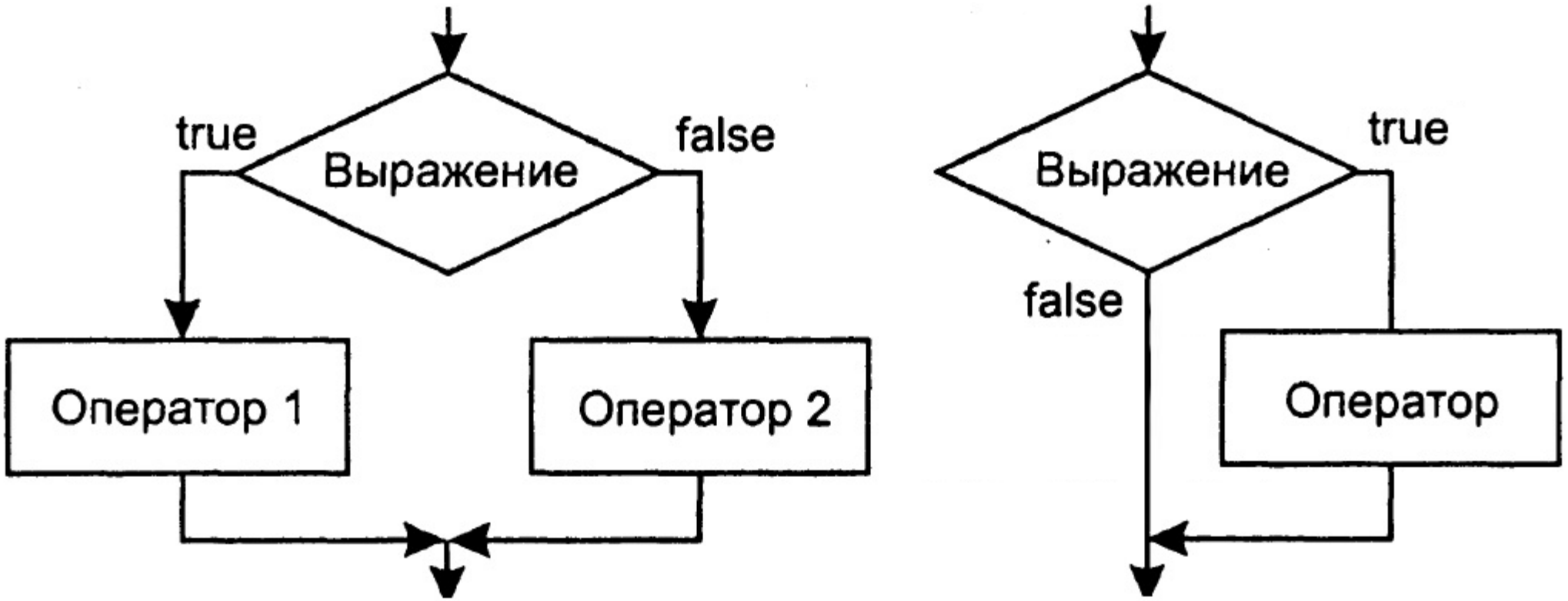
Алгоритмизация и программирование

Лекция 2 (C++)

Условный оператор



Структурная схема условного оператора

Тип bool

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

* Маленькими буквами

Преобразование типа bool

Преобразование bool в другие 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 других типов в bool:

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

```
* Преобразование происходит автоматически
```

Преобразование типа bool

```cpp
#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

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
```

Логические операторы. Возможные ошибки

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

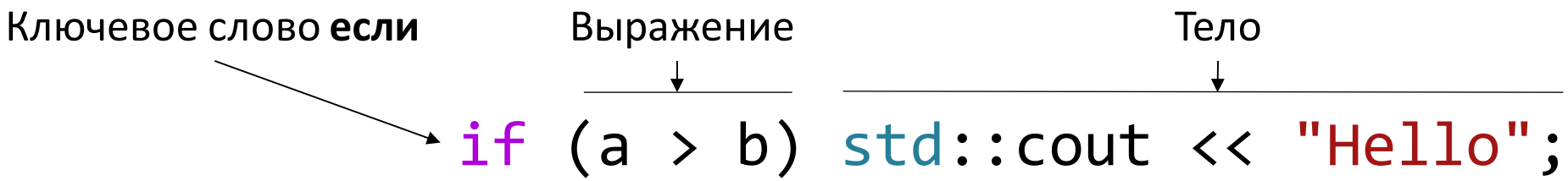
Последнее выражение вычисляется последовательно:

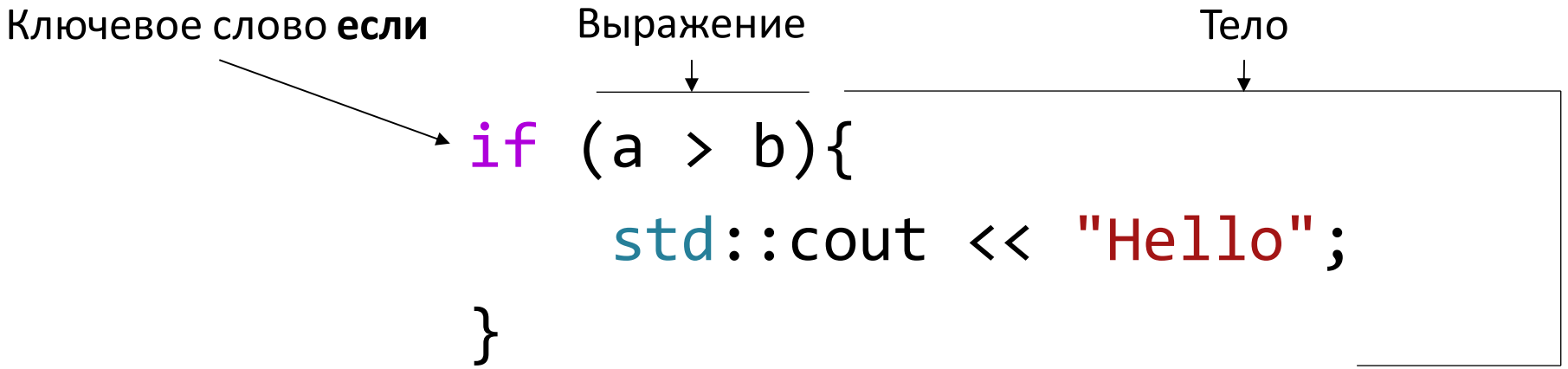
```
  -1 < a < 2
(-1 < a) < 2
    true < 2
        true
```

Если хотите получить результат по математическим правилам пишите:

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

if


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

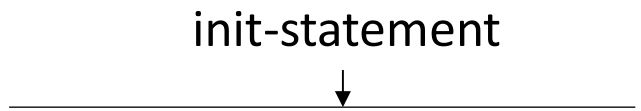

Ключевое слово **если**
Выражение
Тело
if (a > b){
std::cout << "Hello";
}

if (выражение)

Ожидается, что выражение в скобочках типа `bool`, поэтому будет попытка неявно преобразовать его к `bool`.

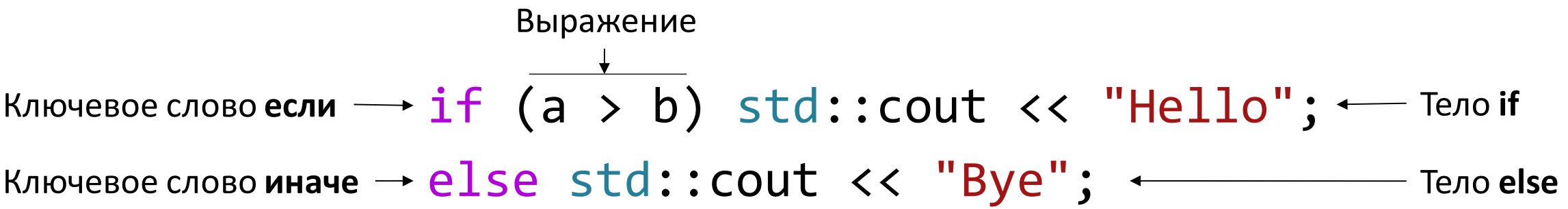
Если преобразование не допустимо, то – ошибка.

