

Основы программирования.  
Учебное пособие

Притчин И.С.

Июль 2021 – Май 2022

# Оглавление

<b>1 Алгоритмы. Структурное программирование</b>	<b>12</b>
1.1 Алгоритм. Свойства алгоритмов . . . . .	12
1.1.1 Свойства алгоритма . . . . .	13
1.1.2 Способы записи алгоритмов . . . . .	13
1.1.3 Словесно-формульная запись . . . . .	13
1.1.4 Блок-схемы (графический способ) . . . . .	14
1.1.5 Псевдокод . . . . .	15
1.1.6 Программа . . . . .	15
1.2 Структурное программирование . . . . .	23
1.3 Жизненный цикл программного обеспечения . . . . .	25
<b>2 Типы данных. Переменные</b>	<b>30</b>
2.1 Целочисленные типы данных . . . . .	31
2.1.1 <i>char / unsigned char</i> . . . . .	31
2.1.2 <i>int / unsigned int</i> . . . . .	35
2.1.3 <i>short int / unsigned short int; long int / unsigned long int; long long int / unsigned long long int</i> . . . . .	36
2.2 Вещественные типы данных: <i>float, double</i> . . . . .	37
2.3 Пустой тип: <i>void</i> . . . . .	38
2.4 Объявление и инициализация переменных . . . . .	39
2.5 Массивы . . . . .	40
2.6 Указатели . . . . .	42
2.7 Строки . . . . .	44
2.8 Константы . . . . .	45
<b>3 Линейные алгоритмы</b>	<b>50</b>
3.1 Задача о сумме . . . . .	50
3.2 Задача обмена значений . . . . .	52
3.3 Задача о последней цифре числа . . . . .	54
3.4 Задача о вычислении расстояния между двумя точками . . . . .	55
3.5 Задача о генерации числа в заданном диапазоне . . . . .	56
3.6 Задача о сумме арифметической прогрессии . . . . .	58
3.7 Задача перевода температуры из градусов Фаренгейта в градусы Цельсия	59
<b>4 Операции</b>	<b>62</b>
4.1 Арифметические операции . . . . .	62
4.1.1 Унарный минус и унарный плюс . . . . .	62
4.1.2 Сложение и вычитание . . . . .	62
4.1.3 Умножение . . . . .	66
4.1.4 Деление и остаток от деления . . . . .	67
4.1.5 Приоритеты арифметических операций . . . . .	67

4.2	Побитовые операции . . . . .	68
4.2.1	Побитовое И . . . . .	68
4.2.2	Побитовое ИЛИ . . . . .	69
4.2.3	Побитовое НЕ . . . . .	70
4.2.4	Побитовое исключающее ИЛИ . . . . .	71
4.2.5	Побитовый сдвиг вправо . . . . .	71
4.2.6	Побитовый сдвиг влево . . . . .	72
4.2.7	Приоритеты побитовых операций . . . . .	74
4.3	Логические операции и операции сравнения . . . . .	75
4.4	Приоритеты операций . . . . .	76
4.5	Приведение типов . . . . .	78
4.5.1	Неявное приведение типов . . . . .	78
4.5.2	Явное приведение типов . . . . .	82
4.5.3	Последствия при приведении типов . . . . .	82
<b>5</b>	<b>Разветвляющиеся алгоритмы</b>	<b>87</b>
5.1	Условный оператор <i>if</i> . . . . .	87
5.1.1	Поиск модуля числа . . . . .	87
5.1.2	Поиск максимального значения из двух . . . . .	89
5.1.3	Поиск максимального значения из трёх . . . . .	92
5.1.4	Упорядочивание двух чисел . . . . .	97
5.1.5	Решение линейного уравнения . . . . .	98
5.2	Цепочки операторов <i>if-else-if</i> . . . . .	98
5.2.1	Задача о социальной поддержке . . . . .	98
5.2.2	Решение квадратного уравнения . . . . .	101
5.3	Оператор <i>switch</i> . . . . .	102
5.3.1	Задача о зимнем месяце . . . . .	103
<b>6</b>	<b>Циклические алгоритмы</b>	<b>107</b>
6.1	Цикл <i>for</i> . . . . .	107
6.1.1	Вычисление суммы последовательности . . . . .	111
6.1.2	Поиск значений выражений . . . . .	112
6.1.3	Поиск максимума вводимой последовательности . . . . .	116
6.1.4	Поиск последнего нечетного элемента . . . . .	117
6.1.5	Поиска максимального значения последовательности и их количества . . . . .	118
6.2	Цикл <i>while</i> . . . . .	120
6.2.1	Подсчёт количества цифр в числе . . . . .	121
6.2.2	Подсчёт количества единиц в двоичном представлении числа $x$ .	122
6.2.3	Изменение порядка цифр в числе на обратный . . . . .	123
6.2.4	Подсчёт количества положительных и отрицательных чисел последовательности . . . . .	124
6.2.5	Первый элемент рекуррентно заданной последовательности больше $x$ . . . . .	125
6.2.6	Поиск суммы последних $m$ цифр числа $n$ . . . . .	126
6.2.7	Получение средней цифры числа, если количество цифр в числе нечетно . . . . .	127
6.2.8	Проверка числа на простоту . . . . .	129
6.2.9	Поиск второго максимального значения . . . . .	132

6.2.10	Максимальное количество подряд идущих положительных значений . . . . .	134
6.2.11	Максимальное количество знакочередующихся элементов последовательности . . . . .	136
6.2.12	Подпоследовательность с максимальной суммой . . . . .	137
6.2.13	Поиск элемента находящегося перед первым четным . . . . .	138
6.2.14	Поиск элемента, находящегося после последнего минимального .	139
6.3	Цикл <i>do-while</i> . . . . .	142
6.3.1	Поиск количества цифр в числе . . . . .	142
6.4	Выбор подходящего цикла . . . . .	143
6.5	Операторы <i>break</i> , <i>goto</i> , <i>continue</i> . . . . .	144
6.5.1	<i>break</i> . . . . .	144
6.5.2	<i>continue</i> . . . . .	145
6.5.3	<i>goto</i> . . . . .	146
<b>7</b>	<b>Функции</b>	<b>150</b>
7.1	Функции в математике . . . . .	151
7.2	Структурное программирование. Метод пошаговой детализации . . . . .	152
7.2.1	Задача о кирпиче . . . . .	152
7.2.2	Поиск максимального положительного значения из двух чисел .	156
7.2.3	Задача о треугольнике . . . . .	157
7.3	Определение функции . . . . .	158
7.4	Формальные и фактические параметры функции . . . . .	159
7.5	Действия, производимые при вызове функции . . . . .	160
7.6	Возвращаемое значение функции . . . . .	163
7.7	Передача аргументов . . . . .	166
7.8	Решения задач при помощи нерекурсивных функций . . . . .	171
7.8.1	Алгоритмы возведения в степень . . . . .	175
7.8.2	Алгоритм Евклида . . . . .	177
7.8.3	Вывод числа в 16-ой системе счисления. . . . .	178
7.8.4	Задача о счастливом билете. . . . .	180
7.8.5	Задача о разбиении числа. . . . .	181
7.8.6	Задача о совершенных числах. . . . .	183
7.8.7	Быки и коровы. . . . .	184
<b>8</b>	<b>Рекурсия</b>	<b>192</b>
8.1	Виды рекурсии . . . . .	193
8.2	Решение задач с использованием рекурсии . . . . .	195
8.2.1	Вычисление значения $n!$ . . . . .	195
8.2.2	Вывод двоичного представления числа . . . . .	196
8.2.3	Вычисление суммы массива . . . . .	197
8.2.4	Алгоритм однопроходного удаления . . . . .	201
8.2.5	Проверка массива на палиндром . . . . .	201
8.2.6	Сортировка выбором . . . . .	202
8.2.7	Вычисление многочлена в точке . . . . .	203
8.2.8	Бинарный поиск . . . . .	203

<b>9 Работа в CLion</b>	<b>208</b>
9.1 Статический анализ кода . . . . .	208
9.2 Отладчик . . . . .	210
9.3 Автодополнение . . . . .	217
9.4 Выделение программных объектов . . . . .	219
9.5 Горячие клавиши . . . . .	222
9.6 Шаблоны кода . . . . .	224
9.7 Шаблон-окружение . . . . .	226
9.8 Вывод . . . . .	228
<b>10 Алгоритмы обработки одномерных массивов</b>	<b>229</b>
10.1 Алгоритмы, не зависящие от упорядоченности . . . . .	229
10.1.1 Ввод массива . . . . .	229
10.1.2 Вывод массива . . . . .	232
10.1.3 Поиск позиции элемента, удовлетворяющему условию . . . . .	233
10.1.4 Поиск количества элементов, удовлетворяющему условию . . . . .	236
10.1.5 Поиск максимального количества подряд идущих элементов, удовлетворяющих условию . . . . .	238
10.1.6 Однопроходный алгоритм удаления . . . . .	239
10.1.7 Вставка элемента с сохранением относительного порядка других элементов . . . . .	243
10.1.8 Удаление элемента с сохранением относительного порядка других элементов . . . . .	245
10.1.9 Обращение элементов массива . . . . .	246
10.1.10 Проверка упорядоченности массива по неубыванию . . . . .	246
10.2 Неупорядоченные массивы . . . . .	248
10.2.1 Добавление элемента в конец массива . . . . .	248
10.2.2 Удаление элемента из массива без сохранения относительного порядка других элементов . . . . .	248
10.3 Одномерные массивы и работа с динамической памятью . . . . .	249
10.3.1 Ввод и вывод массива (вариация 1) . . . . .	251
10.3.2 Ввод и вывод массива (вариация 2) . . . . .	253
10.4 * Передача функций, как параметров . . . . .	255
10.5 * Массив префиксных сумм . . . . .	261
10.5.1 Поиск количества подотрезков с нулевой суммой . . . . .	262
10.6 * Разностные массивы . . . . .	263
10.6.1 Добавление на отрезке . . . . .	264
10.6.2 Добавление арифметической прогрессии на отрезке . . . . .	265
10.7 Решения задач на одномерные массивы с использованием принципа пошаговой детализации . . . . .	266
10.7.1 Сортировка подпоследовательности до первого вхождения нуля . . . . .	266
10.7.2 Получение упорядоченной последовательности из неуникальных элементов . . . . .	268
10.7.3 Сортировка с шагом . . . . .	271
10.7.4 Подсчёт количества вхождений элементов . . . . .	274
10.7.5 Сортировка точек по удалённости от другой точки . . . . .	276
10.7.6 Вывод чисел, непринадлежащих последовательности . . . . .	277
10.7.7 Сортировка элементов массива, выбранных по критерию . . . . .	278

<b>11 Поиск</b>	<b>283</b>
11.1 Быстрый линейный поиск . . . . .	283
11.2 Линейный поиск в отсортированном массиве . . . . .	283
11.3 Блочный поиск. <i>Sqrt</i> -декомпозиция . . . . .	284
11.4 Бинарный поиск (вариация №1) . . . . .	288
11.5 Бинарный поиск (вариация №2) . . . . .	290
11.6 Бинарный поиск по критерию . . . . .	292
11.7 Решение задач на бинарный поиск . . . . .	294
11.7.1 Поиск позиции первого значения равному $x$ . . . . .	294
11.7.2 Поиск позиции последнего значения равному $x$ . . . . .	294
11.7.3 Задача о верёвках . . . . .	295
11.7.4 Задача о ксероксах . . . . .	296
11.7.5 Задача об уравнении . . . . .	297
11.7.6 Задача о сборе . . . . .	298
11.7.7 Задача о коровах и стойлах . . . . .	299
11.7.8 Поиск максимального среднего арифметического на отрезке длины не менее $D$ . . . . .	300
11.8 Тернарный поиск . . . . .	304
<b>12 Сортировки</b>	<b>306</b>
12.1 Обезьянья сортировка . . . . .	306
12.2 Сортировка выбором . . . . .	307
12.3 Сортировка вставками . . . . .	310
12.4 Обменная сортировка . . . . .	312
12.5 Сортировка расческой . . . . .	313
12.6 Сортировка подсчётом . . . . .	315
12.7 Цифровая сортировка . . . . .	318
12.8 Сортировка слиянием . . . . .	321
12.9 Куча и сортировка кучей . . . . .	324
12.10 Быстрая сортировка . . . . .	332
<b>13 Система непересекающихся множеств</b>	<b>334</b>
13.1 Наивный подход . . . . .	335
13.2 Модификация №1 - Переход к спискам . . . . .	336
13.3 Модификация №2 - Переход к деревьям . . . . .	337
13.4 Задача про людей . . . . .	340
13.5 Алгоритм Краскала . . . . .	341
<b>14 Множества. Представление числовых множеств в ЭВМ</b>	<b>345</b>
14.1 Организация библиотек в языке программирования С . . . . .	345
14.2 Представление числовых множеств и мульти множеств . . . . .	351
14.2.1 Реализация множества на массиве битов . . . . .	352
14.2.2 Реализация множества и мульти множества на массиве типа <i>char</i> . . . . .	358
14.2.3 Множество на неупорядоченном массиве . . . . .	359
14.2.4 Множество на упорядоченном массиве . . . . .	361
<b>15 Одномерное динамическое программирование</b>	<b>363</b>
15.1 Сумма чисел от 1 до $n$ . . . . .	365
15.2 Техники кэширования: мемоизация и табуляция . . . . .	366
15.3 Задача о зайце . . . . .	368

15.4 Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет . . . . .	371
15.5 Вычисление минимального количества монет, необходимого для выдачи сдачи $n$ рублей известными номиналами монет . . . . .	373
15.6 Получение произвольного минимального по количеству набора монет известного номинала, которые можно сдать сдачу $n$ рублей . . . . .	374
15.7 Задача о игре . . . . .	378
15.8 Задача о зайце с запрещенными ступеньками . . . . .	379
15.9 Поиск возрастающей подпоследовательности наибольшей длины . . . . .	382
15.10 Поиск оптимального города, для доставки мусора . . . . .	384
<b>16 Двумерное динамическое программирование</b>	<b>386</b>
16.1 Получение количества двоичных последовательностей длины $n$ , не содержащих две единицы подряд . . . . .	386
16.2 Количество путей из верхнего левого угла в правый нижний на поле размера $n$ на $m$ . . . . .	387
16.3 Минимальная сумма при движении по полю размера $n$ на $m$ . . . . .	391
16.4 Поиск пути с минимальными суммарными перепадами высот . . . . .	392
16.5 Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет с ограничением на количество монет определенного номинала . . . . .	393
16.6 Вычисление минимального количества монет, необходимое для выдачи сдачи $n$ рублей известными номиналами монет с ограничением на количество монет определенного номинала . . . . .	394
16.7 Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет с ограничением на количество монет . . . . .	396
16.8 Задача об оптимальном спуске с горы . . . . .	396
16.9 Задача о счастливых билетах . . . . .	398
16.10 Задача о покупке билетов . . . . .	399
16.11 О игроках . . . . .	401
16.12 О художниках . . . . .	404
16.13 Поиск подпоследовательности наибольшей длины с неотрицательными префиксными суммами . . . . .	407
16.14 Задача о рюкзаке . . . . .	408
<b>17 Продвинутые техники в динамическом программировании при решении олимпиадных задач</b>	<b>411</b>
17.1 Matrix Exponentiation . . . . .	411
17.1.1 Числа Фибоначчи . . . . .	411
17.1.2 Задача о треугольниках . . . . .	412
17.1.3 Расшифровка генома . . . . .	413
17.1.4 Тетраэдр . . . . .	414
<b>18 Лабораторные работы</b>	<b>415</b>
18.1 Лабораторная работа «Стандартный ввод и вывод» . . . . .	416
18.2 Лабораторная работа «Алгоритмы разветвляющейся структуры» . . . . .	421
18.3 Лабораторная работа «Алгоритмы разветвляющейся структуры» . . . . .	432
18.4 Лабораторная работа «Циклы. Введение в тестирование» . . . . .	436
18.5 Лабораторная работа «Циклы» . . . . .	445
18.6 Лабораторная работа «Введение в функции» . . . . .	447

