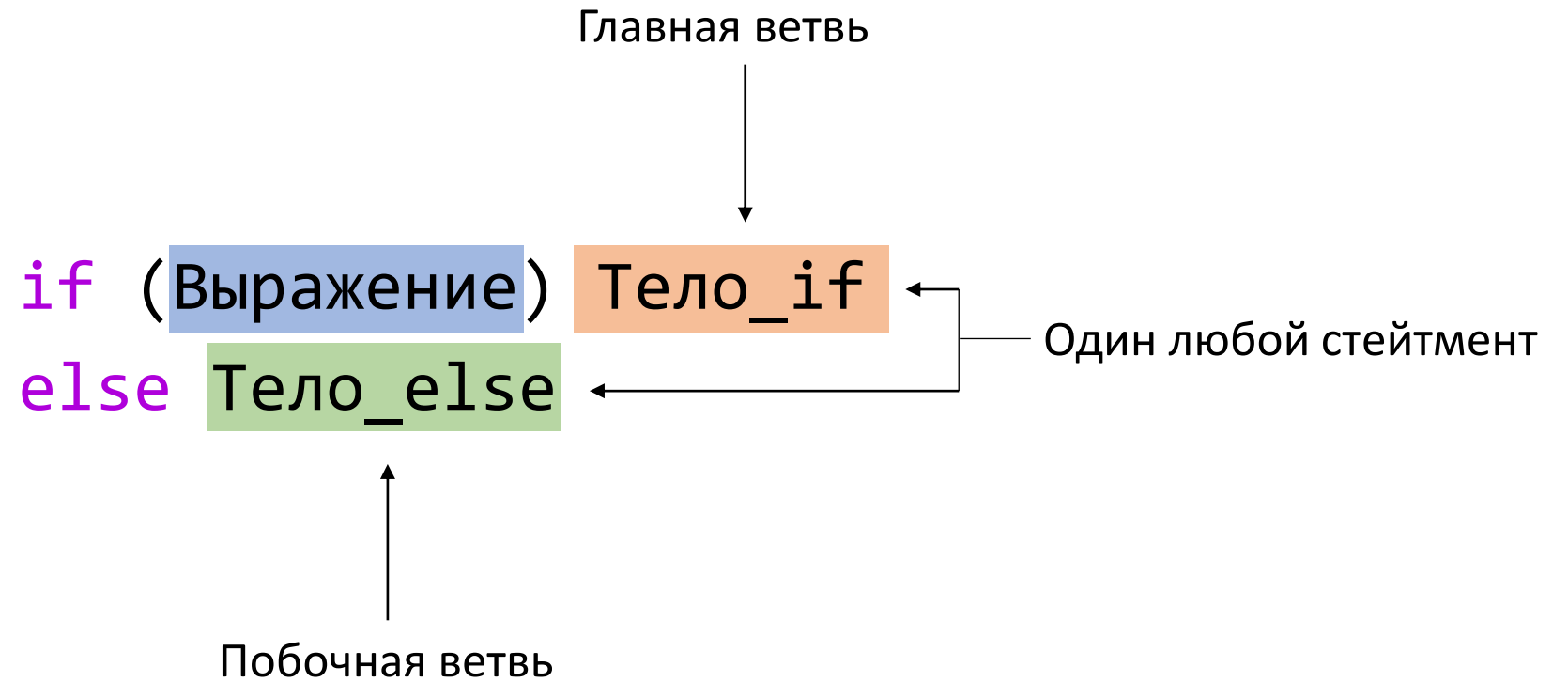
Тело

The diagram shows the components of a C++ if statement with a block body. An arrow points from the label 'Ключевое слово если' to the word 'if'. Another arrow points from 'Выражение' to the expression '(a > b)'. A third arrow points from 'Тело' to the block of code '{ std::cout << "Hello"; }'. The code is: `if (a > b){
 std::cout << "Hello";
}`

if(инициализация; проверка)

init-statement
↓
if (int res = a > b; res) std::cout << "Hello";
else std::cout << "Bye";

if-else



if-else

Выражение



Ключевое слово **если** → **if** (a > b) **std::cout** << "Hello"; ← Тело **if**

Ключевое слово **иначе** → **else** **std::cout** << "Bye"; ← Тело **else**

Выражение



Ключевое слово **если** → **if** (a > b){
 std::cout << "Hello"; ← Тело **if**

Ключевое слово **иначе** → } **else** {
 std::cout << "Bye"; ← Тело **else**
}

if и область видимости

```
int main() {  
    if (1) int a = 1;  
  
    cout << a;  
}
```

```
int main() {  
    {  
        if (1) int a = 1;  
    }  
    cout << a;  
}
```

```
int main() {  
    {  
        int a = 1;  
    }  
    cout << a;  
}
```

if и область видимости

```
int main() {  
    if (?) int a = 1;  
    else int a = 2;  
  
    cout << a;  
}
```

