

Алгоритмизация и программирование

Лекция 1-3

О преподавателях

Чабанов Владимир Викторович, старший преподаватель Кафедры компьютерной инженерии и моделирования Физико-технического института.

Кафедра: 310А

E-mail: chabanov.vv@cfuv.ru

VK: <https://vk.com/id444710087>

О курсе

- Год поступления студентов: 2021;
- Лекции: 8 часов;
- Практические занятия: 8 часов;
- Аттестация: экзамен;
- Основной язык: C++;

** Академический час - 45 минут, т.е. пол пары.*

Дни 1 - 10

Выучить переменные, константы, массивы, строки, выражения, функции...



Дни 11 - 21

Выучить потоки, указатели, ссылки, классы, объекты, наследование, полиморфизм...



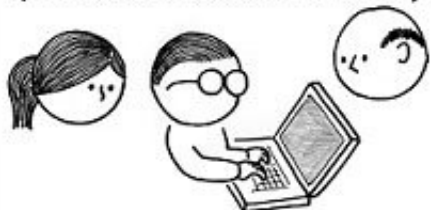
Дни 22 - 697

Много программировать для себя. Иногда взламывать что-то, но все время учиться на ошибках.



Дни 698 - 3648

Общаться с другими программистами. Работать над проектами с ними. Учиться у них.



Дни 3649 - 7781

Выучить продвинутую теоретическую физику и сформулировать теорию квантовой гравитации.



Дни 7782 - 14611

Выучить биохимию, молекулярную биологию, генетику...



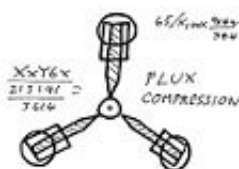
День 14611

Использовать знания по биологии для создания омолаживающего зелья.



День 14611

Использовать знания по физике для создания поточного конденсатора и вернуться в день 21.



День 21

Заменить себя-из-прошлого.



Материалы курса

- Курс на мудле: <https://moodle.cfuv.ru/course/view.php?id=21690>;
- Материалы на GitHub: https://github.com/VladimirChabanov/alg_and_prog_zo;

Система оценивания

Фрагмент приказа №135 от 11.02.2020 "Об утверждении Порядка применения балльно-рейтинговой системы оценивания успеваемости обучающихся по программам ВО в ФГАОУ ВО "КФУ им. В.И. Вернадского":

3.13. Экзамен

3.13.1. Рейтинговые баллы промежуточной аттестации по дисциплине (модулю) $B_{ПА}$, изучаемой в одном семестре, определяется по сумме баллов текущего контроля успеваемости в семестре $B_{сем}$, и баллов, полученных при сдаче экзамена $B_{ЭКЗ}$, т. е.

$$B_{ПА} = B_{сем} + B_{ЭКЗ}.$$

Система оценивания

3.13.2. Ответ на экзамене оценивается в диапазоне от 0 до 40 ($0 \leq B_{\text{ЭКЗ}} \leq 40$).

Ответ на экзамене, оцененный меньше 20 баллов считается неудовлетворительным – обучающемуся за экзамен выставляется нулевой рейтинговый балл ($B_{\text{ЭКЗ}} = 0$).

Шкала соответствия рейтингового балла за экзамен
и оценки в 5-балльной системе

Оценка за экзамен в 5-балльной системе	Рейтинговый балл за экзамен ($B_{\text{ЭКЗ}}$)
отлично	$35 \leq B_{\text{ЭКЗ}} \leq 40$
хорошо	$28 \leq B_{\text{ЭКЗ}} < 35$
удовлетворительно	$20 \leq B_{\text{ЭКЗ}} < 28$
неудовлетворительно	$0 \leq B_{\text{ЭКЗ}} < 20$

Система оценивания

3.13.3. Экзаменационная оценка по дисциплине (модулю) выставляется в соответствии со шкалой пересчета рейтинговых баллов в оценку по 5-балльной системе.

Шкала пересчета рейтингового балла по дисциплине (модулю)
в экзаменационную оценку в 5-балльной системе

Рейтинговый балл по дисциплине ($B_{ПА}$)	Оценка по дисциплине в 5-балльной системе
$90 \leq B_{ПА} \leq 100$	отлично
$71 \leq B_{ПА} < 90$	хорошо
$53 \leq B_{ПА} < 71$	удовлетворительно
$0 \leq B_{ПА} < 53$	неудовлетворительно

Система оценивания

Балл за **работу в семестре** определяется как сумма баллов по всем контрольным точкам (55 баллов) + бонусные баллы (5 баллов):

Баллы за **экзамен** проходит в 2 этапа:

- Тестирование. Проверяет полноту освоения курса (простые вопросы, но по всем темам);
- Опрос по билетам. Проверяет глубину освоения материала (несколько вопросов, но подробно).

Каждый этап оценивается отдельно, по 100 балльной шкале. Итоговая оценка за экзамен определяется по формуле:

$$B_{\text{экз}} = 40 \cdot \frac{B_1}{100} \cdot \frac{B_2}{100}$$

Если за первый этап получена оценка ниже 50 баллов, то за весь экзамен выставляется оценка *НЕудовлетворительно*, т.к. $B_{\text{экз}}$ гарантировано будет меньше 20 баллов.

Практика

Практические и контрольные задания размещены в системе [Яндекс.Контест](#).

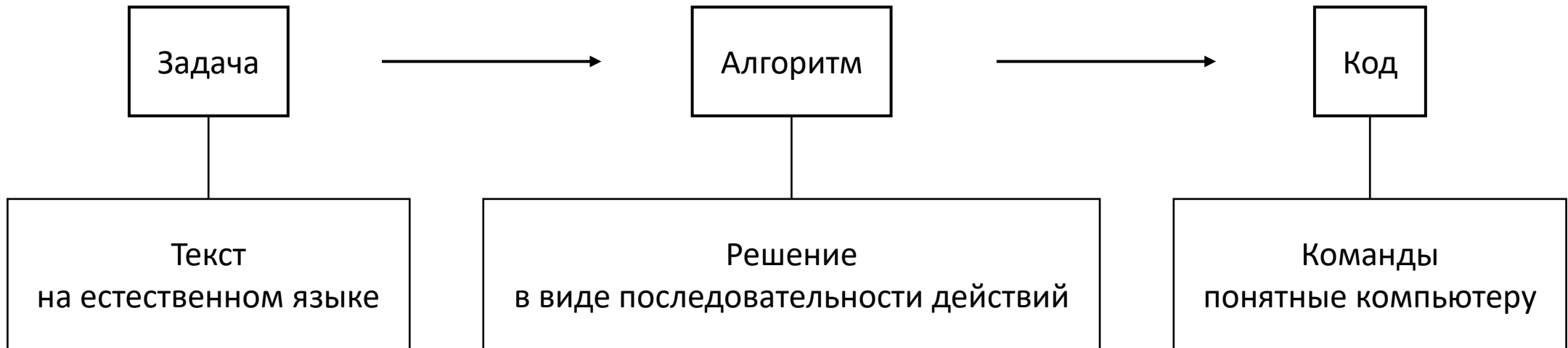
Доступ к практическим заданиям: [заполните форму](#);

О чём предмет

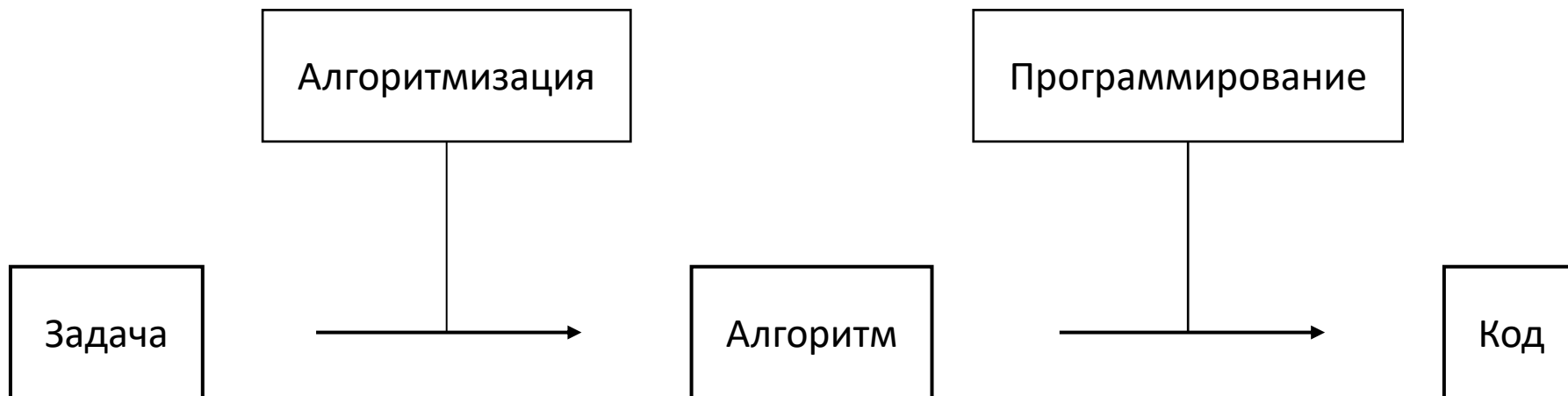
О предмете



О предмете



О предмете



На каком языке будем писать

C++



C++ - компилируемый, статически типизированный язык программирования общего назначения.

Основной принцип: zero-overhead

Создан: в начале 80-х (появление: 1983; выпуск: 1985)

Автор: Бьёрн Страуструп

Где писать код

Онлайн-компиляторы

[Wandbox](#)

- доступно большое количество языков (не только C++);

[Compiler Explorer](#)

- доступно большое количество языков (не только C++);
- для C++ доступно множество различных компиляторов в том числе экспериментальных;
- позволяет посмотреть ассемблерный код и сравнить его для разных вариантов сборки;
- есть встроенная поддержка некоторых популярных библиотек;

[OnlineGDB](#)

- можно запустить дебагер.

Локально

Visual Studio

- доступно большое количество языков (не только C++);
- "всё включено" (компилятор, отладчик, профилировщик);
- есть community версия;

Что такое код/программа на C++

Что такое код?

```
#include <iostream>

int main(){
    std::cout<<"Hello World";
    return 0;
}
```

Код – это текст, который написан в соответствии с "правилами" языка – стандартом языка.