18.7 Лабораторная работа «Обработка одномерных массивов с использованием функций» . . . . .	454
18.8 Лабораторная работа «Бинарный поиск» . . . . .	461
18.9 Лабораторная работа «Рекурсивные функции» . . . . .	465
18.10 Лабораторная работа «Структуры. Функции для работы со структурами» . . . . .	467
18.11 Лабораторная работа «Множества» . . . . .	472
18.12 Лабораторная работа «Реализация структуры данных «Вектор» . . . . .	479
18.13 Лабораторная работа «Работа с многомерными массивами» . . . . .	498
18.14 Лабораторная работа «Работа со строками» . . . . .	510
18.15 Лабораторная работа «Оценка сложности алгоритмов сортировки по времени» . . . . .	532
18.16 Лабораторная работа «Потоки. Ссылки» . . . . .	546
18.16.1 Буферизация в выходных потоках . . . . .	547
18.16.2 Некоторые особенности вывода . . . . .	549
18.16.3 Ввод . . . . .	549
18.16.4 Файловый ввод / вывод . . . . .	552
18.16.5 <i>sstream</i> . . . . .	554
18.16.6 Определение операций ввода / вывода для произвольных типов . . . . .	555
18.16.7 Ссылки . . . . .	558
18.17 Лабораторная работа «Исключения» . . . . .	560
18.18 Лабораторная работа «Решение простейших задач на C++» . . . . .	568
18.19 Лабораторная работа «Векторы» . . . . .	570
18.19.1 Массивы в стиле C <i>std::array</i> . . . . .	570
18.19.2 Векторы <i>std::vector</i> . . . . .	570
18.19.3 Создание вектора . . . . .	571
18.19.4 Ввод и вывод вектора . . . . .	573
18.19.5 Методы для работы с вектором . . . . .	574
18.19.6 <i>std::vector&lt;bool&gt;</i> . . . . .	577
18.20 Лабораторная работа «Стек, очередь, дек» . . . . .	579
18.20.1 Деки <i>std::deque</i> . . . . .	579
18.20.2 Списки <i>std::list</i> , <i>std::forward_list</i> . . . . .	580
18.20.3 Стек <i>std::stack</i> . . . . .	583
18.20.4 Очередь <i>std::queue</i> . . . . .	585
18.20.5 Приоритетные очереди <i>std::priority_queue</i> . . . . .	586
18.21 Лабораторная работа «Структуры в стиле C++» . . . . .	593
18.21.1 Причины появления структур в языках программирования . . . . .	593
18.21.2 Ограничение доступа к полям структуры . . . . .	595
18.21.3 Константность функций-членов . . . . .	601
18.21.4 Конструкторы . . . . .	603
18.21.5 Инициализация полей и структур / классов по умолчанию . . . . .	606
18.21.6 Перегрузка операций ввода и вывода . . . . .	607
18.21.7 Перегрузка арифметических операций . . . . .	608

# List of Listings

1	Решение уравнения $ax + b = 0$ . . . . .	16
2	Обмен двух значений . . . . .	53
3	Вычисление расстояния между двумя точками . . . . .	56
4	Генерация случайных чисел . . . . .	57
5	Решение квадратного уравнения . . . . .	81
6	Поиск максимального значения из двух . . . . .	90
7	Поиск максимального значения из трёх . . . . .	96
8	Упорядочивание двух чисел . . . . .	97
9	Проверка числа на простоту . . . . .	130
10	Алгоритм Джая Кадана . . . . .	137
11	Алгоритм быстрого возведения в степень . . . . .	177
12	Алгоритм Евклида . . . . .	178
13	Вывод чисел в шестнадцатеричной системе счисления (итеративно) . . . . .	179
14	Вывод чисел в шестнадцатеричной системе счисления (рекурсивно) . . . . .	179
15	Проверка на уникальность элементов массива $O(n^2)$ . . . . .	188
16	Вычисление значения факториала (итеративно) . . . . .	195
17	Вычисление значения факториала (рекурсивно) . . . . .	195
18	Вычисление суммы массива с начала последовательности (рекурсивно) . . . . .	199
19	Вычисление суммы массива с конца последовательности (рекурсивно) . . . . .	199
20	Вычисление суммы массива путём разбиения на подзадачи (рекурсивно) . . . . .	200
21	Алгоритм однопроходного удаления (рекурсивно) . . . . .	201
22	Проверка последовательности на палиндром (рекурсивно) . . . . .	201
23	Сортировка выбором (рекурсивный) . . . . .	202
24	Вычисление многочлена в точке / Схема Горнера (рекурсивно) . . . . .	203
25	Бинарный поиск (рекурсивно) . . . . .	204
26	Ввод массива размера $n$ . . . . .	231
27	Вывод массива размера $n$ . . . . .	233
28	Поиск позиции первого отрицательного элемента в массиве . . . . .	235
29	Подсчёт количества элементов, удовлетворяющих условию . . . . .	237
30	Поиск максимального количества подряд идущих элементов, удовлетворяющих условию. . . . .	238
31	Алгоритм однопроходного удаления (итеративный) . . . . .	242
32	Вставка элемента с сохранением порядка элементов . . . . .	243
33	Удаление элемента с сохранением относительного порядка других элементов . . . . .	245
34	Обращение элементов массива . . . . .	246
35	Проверка упорядоченности массива по неубыванию . . . . .	247
36	Добавление элемента в конец массива . . . . .	248
37	Удаление элемента из массива без сохранения относительного порядка других элементов . . . . .	248
38	Ввод элементов массива до первого нуля . . . . .	254

40	Линейный поиск в отсортированном массиве . . . . .	283
39	Линейный поиск в неотсортированном массиве . . . . .	284
41	Поиск позиции первого элемента большего или равного $x$ . . . . .	291
42	Поиск позиции последнего элемента меньшего или равного $x$ . . . . .	292
43	Бинарный поиск по критерию . . . . .	293
44	Тернарный поиск . . . . .	305
45	Обезьяняя сортировка . . . . .	308
46	Сортировка выбором . . . . .	309
47	Сортировка вставками . . . . .	311
48	Пузырьковая сортировка . . . . .	313
49	Сортировка расческой . . . . .	314
50	Сортировка подсчётом . . . . .	318
51	Сортировка слиянием . . . . .	323
52	Операции над кучей . . . . .	331
53	Сортировка кучей . . . . .	332
54	Вычисление чисел Фибоначчи с использованием мемоизации . . . . .	367
55	Вычисление чисел Фибоначчи с использованием табуляции . . . . .	368
56	Задача о количестве способов выдачи сдачи $n$ рублей с определенными номиналами монет . . . . .	373
57	Вычисление минимального количества монет, необходимого для выдачи сдачи $n$ рублей известными номиналами монет . . . . .	375
58	Получение произвольного минимального по количеству набора монет известного номинала, которые можно сдать сдачу $n$ рублей . . . . .	377
59	Количество способов добраться до последней ступеньки с ограничением на занятые ступеньки $O(n * k)$ . . . . .	380
60	Количество способов добраться до последней ступеньки с ограничением на занятые ступеньки $O(n)$ . . . . .	381
61	Поиск оптимального города, для доставки мусора . . . . .	385
62	Получение количества двоичных последовательностей длины $n$ , не содержащих две единицы подряд . . . . .	387
63	Количество путей из верхнего левого угла в правый нижний на поле размера $O(n * m)$ . . . . .	390
64	Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет с ограничением на количество монет определенного номинала . . . . .	395
65	Задача об оптимальном спуске с горы . . . . .	398
66	Задача о счастливых билетах . . . . .	400
67	Задача об игроках . . . . .	403
68	О художниках . . . . .	406
69	Задача о суммах подмножеств . . . . .	410

# Введение

Со времени возникновения ЭВМ технология программирования претерпела существенные изменения. Первоначально программы составлялись в машинных кодах. И создание даже сравнительно простых программ требовало больших затрат времени и усилий. Затем появились алгоритмические языки, значительно повысившие производительность труда программиста. Расширение масштабов решаемых задач потребовало создания особой технологии программирования, которую назвали структурным программированием. Основой этой технологии является структурированный алгоритмический язык Си. Несмотря на появление парадигм, отличных от процедурного, владение основами алгоритмического программирования столь же необходимо программисту-профессионалу, как и раньше.

Настоящее пособие посвящено основам алгоритмизации и программирования на языке Си, что позволит освоить принципы структурного программирования и овладеть базовыми алгоритмами.

Изучение алгоритмизации на языке С охватывает программу дисциплин «Основы программирования» и «Основы алгоритмизации» для специальностей 09.03.01 – Информатика и вычислительная техника и 09.03.04 – Программная инженерия.

## Благодарности

Настоящее пособие посвящается моим учителям, которые вложили много сил и труда, чтобы я смог завершить написанное. Особенно признателен Брусенцевой Валентине Станиславовне, благодаря которой началось (а для кого-то продолжилось) знакомство тысяч студентов с программированием на кафедре программного обеспечения вычислительной техники и автоматизированных систем. Я постарался перенести и её и свой опыт в данный труд.

Не могу не отметить и студентов, которые были осторожны со мной и оставили меня при трезвом уме и памяти. Они так же дополнили своими соображениями материалы, за что им отдельное спасибо. Вы придаёте вдохновения работать дальше. Надеюсь, что это взаимно.

Хочу верить, что наша общая работа позволит сделать мир несколько лучше.

## Ошибки

Несмотря на многочисленные перечитки материала, вполне возможно, что Вы встретите ошибки и некоторые неточности. Был бы рад получить информацию по ним (и по всем прочим конструктивным вопросам) на почту: ISPritchin@gmail.com.

## Программистский фольклор

Как говорят, в каждой шутке есть доля шутки. Вы можете найти отсылки на слова значимых в программировании людей в сносках (мои слова к ним не относятся). Понимание профессионального юмора – важная часть культуры, помогающей лучше проникнуться, а кто такой этот ваш 'программист'. Я использовал в том числе графические работы *GarabatoKid*. [Ссылка на его твиттер](#).

# Глава 1

## Алгоритмы. Структурное программирование

### 1.1 Алгоритм. Свойства алгоритмов

Введём ряд определений:

- **Алгоритм** – конечный набор предписаний (команд), выполнение которых преобразует входные (исходные) данные в выходные (результат) в соответствии с условием задачи.
- **Разработчик** – лицо, которое разрабатывает алгоритм.
- **Исполнитель** – объект, который выполняет шаги алгоритма, преобразуя входные данные в выходные (может быть как одушевлённым, так и неодушевлённым; примеры: человек, ЭВМ).
- **Пользователь** – объект (человек, устройство), который пользуется результатами работы алгоритма.

Исполнитель получает некоторые **входные данные** (исходные данные, подаваемые алгоритму), которые обрабатываются согласно алгоритму, и получаются **выходные данные** (результат работы алгоритма):



На начальном этапе изучения алгоритмизации и программирования важно понимать, что являются входными данными, выходными данными и алгоритмом.

#### Пример 1: Сложение двух чисел

Для того, чтобы сложить два числа, нужно иметь непосредственно сами числа (входные данные). Сумма является результатом операции сложения чисел (выходные данные). А набор инструкций для преобразования двух слагаемых в сумму и является алгоритмом.

### Пример 2: Приготовление блюда

Данный процесс нельзя выполнить, если нет ингредиентов (входные данные). Само блюдо является результатом готовки (выходные данные). Инструкции кулинарной книги - алгоритмом.

### Пример 3: Сборка мебели

Входными данными являются компоненты изделия. Собранная мебель является выходными данными. Инструкции по сборке - алгоритмом.

#### 1.1.1 Свойства алгоритма

Алгоритмы обладают следующими **свойствами**:

- **Детерминированность** – ориентированность на определенного исполнителя, исключающая неоднозначность понимания.  
Пример нарушенного свойства: 'сходи туда – не знаю куда, принеси то – не знаю что'.
- **Массовость** – пригодность для решения задач определенного класса при любых допустимых значениях исходных данных.
- **Дискретность** – пошаговый характер получения результата.
- **Конечность / результативность** – свойство алгоритма получать результат за конечное время<sup>1</sup>.

#### 1.1.2 Способы записи алгоритмов

Существуют следующие способы записи алгоритмов:

- Словесно-формульный (примеры: рецепты в кулинарной книге, инструкции сборки).
- Графический (блок-схемы, структурограммы).
- Псевдокод (компактный, зачастую неформальный язык описания алгоритмов).
- Программа на языке программирования.

Выбор способа записи алгоритма зависит от цели его описания. Алгоритмы могут быть описаны с разной степенью детализации и формализации.

#### 1.1.3 Словесно-формульная запись

Алгоритм в словесно-формульном виде представляет собой упорядоченную последовательность команд, описанных обычным языком с возможным использованием математической нотации. Допускается следующий вариант: нумерация команд начинается с 1, и по крайней мере один раз в описании алгоритма должна быть команда «Конец».

---

<sup>1</sup>Если бы алгоритм выполнялся бесконечно, он не имел смысла.

**Пример**

Описание алгоритма решения уравнения вида  $ax = b$ .

1. Ввод коэффициентов уравнения  $a$  и  $b$ .
2. Если  $a \neq 0$ , перейти к п.8.
3. Если  $b \neq 0$ , перейти к п.6.
4. Вывод: «Любое  $x$  является корнем уравнения.».
5. Перейти к п. 10.
6. Вывод: «Уравнение не имеет корней.».
7. Перейти к п. 10.
8.  $x := b/a$ .
9. Вывод  $x$ .
10. Конец.

#### **1.1.4 Блок-схемы (графический способ)**

Графическая форма записи алгоритма более наглядна, позволяет отчетливо представить все логические связи между частями алгоритма.

Блок-схема алгоритма представляет собой набор геометрических фигур (блоков), соединенных линиями или линиями со стрелками, для указания направления перехода от блока к блоку. Движение от блока к блоку сверху вниз или слева направо считается стандартным. В этом случае стрелки можно не указывать. Если же направление отлично от стандартного, то стрелки обязательны<sup>2</sup>. Необходимая для выполнения очередного действия информация помещается в блок в виде текста или математического обозначения. Перечень блоков, их форма и отображаемые функции установлены ГОСТ 19.701-90 ЕСПД. В таблице 1.1 на странице 15 приведены основные блоки.

**Пример**

Блок-схема алгоритма решения уравнения вида  $ax = b$  представлена на рисунке 1.1 (страница 16).

<sup>2</sup>В примерах данного пособия стрелки будут отсутствовать

Таблица 1.1: Блоки, применяемые при создании блок-схем

Форма	Название	Назначение
	Терминатор	Используется для начальных и конечных блоков основных и вспомогательных алгоритмов
	Процесс	Отображает процесс обработки данных любого вида.
	Предопределённый процесс	Отображает процесс, определенный ранее.
	Решение	Используется для отображения бинарного или множественного ветвления.
	Данные	Используется для ввода и вывода данных. Носитель данных не определен и должен быть указан.
	Модификатор	Используется как заголовок цикла с фиксированным числом повторений.
	Соединитель	Отображает выход из части схемы и вход в другую часть этой схемы. Соответствующие соединители помечаются одним и тем же уникальным обозначением.
	Комментарий	Используется для пояснительных записей

### 1.1.5 Псевдокод

Псевдокод занимает промежуточное место между естественным языком и языком программирования. Он позволяет описывать логику программы на естественном языке, но включать типовые конструкции языка программирования, не заботясь о синтаксических тонкостях.

### 1.1.6 Программа

Программа является способом записи алгоритма при помощи **языка программирования** – строгого набора правил, символов и конструкций, которые позволяют в формульно-словесной форме описывать алгоритмы для обработки данных на ЭВМ.