главная

побочная

```
int main() {  
    {  
        int a = 1;  
    }  
    cout << a;  
}
```

```
int main() {  
    {  
        int a = 2;  
    }  
    cout << a;  
}
```

if и область видимости

```
int main() {  
    if (int a = 1; a)  
        int a = 1;  
  
    cout << a;  
}
```

```
int main() {  
    {  
        if (int a = 1; a)  
            int a = 1;  
    }  
    cout << a;  
}
```

```
int main() {  
    {  
        int a = 1;  
        int a = 1; // He OK  
    }  
    cout << a;  
}
```

if и область видимости

```
int main() {  
    if (int a = 1; a){  
        int a = 1; // Не OK  
    }  
  
    cout << a;  
}
```

```
int main() {  
    if (int a = 1; a){  
        {  
            int a = 1 ; // OK  
        }  
    }  
  
    cout << a;  
}
```

if

```
if (/* условие */)
{
    /* true */
}
```

If-else

```
if (/* условие */)
{
    /* true */
}
else
{
    /* false */
}
```

If-else if

```
if (/* условие 1 */)
{
    /* true */
}
else if (/* условие 2 */)
{
    /* true */
}
else
{
    /* false */
}
```

Вложенный if

```
if (/* условие */)
{
    if (/* условие */)
    {
        /* true */
    }
    else
    {
        /* false */
    }
}
else
{
    if (/* условие */)
    {
        /* true */
    }
    else
    {
        /* false */
    }
}
```

If-else (ошибки)

```
if (a > b);
```

```
if (a > b){/* код */};  
else {/* код */}
```

```
if (a > b)  
    if (a > c) std::cout << "Hello";  
else std::cout << "Bye";
```

Тернарный оператор (?:)

```
variable = a > b ? a : b;
```

The diagram illustrates the execution flow of the ternary operator. It shows the syntax: `/* результат */ = /* условие */ ? /* выражение 1 */ : /* выражение 2 */;`. A horizontal line with a downward arrow labeled "true" points from the condition to the first expression. A horizontal line with an upward arrow labeled "false" points from the condition to the second expression.

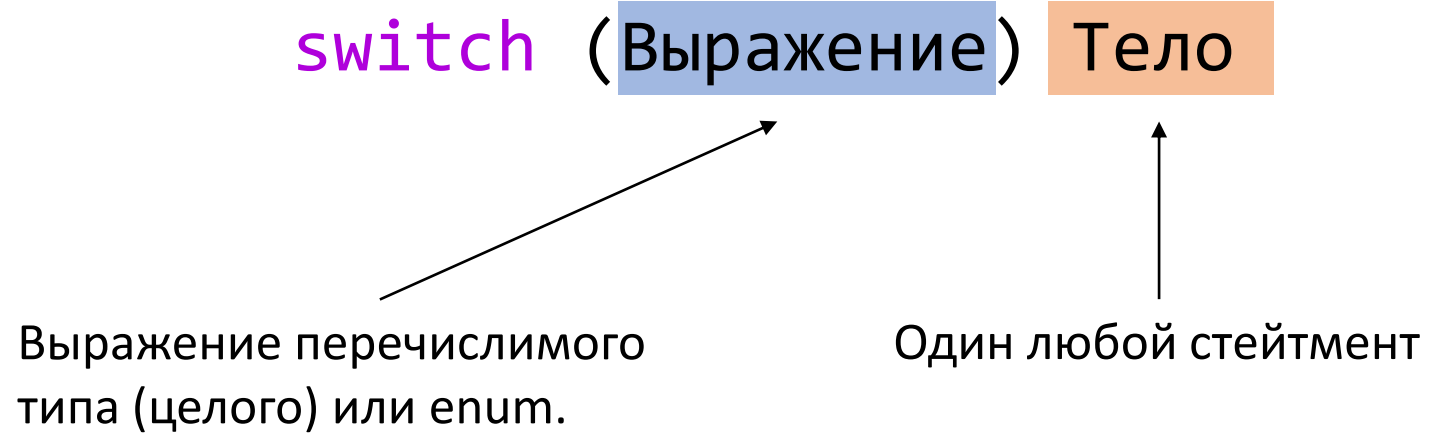
```
/* результат */ = /* условие */ ? /* выражение 1 */ : /* выражение 2 */;
```

* выражение 1 и выражение 2 должны быть одного или приводимого к одному типу

switch

switch (Выражение) Тело

Выражение перечислимого
типа (целого) или enum.