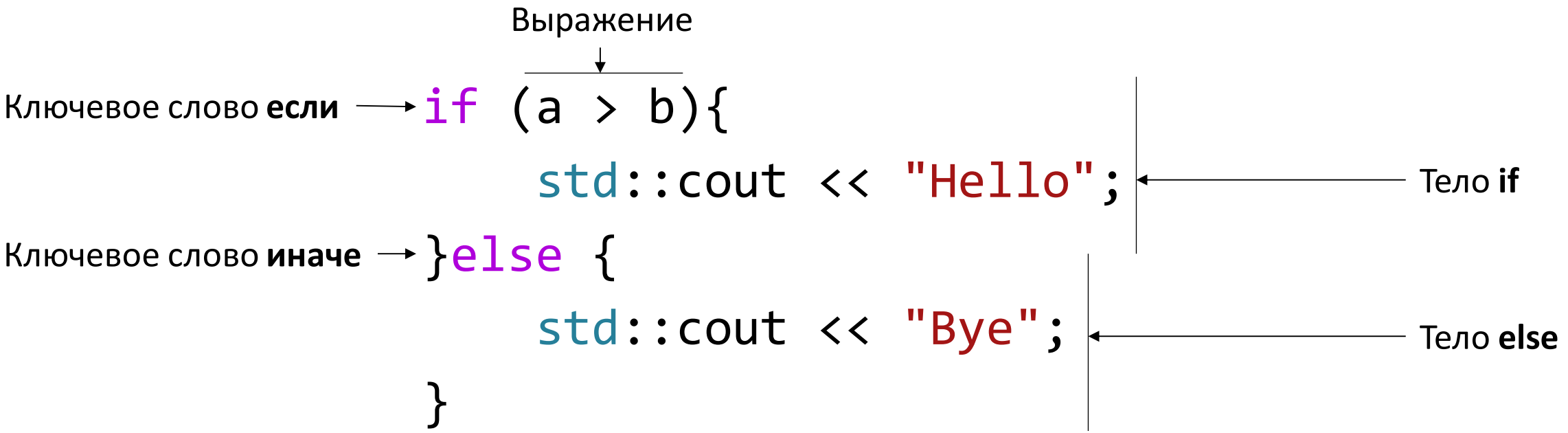
if(инициализация; проверка)




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

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

```
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

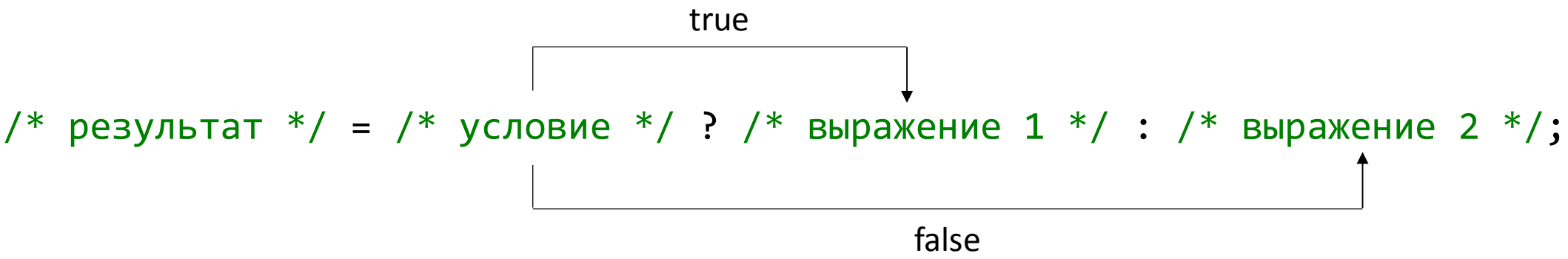
```
if (/* условие */)
{
    if (/* условие */)
    {
        /* true */
    }
    else
    {
        /* false */
    }
}
else
{
    if (/* условие */)
    {
        /* true */
    }
    else
    {
        /* false */
    }
}
```

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";
```

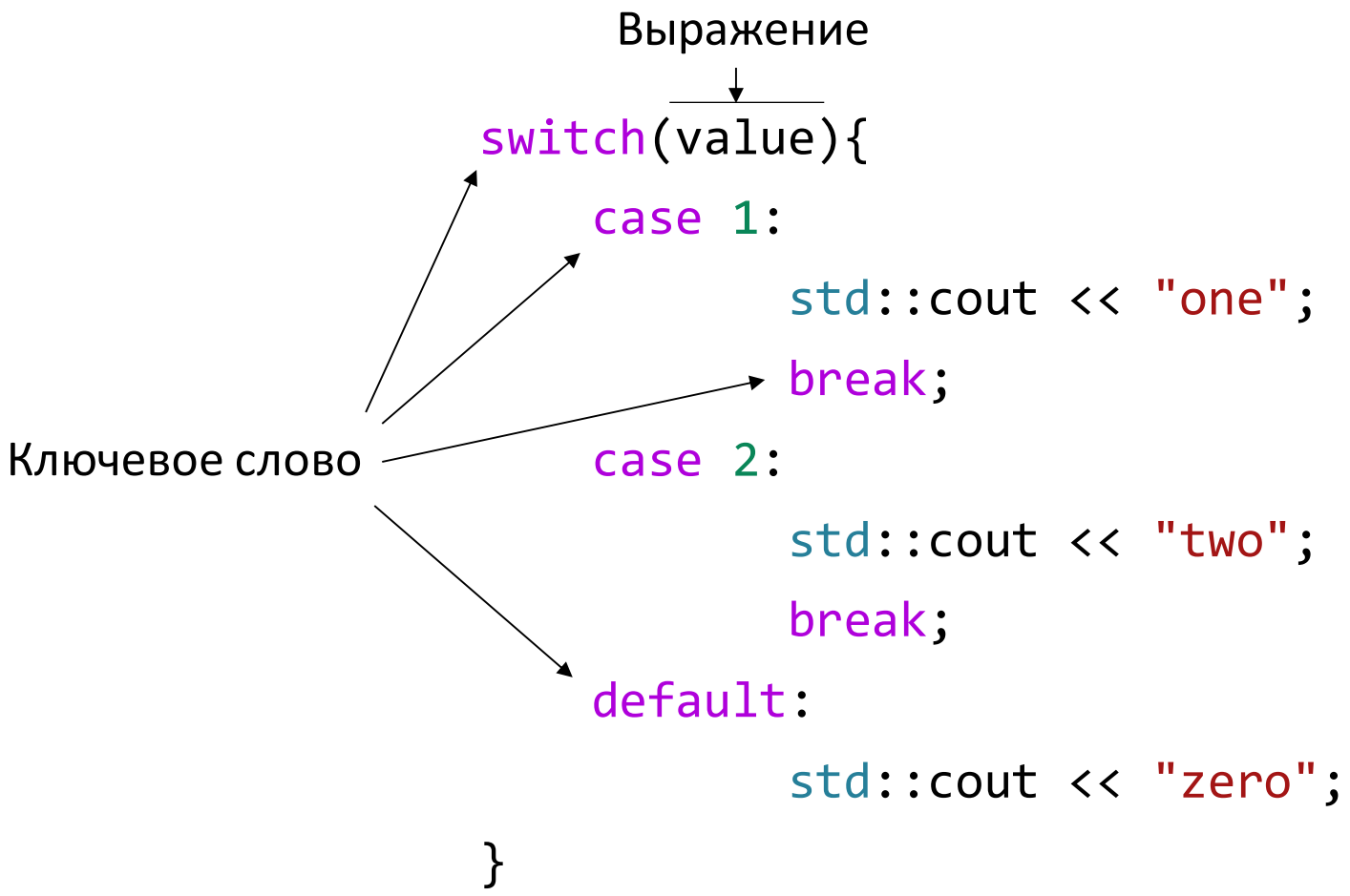
Тернарный оператор (?:)