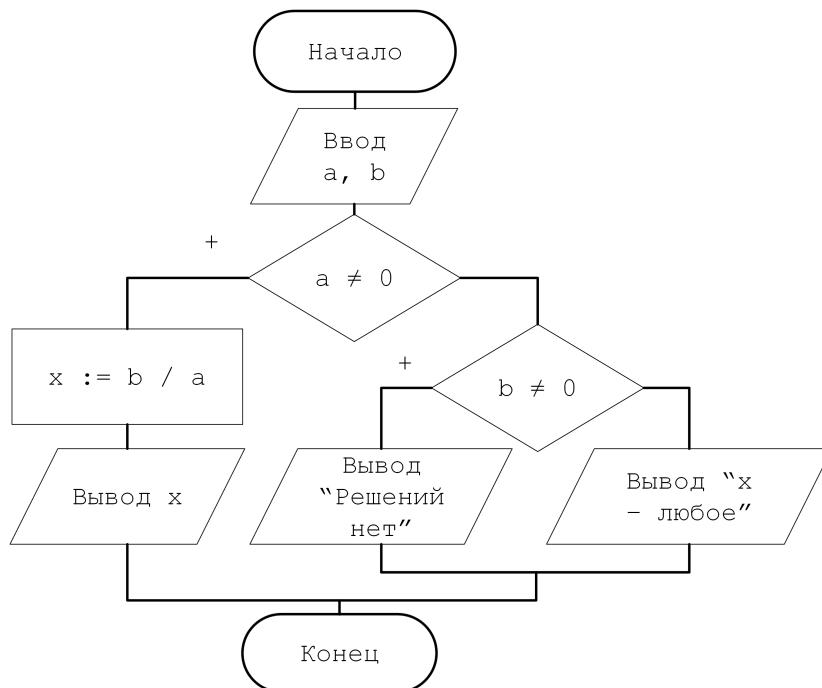


Рис. 1.1: Блок-схема алгоритма решения уравнения вида  $ax = b$

---

### Листинг 1 Решение уравнения $ax + b = 0$

---

```

#include <stdio.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    float a, b;
    scanf("%f %f", &a, &b);

    if (a) {
        float x = b / a;
        printf("%f", x);
    } else {
        if (b)
            printf("решений нет");
        else
            printf("x - любое");
    }

    return 0;
}
  
```

---

Пока что не будем вдаваться в процесс написания данной программы, а лишь отразим тот факт, что перевод из блок-схемы в код является относительно тривиальной задачей. На начальных этапах особое внимание надо уделить умению написания блок-схем.

## История развития языков программирования

Физические принципы работы электронных устройств ЭВМ таковы, что компьютер может воспринимать команды, состоящие только из единиц и нулей — последовательность перепада напряжения, то есть машинный код. На начальной стадии развития ЭВМ человеку было необходимо составлять программы на языке, понятном компьютеру, в машинных кодах. Каждая команда состояла из кода операций и адресов операндов, выраженных в виде различных сочетаний единиц и нулей. Итак, любая программа для процессора выглядела на то время как последовательность единиц и нулей.

Программа «Hello, world!» для процессора архитектуры x86 (ОС MS DOS, вывод при помощи BIOS прерывания int 10h) выглядит следующим образом (в шестнадцатеричном представлении): BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21. Не особо важно, что там происходит, но я с уверенностью могу сказать, что строка "Hello, world!" закодирована в последовательности 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21.

Как показала в дальнейшем практика общения с компьютером, такой язык громоздок и неудобен. При пользовании им легко допустить ошибку, записав не в той последовательности 1 или 0. Кроме того, при программировании в машинных кодах надо хорошо знать внутреннюю структуру ЭВМ, принцип работы каждого блока. И самое плохое в таком языке, что программы на данном языке — очень длинные последовательности единиц и нулей являются машиннозависимыми, то есть для каждой ЭВМ необходимо было составлять свою программу, а также программирование в машинных кодах требует от программиста много времени, труда и повышенного внимания.

Довольно скоро стало понятно, что процесс формирования машинного кода можно автоматизировать. Уже в 1950 году для записи программ начали применять меморический язык — язык *assembly*. Язык ассемблера позволил представить машинный код в более удобной для человека форме: для обозначения команд и объектов, над которыми эти команды выполняются, вместо двоичных кодов использовались буквы или сокращенные слова, которые отражали суть команды.

Ассемблер — язык программирования низкого уровня<sup>3</sup>. В данном случае «низкий уровень» не значит «плохой». Имеется в виду, что операторы языка близки к машинному коду и ориентированы на конкретные команды процессора. Появление языка ассемблера значительно облегчило жизнь программистов, так как теперь вместо рябящих в глазах нулей и единиц, они могли писать программу командами, состоящими из символов, приближенных к обычному языку. Для того времени этот язык был новшеством и пользовался популярностью, так как позволял писать программы небольшого размера.

Середина 50-х годов характеризуется стремительным прогрессом в области программирования. Роль программирования в машинных кодах стала уменьшаться. Начали появляться языки программирования высокого уровня, выступающие в роли посредника между машинами и программистами. Эти языки не были привязаны к определенному типу ЭВМ<sup>4</sup>. Для каждого из них были разработаны собственные компиляторы, осуществляющие процесс компиляции. **Компиляция** — трансляция (перевод) программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке. Одним из первых языков (до сих

---

<sup>3</sup> Язык программирования низкого уровня — язык программирования, который ориентирован на конкретный тип процессора и учитывает его особенности.

<sup>4</sup> Такие языки называются машиннозависимыми.

пор используемым сегодня) является Фортран, который был создан в период с 1954 по 1957 в корпорации IBM. Он предназначался для научных и технических расчетов. Название Fortran является сокращением от FORmula TRANslator (переводчик формул). Со временем появились и другие языки высокого уровня (ALGOL, LISP, COBOL, Pascal, C и др.).

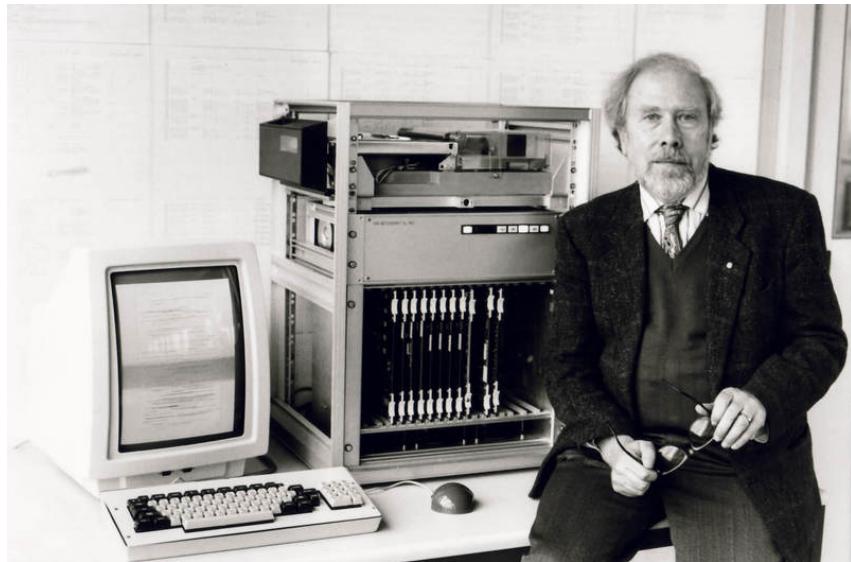


Рис. 1.2: Никлаус Вирт – автор языка Паскаль

Достоинства языков программирования высокого уровня:

- Набор операций, допустимых для использования, не зависит от набора машинных операций, а выбирается из соображений удобства формулирования алгоритмов решения задач определенного класса.
- Конструкции операторов задаются в удобном для человека виде.
- Поддерживается широкий набор типов данных.

Такие языки как Паскаль и С поддерживают парадигму структурного программирования, в основе которой лежит представление программы в виде иерархической структуры блоков. Имеются и другие парадигмы: функциональное, логическое, обобщенное, объектно-ориентированное программирование и т. д. Не будем подробно останавливаться на данных концепциях, а только пробежимся по некоторым языкам:

**1972:** С был разработан и реализован в начале 70-х гг. ХХ в Bell Telephone Laboratories Деннисом Ритчи<sup>5</sup> во время совместной работы с Кеном Томпсоном над операционной системой UNIX. Он был задуман



Рис. 1.3: Деннис Ритчи – разработчик языка С

<sup>5</sup>На одной из олимпиад мне попался вопрос, в котором нужно было по портрету определить деятелей в области информатики. Портрет одного из них оставлен здесь (чтобы вы были готовы к этому).

для решения задач системного программирования, и примерно 80% операционной системы было написано на нём. Основной целью создания этого языка было максимальное приближение программиста к используемым аппаратным средствам с сохранением всех преимуществ языка высокого уровня. Это должно было обеспечить с одной стороны мобильность программного обеспечения, а с другой – его эффективность<sup>6</sup>.

Многие из ведущих в настоящее время языков являются производными от С, включая C++, C#, Java, JavaScript, Perl, PHP и Python<sup>7</sup>. Он также использовался / до сих пор используется крупными компаниями.

**1980: Ada** – язык программирования, созданный в 1979–1980 годах в ходе проекта Министерства обороны США с целью разработать единый язык программирования для встроенных систем (то есть систем управления автоматизированными комплексами, функционирующими в реальном времени). Имелись в виду прежде всего бортовые системы управления военными объектами (кораблями, самолётами, танками, ракетами, снарядами и т. п.).



Рис. 1.4: Бъярн Страуструп – автор языка C++

разработан Гвидо Ван Россумом. Это универсальный язык программирования высокого уровня, созданный для поддержки различных стилей программирования и приятный в использовании. Python по сей день является одним из самых популярных языков программирования в мире, который используют такие компании, как Google, Yahoo и Spotify.

**1995: Java** – это универсальный язык высокого уровня, созданный Джеймсом Гослингом для проекта интерактивного телевидения. Он обладает кроссплатформенной функциональностью и неизменно входит в число самых популярных языков программирования в мире. Java можно найти везде, от компьютеров до смартфонов и парковочных счетчиков.

**1995: JavaScript** был создан Брэнданом Эйхом, этот язык в основном используется для динамической веб-разработки, документов PDF, веб-браузеров и виджетов рабочего стола<sup>9</sup>. Почти каждый крупный веб-сайт использует JavaScript. Gmail, Adobe Photoshop и Mozilla Firefox включают несколько хорошо известных примеров.

<sup>6</sup>Ритчи всегда был за кулисами, его имя не было известно обычательям, но, если заглянуть в компьютер с микроскопом, следы его работы встречаются повсюду (Пол Э. Черуцци).

<sup>7</sup>Говорят, что C++ и Java растут быстрее, чем простой С, но я уверен, что С ещё поживёт. (Деннис Ритчи)

<sup>8</sup>При помощи С вы легко можете выстрелить себе в ногу. При помощи C++ это сделать сложнее, но если это произойдёт, вам оторвёт всю ногу целиком. (Бъярн Страуструп)

<sup>9</sup>Java относится к JavaScript так же, как Сомали к Сомали.

**1983: C++**. Бъярн Страуструп модифицировал язык С в Bell Labs, C++ - это расширение С с такими улучшениями, как классы, виртуальные функции и шаблоны. Он был включен в 10 лучших языков программирования с 1986 года и получил статус Зала славы в 2003 году. C ++ используется в MS Office, Adobe Photoshop, игровых движках и другом высокопроизводительном программном обеспечении<sup>8</sup>.

**1991: Python**. Названный в честь британской комедийной труппы «Монти Пайтон», Python был

**2000: C#.** Разработанный в Microsoft с надеждой на объединение вычислительных возможностей C++ с простотой Visual Basic, C# основан на C++ и имеет много общего с Java. Этот язык используется почти во всех продуктах Microsoft и используется в основном при разработке настольных приложений.

**2009: Golang (Go)** был разработан Google для решения проблем, возникающих из-за больших программных систем. Благодаря своей простотой и современной структуре Go завоевал популярность среди некоторых крупнейших технологических компаний по всему миру, таких как Google, Uber, Twitch и Dropbox.

**2014: Swift.** Разработанный Apple в качестве замены C, C++ и Objective-C, Swift был разработан с целью быть проще, чем вышеупомянутые языки, и оставлять меньше места для ошибок. Универсальность Swift означает, что его можно использовать для настольных, мобильных и облачных приложений. Ведущее языковое приложение Duolingo запустило новое приложение, написанное на Swift.

Несмотря на то, что было создано большое множество языков, нет предпосылок к созданию универсального языка<sup>10</sup>, который подходил бы для решения абсолютно всех задач. Каждый инструмент имеет свою нишу. Когда я вижу, что студенты спорят о том, какой язык круче, представляю, как столяры обсуждают, что лучше для решения всех задач: молоток, пила или киянка.

Говоря о любом языке, можно выделить синтаксис и семантику. **Синтаксис** языка – это набор правил, определяющий допустимые конструкции языка. **Семантика** языка задаёт правила трактования допустимых конструкций.

## Язык программирования С

В данном пособии будет описан язык С – компилируемый статически типизированный язык программирования общего назначения<sup>11</sup>. Первоначально был разработан для реализации операционной системы UNIX, но вследствии был перенесён на множество других платформ. Согласно дизайну языка, его конструкции близко сопоставляются типичным машинным инструкциям, благодаря чему он нашёл применение в проектах, для которых был свойственен язык ассемблера, в том числе как в операционных системах, так и в различном прикладном программном обеспечении для множества устройств — от суперкомпьютеров до встраиваемых систем.

Основные особенности С<sup>12</sup>:

- небольшой язык, его возможности заключены в библиотеках;
- язык ориентирован на процедурное программирование;
- система типов, предохраняющая от бессмысленных операций;

<sup>10</sup>Любой, кто приходит к вам и говорит, что у него есть совершенный язык — либо наивен, либо торгаш (Страуструп).

<sup>11</sup>Языки программирования общего назначения - языки, которые предназначены для написания программного обеспечения в самых различных прикладных областях

<sup>12</sup>Существует два вида языков программирования: одни – все ругают, другими не пользуются.



Рис. 1.5: Гвидо Ван Россум – велиcodушный по-жизненный диктатор до 2018 года – разработчик языка *Python*

- доступ к памяти осуществляется с помощью указателей;
- компилятор способен генерировать компактный и эффективный машинный код.

В то же время С:

- не имеет средства автоматического управления памятью;
- не для первоначального обучения.

Простейшая программа на С<sup>13</sup>:

---

```
int main() {
    return 0;
}
```

---

состоит из функции `main`. Данная программа ничего не получает на вход и ничего не выводит. Она завершается, возвращая значение `0` в операционную систему (которая производит запуск приложения). Код завершения `0` означает, что программа успешно закончила выполнение. Другие коды выхода могут указывать на проблему<sup>14</sup> Наличие оператора возврата `return` не является обязательным; код выхода по умолчанию равен нулю:

---

```
// программа без явного возврата
// так делать можно, но не рекомендуется
int main() {

}
```

---

В примере выше вы могли заметить **комментарии**, которые являются удобочитаемыми аннотациями, помещаемые в исходный код программы. `//` – обозначает начало одностороннего комментария (весь текст в данной строке начиная с `//` будет проигнорирован компилятором). Можно использовать многострочные комментарии `/* */` (будет проигнорирован весь текст между парой `/*` и `*/`). Примеры:

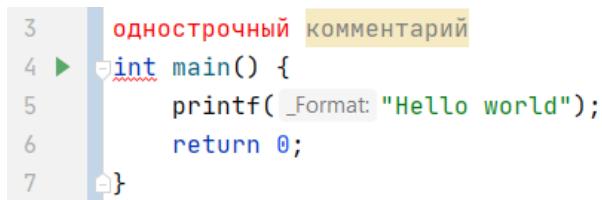
---

```
// односторочный комментарий

/* многострочный
   комментарий */
```

---

Если забыть о существовании такого инструмента и начать 'лепить' пометки без их использования – и компилятор, и среда 'порадуют' вас сообщениями об ошибках:



3       однострочный комментарий  
 4 ► int main() {  
 5           printf(\_Format: "Hello world");  
 6           return 0;  
 7       }

<sup>13</sup>Настоятельно рекомендуя набирать примеры из пособия, проверять какие-то идеи. Автор языка С говорил: "Единственный способ выучить новый язык программирования – это писать на нем программы.". Не волнуйтесь, если что-то не работает. Если бы всё работало, вас бы уволили.