Один любой стейтмент

switch

```
int main() {  
    switch (1) ;  
}
```

```
int main() {  
    switch (1) {  
    }  
}
```

```
int main() {  
    switch (1) {  
        cout << "Hello ";  
        cout << "world";  
        cout << "!";  
    }  
}
```

Labeled statement (Метка)

Стандартом определены 3 типа меток:

- метка для `goto`. Будет рассмотрена далее;
- `case`.
- `default`.

Метки `case` и `default` могут быть использованы только внутри `switch`. При совпадении (`==`) выражения в скобках у `switch` и значения метки `case` поток исполнения перепрыгивает на эту метку пропуская весь код до неё и продолжает исполнение команд **до конца** `switch` игнорируя все остальные метки.

В случае отсутствия совпадения с `case`, используется метка `default`, которая действует аналогично `case`. Метка `default` может быть расположена в любом месте `switch`.

Если внутри `switch` создаются переменные, то не должно быть метки, которая перепрыгивает создание переменной, иначе компилятор сообщит об ошибке.

switch

Выражение
↓
`switch(value){`
 `case 1:` ←
 `std::cout << "one";`
 `case 2:` ←
 `std::cout << "two";`
 `default:` ←
 `std::cout << "zero";`
}

Метка

Jump Statement break (Стейтмент перехода break)

Т.к. после перехода на метку `case` или `default` исполнение кода продолжается до конца `switch`, иногда есть необходимость прервать исполнение кода и выйти из `switch` досрочно. Для этих целей существует стейтмент перехода `break`; Как только он встречается в коде, поток исполнения выходит за пределы текущего блока.


`break` разрешено использовать только в `switch` и в циклах.

В C++ есть только `break` без метки.

switch

Выражение
↓

```
switch(value){  
    case 1:  
        std::cout << "one";  
        break;  
    case 2:  
        std::cout << "two";  
        break;  
    default:  
        std::cout << "zero";  
}
```



switch(инициализация; выражение)

init-statement
↓

```
switch (int res = a > b; res ){  
    case true: std::cout << " :)"; break;  
    default: std::cout << " :(";  
}
```

switch и области видимости

Всё, что было сказано про области видимости в отношении if, справедливо и здесь.

Стоит упомянуть, что case и default - это просто метки, поэтому они не создают свои области видимости. Если нужно, используйте Блоки.

Iteration Statements (Циклы)

В стандарте указано 4 вида:

- `while`. Цикл с предусловием;
- `do-while`. Цикл с постусловием;
- `for`. Цикл со счётчиком;
- Range-based `for` (с C++11). `for` на основе диапазона.

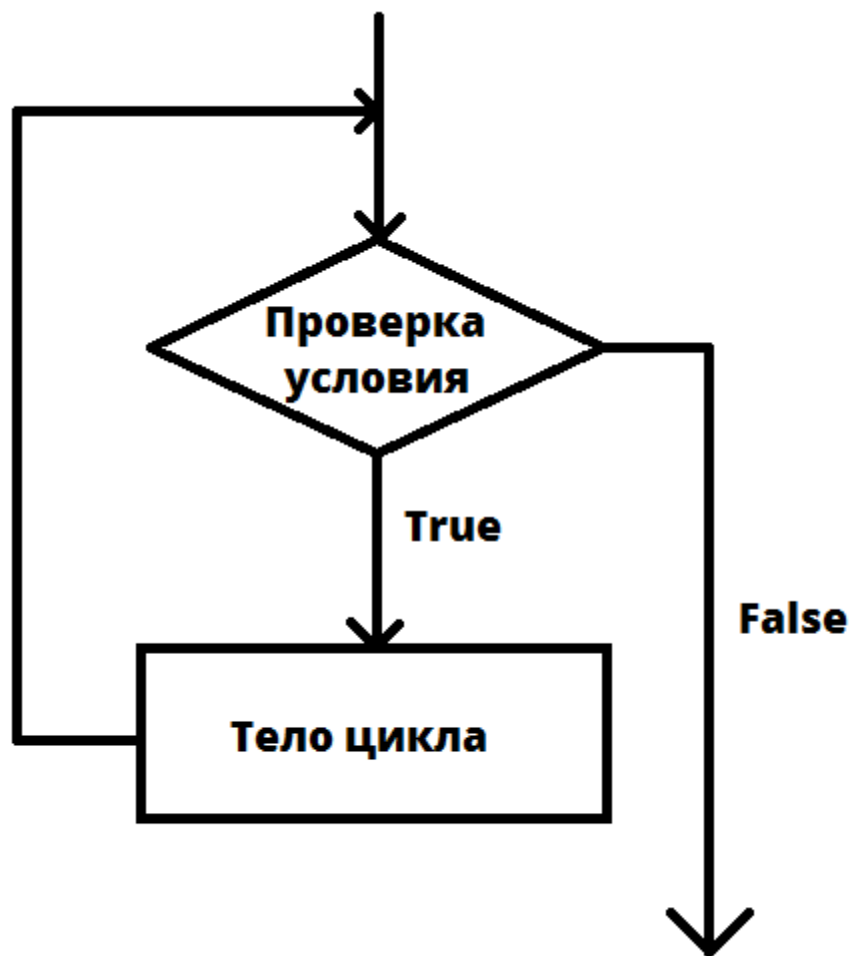
Внимание! Циклы тоже иногда называют операторами, но это стейтменты. Та же история, что и с `if`