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



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

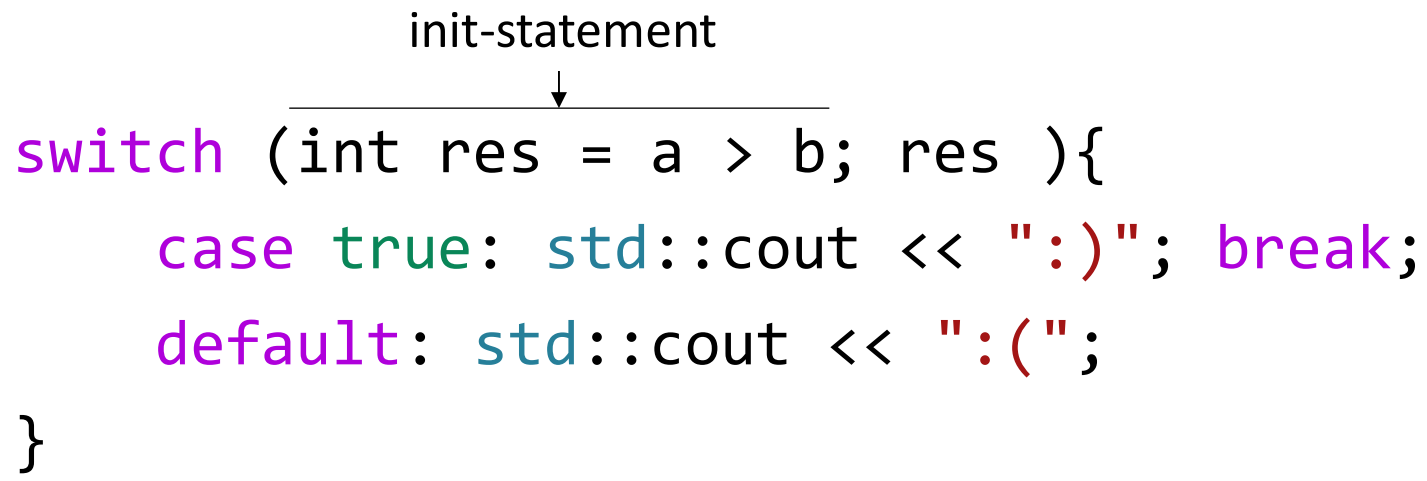
switch


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

switch (выражение)

Выражение перечислимого типа (целого), enum или что-то, что можно преобразовать в эти типы.

switch(инициализация; выражение)



goto

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

* label – обычный идентификатор

Оператор трехстороннего сравнения

Оператор <=>, также известный как оператор "spaceship" (космический корабль), был представлен в стандарте **C++20**. Этот оператор используется для сравнения двух значений и возвращает отрицательное число, ноль или положительное число.

Результатом оператора <=> будет объект типа std::strong_ordering, который может принимать одно из 4 значений:

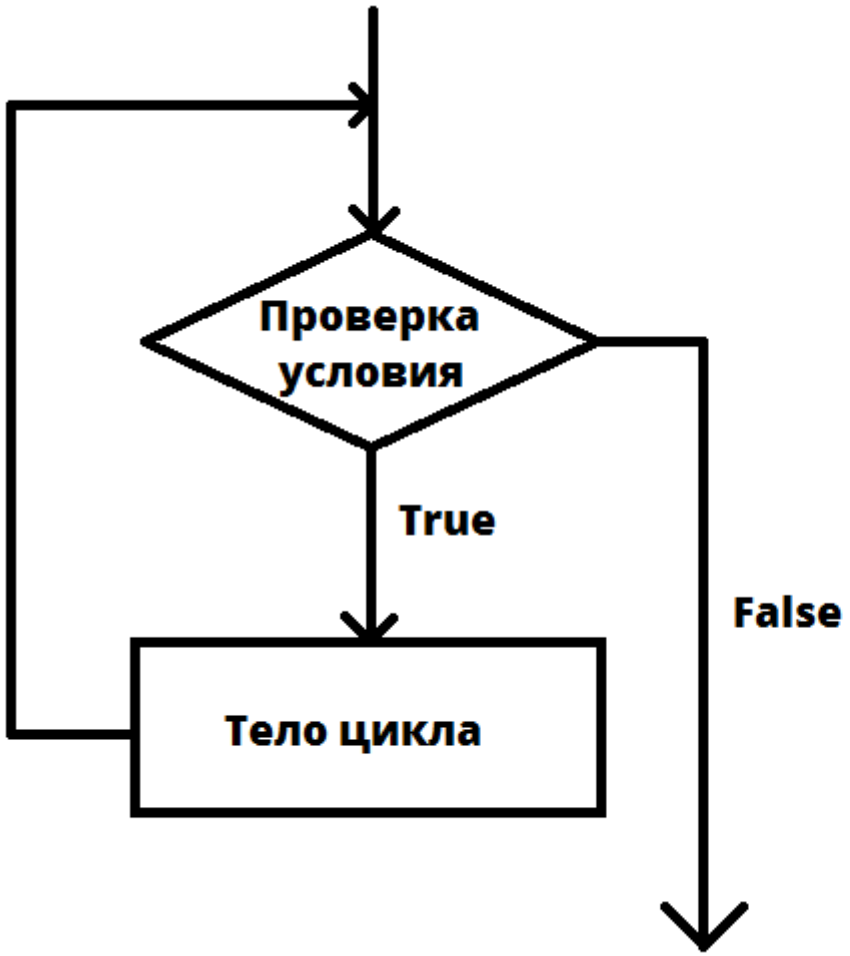
std::strong_ordering::less
std::strong_ordering::equivalent
std::strong_ordering::equal
std::strong_ordering::greater.

Оператор позволяет уменьшить количество сравнений.

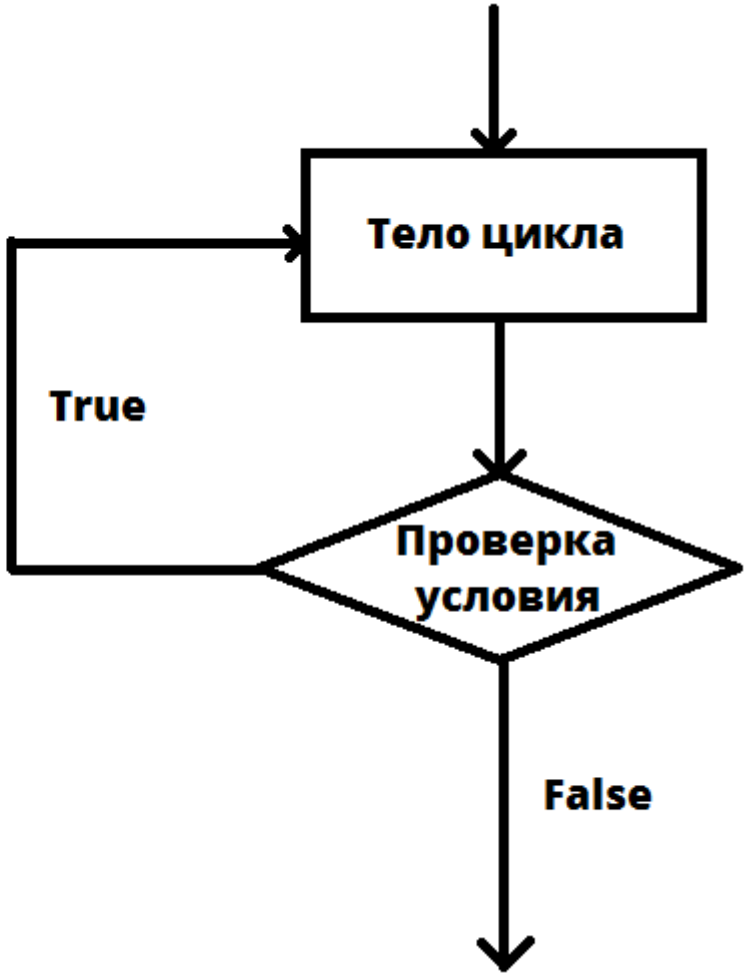
Оператор трехстороннего сравнения

```cpp
#include <iostream>
#include <compare> // для std::strong_ordering

int main() {
    std::string str1 = "hello";
    std::string str2 = "world";

    auto result = str1 <=> str2;  // std::strong_ordering

    if (result == std::strong_ordering::less) {
        std::cout << "Первая строка меньше второй" << std::endl;
    } else if (result == std::strong_ordering::equal) {
        std::cout << "Строки равны" << std::endl;
    } else if (result == std::strong_ordering::greater) {
        std::cout << "Первая строка больше второй" << std::endl;
    }
}
```