<sup>14</sup>Одной из основных причин падения Римской империи было то, что, поскольку в их арифметике не было нуля, они никак не могли сообщать об удачном завершении в своих программах на С. (Роберт Фёрт)

*Hello world*<sup>15</sup> на языке программирования С:

---

```
#include <stdio.h>

int main() {
    printf("Hello world");

    return 0;
}
```

---

В первой строке подключается библиотека `stdio` для стандартного ввода и вывода. Для подключения библиотеки используется ключевое слово `#include`, за которым следует имя библиотеки в `<>` или `" "` кавычках. Первый вариант говорит препроцессору о том, что библиотека располагается в некоторой стандартной директории. А если использованы двойные кавычки – библиотека считается пользовательской, и поиск будет осуществлен и в стандартной директории и в директории проекта.

Так как библиотека ввода/вывода была подключена, нам доступны её функции. Речь о них пойдёт позже. Пока что нам достаточно знать, что в строке 4 посредством функции `printf` будет выведено сообщение *"Hello world"* на экран.

Этап компиляции выполняется следующей цепочкой инструментов:

- Препроцессор. Исходный текст частично обрабатывается — производятся:
  - Замена комментариев пустыми строками
  - Текстовое включение файлов — `#include`
  - Макроподстановки — `#define`
  - Обработка директив условной компиляции — `#if`, `#ifdef`, `#elif`, `#else`, `#endif`
- Компилятор. Выполняет следующие действия:
  - Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.
  - Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора.
  - Семантический анализ. На этой фазе дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их объявлениям, типам данных, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.

---

<sup>15</sup>Хотя небольшие проверочные примеры использовались с тех самых пор, как появились компьютеры, традиция использования фразы «Hello, world!» в качестве тестового сообщения была введена в книге «Язык программирования Си» Брайана Кернигана и Денниса Ритчи, опубликованной в 1978 году.

- Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может происходить на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.
- Генерация кода. Из промежуточного представления порождается код на целевом машинно-ориентированном языке.
- Компоновщик (редактор связей) – инструментальная программа, которая производит компоновку («линковку», «сборку»): принимает на вход один или несколько объектных модулей и собирает из них исполняемый или библиотечный файл-модуль.

Этапы компиляции представлены на рисунке 1.6.

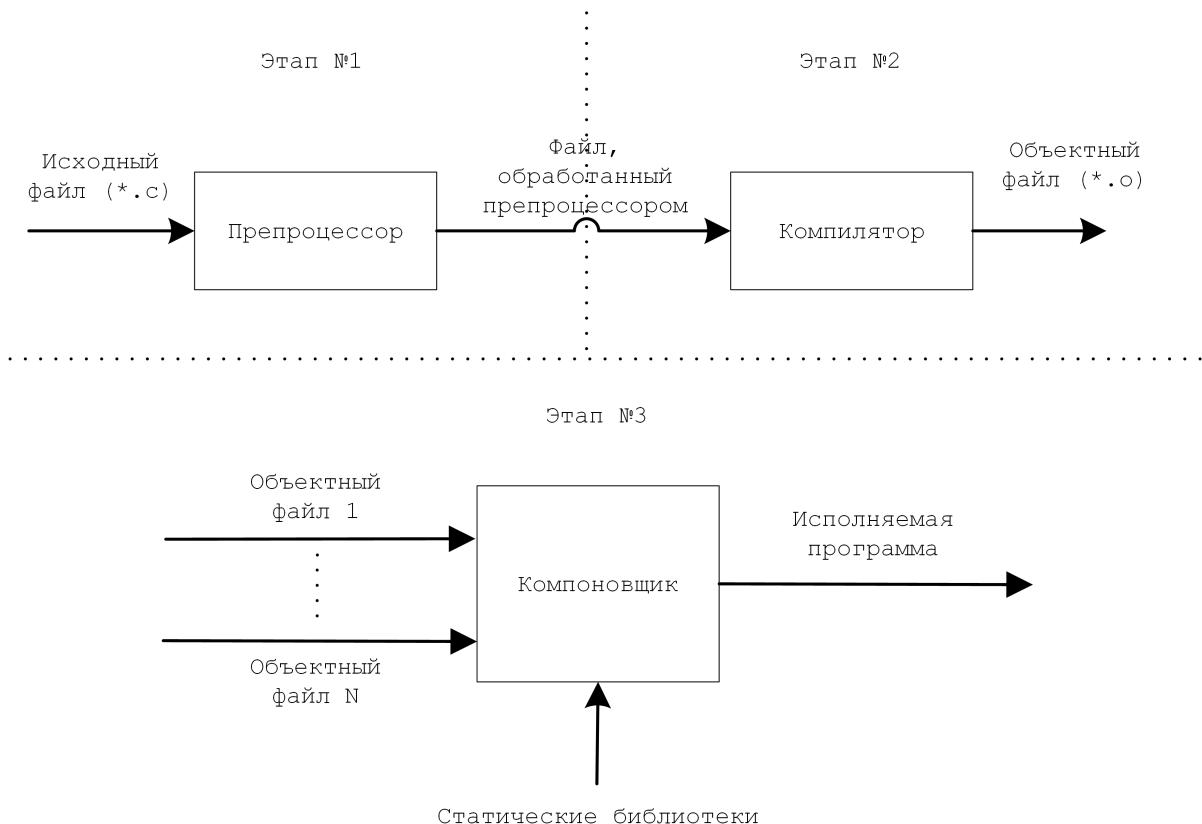


Рис. 1.6: Этапы компиляции

## 1.2 Структурное программирование

Одной из методологий к написанию программ является структурное программирование, основными принципами которого являются:

1. **Разработка алгоритма «сверху вниз» (метод пошаговой детализации).** Начиная со спецификации, полученной в результате анализа задачи, выделяют небольшое число достаточно самостоятельных подзадач, описывают спецификации для каждой и описывают алгоритм в терминах выделенных подзадач. Это будет первый шаг детализации. С каждой из выделенных подзадач поступают так же (второй шаг детализации) и т.д. Таким образом получается

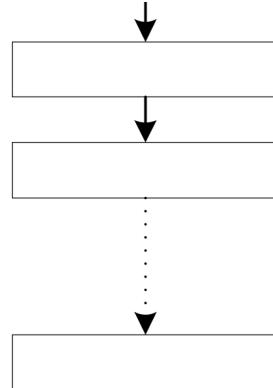
последовательность все более детальных спецификаций, приближающаяся к окончательной версии программы.

2. **Модульность.** Метод пошаговой детализации дает возможность разбить алгоритм на части (модули – подпрограммы), каждая из которых решает самостоятельную подзадачу. Размеры модулей должны быть небольшими, а инструкции, входящие в состав модуля, должны давать исчерпывающее представление о действиях, выполняемых модулем.

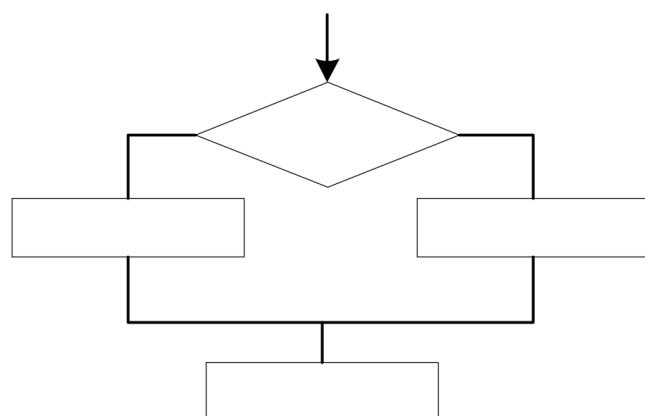
Каждый модуль имеет имя. Связи по управлению между модулями осуществляются посредством обращений к ним по имени, а обмен информацией - через параметры и глобальные переменные.

3. Каждый модуль должен иметь один вход и один выход. Это позволяет упростить стыковку модулей в сложной программе.
4. Логика алгоритма должна опираться на небольшое число достаточно простых базовых управляющих структур:

- Следование



- Развилка



- Цикл

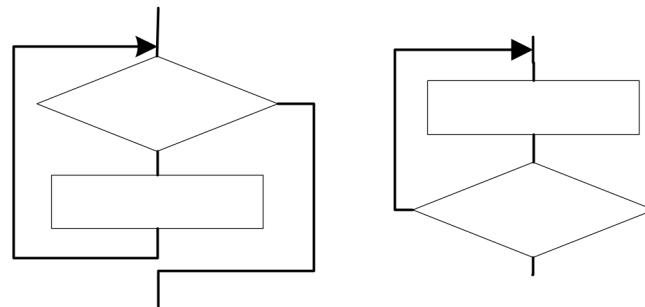
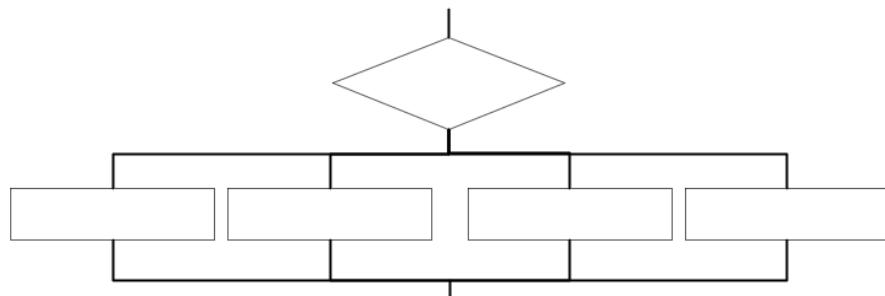


Рис. 1.7: Цикл с предусловием и цикл с постусловием

- Выбор из нескольких альтернатив



Фундаментом структурного программирования является **теорема о структурировании**. Она утверждает, что как бы ни сложна была задача, схема алгоритма может быть представлена с использованием ограниченного числа элементарных управляющих структур.

**Теорема о полноте:** базовые элементарные структуры: следование, разветвление и цикл с предусловием обладают функциональной полнотой, то есть любой алгоритм может быть реализован в виде композиции этих конструкций.

### 1.3 Жизненный цикл программного обеспечения

**Жизненный цикл программного обеспечения** — это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

Существует несколько моделей, описывающих процесс создания программного обеспечения. Мы остановимся на каскадной модели. Её основная особенность — каждый последующий шаг начинается после полного завершения выполнения предыдущего шага.

Выделяют следующие этапы:

- Инициализация (появление идеи о создании ПО),
- Анализ требований<sup>16</sup>,
- Проектирование<sup>17</sup>,

<sup>16</sup> Требования — это перечень того, что необходимо продукту для достижения текущей проектной цели.

<sup>17</sup> Проектирование — процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или её части.

- Кодирование (программирование),
- Тестирование<sup>18</sup> и отладка<sup>19</sup>,
- Эксплуатация и сопровождение<sup>20</sup>.



Как и любая модель она обладает своими достоинствами и недостатками. Каскадная модель используется при разработке проектов с четкими, неизменяемыми в течение жизненного цикла требованиями, понятными реализацией и техническими методиками. Её часто используют при выполнении больших проектов, в которых задействовано несколько больших команд разработчиков. В качестве примера часто указывают ПО для авиации, энергетической промышленности.

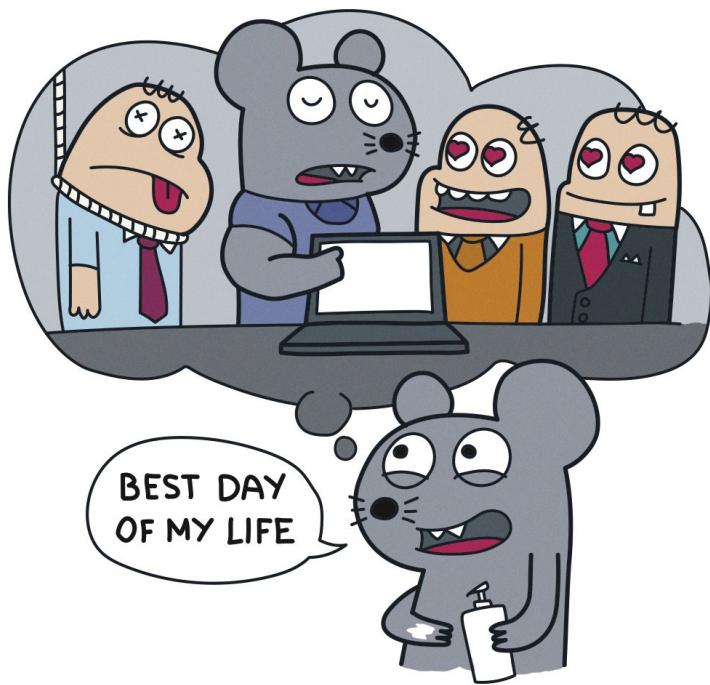
## Резюме

- Алгоритм – это конечный набор предписаний (команд), выполнение которых преобразует входные (исходные) данные в выходные (результат) в соответствии с условием задачи.
- Алгоритмы обладают следующими свойствами: детерминированность, массовость, дискретность, результативность.
- Исполнитель получает входные данные, которые обрабатываются согласно алгоритму, и получаются выходные данные.
- Существует несколько способов записи алгоритмов, выбор которого зависит от цели его описания: словесно-формульный (последовательность команд), графический (блок-схемы), псевдокод (неформальный язык описания алгоритмов),

<sup>18</sup>Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

<sup>19</sup>Отладка - процесс поиска, анализа и устранения причин отказов в программном обеспечении.

<sup>20</sup>Сопровождение - модификация программного продукта после его поставки с целью исправления дефектов, улучшения производительности или других характеристик или для адаптации продукта к изменившемуся окружению.



программа на языке программирования. Таким образом, программа – это способ записи алгоритма при помощи языка программирования.

- На начальной стадии развития ЭВМ человеку было необходимо составлять программы на языке, понятном компьютеру, в машинных кодах. К началу 1950-ых для записи программ начали применять мнемонический язык – ассемблер. Он значительно облегчил жизнь программистов, так как теперь они могли писать программу командами, приближенными к обычному языку.
- В середине 50-х роль программирования в машинных кодах стала уменьшаться, начали появляться языки программирования высокого уровня, не привязанные к определенному типу ЭВМ.
- В настоящее время несмотря на то, что было создано большое множество языков, нет предпосылок к созданию универсального языка, который подходил бы для решения абсолютно всех задач.
- Одной из методологий к написанию программ является структурное программирование, фундаментом которого является теорема о структурировании. Она утверждает, что как бы ни сложна была задача, схема алгоритма может быть представлена с использованием ограниченного числа элементарных управляющих структур.

## Термины и определения

- **Алгоритм** – это конечный набор предписаний (команд), выполнение которых преобразует входные (исходные) данные в выходные (результат) в соответствии с условием задачи.
- **Блок-схема** – это набор геометрических фигур (блоков), соединенных линиями или линиями со стрелками, для указания направления перехода от блока к блоку.
- **Детерминированность** – это ориентированность на определенного исполнителя, исключающая неоднозначность понимания.
- **Дискретность** – это пошаговый характер получения результата.
- **Жизненный цикл программного обеспечения** – это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации.
- **Исполнитель** – это объект, который выполняет шаги алгоритма, преобразуя входные данные в выходные.
- **Компиляция** – это трансляция(перевод) программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке.
- **Компоновщик (редактор связей)** – это инструментальная программа, которая производит компоновку: принимает на вход один или несколько объектных модулей и собирает из них исполняемый или библиотечный файл-модуль.
- **Конечность / результативность** – свойство алгоритма получать результат за конечное время.
- **Массовость** – это пригодность для решения задач определенного класса при любых допустимых значениях исходных данных.
- **Отладка** - процесс поиска, анализа и устранения причин отказов в программном обеспечении.
- **Пользователь** – это объект (человек, устройство), который пользуется результатами работы алгоритма.
- **Препроцессор** – это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора).
- **Проектирование** – процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или её части.
- **Программа** – это способ записи алгоритма при помощи языка программирования.
- **Разработчик** – это лицо, которое разрабатывает алгоритм.