Jump Statement goto (Стейтмент перехода goto)

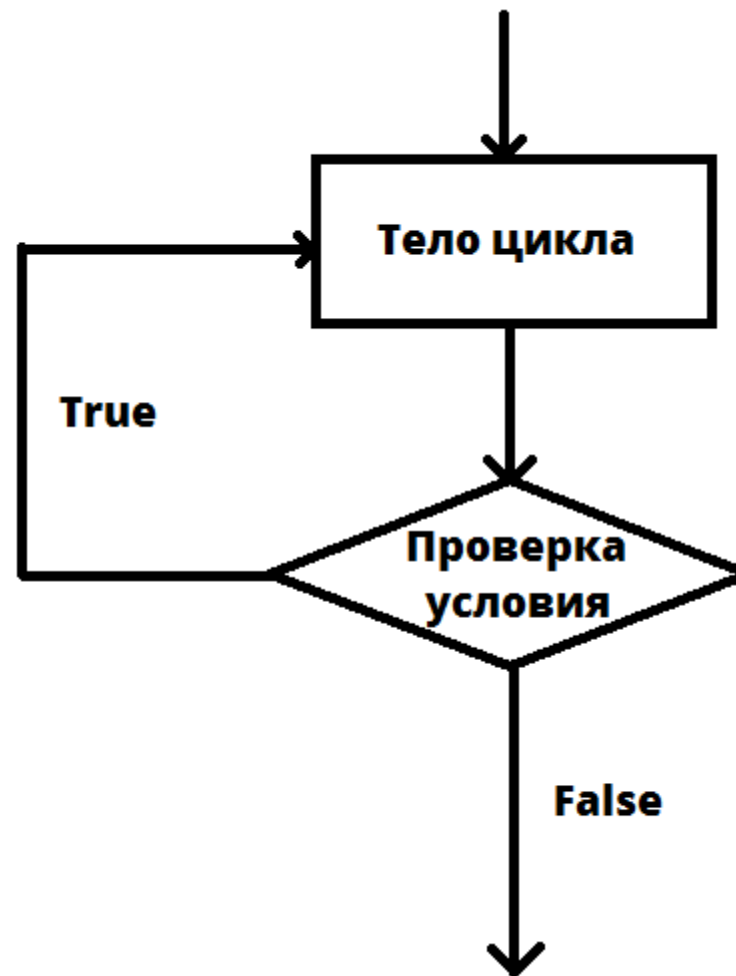
```
label:  
    /* код */  
goto label;
```