Код должен быть сохранён в файл с определённым расширением (для C++: .cpp .h .hpp, ...);


- файл с расширением .cpp – файл с исходным кодом (Source Code File);
- файл с расширением .h – заголовочный файл (Header file);

Стандарт

Официальный сайт Standard C++ Foundation: <https://isocpp.org/>

Buy this standard

Format	Language
✓ PDF	English

CHF 198  Buy

Стандарт – это платный документ.

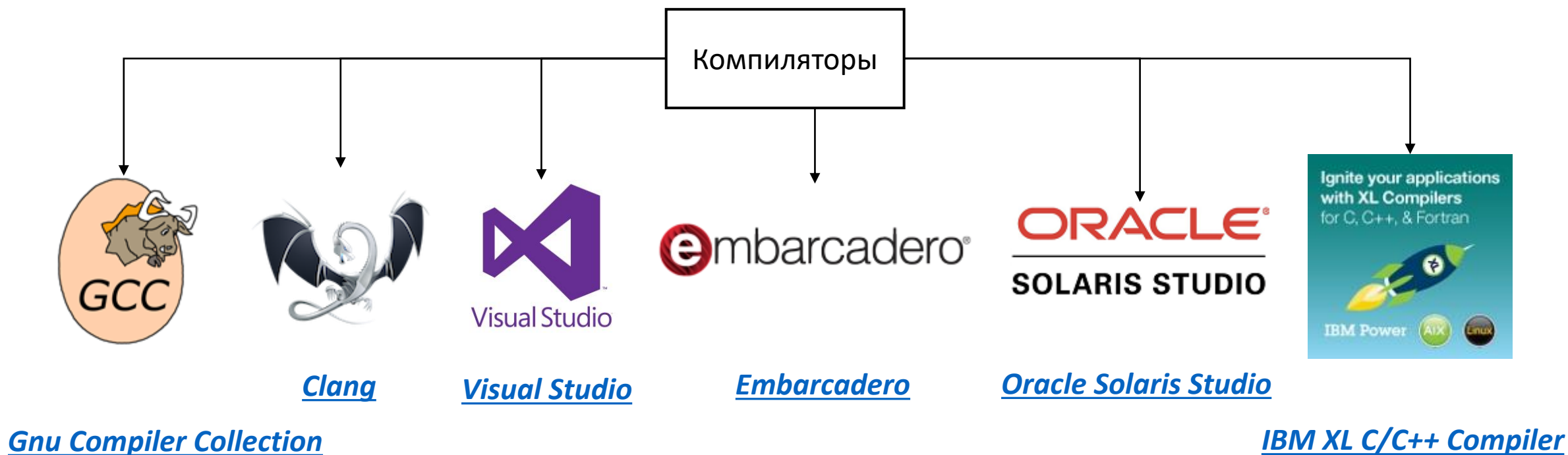
[Черновик стандарта](#) практически не отличаются от самого стандарта.

Стандарт – не учебник по языку, он больше похож на справочник.

Компиляторы

Стандарт – это текстовый документ и он не сможет преобразовать код в исполняемый файл.

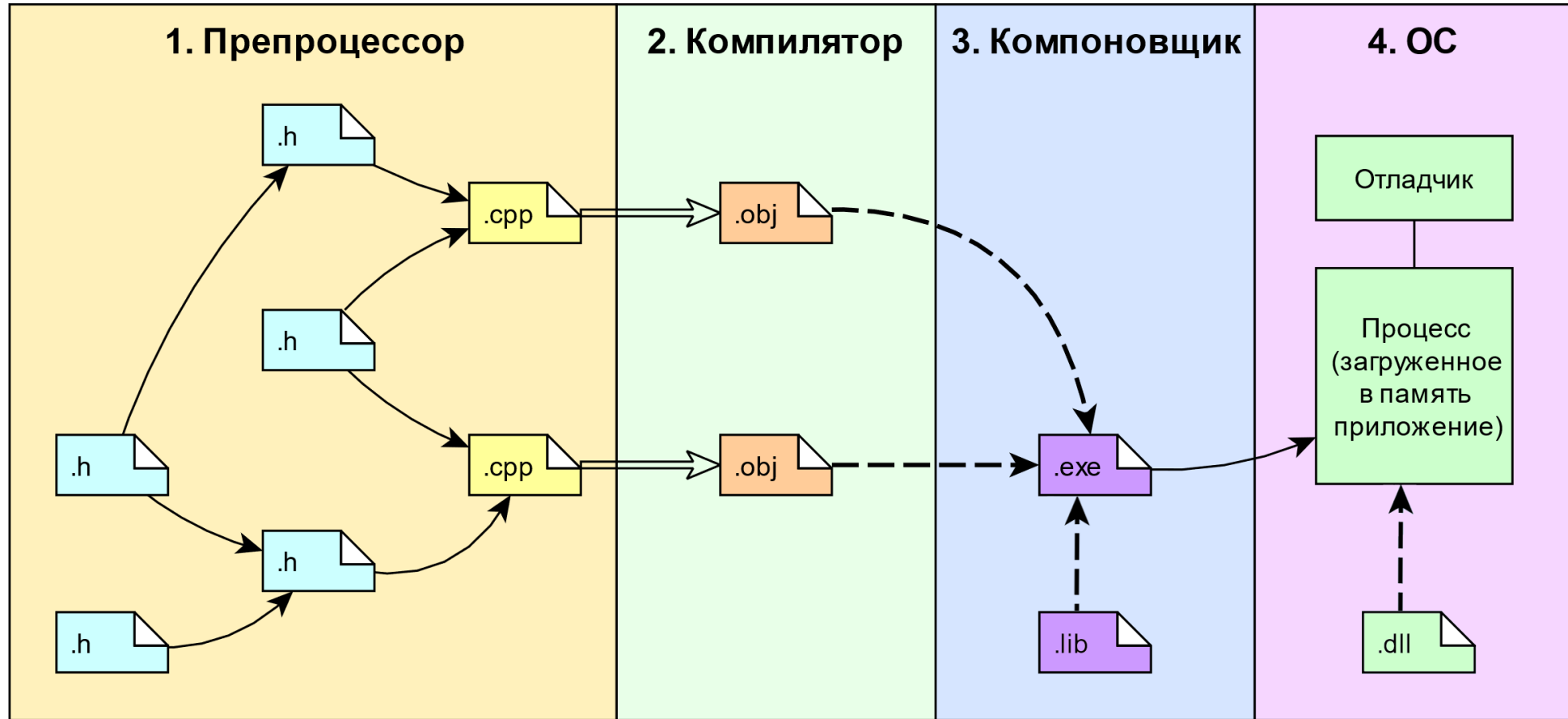
Компилятор – программа, переводящая написанный на языке программирования текст в набор машинных кодов.



Программа

Программа на C++ – это набор текстовых файлов (сpp и h), с исходным кодом. Для получения исполняемой программы (exe) эти файлы передаются компилятору.

Этапы компиляции (трансляции)



Комментарии

<https://wandbox.org/permlink/g919ArA0C3dqefZm>

Ввод вывод

Ввод вывод: [база](#), [сообщения об ошибке](#), [перегрузка операторов](#);

Строки: [база](#), [разновидности подробнее](#);

```
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

size

index based

iterator based

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..."
s.clear()	⇒ s = ""
s.resize(3)	⇒ s = "I'm"
s.resize(20, '?')	⇒ s = "I'm sorry, Dave.?????";
s.insert(4, "very ")	⇒ s = "I'm very sorry, Dave."
s.erase(5, 2)	⇒ s = "I'm srry, Dave."
s[15] = '!'	⇒ s = "I'm sorry, Dave!"
s.replace(11, 5, "Frank")	⇒ s = "I'm sorry, Frank"
s.insert(s.begin(), "HAL: ")	⇒ s = "HAL: I'm sorry, Dave."
s.insert(s.begin()+4, "very ")	⇒ s = "I'm very sorry, Dave."
s.erase(s.begin()+5)	⇒ s = "I'm srry, Dave."
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
                        source iterator range
                        b c d e f g h i j
string( a b c d ) copy/move → a b c d
                        source string object
```

Obtain Iterators

or

Reverse Iterators

```
.begin() → @first
           ↓
           a b c d e f

.end() → @one_behind_last
           ↓
           a b c d e f
           don't use to access elements!

.rbegin() → reverse@last
           ↓
           a b c d e f
           ↓
           .base()

.rend() → reverse@one_before_first
           ↓
           a b c d e f
           ↓
           .base()
           don't use to access elements!
```

String → Number Conversion

```
int      stoi (●, ●, ●);
long     stol (●, ●, ●);
long long stoll(●, ●, ●);
const string&
input string
std::size_t* p = nullptr
output for
number of processed characters

unsigned long stoul (●, ●, ●);
unsigned long long stoull(●, ●, ●);
int base = 10
base of target system;
default: decimal

float      stof (●, ●, ●);
double    stod (●, ●, ●);
long double stold(●, ●, ●);
```

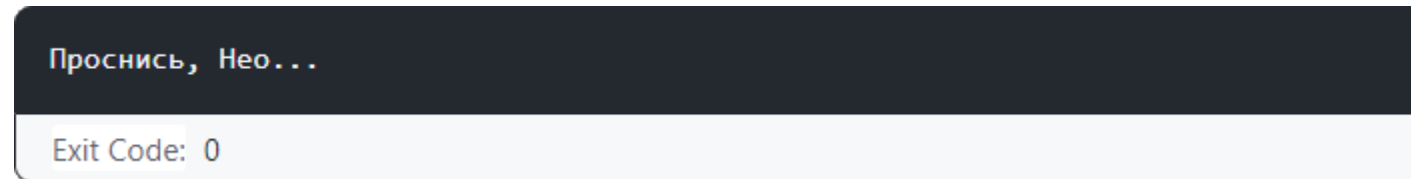
Number → String Conversion

```
string to_string( ● );

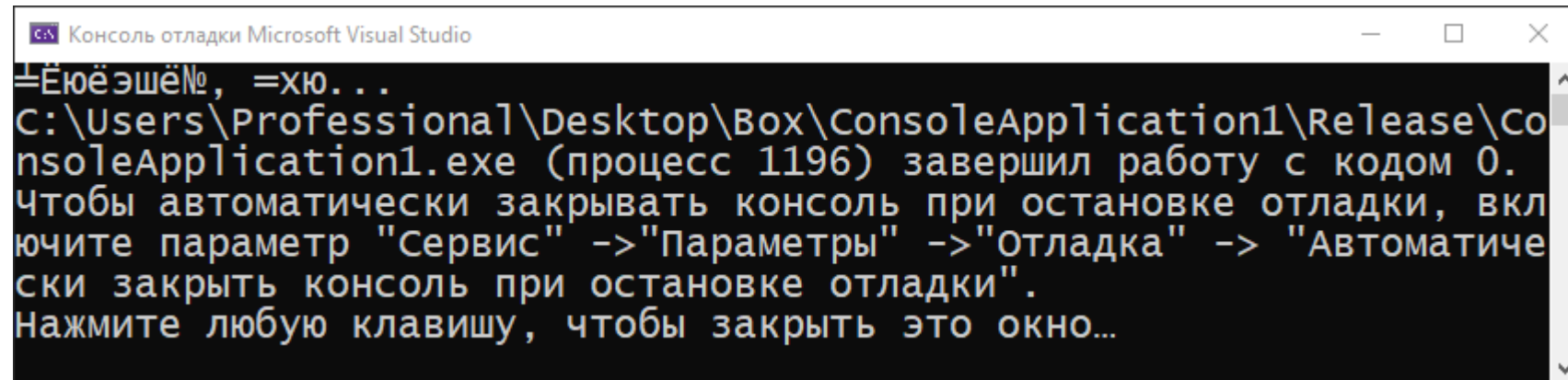
int | long | long long |
unsigned | unsigned long | unsigned long long |
float | double | long double
```