- **Сопровождение** - модификация программного продукта после его поставки с целью исправления дефектов, улучшения производительности или других характеристик или для адаптации продукта к изменившемуся окружению.
- **Тестирование программного обеспечения** — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.
- **Требования** – это перечень того, что необходимо продукту для достижения текущей проектной цели.
- **Языки программирования общего назначения** — это языки, которые предназначены для написания программного обеспечения в самых различных прикладных областях.
- **Язык программирования** – это строгий набор правил, символов и конструкций, который позволяет в формульно-словесной форме описывать алгоритмы для обработки данных на ЭВМ.
- **Язык программирования низкого уровня** – это язык программирования, который ориентирован на конкретный тип процессора и учитывает его особенности.

## Контрольные вопросы

1. Дайте определение термину 'алгоритм'. Перечислите свойства алгоритмов с их определениями.
2. Дайте определение терминам 'разработчик', 'исполнитель', 'пользователь'.
3. Что является входными и выходными данными?
4. Перечислите способы описания алгоритмов. Приведите примеры каждого способа. От чего зависит выбор способа записи алгоритма?
5. Дайте определение терминам 'программа', 'язык программирования'.
6. Что задают 'синтаксис' и 'семантика'?
7. Перечислите основные достоинства и недостатки языка С.
8. Препроцессор, компилятор и компоновщик. Этапы компиляции.
9. При использовании директивы препроцессора `#include` в чем заключается разница между `< >` и `" "`.
10. Перечислите принципы структурного программирования. Перечислите базовые управляющие конструкции.
11. Дайте определение термину 'жизненный цикл программного обеспечения'. Каскадная модель. Её особенности и этапы.

# Глава 2

## Типы данных. Переменные

Напомним, что алгоритм должен преобразовывать входные данные в выходные. В данной главе остановимся на представлении данных в компьютере. Информация в нём представлена последовательностью бит. Каждый из которых может хранить 0 или 1<sup>1</sup>. В процессе написания программ активно оперируют объектами, которые могут иметь тот или иной **тип данных**, определяющий

- Множество допустимых значений, принимаемых переменными данного типа.
- Множество допустимых операций, над переменными данного типа.

Каждая константа<sup>2</sup>, переменная<sup>3</sup>, результат вычисления выражения относится к определенному типу данных. Каждая операция требует operandов определенного типа и формирует результат, тип которого определяется правилами языка. Каждая функция требует аргументов определенного типа и возвращает результат определенного типа.

В языке С стандарта C90 определены следующие базовые типы<sup>4</sup>:

- `char`, `int` – целочисленные типы;
- `float`, `double` – вещественные типы;
- `void` – пустой тип.

С помощью модификаторов `signed`, `unsigned`, `short`, `long` типы могут быть модифицированы. Модификаторы `short`, `long` изменяют размер типа `int`. Модификаторы `signed` (знаковый), `unsigned` (беззнаковый) определяют интерпретацию старшего бита для целочисленных типов.

Далее будут описаны типы данных языка С. Для каждого типа покажем, как выглядит литерал<sup>5</sup> соответствующего типа.

<sup>1</sup>На свете существует 10 типов людей: те, кто понимает двоичную систему счисления, и те, кто не понимает.

<sup>2</sup>**Константа** – это программный объект, не изменяющий своего значения.

<sup>3</sup>**Переменная** – программный объект, изменяющий своё значение.

<sup>4</sup>Именованные константы для всех типов вместе с другими характеристиками машины и компилятора содержатся в стандартных заголовочных файлах `<limits.h>` и `<float.h>`. Стандарты C99 и C11 добавили ещё несколько типов, но они не будут использованы в пособии.

<sup>5</sup>**Литерал** – это константа, тип и значение которой определяется по ее виду.

## 2.1 Целочисленные типы данных

Целочисленные типы данных необходимы для представления целых чисел (чисел, которые не содержат дробной части). В памяти ЭВМ они представлены последовательностью из  $n$ -бит, где старший бит может отводиться как под знак, так и под значение. Рассмотрим обе ситуации.

### 2.1.1 *char / unsigned char*

Под целочисленный тип `char` отводится один байт (8 бит), и он часто используется для хранения символов или небольших целых. В данном типе умещается 256 значений, что изначально хватало для представления наиболее часто употребляемых символов. Когда с клавиатуры считывается символ, то в память компьютера записывается не его графическое представление, а код, соответствующий введенному символу. Соответствие кода и графического отображения имеются в *ASCII*<sup>6</sup>-таблице (таблица 2.1).

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	:	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	-	127	7F	177	

Рис. 2.1: Таблица *ASCII*

Например, при вводе символа '*A*' будет записан его код, равный 65.

**Символьный литерал** – целое (переменная типа `int`), записанное в виде символа, обрамленного одиночными кавычками<sup>7</sup>. Для записи некоторых символов используются, так называемые, *escape*-последовательности. Приведём некоторые из них:

- '＼n' – символ перехода на новую строку;

<sup>6</sup>ASCII (*American standard code for information interchange*) – название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Таблица была разработана и стандартизована в США, в 1963 году.

<sup>7</sup>Имеется небольшое отличие между языками программирования С/C++. В первом же литерал относится к типу `int`, во втором - к типу `char`.

- `'\t'` – символ табуляции;
- `'\b'` – символ удаления предшествующего символа;
- `'\0'` – символ признака конца строки (символ с нулевым значением)<sup>8</sup>;
- `'\'` – апостроф;
- `'\"'` – кавычки;
- `'\\'` – обратный слеш;
- `'%%` – знак процента;
- `'\?'` – знак вопроса;
- `'\ooo'` – восьмеричное значение;
- `'\hhh'` – шестнадцатеричное значение.

Мы можем объявить переменные данного типа и присвоить им символьный литерал или целочисленный литерал<sup>9</sup>:

---

```
char a = 13;
char b = 'a'; // b = 97, так как код буквы 'a' в таблице ASCII равняется 97.
char c = ' '; // c = 32 (код пробела)
char d = 0x41; // d = 65 (целочисленный литерал в 16-ричной СС)
char e = 0101; // e = 65 (целочисленный литерал в 8-ричной СС)
```

---

В листинге<sup>10</sup> выше мы впервые встречаемся с **операцией**<sup>11</sup> **присваивания** (`=`). Она работает следующим образом: вычисляется значение выражения справа от знака присваивания и результат сохраняется в объекте, находящемся слева<sup>12</sup>. Когда встречаетесь с данной операцией, не думайте, что строка 1 гласит: "переменная *a* равна 13". Правильнее прочитать: "присвоить переменной *a* значение 13". Вполне естественной смотрится конструкция:

---

```
x = x + 1 // увеличить значение x на 1
```

---

когда уравнение

$$x = x + 1$$

не имеет решений ни при каких *x*.

Возвращаемся к типу `char`. Если все биты отведены под значение, данная переменная имеет тип `unsigned char`:

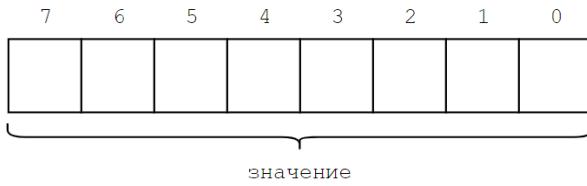
<sup>8</sup>Вместо просто 0 часто используют запись '`\0`', чтобы подчеркнуть символьную природу выражения, хотя и в том и другом случае запись обозначает нуль.

<sup>9</sup>Если вы явно хотите указать, что значение интерпретируется как код символа, присваивайте символьный литерал, вместо его кода

<sup>10</sup>Листинг – бумажная распечатка текста компьютерной программы или её части

<sup>11</sup>Операция – конструкция в языках программирования; специальный способ записи некоторых действий. Операнд – то, чем оперируют операции.

<sup>12</sup>Объект данных – область хранения данных, которая может применяться для удержания значений. Термин *l* – *value* в С применяется для обозначения любого имени или выражения, идентифицирующего конкретный объект данных. Термин *r* – *value* относится к величинам, которые могут быть присвоены модифицируемым *l*-значениям, но которые сами не являются *l*-значениями.



и при его помощи могут быть представлены любые числа из диапазона:

$$E(\text{unsigned char}) \in [0; 255]$$

Количество различных значений, которое может быть представлено переменными данного типа, равняется:

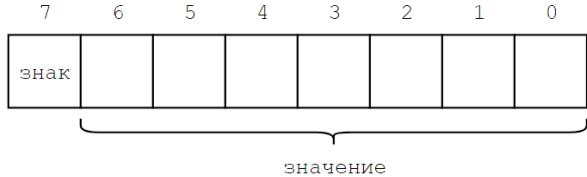
$$|E(\text{unsigned char})| = 2^8$$

Из школьного курса информатики вам должно быть известно, что бит – единица информации, которая может хранить либо значение 0, либо значение 1. Следовательно, если у вас имеется всего один бит, вы можете представить 2 возможных значения.

Если битов становится два, тогда возможные значения, образуемые парой бит:  $00_2 = 0_{10}$ ,  $01_2 = 1_{10}$ ,  $10_2 = 2_{10}$ ,  $11_2 = 3_{10}$ . Всего 4 значения.

Для  $n$ -бит количество представляемых значений находится как  $2^n$ . Отметим ещё раз: **количество представляемых значений  $|E(T)|$  зависит только от количества бит, отводимых для данного типа  $T$ .** Если под значение типа  $T$  отводится 8 бит, то  $|E(T)| = 2^8$ ; если 16 бит -  $|E(T)| = 2^{16}$  и так далее. В теории могут существовать типы с количеством бит не кратным 8. Предположим, если под тип данных отводится 3 бита, то он может принимать  $|E(T)| = 2^3 = 8$  каких-то значений.

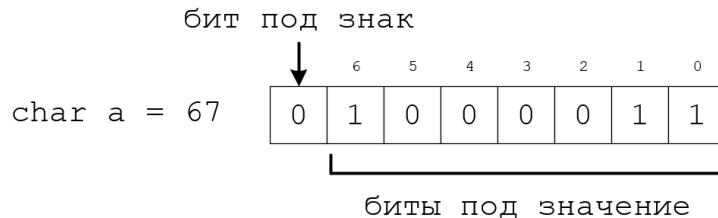
Если старший бит из восьми имеющихся отвести под знак (`signed char`):



тогда под значение останется 7 бит. Можно представить  $2^7$  положительных значений и  $2^7$  отрицательных значений. 0 будет отдан положительным. Значит:

$$E(\text{char}) \in [-2^7; 2^7 - 1] \quad E(\text{char}) \in [-128; 127] \quad |E(\text{char})| = 2^8$$

Затронем момент двоичного представления чисел типа `char`. Если знаковый бит равен нулю, то можно просто перевести число в двоичную систему счисления:



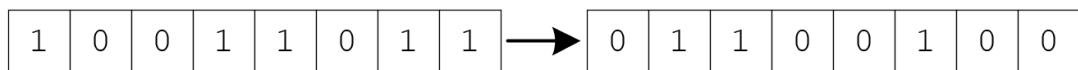
$$1000011_2 = 2^6 + 2^1 + 2^0 = 64 + 2 + 1 = 67_{10}$$

Отрицательные числа представляются в дополнительном коде. Чтобы выполнить перевод в привычную нам форму, необходимо выполнить ряд действий. Пусть дано число:



Для получения эквивалентного числа в десятичной системе счисления необходимо:

- Инвертировать (заменить нули на единицы и наоборот) биты числа:



- Перевести полученное число в десятичную систему счисления:

$$1100100_2 = 2^6 + 2^5 + 2^2 = 64 + 32 + 4 = 100_{10}$$

- Прибавить единицу:

$$a = 100 + 1 = 101$$

- Перед полученным на прошлом этапе значению поставить знак минус:  $-101$ .

Пример объявления<sup>13</sup> переменных:

---

```
signed char a;
unsigned char b;
```

---

По умолчанию все целочисленные типы являются знаковыми, и нет необходимости писать ключевое слово `signed`:

---

```
// данные записи эквивалентны при написании программ остановитесь на варианте
→ без signed
signed char a;
char a;
```

---

Последний вопрос по данному типу – вопрос вывода значений. С типом `char` наиболее часто используются спецификаторы: `%d` и `%c`. Спецификаторы не влияют на двоичное представление символа. Они только определяют, каким образом отобразить значение<sup>14</sup>:

<sup>13</sup>Объявление переменной говорит компилятору, что он должен связать определенное количество байт памяти с именем данной переменной.

<sup>14</sup>Для того, чтобы воспользоваться функцией `printf` необходимо подключить заголовочный файл `stdio.h`

---

```
#include <stdio.h>

int main() {
    char c = 65;

    printf("%d\n", c); // 65
    printf("%c\n", c); // A

    return 0;
}
```

---

Для ввода переменных данного типа не следует использовать спецификатор `%d`.

### 2.1.2 *int / unsigned int*

Тип данных `int` занимает машинное слово. **Машинным словом** называется единица данных, которая выбрана естественной для данной архитектуры процессора. На многих персональных машинах (на момент написания пособия) машинное слово равняется 4 байт (32 бита).

Если старший бит отводится под знак, то остаётся 31 бит под значение. Наибольшим представляемым числом переменными данного типа является  $2^{31} - 1$  (стоит вспомнить про 0). Также можно представить  $2^{31}$  отрицательных значений. Выходит, диапазон составляет:

$$E(\text{int}) \in [-2^{31}; 2^{31} - 1] \quad E(\text{int}) \in [-2147483648; 2147483647] \quad |E(\text{int})| = 2^{32}$$

Если все 32 бита отвести под значение при помощи ключевого слова `unsigned` получим:

$$E(\text{unsigned int}) \in [0; 2^{32} - 1] \quad E(\text{unsigned int}) \in [0; 4294967295] \\ |E(\text{unsigned int})| = 2^{32}$$

Примеры объявления переменных:

---

```
int a;
unsigned int b;
unsigned c;
```

---

При применении `unsigned` можно опустить имя типа `int`.

Для вывода значений типа `int` используется функция `printf` со спецификатором `%d`:

---

```
int a = 54;
printf("%d", a);
```

---

Для типа `unsigned int` используется спецификатор `%u`:

---

```
unsigned a = 54;
printf("%u", a);
```

---

Целочисленные литералы могут быть записаны в десятичной, восьмеричной и шестнадцатеричной системах счисления. Восьмеричная константа имеет префикс 0. Пара символов 0x или 0X является префиксом шестнадцатеричной константы:

---

```
int a = 16; // a = 16
int b = 020; // b = 16
int c = 0xA; // c = 10
int d = 0xFF; // d = 255
```

---

Для вывода числа в восьмиричном виде используется спецификатор %o или %#o:

---

```
printf("%o %#o", 100, 100); // 144 0144
```

---

а для вывода числа в 16-ой системе счисления – спецификатор %x или %#x:

---

```
printf("%x %#x", 110, 110); // 6e 0x6e
```

---

### 2.1.3 *short int / unsigned short int; long int / unsigned long int; long long int / unsigned long long int*

Последние вариации знаковых и беззнаковых типов устроены аналогично. Первый бит может как отводиться под знак, так и нет. Нас будет интересовать их размер в байтах (а диапазоны можно вывести). Тип `short` может использовать меньший объем памяти, чем `int`. Тип `long int` может использовать больший объем памяти, чем `int`<sup>15</sup>. А тип `long long int` может использовать больше памяти чем `long int`:

$$\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$$

Как показывает практика на текущий момент, использование переменных типа `long` не является безопасным. Она может занимать как 4 байта, так и 8 байт в разных операционных системах. Рекомендуется не использовать тип `long`.