* label – обычный идентификатор

Оператор цикла



while



do-while

while

`while` (Выражение) Тело

Результат должен быть
булевого типа или приводимый
к нему

Один любой стейтмент

while

Ключевое слово

Выражение

Тело

```
while (a > b) std::cin >> a;
```

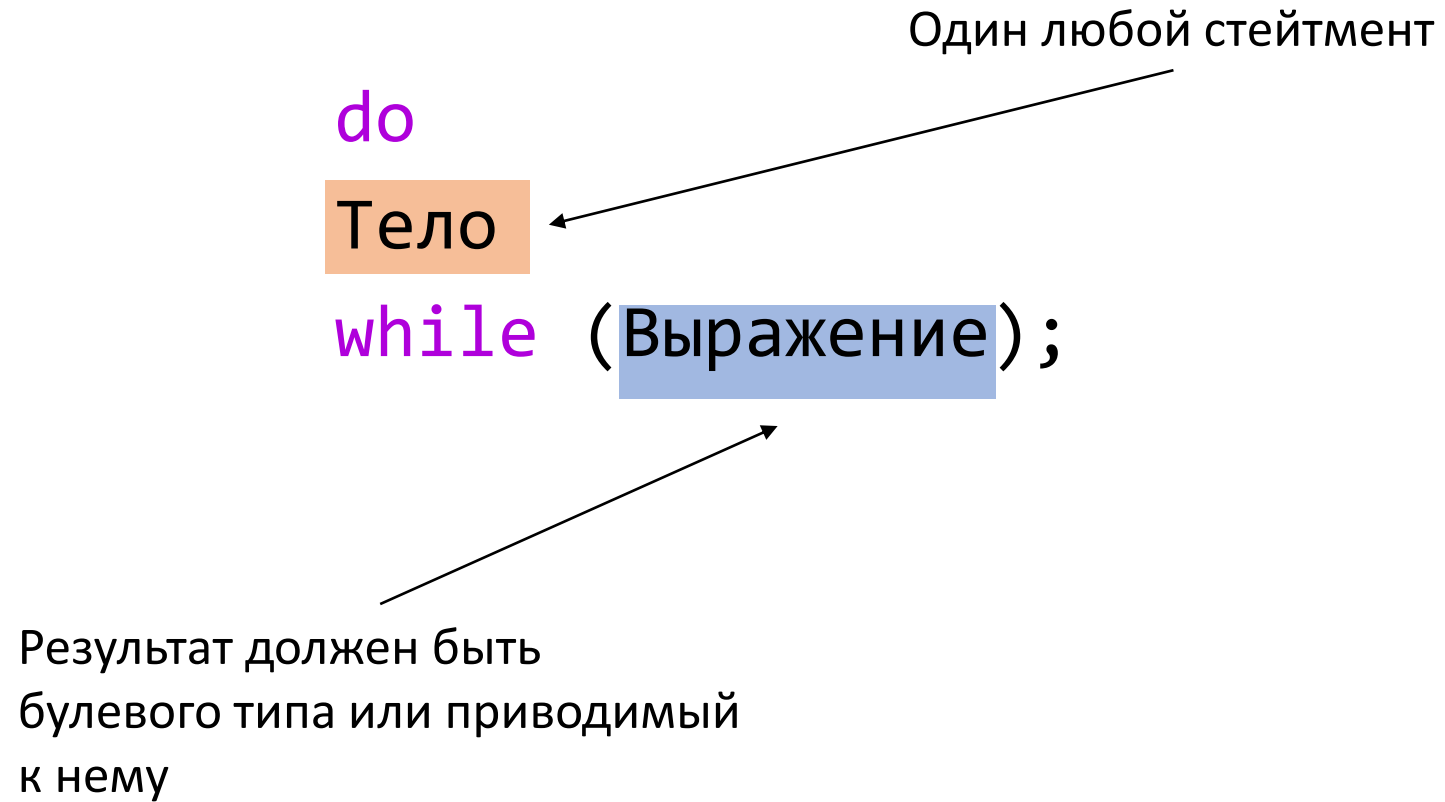
Ключевое слово

Выражение

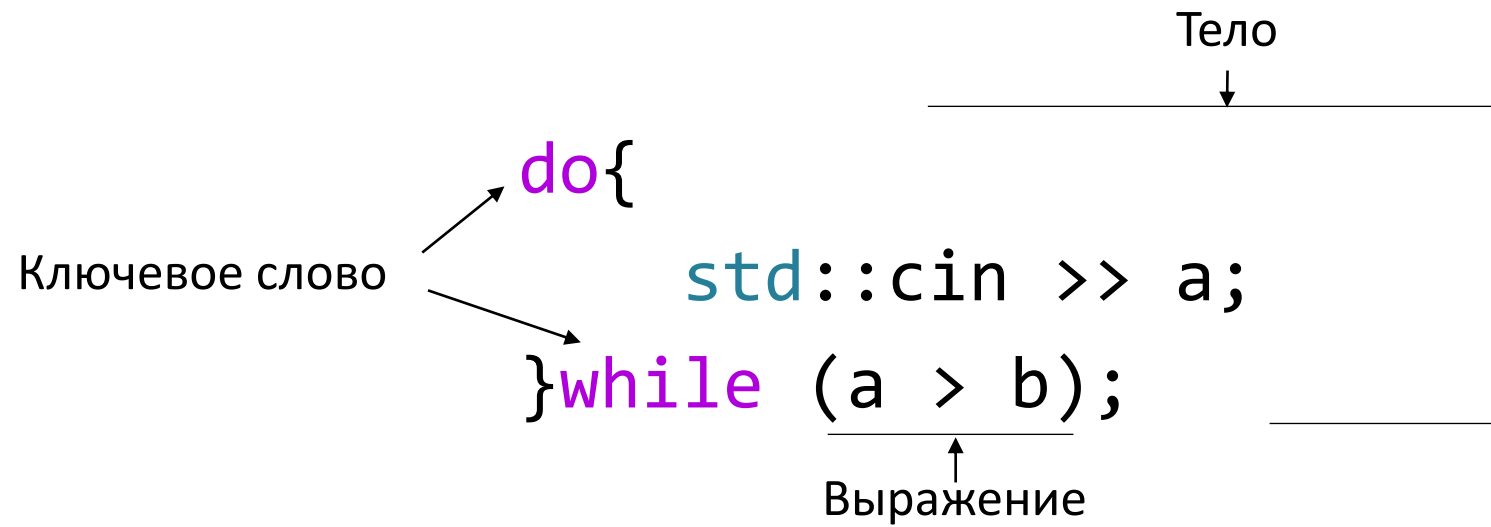
Тело

```
while (a > b){
    std::cin >> a;
}
```