Оператор цикла

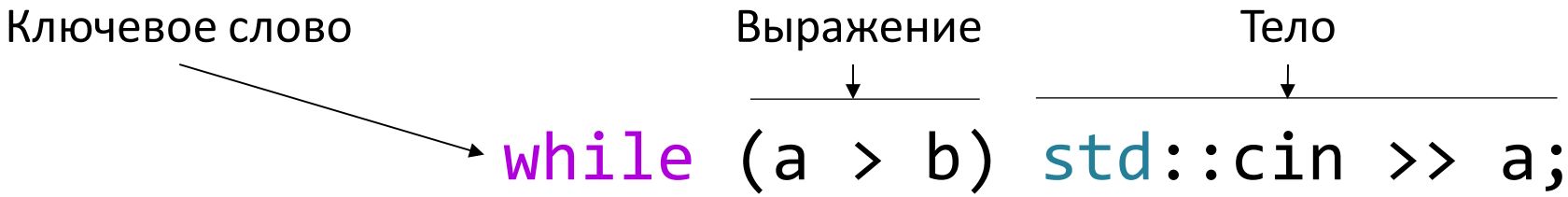


while



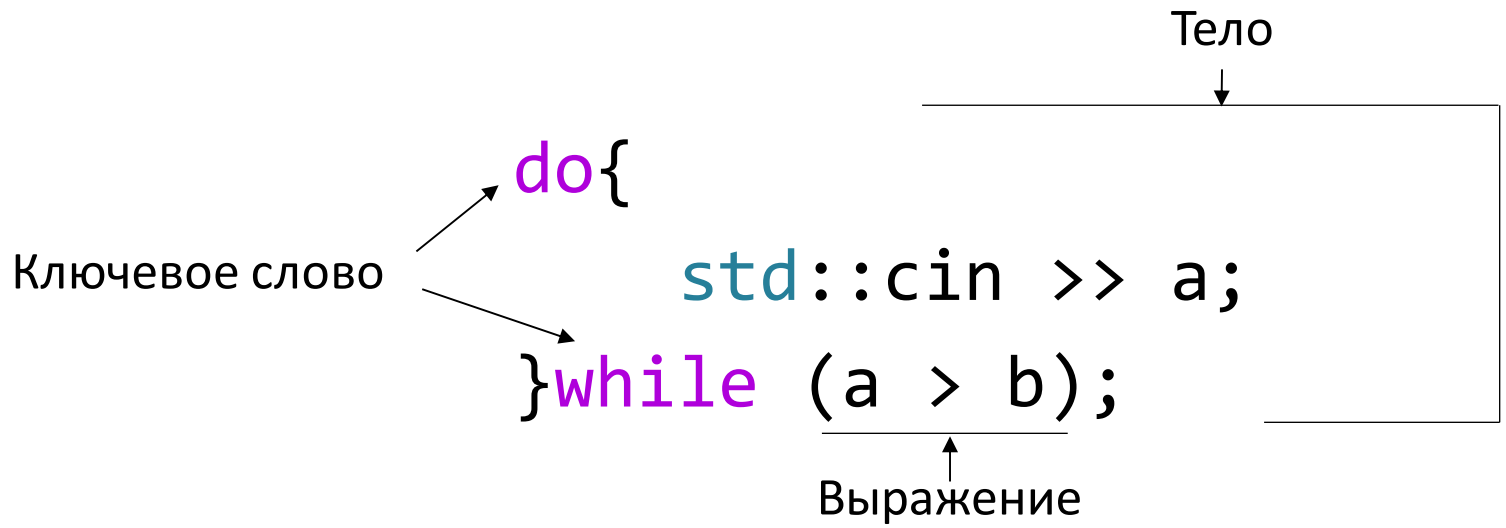
do-while

while


Ключевое слово
Выражение
Тело
while (a > b) std::cin >> a;


Ключевое слово
Выражение
Тело
while (a > b){
std::cin >> a;
}

do-while

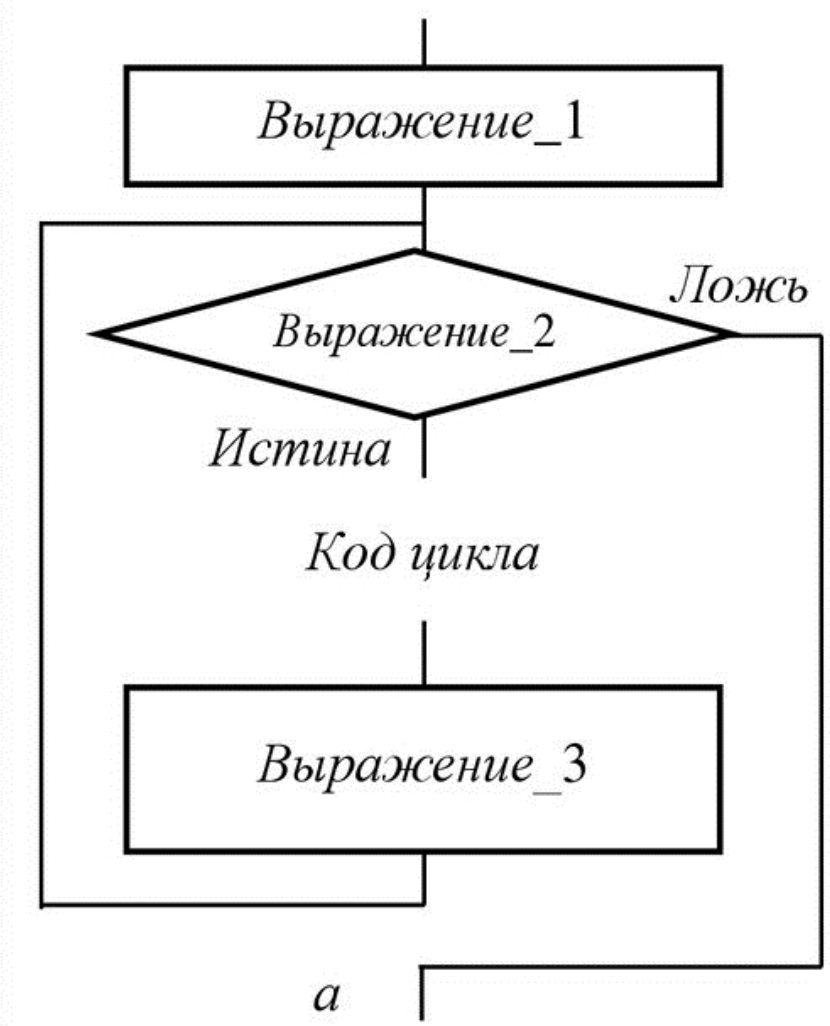


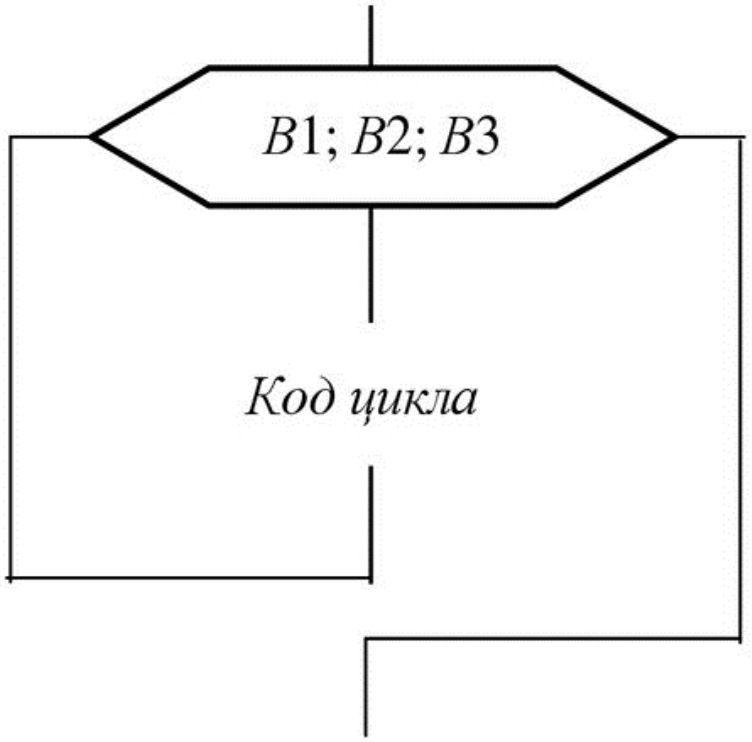
while (выражение)

Ожидается, что выражение в скобочках типа `bool`, поэтому будет попытка неявно преобразовать его к `bool`.