Крокозябры

Unix:



Windows:



setlocale

Магия которая позволяет побороть крокозябры, но работает не всегда.

Это не единственное решение, есть ещё множество вариантов.

Исходники должны быть в кодировке 1251

```
#include <windows.h>

SetConsoleCP(1251);          // установка кодовой страницы win-ср 1251 в поток ввода
SetConsoleOutputCP(1251);    // установка кодовой страницы win-ср 1251 в поток вывода

system("chcp 1251");
```

<https://wandbox.org/permlink/zEhVfp9IF76ObKvA>

Пробельные символы

Символы пробел, табуляция, перевод строки, возврат каретки, новая страница, вертикальная табуляция и новая строка называются пробельными, поскольку они имеют то же самое назначение, что и пробелы между словами и строками в тексте на естественном языке. Эти символы отделяют друг от друга лексемы, например константы и идентификаторы.

Переменная

```
std::string name = "James Bond";
```

Значение ("James Bond")

Идентификатор/имя (name)
может быть 0 или больше

Переменная

Тип (std::string)

Адрес (0x7ffd7ca6b9a0)

Переменная

Создаём переменные:

```
std::string name = "James Bond";  
int answer = 42;  
double trash;
```

Модифицируем:

```
name = "James";  
answer = 43;  
trash = -1.0;
```

Читаем:

```
std::cout << name << ' ' << answer << ' ' << trash << std::endl;
```

Типы данных

Фундаментальные

Числа с плавающей
запятой

- float
- double
- long double

Целочисленные
(интегральные)

- bool
- char
- wchar_t
- char16_t
- char32_t
- signed и unsigned:
 - char
 - short int
 - int
 - long int
 - long long int

`std::nullptr_t`

`void`

Составные

Массивы

Структуры/Классы

Объединения

Перечисления

Указатели

Функции

One definition rule (ODR)

Definition – определение.

```
int answer;
```

```
int answer = 42; // ошибка
```

Массивы

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

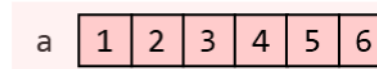
h/cpp/ hackingcpp.com

`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

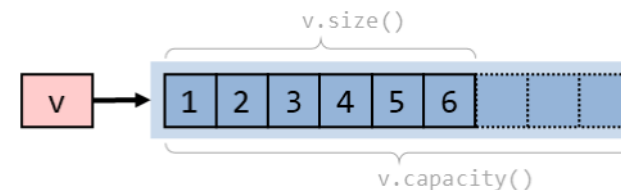
`vector<T>`

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
```



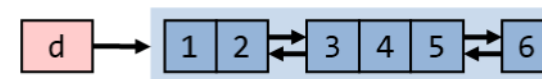
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

`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
```



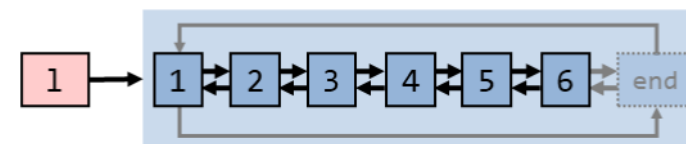
fast insertion/deletion at both ends

`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();
```



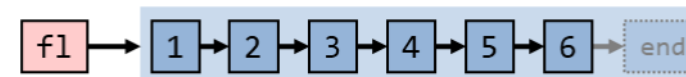
fast splicing; many operations without copy/move of elements

`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>

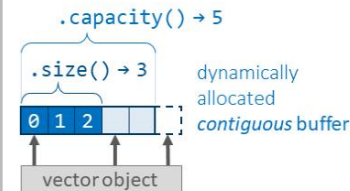
h/cpp hackingcpp.com

Construct A New Vector Object

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

C++17 value type deducible from argument type
vector w {7,4,2}; // vector<int>

Typical Memory Layout



Assign New Content To An Existing Vector

```
vector<int> v1 {8,5,3}; (deep copy from source)
vector<int> v2 {6,8,1,9};
v1 = v2;
new state of v1: [6, 8, 1, 9]
[8, 5, 3].assign({4,1,3,5}) → [4, 1, 3, 5]
[8, 5, 3].assign(2, 1) → [1, 1]
[8, 5, 3].assign(@InBeg, @InEnd) → [2, 1, 1, 2]
source container: [3, 2, 1, 1, 2, 3]
```

Query/Change Size (= Number of Elements)

```
[8, 5, 3].empty() → false
[8, 5, 3].size() → 3
[8, 5, 3].resize(2) → [8, 5, +]
[8, 5, 3].resize(4, 1) → [8, 5, 3, 1]
[8, 5, 3].resize(6, 1) → [8, 5, 3, 1, 1, 1]
[8, 5, 3].clear() → [+, +, +]
```

Query/Grow Capacity (= Memory Buffer Size)

```
[8, 5, 3].capacity() → 4
[8, 5, 3].reserve(6) → [8, 5, 3, +, +, +]
```

Get Element Values $O(1)$ Random Access

```
[2, 8, 5, 3][1] → 8
[2, 8, 5, 3].front() → 2
[2, 8, 5, 3].back() → 3
```

Change Element Values

```
[2, 8, 5, 3][1] = 7 → [2, 7, 5, 3]
[2, 8, 5, 3].front() = 7 → [7, 8, 5, 3]
[2, 8, 5, 3].back() = 7 → [2, 8, 5, 7]
```

Out of Bounds Access

```
[2, 8, 5, 3][6] → Undefined Behavior
[2, 8, 5, 3].at(6) → Throws Exception
std::out_of_range
```

Erase Elements $O(n)$ Worst Case

```
vector<int> v {4,8,5,6};
[4, 8, 5, 6].pop_back() → [4, 8, 5, +]
[4, 8, 5, 6].erase(begin(v)+2) → [4, 8, 6, +]
[4, 8, 5, 6].erase(begin(v)+1, begin(v)+3) → [4, 6, +, +]
```

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
```

Obtain Iterators

$O(1)$ Random Incrementing
[0, 1, 2, 3].begin() → @first
[0, 1, 2, 3].end() → @one_behind_last

Obtain Reverse Iterators

[0, 1, 2, 3].rbegin() → rev@last
[0, 1, 2, 3].rend() → rev@one_before_first

```
v.begin() v.end()
v.rend() v.rbegin()
base() base()
@pos = rev@pos.base() - 1
rev@pos.base()
```

[2, 8, 5, 3].data() → pointer_to_first

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

Append Elements $O(1)$ Amortized Complexity

```
[8, 5, 3].push_back(7) → [8, 5, 3, 7]
```






Insert Elements at Arbitrary Positions $O(n)$ Worst Case

```
vector<int> v {8,5,3};
[8, 5, 3].insert(begin(v), 2) → [2, 8, 5, 3]
[8, 5, 3].insert(begin(v)+1, 7) → [8, 7, 5, 3]
[8, 5, 3].insert(begin(v)+1, 3, 7) → [8, 7, 7, 7, 5, 3]
[8, 5, 3].insert(begin(v)+1, {6,9,7}) → [8, 6, 9, 7, 5, 3]
[8, 5, 3].insert(begin(v)+1, @InBeg, @InEnd) → [8, 1, 8, 9, 5, 3]
source container: [3, 1, 8, 9, 2, 3]
```

Insert & Construct Elements in Place $O(n)$ Worst Case

```
vector<pair<string,int>> v {{"a",1}, {"w",7}};
[a,1][w,7].emplace_back("b",4) → [a,1][w,7][b,4]
[a,1][w,7].emplace(begin(v)+1, "z",5) → [a,1][z,5][w,7]
```