do-while



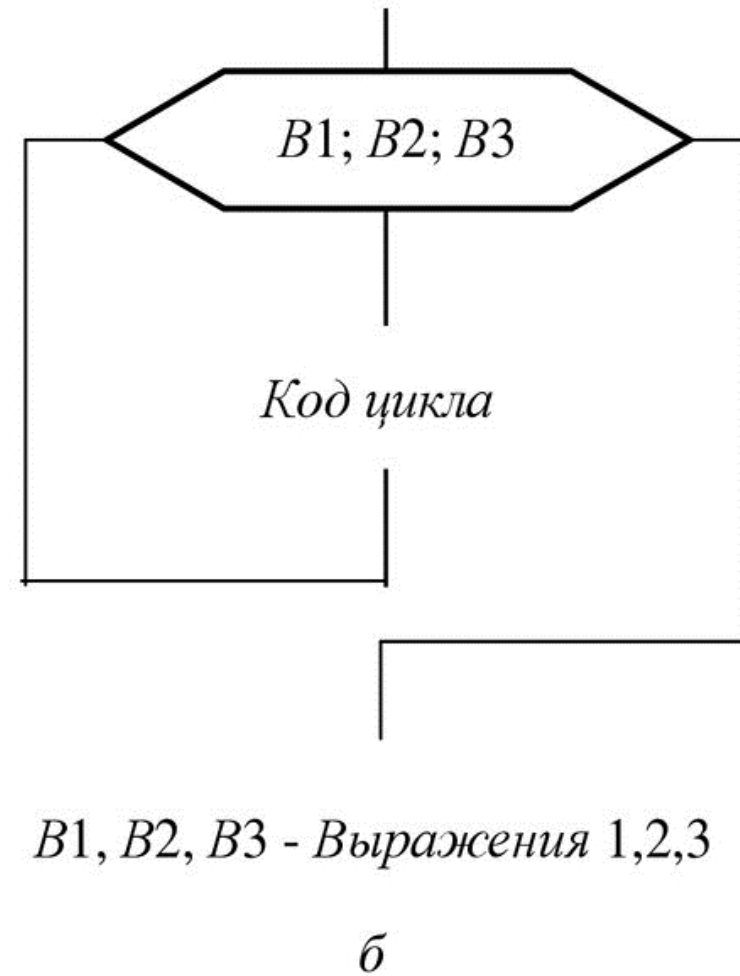
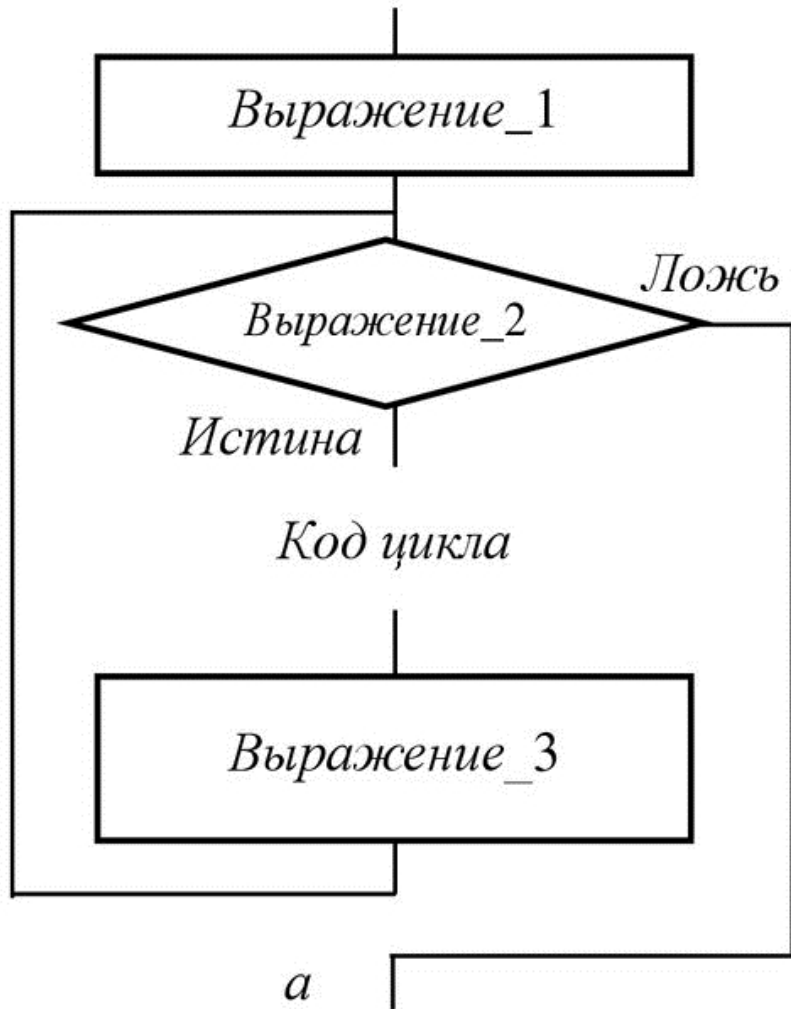
do-while