Если преобразование не допустимо, то – ошибка.

Оператор цикла for

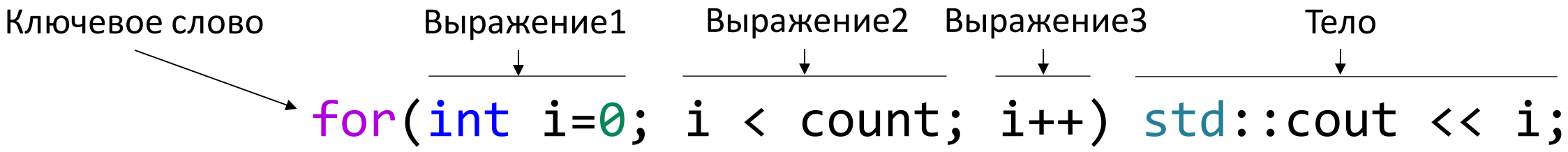


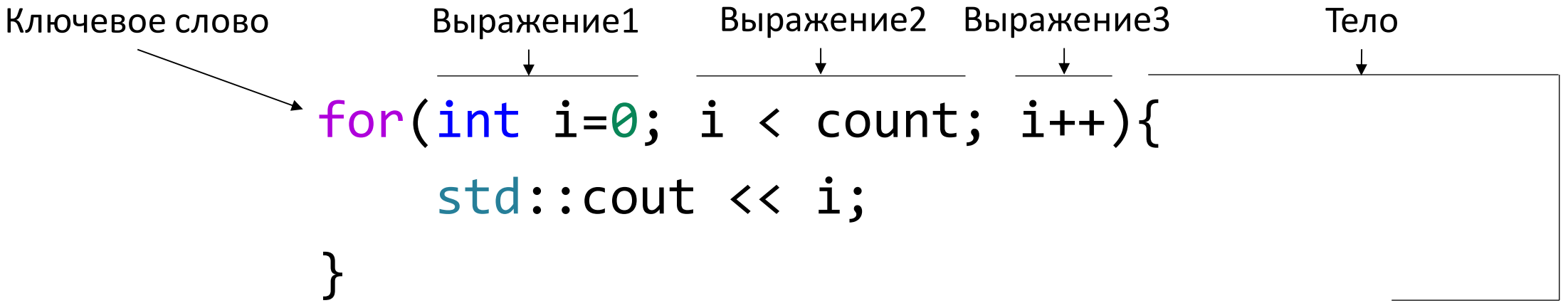


B1, B2, B3 - Выражения 1,2,3

б

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;
}

for (выражение1; выражение2; выражение3)

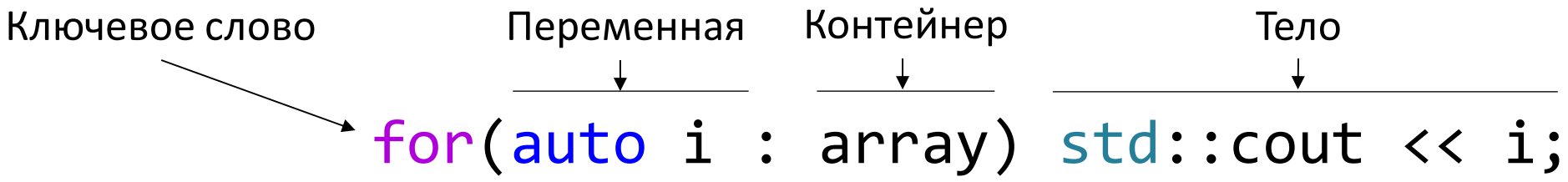
Выражение1 – любое выражение или инициализация переменной. Обычно - инициализация переменной счётчика или нескольких;

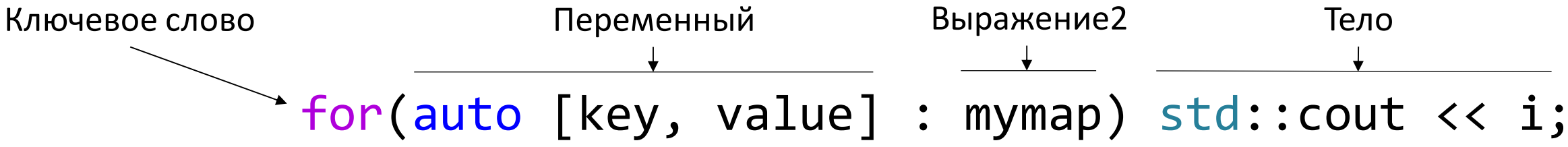
Выражение2 – любое выражение или инициализация переменной. Обычно - выражение проверяющее условие работы цикла. Если выражение не указано, то считается, что оно равно `true`.

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

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

range-based for





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;
}
```

std::string

#include <string>


```
string s = "I'm sorry, Dave.";
```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 indices

non-mutating

s.size()	→ 16	(number of characters)
s[2]	→ 'm'	(character at index 2)
s.find("r")	→ 6	(first match from start)
s.rfind("r")	→ 7	(first match from end)
s.find("X")	→ string::npos	(not found, invalid index)
s.find(' ', 5)	→ 10	(first match after index ≥ 5)
s.substr(4, 6)	→ string{"sorry,"}	
s.contains("sorry")	→ *true*	(C++23)
s.starts_with('I')	→ *true*	(C++20)
s.ends_with("Dave.")	→ *true*	(C++20)
s.compare("I'm sorry, Dave.")	→ 0	(identical)
s.compare("I'm sorry, Anna.")	→ > 0	(same length, but 'D' > 'A')
s.compare("I'm sorry, Saul.")	→ < 0	(same length, but 'D' < 'S')

mutating

	s += " I'm afraid I can't do that."	⇒	s = "I'm sorry, Dave. I'm afraid I can't do that."
	s.append("..")	⇒	s = "I'm sorry, Dave..."
size	s.clear()	⇒	s = ""
size	s.resize(3)	⇒	s = "I'm"
size	s.resize(20, '?')	⇒	s = "I'm sorry, Dave.????";
index based	s.insert(4, "very ")	⇒	s = "I'm very sorry, Dave."
index based	s.erase(5, 2)	⇒	s = "I'm sry, Dave."
index based	s[15] = '!'	⇒	s = "I'm sorry, Dave!"
index based	s.replace(11, 5, "Frank")	⇒	s = "I'm sorry, Frank"
iterator based	s.insert(s.begin(), "HAL: ")	⇒	s = "HAL: I'm sorry, Dave."
iterator based	s.insert(s.begin()+4, "very ")	⇒	s = "I'm very sorry, Dave."
iterator based	s.erase(s.begin()+5)	⇒	s = "I'm sry, Dave."
iterator based	s.erase(s.begin(), s.begin()+4)	⇒	s = "sorry, Dave."

Constructors

```
string{'a','b','c'}                  → a b c
string(4, '$')                       → $ $ $ $
string(@firstIn, @lastIn)            → e f g h
string( a b c d )  copy/move         → a b c d
```

source iterator range: b c d e f g h i j

source string object

Obtain Iterators or Reverse Iterators

```
.begin()  → @first                   a b c d e f
.rbegin() → reverse@last .base()     a b c d e f
.end()    → @one_behind_last         a b c d e f
.rend()   → reverse@one_before_first .base()  a b c d e f
```

don't use to access elements!

String → Number Conversion