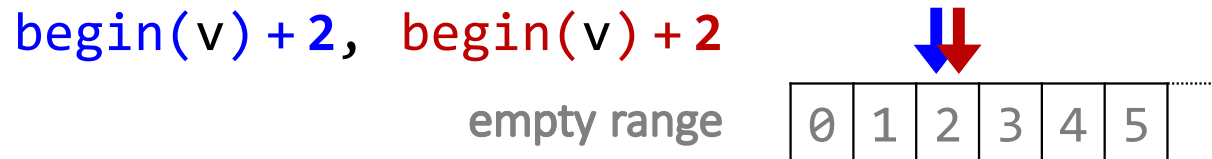
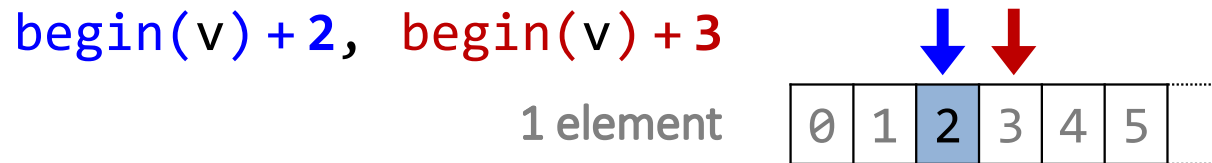
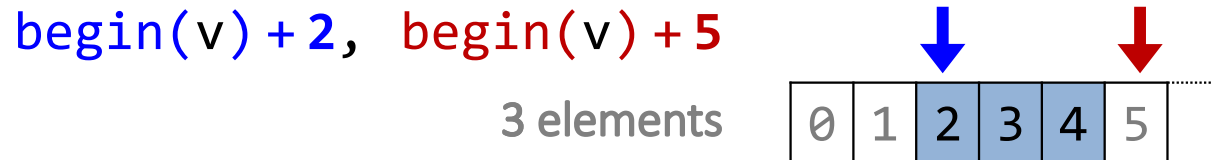
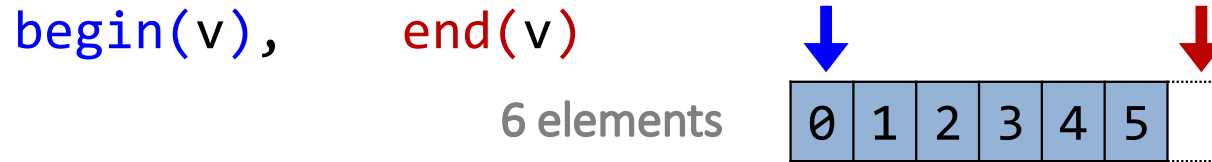
Итераторы (начало)

```
vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9,  };  
auto i = begin(v);   
int x = *i; // x: 1  
++i; // advance by 1   
auto j = begin(v) + 3;   
int y = *j; // y: 4  
auto e = end(v);   
*j = 47; // change element value: 4 → 47
```

DO NOT ACCESS 'END' WITH '*'!
(does not refer to valid memory)
ONLY USE AS POSITION SPECIFIER!

Итераторы (начало)

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



Структуры

Постановка задачи

- Хранить в программе описание характеристик некоторого объекта

Решение I

```
int aliceBirthYear;  
int aliceBirthMonth;  
int aliceBirthDay;  
double aliceHeight;  
double aliceWeight;
```

```
int bobBirthYear;  
int bobBirthMonth;  
int bobBirthDay;  
double bobHeight;  
double bobWeight;
```

Решение I - Проблемы

- Для каждого человека нужно создавать по пять отдельных переменных – **долго, могут быть опечатки**
- Чтобы передать в функцию, нужно перечислить все аргументы – **можно перепутать порядок**

```
print(aliceBirthYear, aliceBirthMonth,  
      aliceBirthDay, aliceHeight, aliceWeight  
);
```

- Как вернуть из функции?

Решение II - Структуры

```
struct human {      // Свой тип данных
    int BirthYear;
    int BirthMonth;
    int BirthDay;
    double Height;
    double Weight;
};    // Точка с запятой обязательно

human alice, bob;    // Создаём переменные
```

Решение II - Структуры

```
struct human {  
    int BirthYear;  
    int BirthMonth;  
    int BirthDay;  
    double Height;  
    double Weight;  
} alice, bob;
```

Решение II - Структуры

```
struct {  
    int BirthYear;  
    int BirthMonth;  
    int BirthDay;  
    double Height;  
    double Weight;  
} alice, bob;
```


Где можно объявлять структуры?

- Внутри функций

```
void func(){  
    struct num{int i;} var;  
};
```

- Вне функций

```
struct num{int i;} var;  
void func(){  
};
```

- Внутри других структур

```
struct num{  
    int i;  
    struct {int k;} j;  
} var;
```

Что может быть членом структуры?

Если можно создать переменную этого типа, то это может быть членом структуры

Например:

- Примитивные типы: `int`, `double`, `char` ...
- Другие структуры;
- Массивы;
- Строки;
- ...

Как работать со структурой

```
struct Data{  
    int Year;  
    int Month;  
    int Day;  
};
```

```
Data now;  
now.Year = 2018;  
now.Day = 9;  
now.Month = 11;
```

Как работать со структурой

```
now.Year = now.Year + 1; // 2019
```

```
cout << now.Day; // 9
```

```
now.Month = now.Day + now.Year; // 2028
```

```
int *p = &now.Month;
```

Инициализация структуры I

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
// joe.id = 1, joe.age = 32, joe.wage = 60000.0  
Employee joe = { 1, 32, 60000.0 };  
// frank.id = 2, frank.age = 28, frank.wage = 0.0  
Employee frank = { 2, 28 };  
Employee frank { 2, 28 };    // C++11
```

Инициализация структуры II C++11/C++14

```
struct Rectangle {  
    double length = 1.0;  
    double width = 1.0;  
};
```

```
int main() {  
    Rectangle x; // length = 1.0, width =  
1.0  
    x.length = 2.0; // Меняем значение  
    return 0;  
}
```

Инициализация структуры III C++11/C++14

```
struct Rectangle {  
    double length = 1.0;  
    double width = 1.0;  
};  
  
int main() {  
    // C++11 – Ошибка; C++14 – Разрешено  
    Rectangle x = {1.0, 1.0};  
  
    return 0;  
}
```

Присваивание значений структурам I

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
Employee joe;  
joe.id = 1;  
joe.age = 32;  
joe.wage = 60000.0;
```


Присваивание значений структурам II

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
Employee joe = {1, 20, 3.0}, mike;  
mike = joe; // Копирование значений joe в mike
```

```
// Присваивание полям joe новых значений C++14  
joe = {2, 22, 6.3};
```

Передача структуры как параметр в функцию

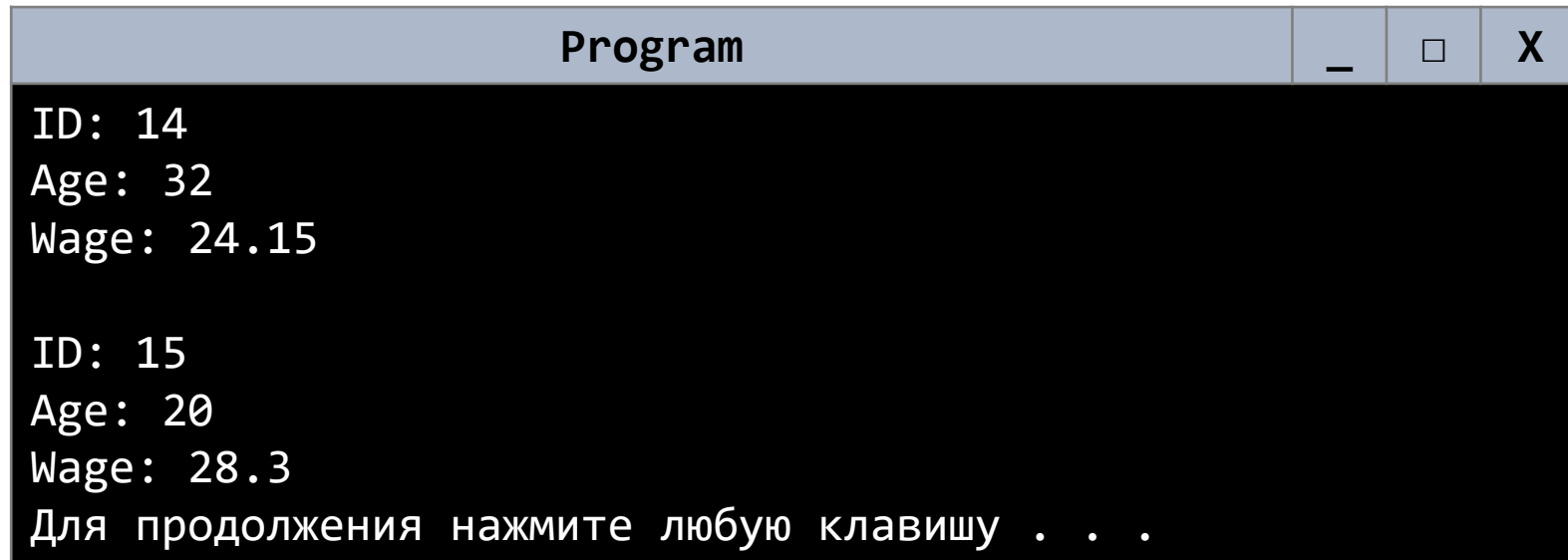
```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
void printInformation(Employee employee) {  
    std::cout << "ID: " << employee.id << "\n";  
    std::cout << "Age: " << employee.age << "\n";  
    std::cout << "Wage: " << employee.wage << "\n";  
}
```

Передача структуры как параметр в функцию

```
int main() {  
    Employee joe = { 14, 32, 24.15 };  
  
    printInformation(joe);  
    std::cout << "\n";  
  
    printInformation({ 15, 20, 28.3 });  
    return 0;  
}
```

Передача структуры как параметр в функцию



```
Program
ID: 14
Age: 32
Wage: 24.15

ID: 15
Age: 20
Wage: 28.3
Для продолжения нажмите любую клавишу . . .
```

The image shows a screenshot of a Windows console application window. The title bar of the window is labeled "Program" and includes standard minimize, maximize, and close buttons. The console output displays two sets of data for different IDs. The first set shows ID: 14, Age: 32, and Wage: 24.15. The second set shows ID: 15, Age: 20, and Wage: 28.3. At the bottom of the console, there is a prompt in Russian: "Для продолжения нажмите любую клавишу . . ." (Press any key to continue).

Передача структуры в функцию через указатель

```
void printInformation(Employee *employee) {  
    std::cout << "ID: " << (*employee).id << "\n";  
    std::cout << "Age: " << (*employee).age << "\n";  
    std::cout << "Wage: " << (*employee).wage << "\n";  
}
```

```
void printInformation(Employee *employee) {  
    std::cout << "ID: " << employee->id << "\n";  
    std::cout << "Age: " << employee->age << "\n";  
    std::cout << "Wage: " << employee->wage << "\n";  
}
```

Возврат структур из функций

```
struct Point3d {  
    double x, y, z;  
};
```

```
Point3d getZeroPoint() {  
    Point3d temp = { 0.0, 0.0, 0.0 };  
    return temp;  
}
```

```
int main() {  
    Point3d zero = getZeroPoint();  
    return 0;  
}
```

Дополнительные сведения

Разные типы

```
struct Point3d {  
    double x, y, z;  
};
```

```
struct Vector3d {  
    double x, y, z;  
};
```

```
Point3d p = { 0.0, 0.0, 0.0 };
```

```
Vector3d v;
```

```
v = p; // Ошибка. У v и p разные типы
```


Массив структур

```
struct Point3d {  
    double x, y, z;  
};
```

```
Point3d p[2] = {{}, {1.0, 2.0, 3.0}};
```

```
p[0].x = 1.0;
```

```
std::cout << p[0].x << ' ' << p[0].y << ' ' << p[0].z;
```

Вложенные структуры