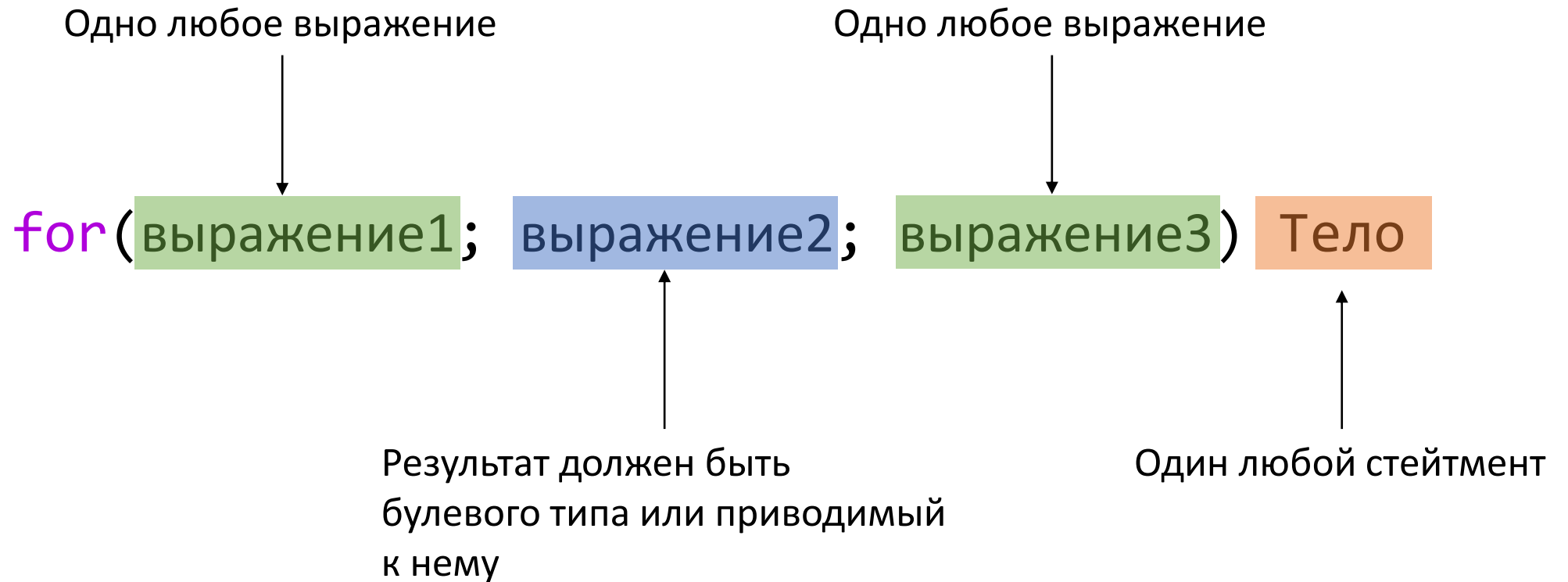
do-while и область видимости

```
int main() {  
    do  
        int a = 1;  
    while (a); // а не существует  
}
```


Оператор цикла for



for



for (выражение1; выражение2; выражение3)

Выражение1 – любое выражение или инициализация переменной. Обычно - инициализация переменной счётчика или нескольких;

Выражение2 – любое выражение или инициализация переменной. Обычно - выражение проверяющее условие работы цикла. Если выражение не указано, то считается, что оно равно true.