Унарная операция `sizeof` позволяет получить размер типа в байтах. Пример использования операции:

---

```
#include <stdio.h>

int main() {
    printf("%d\n", sizeof(short));
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(long));
    printf("%d\n", sizeof(long long));

    return 0;
}
```

---

Восьмеричный префикс является пережитком языка B, который использовали еще во времена компьютера PDP-8 и вездесущих восьмеричных литералов. Языки C/C++ продолжают сомнительную традицию.

<sup>15</sup>Ключевыми словами в прошлых предложениях выступает слово 'может'. Например, переменная типа `short` может быть меньше переменной `int`, но может и не быть.

На моей машине тип `short` равен двум байтам, `int` и `long` – четырём байтам, `long long` – восьми байтам. Вы можете получить отличные значения.

Формат вывода для каждого из типов:

Тип	Спецификатор с модификаторами
<code>short int</code>	<code>%hd</code>
<code>unsigned short int</code>	<code>%hu</code>
<code>int</code>	<code>%d</code>
<code>unsigned int</code>	<code>%u</code>
<code>long</code>	<code>%ld</code>
<code>unsigned long</code>	<code>%lu</code>
<code>long long</code>	<code>%lld</code>
<code>unsigned long long</code>	<code>%llu</code>

Таблица с суффиксами литералов представлена ниже:

Суффикс	Тип
без суффикса	<code>int / long int / long long int</code>
<code>u</code> или <code>U</code>	<code>unsigned int / unsigned long int / unsigned long long int</code>
<code>l</code> или <code>L</code>	<code>long int / long long int</code>
<code>l/U + u/U</code>	<code>unsigned long int / unsigned long long int</code>
<code>ll</code> или <code>LL</code>	<code>long long int</code>
<code>ll/LL + u/U</code>	<code>unsigned long long int</code>

## 2.2 Вещественные типы данных: `float`, `double`

**Число с плавающей запятой** более или менее соответствует тому, что математики называют вещественным числом. Примерами чисел с плавающей запятой могут выступать 2.64, 3.15E3, -1,4e-3, 4.00.

Вещественные типы данных представлены типами `float` (4 байта), `double` (8 байт). Нам известно, что количество вещественных чисел бесконечно. Но компьютер не может представить бесконечное количество чисел в ограниченной памяти.

Тип `float` гарантирует, что 6-7 знаков после запятой могут быть сохранены. Чтобы оценить их количество, число нужно записать в экспоненциальной форме. Например:

$$144.78 = 1.4478 * 10^2 \quad (4 \text{ знака после запятой})$$

$$0.0013 = 1.3 * 10^{-3} \quad (1 \text{ знак после запятой})$$

Диапазон:

$$E(\text{float}) = [3,4 * 10^{-38}; 3,4 * 10^{38}]$$

Тип `double` позволяет представить 13 знаков после запятой. Диапазон:

$$E(\text{double}) = [1,7 * 10^{-308}; 1,7 * 10^{308}]$$

Обычно его бывает достаточно (но если вам и этого не хватает, возможно вам сможет помочь `long double`).

Объявление переменных выглядит следующим образом:

---

```
float a;
double b;
```

---

Проблему с точностью представления вещественных чисел можно заметить в листинге:

---

```
#include <stdio.h>

int main() {
    float a = 123456789;
    float b = 1;
    float c = a + b;

    printf("%f %f", a, c);

    return 0;
}
```

---

Программа дважды выведет число 123456792.000000. Сложение с единицей было проигнорировано (оно никак не влияет на 6-7 цифр после запятой). Значение 123456789 не смогло быть сохранено, так как в экспоненциальной форме имеет 8 знаков после запятой. Представление первых 6 цифр корректно. Дальше замечаем несколько странное округление:

```
123456792.000000 123456792.000000
Process finished with exit code 0
```

Константы с плавающей точкой имеют десятичную точку (123.4), или экспоненциальную часть (1e-2), или же и то и другое. Если у них нет суффикса, считается, что они принадлежат к типу `double`. Окончание `f` или `F` указывает на тип `float`, а `l` или `L` — на тип `long double`.

Функция `printf` предоставляет следующие спецификаторы вывода:

---

```
float a = 1.424;
double b = 5242e23;
printf("%f\n", a); // 1.424000
printf("%e\n", a); // 1.424000e+000

// в следующем выводе видна проблема представления вещественных чисел
printf("%f\n", b); // 52420000000000000000000000000000.000000
printf("%e\n", b); // 5.242000e+026
```

---

## 2.3 Пустой тип: `void`

Тип `void` имеет пустой набор значений и операций. Данный тип используется в особых случаях, речь о которых пойдёт позже по ходу изучения языка.

Работа программиста и шамана имеет много общего — оба бормочут непонятные слова, совершают непонятные действия и не могут объяснить, как оно работает.

## 2.4 Объявление и инициализация переменных

Все переменные должны быть объявлены раньше, чем будут использоваться. Когда вы пишите:

---

```
// <тип> <переменная>
int a;
```

---

то объявляете переменную `a` типа `int`. **Объявление** переменной говорит компилятору, что он должен связать определенное количество байт памяти для хранения значения переменной `a` (в данном случае – объем памяти, равный машинному слову; рисунок 2.2):

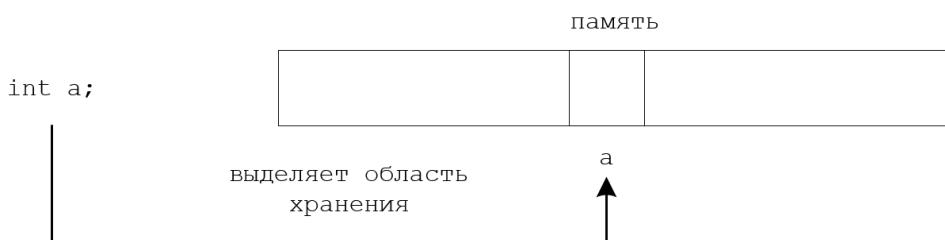


Рис. 2.2: Объявление переменной

Одновременно можно объявить несколько переменных одного типа:

---

```
int a, b;
```

---

но программисты нередко объявляют каждую переменную с новой строки. Это даёт чуть больше возможностей для комментирования кода.

В примерах выше мы только объявили переменные, но не инициализировали их (не присвоили начальное значение). Переменные `a` и `b` называются **неинициализированными**.

Под **инициализацией объекта** будем понимать установление его начального значения (рисунок 2.3).

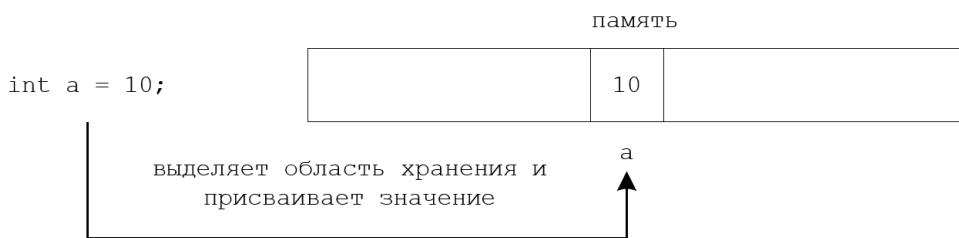


Рис. 2.3: Объявление с инициализацией

Значение неинициализированной переменной неопределено. Можно было бы предложить инициализировать целые значения нулем. Но может на текущий момент мне не нужна инициализация? Тогда будут произведены лишние операции, влияющие на продуктивность приложения. Поэтому от идеи автоматической инициализации (за исключением редких случаев) в языке программирования С отказались.

Существует рекомендация, которая предписывает инициализировать переменные при объявлении (если это возможно):

---

```
int a = 10;
int b = 10, c = 63.
```

---

Иногда значение переменной будет получено извне (например, с клавиатуры). В этом случае инициализация при объявлении будет излишней (зачем записывать какое-либо значение в переменную, если оно будет переписано позже?!). Программа ниже выполняет ввод и вывод значения переменной *a*:

---

```
int a;
scanf("%d", &a);

printf("%d", a);
```

---

Переменные могут быть инициализированы результатом вычисленного выражения:

---

```
int a = 10;
int b = 2*a - 5; // b = 15
```

---

При попытке сохранить в переменную значение, которое выходит за диапазон его значений, современными средствами разработки выдаётся предупреждение (рисунок 2.4). Никогда не игнорируйте их. Объяснение данному преобразованию можно найти на рисунке 2.5.

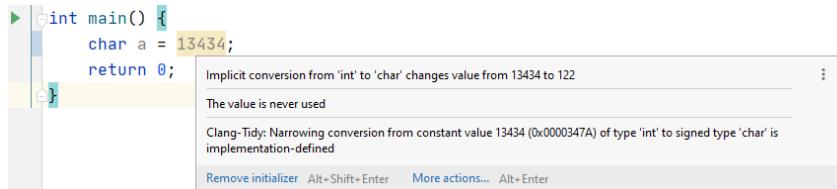


Рис. 2.4: Предупреждение среды о неявном преобразовании значения 13434 в 122.

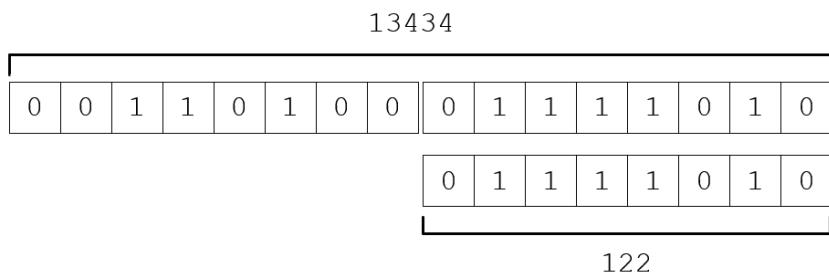


Рис. 2.5: В процессе присваивания старший байт числа 13434 будет отброшен. Остается значение 122.

## 2.5 Массивы

**Массив** это непрерывный участок памяти, содержащий последовательность объектов одного типа. Количество элементов в массиве называется его **размером**. Переменная-массив имеет тип – указатель на базовый элемент массива.

Чтобы объявить массив, нужно после его имени указать размер:

---

```
int a[10];      // массив из 10 элементов
int b[100];     // массив из 100 элементов
int c[10][10];  // массив из 10 на 10 элементов
```

---

Каждый элемент массива характеризуется:

1. Адресом элемента – адресом ячейки памяти, в которой расположен элемент;
2. Индексом элемента – порядковым номером элемента в массиве;
3. Значением элемента.

Обращение к элементам массива осуществляется через имя массива и в квадратных скобках указывается индекс (порядковый номер). Нумерация элементов массива в языке программирования С производится с нуля<sup>16</sup>. Отображение на память представлено на рисунке 2.6.

---

```
int a[3];
a[0] = 1;
a[1] = 4;
a[2] = 6;
// a[3] - ошибка; выход за пределы массива
```

---

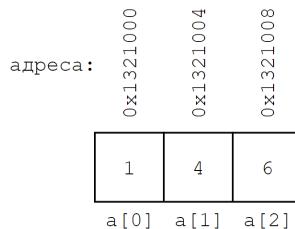


Рис. 2.6: Расположение элементов массива в памяти

Массив может быть инициализирован при объявлении:

---

```
int a[5] = {1, 4, 2};
// a[0] = 1, a[1] = 4, a[2] = 2, a[3] = 0, a[4] = 0;
// если в списке инициализации элементов меньше, чем размер массива,
// то они получают значение 0
```

---

Приведём небольшой фрагмент, в котором покажем работу с массивом:

---

```
#include <stdio.h>

int main() {
    // объявление и инициализация массива размера 3
    // компилятор сам может вычислить размер массива, если
    // имеется список инициализации
    int a[] = {1, 4, 2};
```

---

<sup>16</sup>Нашел ответ на вопрос по вечному спору – с чего должен начинаться индекс массива – с «0» или «1». Считаю, что мое компромиссное решение – «0,5» — было отвергнуто без надлежащего изучения.

```

// вывод a[0]
printf("%d\n", a[0]);

// присвоение a[1] значения 100
a[1] = 100;

// ввод значения a[2]
scanf("%d", &a[2]);

// вывод трёх элементов массива
printf("%d ", a[0]);
printf("%d ", a[1]);
printf("%d ", a[2]);

return 0;
}

```

Результат работы программы:

```

1
40
1 100 40
Process finished with exit code 0

```

## 2.6 Указатели

**Указатели** – это переменные, которые хранят в себе значение адреса. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Для того, чтобы объявить указатель, необходимо перед именем переменной поставить звёздочку (\*):

---

```

int a;      // a - простая целочисленная переменная
int *pa;   // pa - указатель на целое (int)

```

---

Основные операции, применяемые при использовании указателей:

- операция косвенного доступа (\*) позволяет получить значение объекта по его адресу – определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
- операция взятия адреса (&) (амперсанд) позволяет получить адрес переменной.

Указатель без инициализации хранит мусор, как и любая другая переменная. Но в то же время, этот “мусор” вполне может оказаться валидным адресом. Иногда мы хотим явно проинициализировать таким значением, которое бы показывало, что указатель на данный момент не указывает на какой-то полезный для нас адрес. Для этого применяется макрос `NULL`:

---

```
int *a = NULL;
```

---

При объявлении указатель можно проинициализировать адресом другой переменной (рисунок 2.7):

---

```
int a = 10;      // a - простая целочисленная переменная
int *pa = &a;    // pa - указатель на целое (int)
```

---

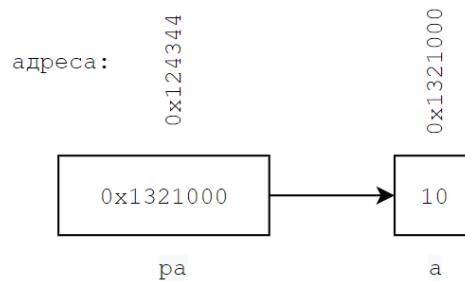


Рис. 2.7: Переменная *pa* хранит адрес переменной *a*

Изменить значение по адресу *pa* при помощи операции косвенного доступа:

---

```
*pa = 20;
```

---

Теперь значение переменной *a* равняется 20.

Ещё один пример программы (результат выполнения на рисунке 2.8):

---

```
#include <stdio.h>

int main() {
    int a = 10;
    int *pa = &a; // указатель хранит адрес переменной a
    printf("%d %d\n", pa, &a);
    printf("%d %d\n", *pa, a);

    *pa = 4;      // изменяем значение по адресу pa
    printf("%d %d\n", *pa, a);

    a = 1;        // изменяем значение переменной a
    printf("%d %d\n", *pa, a);

    return 0;
}
```

---

— Чем отличается программист от политика? — Программисту платят деньги за работающие программы.

```

7143112 7143112
10 10
4 4
1 1

Process finished with exit code 0

```

Рис. 2.8: Результат выполнения программы

Крайне важна следующая связь: имя массива является указателем на нулевой элемент массива:

---

```

int a[3] = {1, 4, 2};
*a = 3;
printf("%d", a[0]); // a[0] = 3;

```

---

Пока что применение указателей не кажется чем-то полезным, однако вся мощь данного инструмента будет показана позже, когда мы обсудим работу с динамической памятью.

## 2.7 Строки

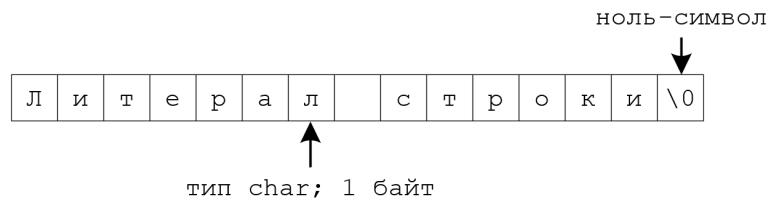
В языке программирования С отсутствует отдельный строковый тип. **Символьные строки** в нём являются последовательностью из одного или большего количества символов (массив символов). Строковые константы обрамляются в двойные кавычки<sup>17</sup>:

---

"Литерал строки"

---

Пример выше представляет собой в памяти массив символов и занимает 15 байт:



Строка должна заканчиваться ноль-символом ('\\0'), который является признаком конца строки. Поэтому требуется выделить на один байт памяти больше под её хранение:

---

```

char s[6] = "Hello";
char t[] = "world"; // размер массива t = 6.