```
struct Employee {  
    short id;  
    int age;  
    float wage;  
};
```

```
struct Company {  
    Employee CEO;    // CEO – это структура  
    int numberOfEmployees;  
};
```

```
Company myCompany = {{ 1, 42, 60000.0f }, 5 };  
std::cout << myCompany.CEO.id;
```

Размер структуры и выравнивание I

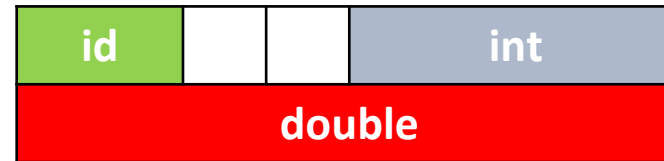
```
struct Employee {  
    short id;      // sizeof(short) == 2  
    int age;       // sizeof(int) == 4  
    double wage;   // sizeof(double) == 8  
};  
  
sizeof(Employee); // 16 != ( 2 + 4 + 8  
)
```

Размер структуры и выравнивание II

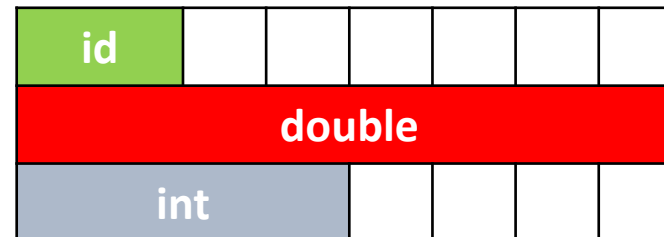
```
struct Employee {  
    short id;      // sizeof(short) == 2  
    double wage;   // sizeof(double) == 8  
    int age;       // sizeof(int) == 4  
};  
  
sizeof(Employee); // 24 != ( 2 + 4 + 8  
)
```

Размер структуры и выравнивание II

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```



```
struct Employee {  
    short id;  
    double wage;  
    int age;  
};
```



Перечисления

enum (перечисления)

```
void set_color(/*какой тип?*/ color){  
    /* Код */  
}
```

```
void set_color(int color){  
    /* Код */  
}
```

```
void set_color(std::string color){  
    /* Код */  
}
```

enum

```
enum Color
{
    // Список перечислителей
    color_black,
    color_red,
    color_blue,
    color_green,
    color_white,
    color_cyan,
    color_yellow,
    color_magenta, //<- запятую можно оставить
}; // <- точкой с запятой

// Определяем несколько переменных перечислимого типа Color
Color paint = color_white;
Color house(color_blue);
Color apple { color_red };
```


enum

```
void set_color(Color color){  
    /* Код */  
}  
  
set_color(color_red); // нормально  
set_color(1);         // ошибка
```

enum

```
enum Color
{
    red,
    blue, // blue помещается в глобальное пространство имен
    green
};

enum Feeling
{
    happy,
    tired,
    blue // ошибка, blue уже использовался в enum Color в глобальном пространстве имен
};
```

enum

```
// определяем новое перечисление с именем Animal
enum Animal
{
    animal_cat = -3,
    animal_dog, // присвоено -2
    animal_pig, // присвоено -1
    animal_horse = 5,
    animal_giraffe = 5, // имеет то же значение, что и animal_horse
    animal_chicken // присвоено 6
};

int pet = animal_pig;
Animal mypet = animal_pig;

std::cout << mypet; // перед передачей в std::cout вычисляется как int
std::cin >> mypet;  // вызовет ошибку компилятора
```

enum

```
// Использовать в качестве базы для перечисления
// 8-битный целочисленный тип без знака.
enum Color : std::uint_least8_t
{
    color_black,
    color_red,
    // ...
};
```

enum class

```
enum class Color
{
    red,
    blue, // blue помещается в пространство имен Color
    green
};

enum class Feeling
{
    happy,
    tired,
    blue // нормально, blue помещается в пространство имен Feeling
};

Color color = Color::blue;
Color color = blue; // ошибка
```

Объединения

union (объединения)

```
// Структура
struct building
{
    std::string owner;
    std::string city;
    int amountRooms;
    float price;
};

building apartment;
apartment.owner = "John";
apartment.city = "NY";
apartment.amountRooms = 10;
apartment.price = 100;
```

```
// Объединение
union building
{
    std::string owner;
    std::string city;
    int amountRooms;
    float price;
};

building apartment;
// Только что-то одно
```

Объединение занимает в памяти столько места, сколько занимает самое большое его поле. После инициализации менять тип данных перечисления нельзя (хотя иногда можно)

Указатели

Указатель

Составной тип данных предназначенный для хранения адреса некоторой программной сущности:

```
int a;                // Хранит целое число
int* ptr_a = nullptr; // Хранит адрес переменной типа int

std::string str;       // Хранит строку
std::string* ptr_str;  // Хранит адрес переменной типа std::string

double* ptr_d;         // Хранит адрес переменной типа double
double** ptr_ptr_d;    // Хранит адрес переменной типа double*
```

& | Оператор взятия адреса

Позволяет узнать адрес в памяти, по которому находится программная сущность:

```
int a;  
int* ptr_a = &a;
```

```
std::string str;  
std::string* ptr_str = &str;
```

```
double d;  
double* ptr_d = &d;  
double** ptr_ptr_d = &ptr_d;
```

```
ptr_d = &a;          // Не соответствие типов  
ptr_a = ptr_str;     // Не соответствие типов
```

```
ptr_a = a;           // Не соответствие типов  
a = ptr_a;           // Не соответствие типов
```

* | Оператор разыменования (dereference)

Позволяет получить доступ к программной сущности по адресу:

```
int a;  
int* ptr_a = &a;  
int b = *ptr_a;
```

```
double d;  
double* ptr_d = &d;  
double** ptr_ptr_d = &ptr_d;  
double c = **ptr_ptr_d;
```

```
*ptr_a = 10; // Значение a изменилось на 10
```

Адресная арифметика

Позволяет получить доступ к другому адресу:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int* ptr_a = &a[0];  
int* end_a = &a[9];
```

```
ptr_a = ptr_a + 1; // a[1]  
ptr_a = ptr_a + 2; // a[3]  
ptr_a++;           // a[4]  
ptr_a--;           // a[3]  
ptr_a = ptr_a - 3; // a[0]
```

```
int size = end_a - ptr_a; // 10
```

Ссылки

Ссылка (на l-value)

Альтернативное имя для переменной:

```
int a;           // Хранит целое число  
int& fer_a = a;  // Ничего не хранит, просто имя
```

```
std::string str;           // Хранит строку  
std::string& fef_str = str; // Ничего не хранит, просто имя
```

```
int& fer_a2 = fer_a; // Тоже ссылка на a
```

```
int& fer_a;           // Нельзя  
int&& fer_fer_a;       // Другой тип ссылки  
int&&& fer_fer_fer_a; // Не существует
```

Псевдонимы

Псевдоним | альтернативное имя

```
// typedef
typedef int integer;
integer a = 1;

typedef int vector[10];
vector vect;
vector matrix[5]; // arr[5][10]

typedef int (*foo)(int a, int b);

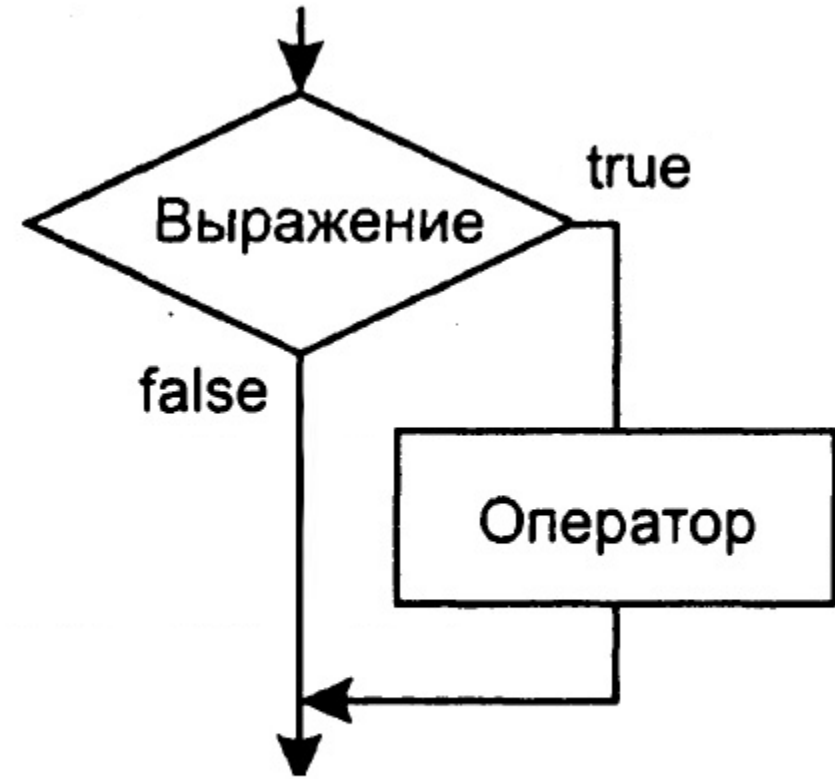
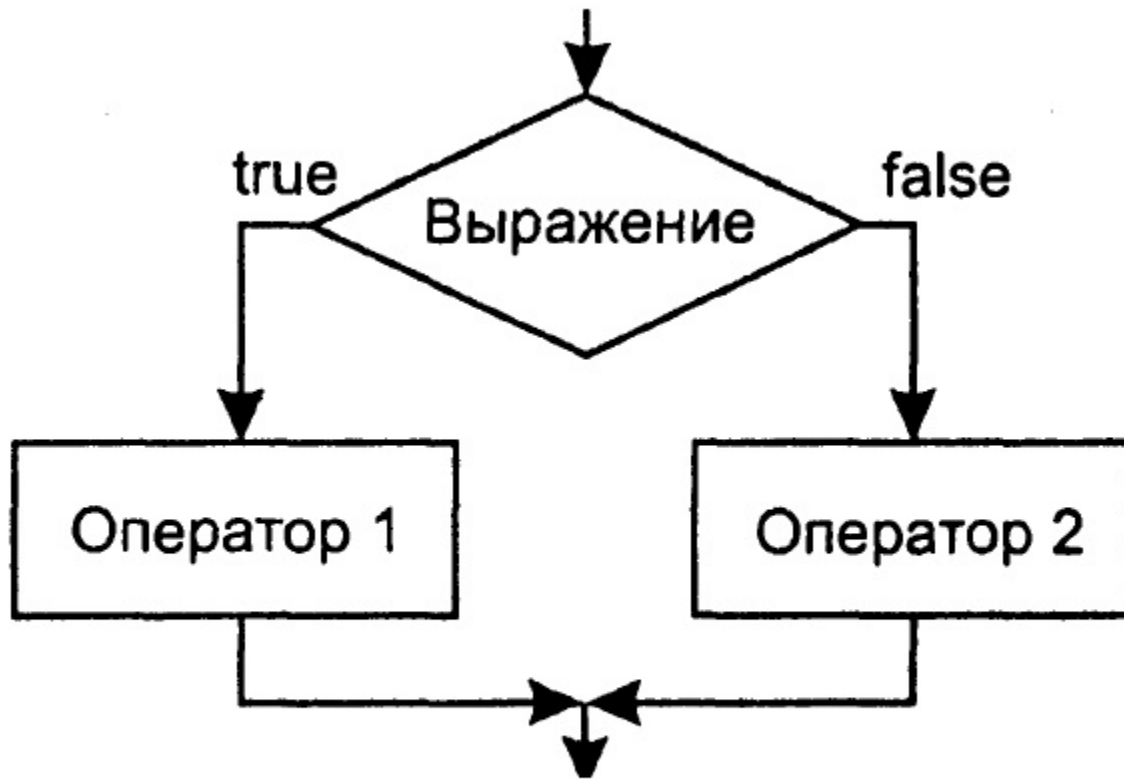
// using
using integer = int;

using vector = int[10];
using matrix = vector[5];

using foo = int (*)(int a, int b);
```