Выражение3 – выражение. Обычно инкремент/декремент счётчика(ов).

* каждое из выражение не обязательное (можно не писать), но точки с запятой писать нужно.

for

Ключевое слово Выражение1 Выражение2 Выражение3 Тело

↓ ↓ ↓ ↓

```
for(int i=0; i < count; i++) std::cout << i;
```

Ключевое слово Выражение1 Выражение2 Выражение3 Тело

↓ ↓ ↓ ↓

```
for(int i=0; i < count; i++){  
    std::cout << i;  
}
```

range-based for

Ключевое слово Переменная Контейнер Тело

↓ ↓ ↓

```
for(auto i : array) std::cout << i;
```

Ключевое слово Переменный Выражение2 Тело

↓ ↓ ↓ ↓

for(auto [key, value] : mymap) std::cout << i;

for (range-declaration : range-expression)

range-expression – любое выражение, представляющее последовательность элементов (либо массив, либо объект, для которого определены методы или функции `begin` и `end`) или список инициализации.

range-declaration – объявление именованной переменной, тип которой является типом элемента последовательности, представленного **range-expression**, или ссылкой на этот тип. Часто использует спецификатор `auto` для автоматического определения типа.

for(инициализация; range-declaration : range-expression)

```
for(auto list = {1,2,3}; auto i : list){  
    std::cout << i;  
}
```

for и области видимости

Также, как и `if` и `while`, так и `for` создаёт свою область видимости.

```
int main() {  
    for (int i=0; i<10; i++){  
        int i = -1; // Не OK, нарушаем ODR  
        cout << i;  
    }  
}
```

```
int main() {  
    for (int i=0; i<10; i++){  
        {  
            int i = -1; // OK  
            cout << i;  
        }  
    }  
}
```


Jump Statements (Стейтменты перехода)


В стандарте указано 5 видов:

- `break`;
- `continue`;
- `return`. Прерывание и выход из функции (будет рассмотрен позже);
- `co_return`. Приостановка и выход из корутины (будет рассмотрен позже);
- `goto`.

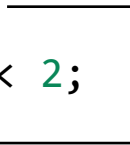
break

Разрешено использовать только в switch и циклах. Приводит к немедленному выходу за пределы switch или цикла соответственно.


```
for (int i=0; i<10; i++){  
    cout << 1;  
    break;  
    cout << 2;  
}
```



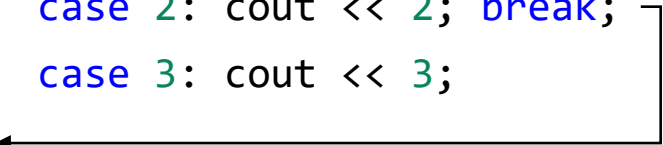
```
while(true){  
    cout << 1;  
    break;  
    cout << 2;  
}
```



```
do{  
    cout << 1;  
    break;  
    cout << 2;  
}while(true);
```



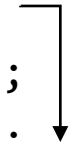
```
switch(i){  
    case 1: cout << 1;  
    case 2: cout << 2; break;  
    case 3: cout << 3;  
}
```



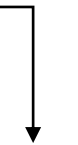
continue

Разрешено использовать только в циклах. Приводит к пропуску кода, расположенного после `continue` и до конца тела цикла. Цикл не прерывается.

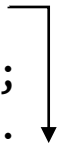
```
for (int i=0; i<10; i++){  
    cout << 1;  
    continue;  
    cout << 2;  
    cout << 3;  
}
```



```
while(true){  
    cout << 1;  
    continue;  
    cout << 2;  
    cout << 3;  
}
```



```
do{  
    cout << 1;  
    continue;  
    cout << 2;  
    cout << 3;  
}while(true);
```



Exception Handling Statements (Стейтменты обработки исключений)

```
try {  
    // Код который может бросить исключение  
} catch (const ExceptionType& e) {  
    // Обработка исключения  
}
```

```
// Бросаем исключение  
throw statement;  
throw exception_object;
```