```

---

Именно по ноль-символу можно понять, когда достигнут конец строки.

В строках можно использовать escape-последовательности, но при этом они в апострофы не заключаются:

<sup>17</sup>Нужно понимать, что символ 'x' и строка "x" - разные вещи. Первое – одиночный символ, второе – строка из двух символов: 'x' и '\0'.

---

```
char s[] = "Hello\t!"; // размер массива s = 8 элементам:
                      // 5 букв, табуляция, знак восклицания и ноль-символ
```

---

Если возникает необходимость считать строку с клавиатуры, нужно сначала создать массив под её хранение и использовать функцию `gets`:

---

```
#include <stdio.h>

int main() {
    char name[15];
    gets(name);

    printf("Hello %s", name); //вместо \%s будет выведено значение name

    return 0;
}
```

---

Если необходимо вывести длинную строку-литерал, её можно 'разбить' на несколько строк:

---

```
printf("It's a long string.
      "A very long string");
```

---

## 2.8 Константы

**Именованная константа** – это переменная, которой при объявлении присвоено значение, которое не может быть изменено. При помощи ключевого слова `const` можно создать именованную константу:

---

```
const int a = 10;
// int const a = 10; работает аналогично,
```

---

Переменная `a` является именованной константой. Когда используется ключевое слово `const`, компилятор следит за тем, чтобы значение переменной не было изменено. При осуществлении таких попыток будут выданы ошибки (рисунок 2.9) от среды разработки и программа не будет компилироваться.

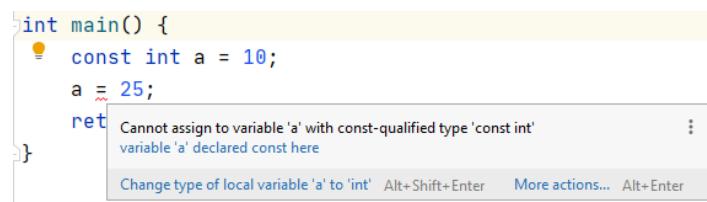


Рис. 2.9: Попытка выполнить изменение константы

Стоит отметить вопрос применения ключевого слова `const` к указателям:

---

```

int main() {
    int a = 10;

    // p - может указывать на int переменные
    // (а в С - ещё и на const int переменные, но не в C++)
    // значение указателя может быть изменено
    // значение по указателю может быть изменено
    int *p;

    // p1 - может указывать на int и const int переменные
    // значение указателя может быть изменено
    // значение по указателю не может быть изменено
    const int *p1;
    // Допустимо: int const *p1;

    // p2 - может указывать на int и const int переменные
    // (а в С - ещё и на const int переменные, но не в C++)
    // значение указателя должно быть задано при инициализации и не может
    // быть изменено
    // значение по указателю может быть изменено
    int * const p2 = &a;

    // p3 - может указывать на int и const int переменные
    // значение указателя должно быть задано при инициализации и не может
    // быть изменено
    // значение по указателю не может быть изменено
    const int * const p3 = &a;
    // Допустимо: int const * const p3 = &a;

    return 0;
}

```

---

В языке имеется способ изменения константы, объявленной как `const` через её указатель<sup>18</sup>:

---

```

#include <stdio.h>

int main() {
    const int b = 10;
    int *a = &b;
    *a = 20;

    printf("%d", b);

    return 0;
}

```

---

При этом вы получите только предупреждение от компилятора.

---

<sup>18</sup>Такое поведение недопустимо в C++.

В языке С существует ещё один способ объявление констант – через макросы.

---

```
#include <stdio.h>

#define N 10

int main() {
    printf("%d", N);

    return 0;
}
```

---

Макрос `#define` заставляет препроцессор выполнить макроподстановку: замену всех вхождений идентификатора `N` на `10`. А использование `#include` потребует от препроцессора вставить содержимое файла `stdio.h`. С точки зрения выполнимых программой действий она будет эквивалентна следующей:

---

```
// содержимое <stdio.h>

int main() {
    printf("%d", 10); // выполнена замена N на 10

    return 0;
}
```

---

## Резюме

- В процессе написания программ оперируют объектами, которые могут иметь тот или иной тип данных, определяющий множество допустимых значений и множество допустимых операций.
- В языке С стандарта C90 определены следующие базовые типы:
  - `char`, `int` – целочисленные типы;
  - `float`, `double` – вещественные типы;
  - `void` – пустой тип.
- Базовые типы могут быть модифицированы с помощью модификаторов: `short`, `long` изменяют размер типа `int`; `signed`, `unsigned` определяют интерпретацию старшего бита для целочисленных типов.
- Все переменные обязаны быть объявлены раньше, чем будут использоваться.
- При объявлении рекомендуется присваивать начальное значение переменным (если это возможно). Объявления должны находиться максимально близко к месту первого их использования.
- Массив – это непрерывный участок памяти, содержащий последовательность объектов одного типа.
- Элементы массива в языке С нумеруются с нуля, а имя массива – указатель на нулевой элемент.

- Указатели – это переменные, которые хранят в себе значение адреса. Над ними в языке С определены две операции: косвенный доступ и взятие адреса.
- Строковый тип в языке С отсутствует, вместо него используется массив символов.
- В С можно задать константы при помощи ключевого слова `const` и при помощи макроса `#define`. В первом случае компилятор следит за тем, чтобы значение переменной не было изменено, а во втором макрос `#define` заставляет препроцессор выполнить макроподстановку.

## Термины и определения

- **Константа** – это программный объект, не изменяющий своего значения.
- **Листинг** – это бумажная распечатка текста компьютерной программы или её части.
- **Литерал** – это константа, тип и значение которой определяется по ее виду.
- **Массив** – это непрерывный участок памяти, содержащий последовательность объектов одного типа.
- **Машинное слово** – это единица данных, которая выбрана естественной для данной архитектуры процессора.
- **Объект данных** – это область хранения данных, которая может применяться для удержания значений.
- **Операнд** – это то, чем оперируют операции.
- **Операция** – это конструкция в языках программирования; специальный способ записи некоторых действий; некоторое действие, выполняемое над операндами.
- **Переменная** – это программный объект, изменяющий своё значение.
- **Символьный литерал** – это целое (переменная типа `int`), записанное в виде символа, обрамленного одиночными кавычками.
- **Указатели** – это переменные, которые хранят в себе значение адреса.

## Контрольные вопросы

1. Что определяет тип данных? Перечислите базовые типы данных С.
2. Целочисленный тип `char`. Вывод диапазона значений `char / unsigned char`.
3. Целочисленный тип `int`. Вывод диапазона значений.
4. Определение «Машинное слово». Связь между размерами целочисленных типов. Унарная операция для определения размера типа.

5. Представление отрицательных целых чисел в языке программирования С. Получение значения по двоичному представлению.
6. Вещественные типы данных. Число с плавающей запятой. Проблемы вещественной арифметики (приведите пример). Количество значений, представляемых переменными вещественного типа.
7. В каких системах счисления могут быть представлены целочисленные литералы? Приведите примеры.
8. Приведите примеры *escape*-последовательностей.
9. В каком случае инициализация переменной при ее объявлении будет излишней?
10. Массивы в С. Объявление. Инициализация.
11. Указатели в С. Операции взятия адреса и косвенного доступа. Макрос `NULL`.
12. В чем заключается разница между литералами `'x'` и `"x"`?
13. В чем заключается разница между объявлением констант при помощи макроса `#define` и при помощи ключевого слова `const`?

# Глава 3

## Линейные алгоритмы

**Линейными алгоритмами** являются алгоритмы, все шаги которых выполняются вне зависимости от каких-либо условий.

Задач, решения которых могут быть описаны линейными алгоритмами, не так уж и много, но этого будет достаточно, чтобы обсудить начальные моменты и проектирования, и программирования.

### 3.1 Задача о сумме

#### Задача о сумме

Вычислить значение суммы переменных  $a$  и  $b$ , вводимых с клавиатуры.

Решение каждой задачи будем описывать и блок-схемой и программой. Блок-схема алгоритма представлена на рисунке 3.1.

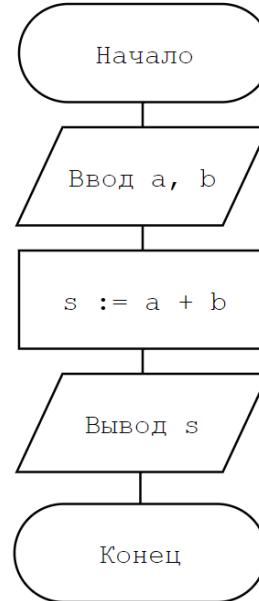


Рис. 3.1: Блок-схема алгоритма задачи о сумме

Алгоритм описан 5 блоками. Напишем пустую программу на С (и уже можем считать, что блок 'начало' и блок 'конец' закодированы):

---

```
#include <stdio.h>

int main() {
    return 0;
}
```

---

Следующим действием необходимо оценить, какие переменные имеются в блоках. В данном случае, их 3: переменные `a` и `b` вводятся с клавиатуры, а в переменной `s` сохранён результат. Значения `a` и `b` являются входными данными для этой задачи. Объявим их:

---

```
#include <stdio.h>

int main() {
    int a, b;

    return 0;
}
```

---

Для чтения переменных используется функция `scanf`, первым параметром которой выступает управляющая строка, а последующие её параметры – адреса областей памяти, в которые необходимо произвести запись значений (о вводе и выводе будет подробно рассказано в следующих главах):

---

```
scanf("%d %d", &a, &b);
```

---

`&a` – операция взятия адреса для переменной `a`<sup>1</sup>. Ранее рассматривалось, какие спецификаторы требуются для каких типов. Для переменных целого типа `int` можно использовать `%d`.

Результат операции сложения может быть присвоен переменной `s` при её инициализации:

---

```
int s = a + b;
```

---

Будет вычислено значение суммы переменных `a` и `b`, и она будет записана в переменную `s`.

Выведем результат при помощи `printf`:

---

```
printf("Сумма = %d", s);
```

---

Вместо `%d` будет подставлено значение переменной `s`. Если `s = 4`, на экране будет отображено сообщение: "Сумма = 4". Можно вывести и несколько переменных, например:

---

```
int a = 1;
int b = 3;
printf("a = %d, b = %d", a, b); // вывело бы сообщение "a = 1, b = 3"
```

---

<sup>1</sup>`a` – переменная, `&a` – значение адреса, где хранится переменная `a`.

Добавим в программу объявление и вычисление переменной `s`, поддержку вывода русского языка и получим:

---

```
#include <stdio.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    int a, b;
    scanf("%d %d", &a, &b);

    int s = a + b;

    printf("Сумма = %d", s);

    return 0;
}
```

---

В исходном тексте программы оставлены пустые строки. Они не являются обязательными, но могут быть использованы для улучшения читаемости. В примере выше они разделяют ввод, обработку и вывод.

В результате запуска программы получим (зеленым цветом описаны вводимые значения, черным – выводимые):

```
1 3
Сумма = 4
Process finished with exit code 0
```

## 3.2 Задача обмена значений

### Задача обмена значений

Обменять значение переменных  $a$  и  $b$ <sup>a</sup>.

---

<sup>a</sup>Эта задача может решаться как с использованием, так и без использования третьей переменной.

Блок-схема алгоритма представлена на рисунке 3.2 (страница 53).

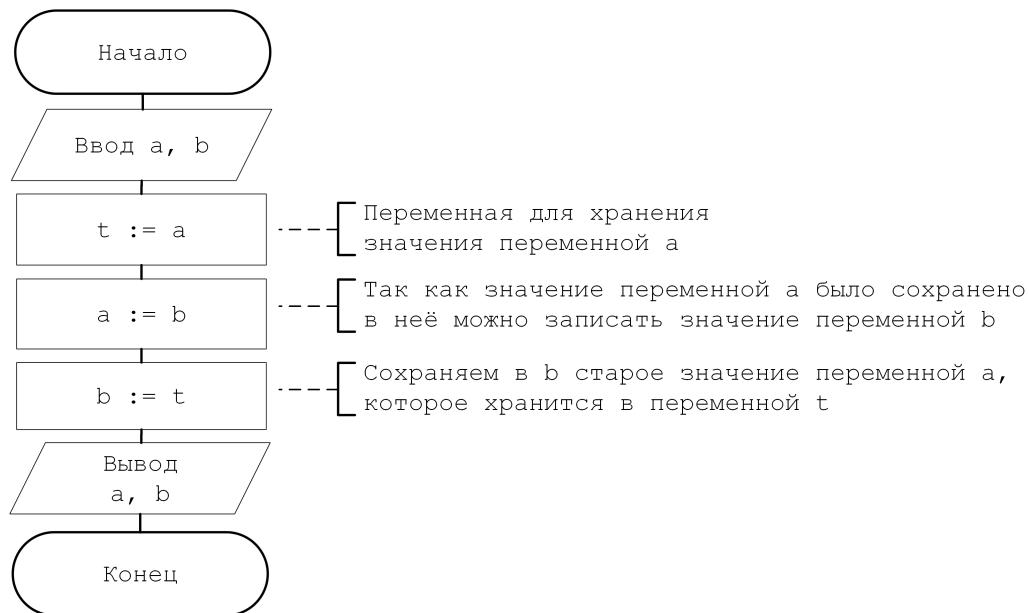


Рис. 3.2: Блок-схема алгоритма задачи обмена значений

Данная задача решается с использованием третьей переменной `t`. Будет ошибочно написать инструкции:

- `a := b;`
- `b := a;`

так как после первого присваивания, значение переменной `a` будет затёрто переменной `b`. А вторая инструкция будет вовсе бесполезной.

Код программы:

---

### Листинг 2 Обмен двух значений

---

```

#include <stdio.h>

int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    int t = a;
    a = b;
    b = t;

    printf("a = %d b = %d", a, b);

    return 0;
}

```

---

Когда вы видите, что компьютер пишет: 'Вы не робот?', подумайте – вдруг он просто хочет создать семью?

Результат работы программы:

```
1 3
a = 3 b = 1
Process finished with exit code 0
```

### 3.3 Задача о последней цифре числа

#### Задача о последней цифре числа

Вычислить последнюю цифру числа  $x$ , вводимого с клавиатуры.

Блок-схема алгоритма представлена на рисунке 3.3 (страница 54).

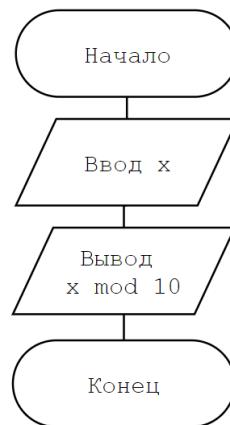


Рис. 3.3: Блок-схема алгоритма задачи о последней цифре числа

Внимание в данной задаче нужно уделить двум моментам:

- в качестве аргумента для выводимого значения может выступать более сложные выражения;
- операции вычисления остатка от целочисленного деления (и их обозначению в коде и блок-схемах).

Код:

---

```
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    printf("last digit = %d", x % 10);

    return 0;
}
```

---

### 3.4 Задача о вычислении расстояния между двумя точками

#### Задача о вычислении расстояния между двумя точками

Вычислить расстояние между точками  $p_1$  с координатами  $(x_1, y_1)$  и  $p_2$  с координатами  $(x_2, y_2)$ .

Расстояние между двумя точками может быть вычислено по формуле:

$$d_1 = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Закодируем её. Блок-схема алгоритма представлена на рисунке 3.4 (страница 55).