Управление потоком исполнения

Условный оператор



Структурная схема условного оператора

if

Ключевое слово **если**

Выражение

Тело



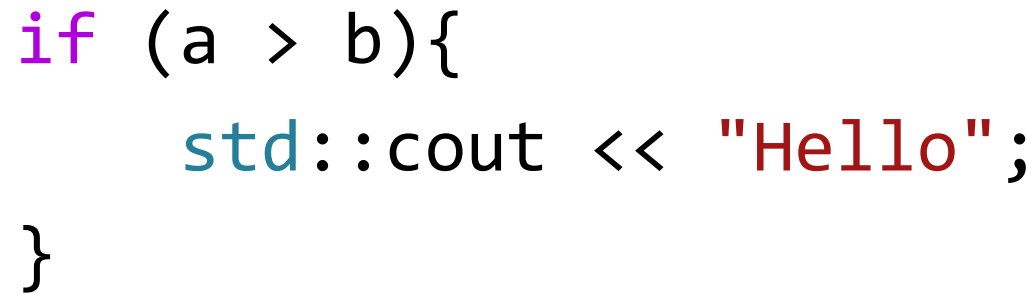
The diagram shows a single-line if statement: `if (a > b) std::cout << "Hello";`. An arrow points from the label 'Ключевое слово **если**' to the word `if`. A horizontal line above the expression `(a > b)` has a downward arrow pointing to it from the label 'Выражение'. Another horizontal line above the body `std::cout << "Hello";` has a downward arrow pointing to it from the label 'Тело'.

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

Ключевое слово **если**

Выражение

Тело



The diagram shows a multi-line if statement: `if (a > b){ std::cout << "Hello"; }`. An arrow points from the label 'Ключевое слово **если**' to the word `if`. A horizontal line above the expression `(a > b)` has a downward arrow pointing to it from the label 'Выражение'. A horizontal line above the body `std::cout << "Hello";` has a downward arrow pointing to it from the label 'Тело'. A vertical line on the right side of the body connects the 'Тело' label line to the closing brace `}`.

```
if (a > b){  
    std::cout << "Hello";  
}
```

if (выражение)

Ожидается, что выражение в скобках типа `bool`, поэтому будет попытка неявно [преобразовать его к `bool`](#).

Если преобразование не допустимо, то – ошибка.

if(инициализация; проверка)

init-statement
↓
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)

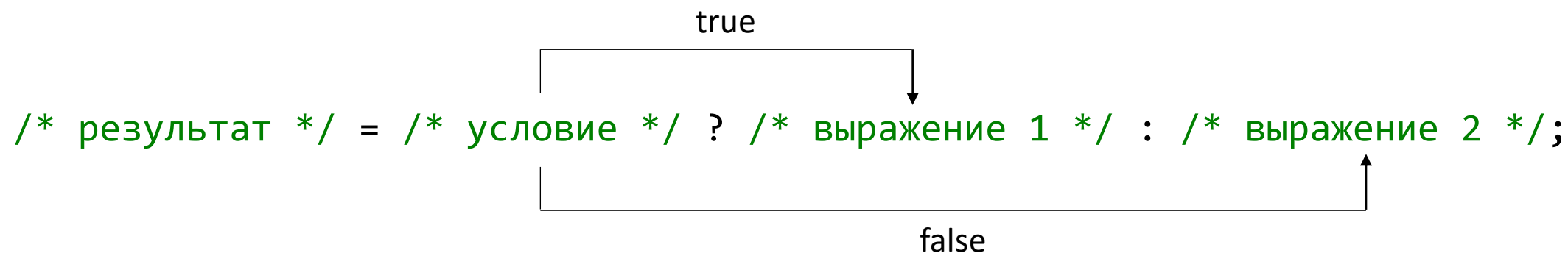
```
if (a > b);
```

```
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

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

```
int a = 5;  
(a > 1) and (a < 10) // true  
(a == 5) or (a == 10) // true  
not (a == 10) // true  
-1 < a < 2 // true
```

Последнее выражение вычисляется последовательно: $(-1 < a) < 2 \rightarrow \text{true} < 2$
если хотите получить результат по математическим правилам пишите: $(-1 < a) \text{ and } (a < 2)$

```
(a > 1) and (a < 3) // false  
(a == 1) or (a == 3) // false  
not (a == 5) // false
```

switch

Выражение
↓

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

Ключевое слово