```
int         stoi (●,●,●);
long        stol (●,●,●);
long long   stoll(●,●,●);
unsigned long       stoul (●,●,●);
unsigned long long  stoull(●,●,●);
float       stof (●,●,●);
double      stod (●,●,●);
long double stold(●,●,●);
```

- const string& — input string
- std::size_t* p = nullptr — output for number of processed characters
- int base = 10 — base of target system; default: decimal

Number → String Conversion

```
string to_string( ● );
```

int | long | long long | unsigned | unsigned long | unsigned long long | float | double | long double

Пример работы с std::string

```cpp
#include <iostream>
#include <string>

int main() {
    std::string word = "Hello";

    // Получение символа по индексу
    char letter = word[1];
    std::cout << "Символ по индексу 1: " << letter << std::endl;

    // Изменение символа по индексу
    word[0] = 'J';
    std::cout << "Измененное слово: " << word << std::endl;

    // Получение длины строки
    int length = word.length();
    std::cout << "Длина строки: " << length << std::endl;

    // Использование цикла для обращения к каждому символу по индексу
    for (int i = 0; i < length; i++) {
        std::cout << "Символ по индексу " << i << ": " << word[i] << std::endl;
    }
}
```

Считывание всей строки до конца

```
#include <iostream>
#include <string>

int main() {
    std::string input;
    std::cout << "Введите строку: ";

    // Считывание строки со стандартного ввода
    std::getline(std::cin, input);

    std::cout << "Вы ввели: " << input << std::endl;
}
```

Массивы

Статические :

```
int arr[10];
```

Динамические:

```
int* arr = new int[10];
delete[] arr;
```

STL:

```
std::array<int, 10> arr;
std::vector<int> arr(10);
```

C++ Standard Library **Sequence Containers**

array<*T*, *size*>

fixed-size array

#include <array>

```
std::array<int,6> a {1,2,3,4,5,6};
cout << a.size();    // 6
cout << a[2];        // 3
a[0] = 7;            // 1st element ⇒ 7
```

contiguous memory; random access; fast linear traversal

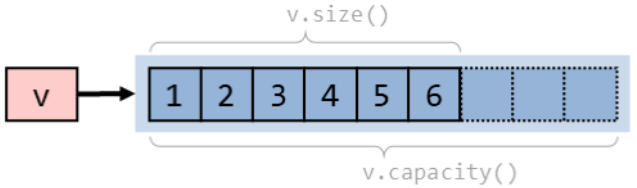
vector<*T*>

dynamic array

C++'s "default" container

#include <vector>

```
std::vector<int> v {1,2,3,4,5,6};
v.reserve(9);
cout << v.capacity();      // 9
cout << v.size();          // 6
v.push_back(7);            // appends '7'
v.insert(v.begin(), 0);    // prepends '0'
v.pop_back();              // removes last
v.erase(v.begin()+2);      // removes 3rd
v.resize(20, 0);           // size ⇒ 20
```



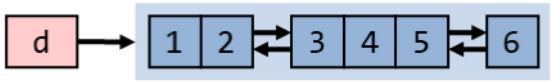
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

deque<*T*>

double-ended queue

#include <deque>

```
std::deque<int> d {1,2,3,4,5,6};
// same operations as vector
// plus fast growth/deletion at front
d.push_front(-1);   // prepends '-1'
d.pop_front();      // removes 1st
```



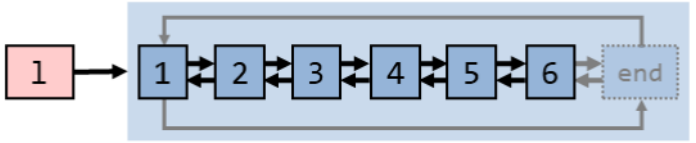
fast insertion/deletion at both ends

list<*T*>

doubly-linked list

#include <list>

```
std::list<int> l {1,5,6};
std::list<int> k {2,3,4};
// O(1) splice of k into l:
l.splice(l.begin()+1, std::move(k))
// some special member function algorithms:
l.reverse();
l.sort();
```



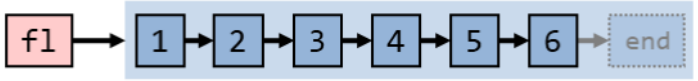
fast splicing; many operations without copy/move of elements

forward_list<*T*>

singly-linked list

#include <forward_list>

```
std::forward_list<int> fl {2,2,4,5,6};
fl.erase_after(begin(fl));
fl.insert_after(begin(fl), 3);
fl.insert_after(before_begin(fl), 1);
```



lower memory overhead than std::list; only forward traversal

std::vector<*ValueType*>

C++'s "default" dynamic array

`#include <vector>`

h/cpp hackingcpp.com

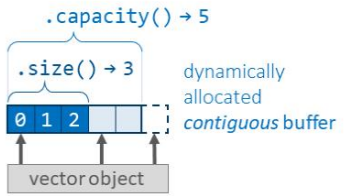
Construct A New Vector Object

```
vector<int> v1 {2,9,1,8,5,4}
vector<int> v2 (begin(v1)+3, end(v1))
vector<int> v3 (5,3)
vector<int> deep_copy_of_v1 (v1)
```

C++17 value type deducable from argument type

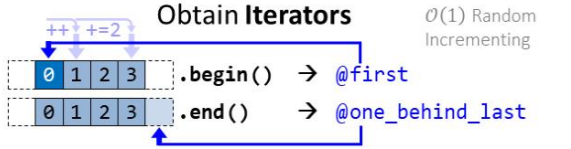
```
vector w {7,4,2};   // vector<int>
```

Typical Memory Layout

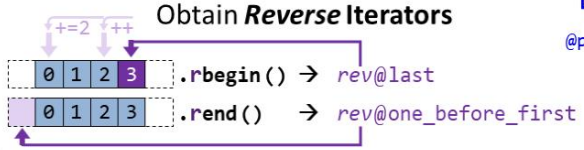


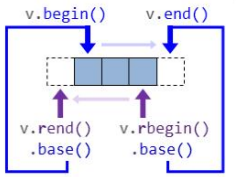
Obtain Iterators

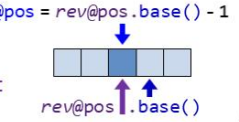
$\mathcal{O}(1)$ Random Incrementing



Obtain *Reverse* Iterators







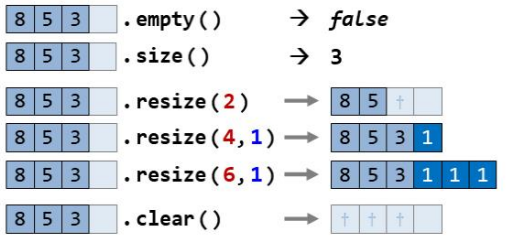
Assign New Content To An Existing Vector

```
vector<int> v1 {8,5,3};
vector<int> v2 {6,8,1,9};
v1      =     v2;
```

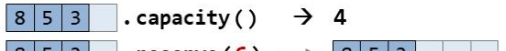
(deep copy from source)

new state of v1

Query/Change Size (= Number of Elements)



Query/Grow Capactiy (= Memory Buffer Size)



Avoid expensive memory allocations: `.reserve` capacity before appending / inserting if you know the (approximate) number of elements to be stored in advance!

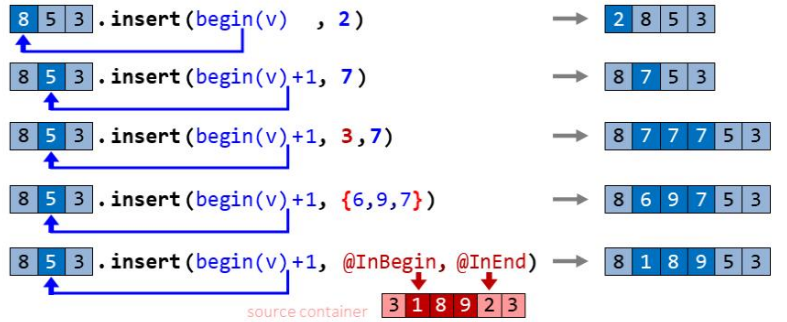
Append Elements

$\mathcal{O}(1)$ Amortized Complexity

Insert Elements at Arbitrary Positions

$\mathcal{O}(n)$ Worst Case

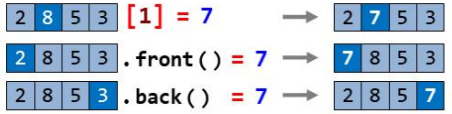
```
vector<int> v {8,5,3};
```



Get Element Values

$\mathcal{O}(1)$ Random Access

Change Element Values



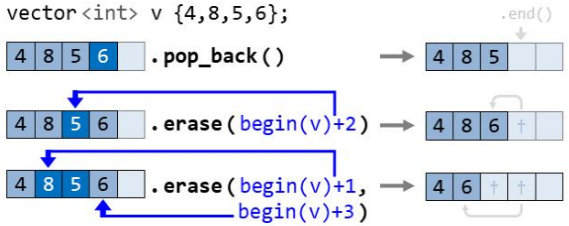
Out of Bounds Access

Erase Elements

$\mathcal{O}(n)$ Worst Case