```
graph LR; V[Выражение] --> value; K[Ключевое слово] --> switch; K --> case1; K --> case2; K --> default;
```

switch (выражение)

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

switch(инициализация; выражение)

init-statement
↓

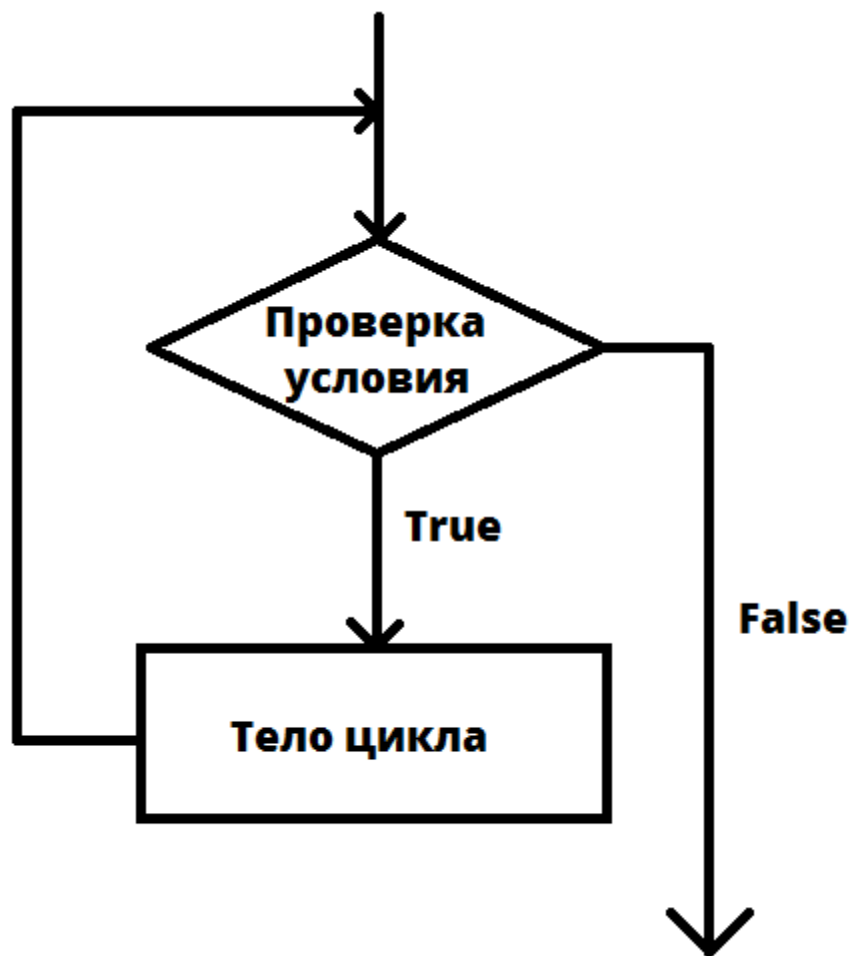
```
switch (int res = a > b; res ){  
    case true: std::cout << ":)"; break;  
    default: std::cout << ":(  
}
```

goto

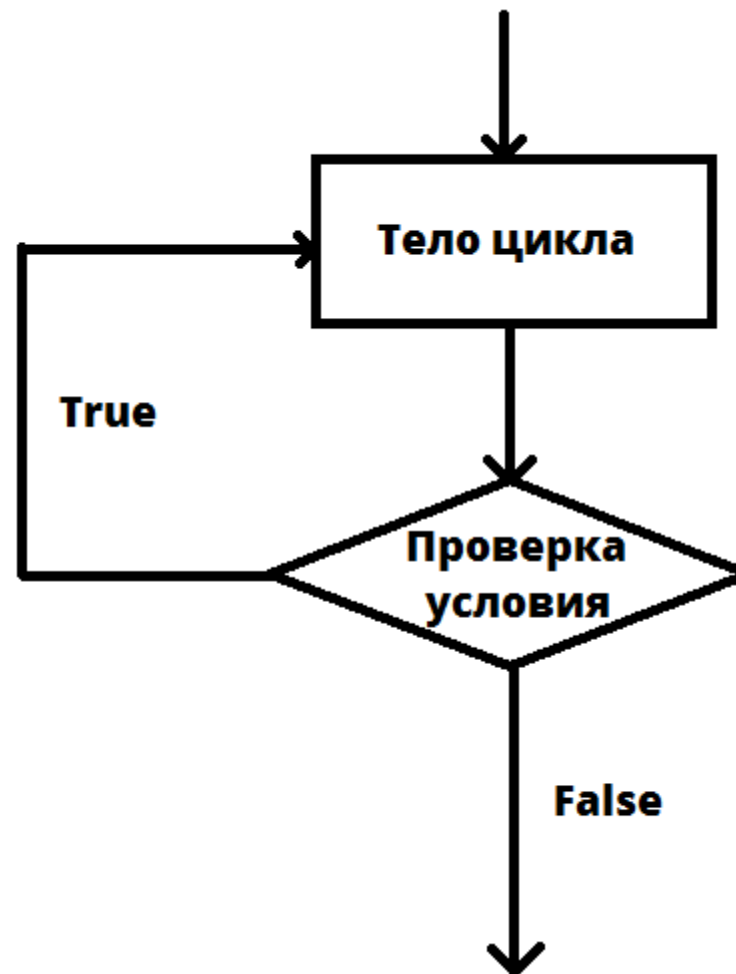
```
label:  
    /* код */  
goto label;
```

* label – обычный идентификатор

Оператор цикла



while



do-while

while

Ключевое слово

Выражение

Тело

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

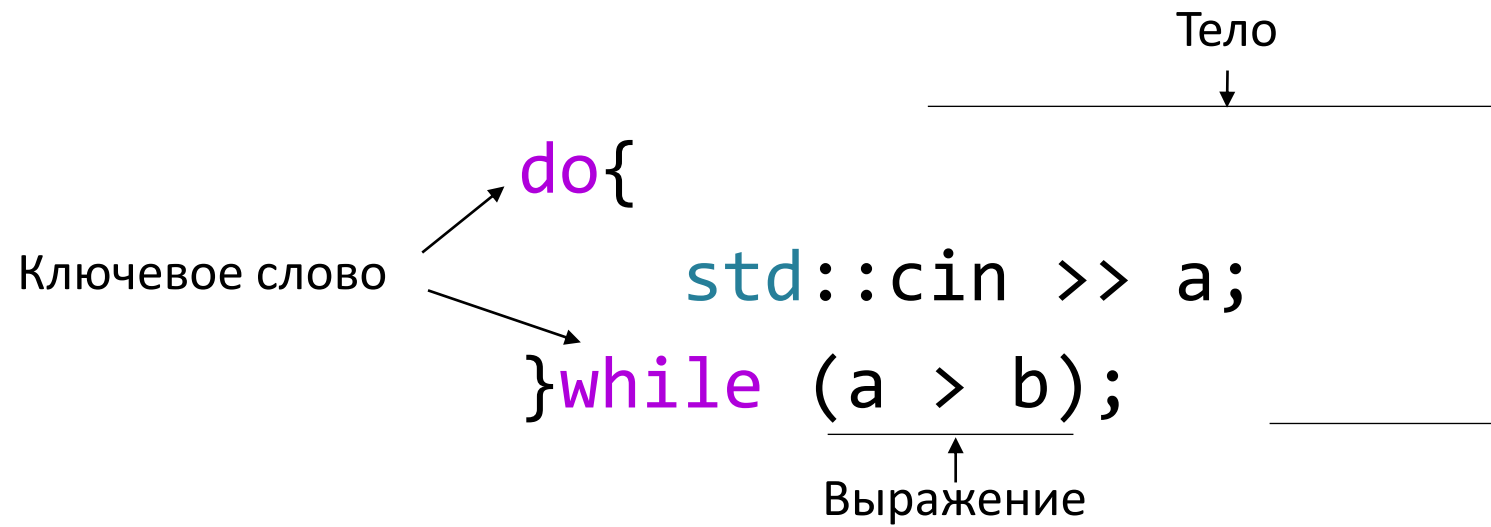
Ключевое слово

Выражение

Тело

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

do-while

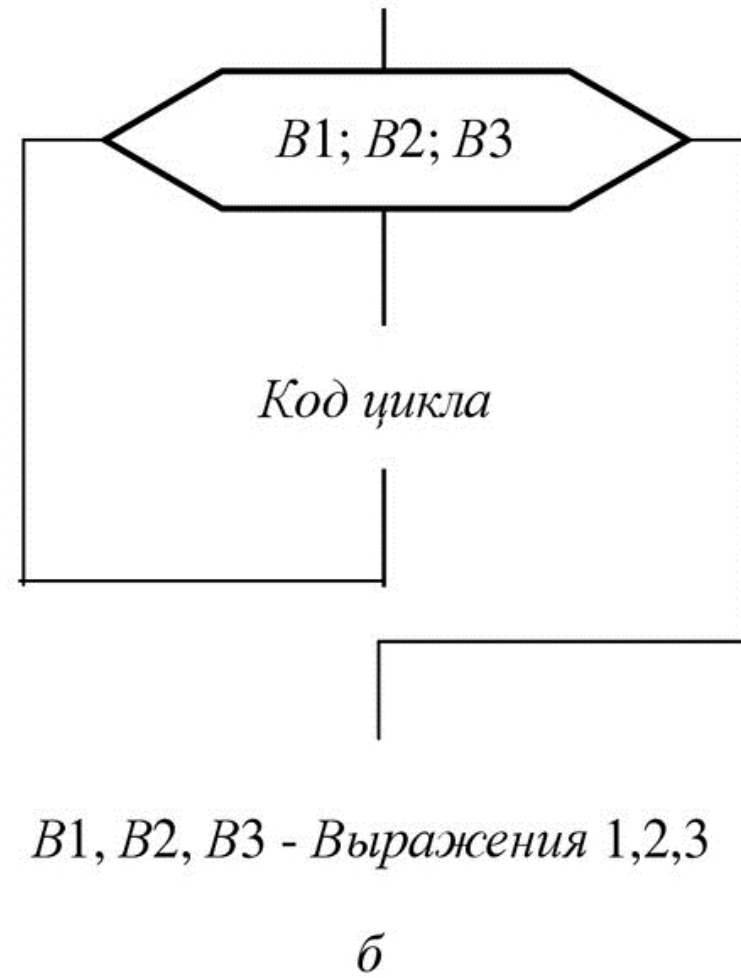
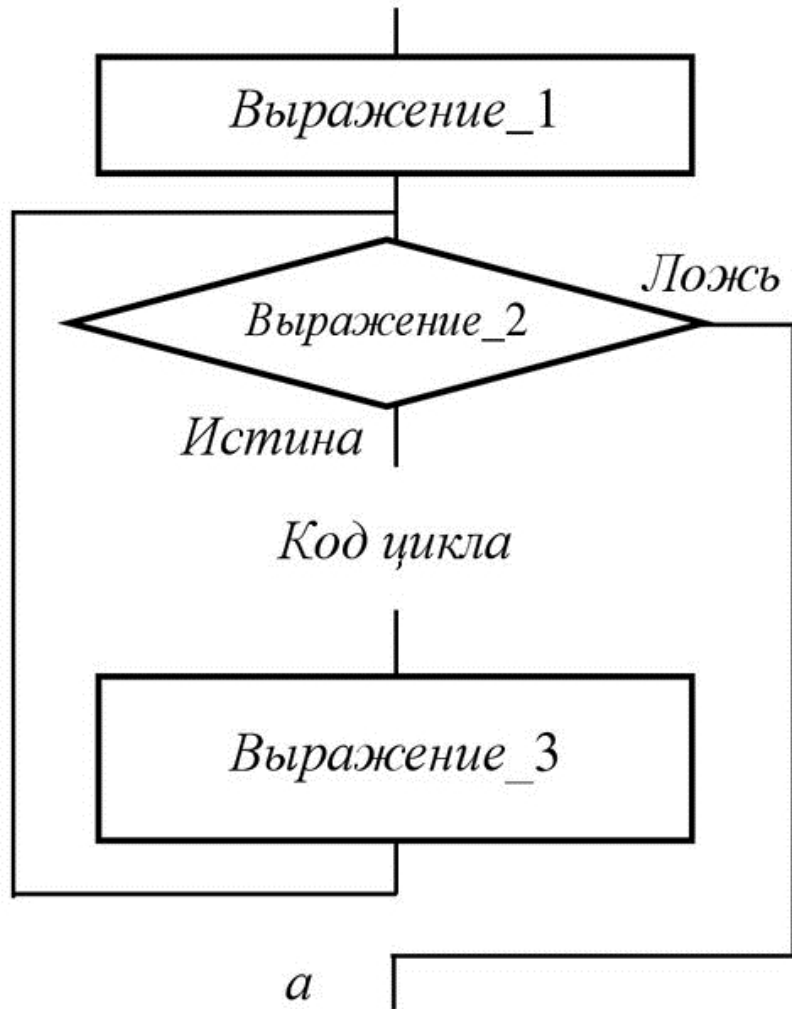


while (выражение)

Ожидается, что выражение в скобках типа `bool`, поэтому будет попытка неявно [преобразовать его к `bool`](#).

Если преобразование не допустимо, то – ошибка.

Оператор цикла for



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

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.

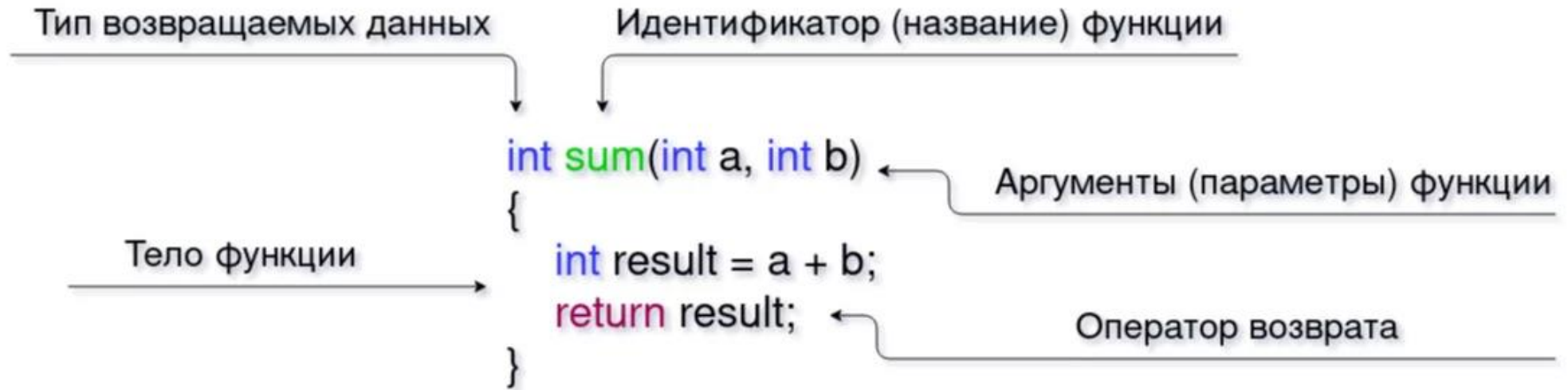
range-declaration – объявление именованной переменной, тип которой является типом элемента последовательности, представленного **range-expression**, или ссылкой на этот тип. Часто использует спецификатор `auto` для автоматического определения типа.

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

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

Функции

Функция



Функция

Объявление (declaration) функции вводит имя функции и ее тип в область видимости (scope);

```
int sum(int a, int b);           auto sum(int a, int b) -> int;
```

Определение (definition) функции связывает имя/тип функции с её телом;

```
int sum(int a, int b)           auto sum(int a, int b) -> int
{                                {
    int result = a + b;         int result = a + b;
    return result;              return result;
}
```

<https://wandbox.org/permlink/DAISkDauoPcyNOdF>

Объявление функции

Прототипом функции в языке Си или C++ называется объявление функции, не содержащее тела функции, но указывающее имя функции, арность, типы аргументов и тип возвращаемых данных.

```
int sum(int a, int b);
```

Сигнатура функции – это части прототипа функции, которые компилятор использует для выполнения разрешения перегрузки.

```
sum(int, int);
```

Формальные параметры (параметры) – это собственно параметры указанные в прототипе/сигнатуре функции (в данном случае `a` и `b`).

Вызов функции

Вызов функции - передача управления потоком исполнения команд в другую точку программы с последующим возвратом в точку вызова.

```
int main()  
{  
    auto res = sum(2, 2);  
    std::cout << res << std::endl;  
}
```

Фактические параметры (аргументы) – конкретные значения, которые передаются формальным параметрам (в данном случае 2 и 2).

Код внутри функции

Время жизни и область видимости локальных переменных

Область видимости локальных переменных, в том числе и параметров. От точки объявления до конца области видимости. Конец области видимости определяется либо концом функции либо концом блока.

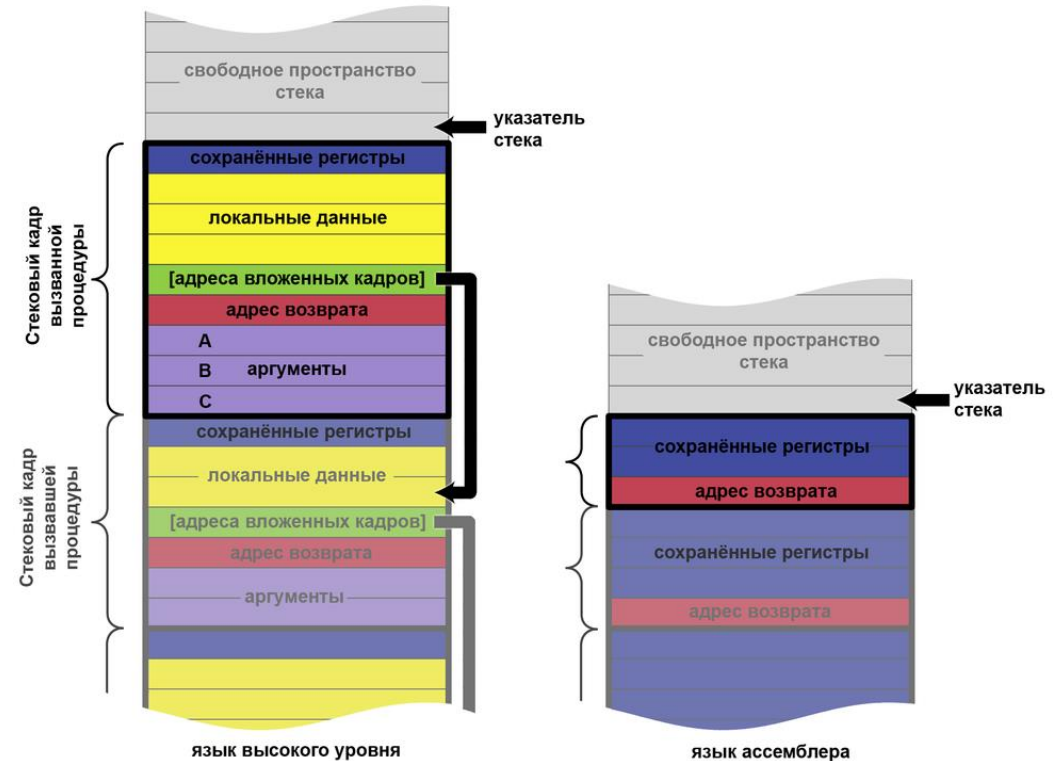
Локальные переменный функции, в том числе и параметры, живут от момента создания до момента выхода из области видимости. Кроме static переменных.

Стек вызова функций

Стек вызова функций

Стек вызовов (стек) может использоваться для различных нужд, но основное его назначение — отслеживать место, куда каждая из вызванных процедур должна вернуть управление после своего завершения. Для этого при вызове процедуры (командами **вызова**) в **стек** заносится адрес команды, следующей за командой **вызова** («адрес возврата»).

Механика вызова функции



Передача данных в функцию

Параметры

Передача данных **по значению**. Создаёт локальную копию передаваемых данных.

```
void swap(int a, int b){  
    int t = a;  
    a = b;  
    b = t;  
}
```

Передача данных **по ссылке**. Создаёт дополнительное имя для переменной переданной в качестве аргумента.

```
void swap(int& a, int& b){  
    int t = a;  
    a = b;  
    b = t;  
}
```

Передача данных **по указателю**. Создаёт копию, но не данных, а адреса по которому они находятся.

```
void swap(int* a, int* b){  
    int t = *a;  
    *a = *b;  
    *b = *t;  
}
```

const

Квалификатор **const** запрещает изменять параметры.

```
void swap(const int a, const int b){  
    int t = a;  
    a = b;  
    b = t;  
}
```

Параметры функции main

Без параметров

```
int main();
```

Доступ к параметрам запуска программы

```
int main(int argc, char const *argv[]);
```

Доступ к параметрам запуска и переменным окружения

```
int main(int argc, char const *argv[], char const *envp[]);
```

Получение данных из функции

Оператор return

Оператор **return** осуществляет прерывание исполнения текущей функции и возврат потока исполнения в точку вызова.

Для void функций не обязателен. Функция завершится после выполнения последней команды в теле функции.

Для не void функций обязателен. После оператора return должно быть указано значение того же (или приводимое) типа, что и в прототипе. Это значение вернётся в качестве результата в вызывающую функцию.

В функции main разрешено не указывать. В этом случае результат будет 0.

Может присутствовать в теле функции множество раз.

```
int sum(int a, int b)
{
    int result = a + b;
    return result;
}
```

```
void print_hello(){
    std::cout << "Hello" << std::endl;
}
```


Возвращаемое значение

Возврат данных **по значению**. Создаёт копию возвращаемых данных и отдаёт наружу.

```
int sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

Возврат данных **по ссылке**. Даёт доступ в нижнему коду к локальной переменной функции.

```
int& sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

Возврат данных **по указателю**. Передаёт наружу информацию об адресе, по которому лежат данные.

```
int* sum(int a, int b){  
    int result = a + b;  
    return &result;  
}
```

const

Квалификатор **const** не играет роли если возврат по значению. В остальных случаях запрещает изменение данных.

```
const int sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

Значение возвращаемое main

Согласно стандарту тип возвращаемого значения функции `main` должен быть только `int`.

```
int main();  
int main(int argc, char const *argv[]);  
int main(int argc, char* argv[], char* envp[]);
```

Только для функции `main` разрешается не указывать оператор `return`, для всех остальных не `void` функций `return` обязателен. В случае, если в `main` нет оператора `return`, то гарантируется, что она вернёт 0.

Значение, которое возвращает функция `main` передаётся операционной системе как результат работы программы.

Это значение в общем случае не влияет ни на что, но его можно использовать в shell-скриптах.

По соглашению если программа вернёт 0, то считается, что она завершилась корректно, а любые другие значения — это не корректное завершение программы. При этом значение для каждого кода ошибки разработчик придумывает по своему желанию (и описывает в документации).

Рекурсия

Рекурсия

Рекурсия — состоит в определении, описании, какого-либо объекта или процесса через самого себя. Функция может содержать вызов себя непосредственно или косвенно.

```
long long fact(int x) {  
    if (x < 2) return 1;  
    else return x * fact(x-1);  
}
```

```
int fib(int N)  
{  
    if (N == 1 || N == 2) return 1;  
    return fib(N - 1) + fib(N - 2);  
}
```

Цикл и рекурсия взаимозаменяемы.

Перегрузка

Перегрузка

В широком смысле **перегрузка** (overloading) — это возможность одновременно использовать несколько функций с одним именем. Компилятор различает их благодаря тому, что они имеют разный набор параметров. В точке вызова компилятор анализирует сигнатуру функции и определяет, какая конкретно функция должна быть вызвана

```
int sum(int a, int b);  
int sum(std::vector<int> arr);
```

Перегрузить по возвращаемому типу нельзя.

Шаблон функции

Шаблон функции

Шаблоны функций представляют некоторый образец, по которому можно создать конкретную функцию, специфическую для определенного типа.

```
template<class T>
T _min(T a, T b) {
    if (a < b) return a;
    return b;
}
```

До момента инстанцирования функции, она не существует.

Проектирование функций

Проектирование функций

- Функции должны быть небольшого размера. Не более одного экрана.
- Имя функции должно однозначно говорить, что эта функция делает.
- **Принцип единой ответственности** (Single Responsibility Principle) – это **принцип**, который гласит, что каждый модуль, класс или **функция** в компьютерной программе должны нести **ответственность** за одну часть функциональности этой программы, и она должна инкапсулировать эту часть.
- По возможности нужно стараться писать чистые функции.
- По возможности нужно писать простой код.
- [Ещё про функции](#)