```
vector<int> v {4,8,5,6};
```

.end()



Shrink The Capacity

(might be inefficient)

Erasing, resizing or clearing will not shrink the capacity!

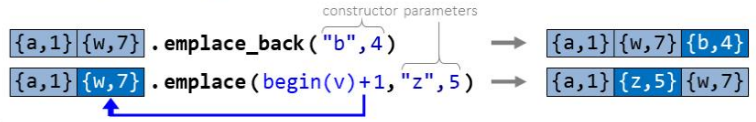
```
vector<int> v (1024,0);   // capacity is at least 1024
v.resize(40);             // capacity unchanged!
v.shrink_to_fit();        // may shrink (not guaranteed)
v.swap(vector<int>(v));   // shrinks but has copy overhead
```

Insert & Construct Elements in Place

$\mathcal{O}(n)$ Worst Case

```
vector<pair<string,int>> v {{"a",1},{"w",7}};
```

constructor parameters



Пример работы с std::vector

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers;

    // Добавление элементов в вектор
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Обращение к элементам вектора по индексу
    for (int i = 0; i < numbers.size(); ++i) {
        std::cout << numbers[i] << " ";
    }

}
```

Пример работы с std::vector

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<std::string> words = {"apple", "banana", "orange", "grape"};

    std::string searchWord;
    std::cout << "Введите слово для поиска: ";
    std::cin >> searchWord;

    bool wordFound = false;

    // Поиск слова в векторе
    for (const auto& word : words) {
        if (word == searchWord) {
            wordFound = true;
            break;
        }
    }

    if (wordFound) {
        std::cout << "Слово найдено в векторе" << std::endl;
    } else {
        std::cout << "Слово не найдено в векторе" << std::endl;
    }

}
```

Пример работы с std::vector - матрица

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<std::vector<int>> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    // Вывод элементов двумерного массива с перебором индексов
    for (size_t i = 0; i < matrix.size(); ++i) {
        for (size_t j = 0; j < matrix[i].size(); ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Итераторы (начало)

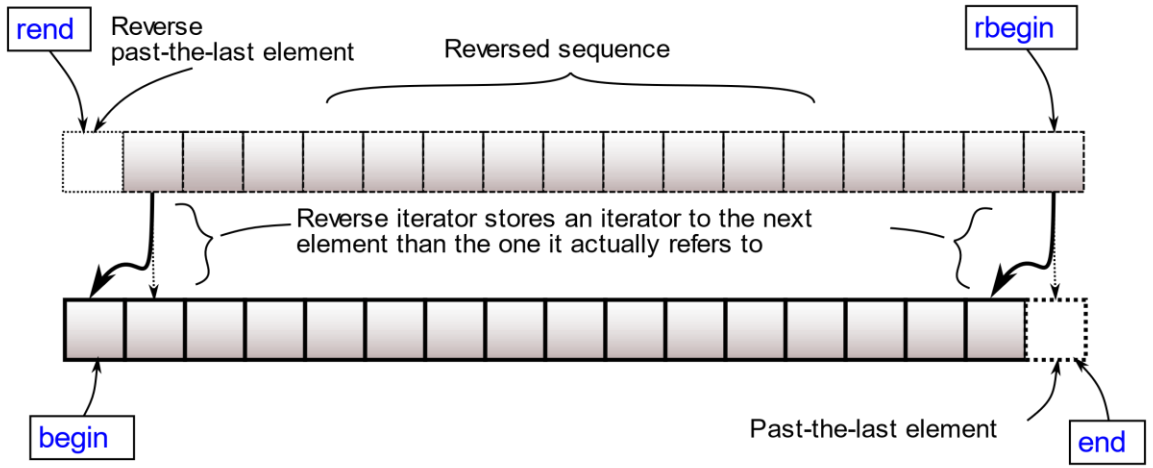
Итератор — это структура данных, предназначенная, для того чтобы перебирать элементы контейнера (последовательности), при этом не задумываясь, с каким именно контейнером происходит работа.

Получить итераторы у контейнера можно через:

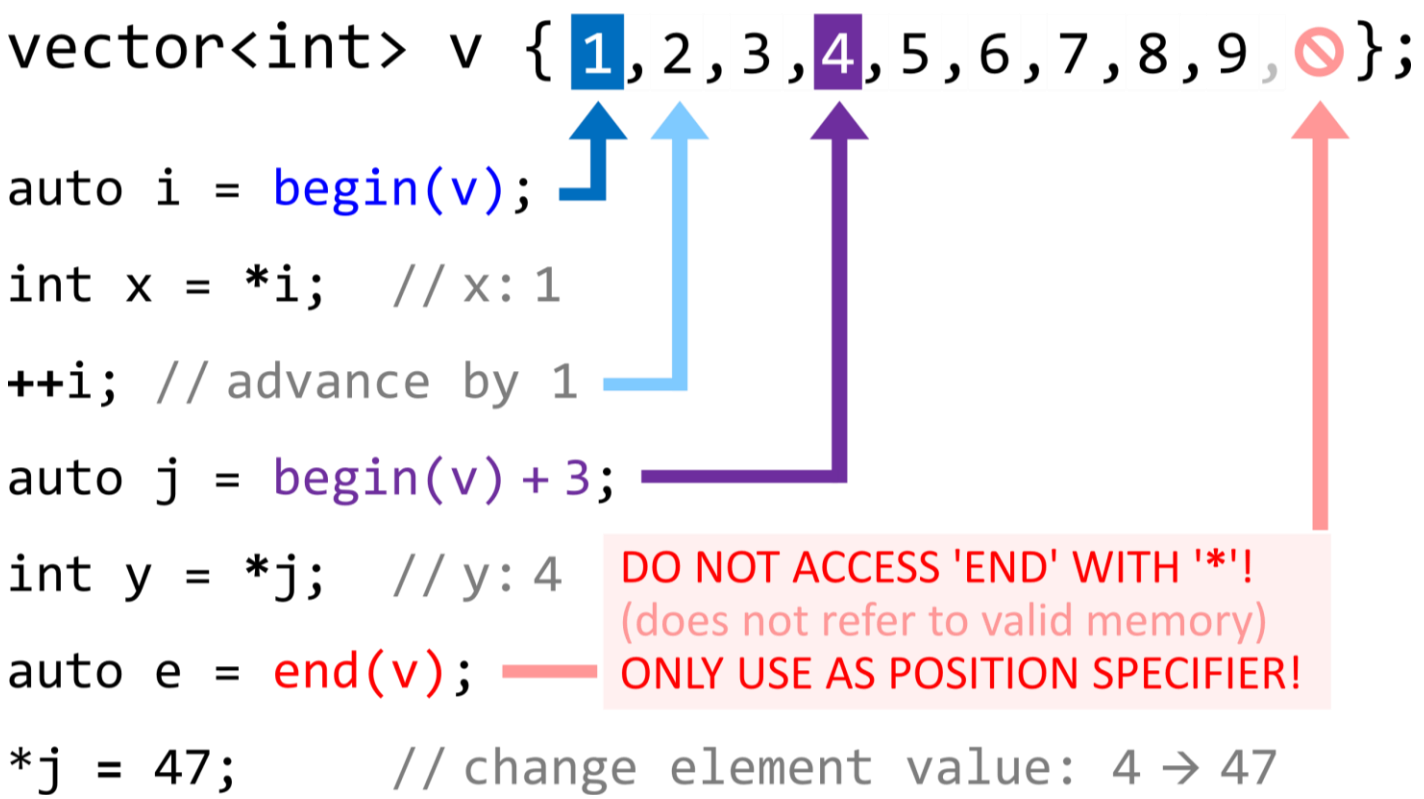
- методы `begin()` и `end()`: `array.begin();`
- одноимённые функции: `begin(array);`

Чтобы ходить по контейнеру в обратном направлении используются реверс-итераторы: `rbegin()` и `rend()`.

Итераторы (начало)



Итераторы (начало)

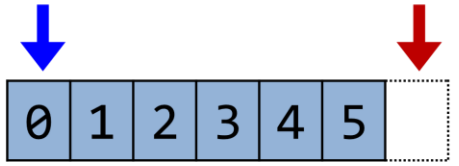


Итераторы (начало)

```
std::vector<int> v {0, 1, 2, 3, 4, 5};
```

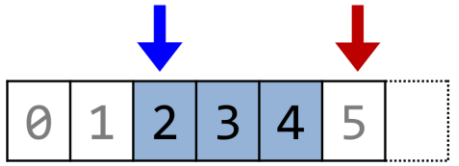
`begin(v)`, `end(v)`

6 elements



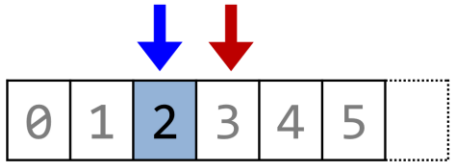
`begin(v)+2`, `begin(v)+5`

3 elements



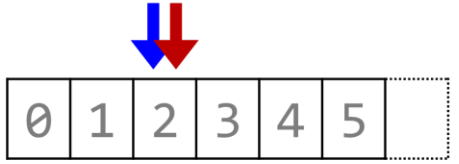
`begin(v)+2`, `begin(v)+3`

1 element



`begin(v)+2`, `begin(v)+2`

empty range



C++ Standard Library Algorithms

Iterator Ranges — @begin+3, @begin+6, @begin, @end

distance(@begin, @end) → element_count <iterator>

Non-Modifying Sequence Operations

min_element(@begin, @end) → @minimum

minmax_element(@begin, @end) → {@minimum, @maximum} C++11

any_of (@begin, @end, f(○)↦bool) →
all_of (@begin, @end, f(○)↦bool) →
none_of(@begin, @end, f(○)↦bool) → C++11
- true, if *f* yields true for *any*, *all* or *none* elements in the input range
- false otherwise

find_if(@begin, @end , f(○)↦bool)→
find (@begin, @end , value)→
- @1st match
- @end if no match

count_if(@begin, @end , f(○)↦bool) → number of occurrences
count (@begin, @end , value) →

equal(@begin1, @end1, @begin2) → true if all elements in both ranges are equal

mismatch(@begin1, @end1, @begin2) → {@mismatch_in1, @mismatch_in2}

search(@beg1, @end1, @beg2, @end2) →
- @1st occurrence of sequence 2 inside sequence 1
- @end1 otherwise

Binary Search On *Sorted* Sequences ⇒ $\mathcal{O}(\log n)$

lower_bound(@begin, @end, value)→ sorted input!
- @1st element **not** < value
- @end if no such element exists

upper_bound(@begin, @end, value)→ sorted input!
- @1st element > value
- @end if no such element exists

equal_range(@begin, @end, value) → {
- @1st item **not** < value or @end if none such found),
- @1st item > value or @end if none such found**}**

sorted input! < value > value

Reordering Elements

reverse(@begin, @end)

sort(@begin, @end, f(○,○)↦bool)
sort(@begin, @end, std::less)

stable_sort(@begin, @end, compare(○,○)↦bool) =○<○
compare_case_insensitive
"stable" = preserves the relative order of equivalent elements

nth_element(@begin, @nth, @end) < nth ≥ nth
element at nth position → element that would be in that position in a sorted sequence

partition(@begin, @end, f(○)↦bool) → @part2 is_odd

rotate(@begin, @newfst, @end) → @old_begin

next_permutation(@begin, @end) → *true* if new permutation is lexicographically greater

shuffle(@begin, @end, random_engine)

Manipulate *Sorted* Sequences ⇒ $\mathcal{O}(n)$

merge($@1_{beg}$, $@1_{end}$, $@2_{beg}$, $@2_{end}$, @out) sorted input! sorted output

set_union($@1_{beg}$, $@1_{end}$, $@2_{beg}$, $@2_{end}$, @out)

Changing Values

copy(@begin, @end, @out)

transform(@begin, @end, @out, f(○)↦■)

generate(@begin, @end, f()↦■) ascending_step_2

replace(@begin, @end, old, new)

replace_if(@begin, @end, f(○)↦bool, new) is_even

remove (@begin, @end, value) → @end_of_
remove_if(@begin, @end, f(○)↦bool) → remaining is_even

unique(@begin, @end) → @end_of_remaining

erase(*container* , value) → erased_count C++20

Numeric Algorithms <numeric>

reduce(@begin, @end, ω=●{}, ⊕=○+○)
reduce(@begin, @end, ω, ⊕(□,○)↦■) → ω+●₂+●₀+●ₙ+⋯+●₁+●₃ C++17 arbitrary evaluation order!

transform_reduce(@●,@●, @♦,ω,⊕=□+○ ,⊗=○×◊)
transform_reduce(@●,@●, @♦,ω,⊕(□,Δ)↦■,⊗(○,◊)↦▲) → $R_{\otimes}$
transform_reduce(@●,@●, ω,⊕(□,Δ)↦■,f(○)↦▲) → R_f C++17 arbitrary evaluation order!

$R_f = \omega + f(\bullet_0) + f(\bullet_1) + \cdots$ $R_{\otimes} = \omega + (\bullet_0 \times \blacklozenge_0) + (\bullet_1 \times \blacklozenge_1) + \cdots$

inclusive_scan(@beg,@end,@out, ⊕=□+○)
inclusive_scan(@beg,@end,@out, ⊕(□,○)↦■, ω) → @o_{end} C++17

Sequence Queries

all_of (C++11)
any_of (C++11)
none_of (C++11)
count
count_if
find
find_if
find_if_not (C++11)
find_end
find_first_of
adjacent_find
for_each
for_each_n (C++17)
sample (C++20)
equal
mismatch
search
search_n
lexicographical_compare
lexicographical_compare_three_way (C++20)

Reordering Elements

reverse
reverse_copy
rotate
rotate_copy
shift_left (C++20)
shift_right (C++20)
shuffle (C++11)
swap
swap_ranges
iter_swap

Partitioning

is_partitioned
partition
stable_partition
partition_copy
partition_point

Permutations

is_permutation (C++11)
next_permutation
prev_permutation

Sorting

sort
stable_sort
partial_sort
partial_sort_copy
is_sorted
is_sorted_until
nth_element

Changing Elements

copy
copy_backward
copy_if (C++11)
copy_n (C++11)
move (C++11)
move_backward (C++11)
fill
fill_n
generate
generate_n
transform
replace
replace_if
replace_copy
replace_copy_if
remove
remove_if
remove_copy
remove_copy_if
unique
unique_copy

Binary Search on Sorted Ranges

binary_search
lower_bound
upper_bound
equal_range
includes

Merging of Sorted Ranges

merge
inplace_merge
set_union
set_intersection
set_difference
set_symmetric_difference

Heaps

make_heap
sort_heap
push_heap
pop_heap
is_heap (C++11)
is_heap_until (C++11)

Minimum/Maximum

min
max
min_element
max_element
minmax (C++11)
minmax_element (C++11)
clamp (C++17)

Numeric <numeric>

accumulate
adjacent_difference
inner_product
partial_sum
iota (C++11)
reduce (C++17)
inclusive_scan (C++17)
exclusive_scan (C++17)
transform_reduce (C++17)
transform_inclusive_scan (C++17)
transform_exclusive_scan (C++17)

h/cpp hackingcpp.com

v3.0