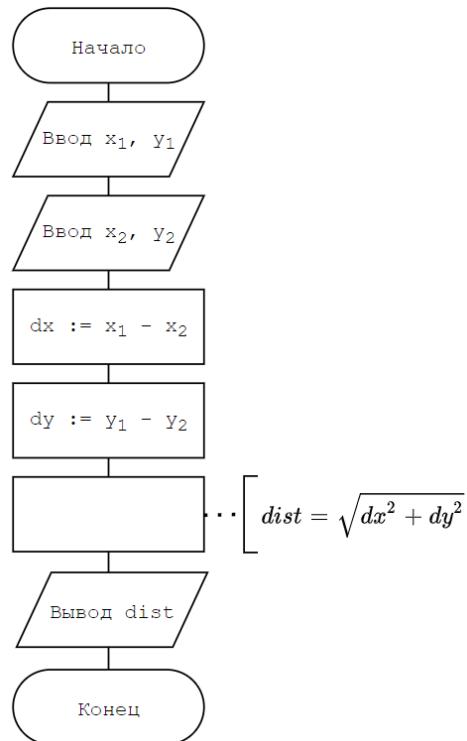


Рис. 3.4: Блок-схема алгоритма задачи о вычислении расстояния между двумя точками

Для того чтобы воспользоваться функциями из математической библиотеки, необходимо их импортировать из `math.h`.

---

### Листинг 3 Вычисление расстояния между двумя точками

---

```
#include <stdio.h>
#include <math.h>

int main() {
    int x1, x2, y1, y2;
    scanf("%d %d", &x1, &y1);
    scanf("%d %d", &x2, &y2);

    int dx = x2 - x1;
    int dy = y2 - y1;
    float distance = sqrt(dx*dx + dy*dy);

    printf("distance = %f", distance);

    return 0;
}
```

---

Результат работы программы представлены на рисунке 3.5.

```
1 1
4 5
dist = 5.000000
Process finished with exit code 0
```

Рис. 3.5: Результаты работы программы

## 3.5 Задача о генерации числа в заданном диапазоне

### Задача о генерации числа в заданном диапазоне

С клавиатуры вводятся числа  $a$  и  $b$ . Необходимо сгенерировать случайное число из промежутка от  $a$  до  $b$ .

Для того чтобы сгенерировать случайное значение, необходимо использовать генератор псевдослучайных чисел. Имеется ряд алгоритмов, которые позволяют делать это. Когда кто-то акцентирует внимание на 'псевдослучайности', он хочет отметить, что числа сами по себе не являются случайными (они выдаются определенным алгоритмом), но если на них посмотреть, то вроде как случайные<sup>2</sup>.

Для работы со случайными числами используются функции `rand` и `srand`:

- `srand()` задаёт так называемое зерно генератора случайных чисел (некоторое значение, которое будет использоваться для генерации всей последовательности). Если запускать программу с одним и тем же зерном, будут генерироваться одни и те же числа.
- `rand()` генерирует следующее случайное число в последовательности. Оно будет находиться в диапазоне от 0 до `RAND_MAX` (константа и функции `srand`,

---

<sup>2</sup>Когда мастер Угвэй из мультфильма 'Кунг-фу Панда' говорит фразу 'случайности не случайны' – это он скрыто упоминает принципы работы генератора псевдослучайных чисел.

`rand` определены в `<stdlib.h>`; значение `RAND_MAX` в моей системе она равна  $7FFF_{16}$ ).

Рассмотрим на примерах<sup>3</sup>:

---

#### Листинг 4 Генерация случайных чисел

---

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    int a = rand(); // генерация чисел от 0 до RAND\_MAX
    int b = rand() % 10; // генерация чисел от 0 до 9
    int c = rand() % 10 + 3; // генерация чисел от 3 до 12

    printf("%d %d %d ", a, b, c);

    a = 5;
    b = 10;

    int d = rand() % a; // генерация чисел от 0 до a - 1
    int e = rand() + b; // генерация чисел от b до RAND\_MAX + b
    int f = rand() % b + a; // генерация чисел от a до a + b - 1
    int g = a + rand() % (b - a + 1); // генерация чисел от a до b

    printf("%d %d %d %d", d, e, f, g);

    return 0;
}
```

---

На моей машине были получены значения: 41 7 7 0 19179 9 5. Сколько бы раз я не запускал программу – она будет выдавать одну и ту же последовательность (и будет так делать всегда, пока вы не измените зерно генерации). Иногда воспроизводимость случайной последовательности является крайне важной особенностью алгоритма генерации.

Если необходимо сгенерировать другую воспроизводимую последовательность, перед началом генерации нужно указать другое зерно:

---

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    srand(42);
    ...
}
```

---

Но если вам хочется, чтобы и зерно было всегда 'случайным' – вы можете использовать функцию `time`<sup>4</sup>, которая определена в `<time.h>`:

---

<sup>3</sup>Напомню, `%` – операция поиска остатка от деления:  $145 \% 10 = 5$

<sup>4</sup>Функция возвращает текущее календарное значение времени в секундах.

---

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main() {
    srand(time(0));
    ...
}
```

---

## 3.6 Задача о сумме арифметической прогрессии

### Задача о сумме арифметической прогрессии

Необходимо вычислить сумму арифметической прогрессии по заранее известным значениям  $a_1$ ,  $a_n$ ,  $d$ .

Можно воспользоваться следующей формулой:

$$s = \frac{a_1 + a_n}{2}n$$

но нам неизвестно  $n$ . Оно может быть вычислено как

$$n = \frac{a_n - a_1}{d} + 1$$

На первый взгляд кажется, что данный код решит задачу:

---

```
#include <stdio.h>

int main() {
    int a1, an, d;
    scanf("%d %d %d", &a1, &an, &d);

    int n = (an - a1) / d + 1;
    int s = (a1 + an) / 2 * n;

    printf("%d", s);

    return 0;
}
```

---

Однако программа найдёт без ошибок сумму арифметической прогрессии только для тех случаев, когда  $a_1 + a_n$  четно, что не всегда является истинным. Акцентируем внимание на следующей строчке:

---

```
int s = (a1 + an) / 2 * n;
```

---

В первую очередь выполнится сложение двух целых. И переменная  $a1$ , и переменная  $an$  являются переменными целочисленного типа. Результат любого выражения имеет тип. Значение выражения  $a1 + an$  имеет тип (`int` в данном случае).

Значение 2 – литерал целочисленного типа, на который осуществляется деление (/ - операция деления). Для языка С применимо следующее правило: если и первый операнд и второй операнд относятся к целочисленному типу, будет выполнено целочисленное деление, но если хотя бы один из них будет вещественным – выполнится вещественное деление (как на калькуляторе).

Предположим, что у нас была прогрессия:

$$a_1 = 1 \quad a_2 = 2 \quad a_3 = 3 \quad a_4 = 4 \quad n = 4$$

Тогда значение переменной `s` будет вычисляться так:

---

```
int s = (a1 + an) / 2 * n; // (1 + 4) / 2 * 4 -> 5 / 2 * 4 -> 2 * 4 -> 8
```

---

Данная сумма прогрессии не соответствует действительности. Лучшим решением будет воспользоваться свойством, что или  $a_1 + a_n$  или  $n$  будет четным при любых исходных данных. Тогда если переписать выражение в следующем виде:

---

```
int s = (a1 + an) * n / 2;
```

---

проблема наблюдаться не будет, так как  $(a_1 + a_n)n$  является четным и точно делится на 2.

Таким образом, код функции `main`:

---

```
#include <stdio.h>

int main() {
    int a1, an, d;
    scanf("%d %d %d", &a1, &an, &d);

    int n = (an - a1) / d + 1;
    int s = (a1 + an) * n / 2;

    printf("%d", s);

    return 0;
}
```

---

### 3.7 Задача перевода температуры из градусов Фаренгейта в градусы Цельсия

**Задача перевода температуры из градусов Фаренгейта в градусы Цельсия**

С клавиатуры вводится температура в градусах Фаренгейта. Необходимо вывести эквивалентную температуру в градусах Цельсия.

Перевод можно осуществить по формуле:

$$C = \frac{F - 32}{9} * 5$$

Рассмотрим 2 случая. Случай первый: F – переменная типа int:

---

```
#include <stdio.h>

int main() {
    int F;
    scanf("%d", &F);
    // int C = (F - 32) / 9 * 5; Не сработает, так как деление целочисленное
    // Предположим, будет введено F = 40. Тогда выполняются следующие расчеты:
    // (40 - 32) / 9 * 5 -> 8 / 9 * 5 -> 0 * 5 -> 0

    // Ситуацию можно решить одним из следующих способов:
    // 1. Представить 9 как литерал вещественного типа
    // (40 - 32) / 9.0 * 5 -> 8 / 9.0 * 5 -> 0.888889 * 5 -> 4.444445
    float C = (F - 32) / 9.0 * 5;
    // 2. Представить 32 как литерал вещественного типа:
    // float C = (F - 32.0) / 9 * 5;
    // (40 - 32.0) / 9 * 5 -> 8.0 / 9 * 5 -> 0.888889 * 5 -> 4.444445
    // 3. Заменить / 9 * 5 на / 1.8
    // float C = (F - 32) / 1.8;

    printf("%f", C);
    return 0;
}
```

---

Случай второй: F – переменная типа float:

---

```
#include <stdio.h>

int main() {
    float F;
    scanf("%f", &F);
    float C = (F - 32) / 9 * 5;
    // (40.0 - 32) / 9 * 5 -> 8.0 / 9 * 5 -> 0.888889 * 5 -> 4.444445
    printf("%f", C);
    return 0;
}
```

---

При решении данной задачи следует сделать вывод: обращайте внимание на типы выражений в контексте операции деления.

## Резюме

- Линейные алгоритмы – алгоритмы, все шаги которых выполняются вне зависимости от каких-либо условий.
- Чтение переменных происходит с помощью функции `scanf`, а вывод – с помощью функции `printf` объявленных в стандартной библиотеке `stdio.h`.
- Для вывода сообщений на русском языке необходимо включить поддержку вывода русского языка.
- В исходном тексте программы рекомендуется оставлять пустые строки, если это улучшает читаемость.
- Для использования функций из математической библиотеки, необходимо их импортировать из `math.h`.
- Для генерации случайного значения необходимо использовать генератор псевдослучайных чисел.
- При присваивании переменной значения, прежнее значение стирается.
- Правило выполнения деления в С: если первый и второй операнды относятся к целочисленному типу, будет выполнено целочисленное деление, в противном случае выполнится вещественное деление.

## Контрольные вопросы

1. Что такое линейный алгоритм? Приведите примеры.
2. Функция ввода `scanf` и её параметры.
3. Функция вывода `printf` и её параметры.
4. Подключение библиотеки в С. Какую библиотеку необходимо подключить для использования математических функций?
5. Функции для генерации случайных чисел. Для чего используются функции `srand` и `rand`?
6. Правило выполнения деления в С.
7. Напишите программу
  - обмена двух значений,
  - вычисляющую периметр треугольника по трем сторонам,
  - вычисляющую площадь треугольника по трем сторонам,
  - вычисляющую среднее геометрическое двух чисел.

# Глава 4

## Операции

В языке программирования С существует большое количество различных операций. **Операция** – это некоторое действие, которое выполняется над **операндом**. В зависимости от количества operandов, они делятся на:

1. Унарные (один operand),
2. Бинарные (два operandы),
3. Тернарные (три operandы).

**Выражение** – это комбинация operandов и операций. Выражение определяет способ вычисления **значения выражения**.

### 4.1 Арифметические операции

#### 4.1.1 Унарный минус и унарный плюс

Никаких хитростей у унарного минуса нет. Операция просто меняет знак числа:

```
char a = 42;
printf("%d", -a); // выведет -42
```

А унарный плюс вообще ничего не делает (но существует):

```
int a = +3;
```

#### 4.1.2 Сложение и вычитание

С сложением и вычитанием дела обстоят несколько сложнее. Предположим, что мы складываем два целочисленных operandов одного и того же типа:

```
char a = 28;
char b = 10;
char c = a + b;
char d = a + 100;

printf("%d %d", c, d);
```

С имеет мощь ассемблера и удобство... ассемблера (Деннис Ритчи).

Попробуем пока посчитать вручную для того, чтобы лучше понять, что именно происходит, когда складываются числа:

$$\begin{array}{r}
 + \boxed{\begin{array}{ccccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array}} \boxed{28} \\
 \hline
 \boxed{\begin{array}{ccccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{array}} \boxed{10} \quad \boxed{38}
 \end{array}$$

Первый пример Америку не открыл. А теперь второй:

$$\begin{array}{r}
 + \boxed{\begin{array}{ccccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}} \boxed{28} \\
 \hline
 \boxed{\begin{array}{ccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}} \boxed{100}
 \end{array}$$

Я специально не вписал ответ. Посмотрим в программу. Результат сложения переменной `a` и 100 записывается в переменную типа `char`. Переменная `char` является знаковой. И старший бит в ней отводится под знак. В качестве старшего бита в результате стоит единичка. Это уже говорит нам о том, что ответ получается отрицательный. Чтобы понять, какое же число там получилось, сделаем следующее:

1. Инвертируем результат ( $10000000 \rightarrow 01111111$ ).
2. Прибавим к нему единицу ( $01111111 + 1 = 10000000$ ).
3. Переведём в 10-ю систему счисления и перед результатом поставим знак минус  $10000000 \rightarrow -128^1$ .

Однако стоит отметить, что если сложение значений `28` и `100` происходило в `unsigned char`, то мы действительно бы получили `128`.

Неприятности ожидают и в том случае, если результат превышает максимум для знакового целого:

---

```

unsigned char a = 255;
unsigned char b = 1;
unsigned char c = a + b;

printf("%d", c);

```

---

Снова посчитаем столбиком:

$$\begin{array}{r}
 + \boxed{\begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}} \boxed{255} \\
 \hline
 \boxed{\begin{array}{ccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}} \boxed{1}
 \end{array}$$

<sup>1</sup>Магия перестаёт существовать после того, как вы понимаете, как она работает (Тим Бернерс-Ли).

Программирование на С подобно быстрому танцу на полу, только что натёртом воском, среди людей с острыми бритвами в руках.

Ожидали получить 256, но не смогли записать такое число в рамках одного байта и получили 0. Когда мы рассматриваем сложение беззнаковых целых, надо сказать, что оно происходит по модулю (по модулю 256 в случае однобайтовых целых). Т. е. если у вас возникнет соблазн посчитать, сколько будет  $150 + 798$  и этот результат вы запишите в однобайтовое беззнаковое целое, то получите:

$$150 + 798 = 948$$

которые в силу ограниченности памяти беззнакового однобайтного целого компьютер сохранит как

$$948 \bmod 256 = 180$$

Последняя проблема решается достаточно просто – для переменной-результата стоит взять тип побольше, например, `short`:

---

```
unsigned char a = 255;
unsigned char b = 1;
short c = a + b;

printf("%d", c); // 256
```

---

В связи с этим рекомендуется при написании кода приложения тщательно присматриваться к типу переменной и её потенциальным значениям в ходе выполнения программы. Например, если вводится большое количество значений, и мы хотим найти их сумму, то результат рекомендуется записать в переменную типа `long long`.

Опишем классический способ 'выстрелить себе в ногу' при помощи `unsigned int` и операции вычитания. Для этого потребуется некоторое знание функций. Если они вам знакомы, вы сможете понять пример ниже:

---

```
int a[] = {1, 2, 3, 4};
for (size_t i = 3; i >= 0; i--) // size_t - псевдоним для unsigned int
    printf("%d", a[i]);
```

---

Намерения программиста ясны и прозрачны. Он хотел вывести значения массива в обратном порядке. Но не учел тот факт, что после вычитания из нуля единицы, получится значение  $2^{32} - 1$ .

## Корректирующее присваивание

Следующие операции (`+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, `|`) могут быть объединены с операцией присваивания. Вместо того, чтобы писать:

---

```
a = a + 3;
```

---

лучше остановиться на более короткой форме:

---

```
a += 3;
```

---

Данную строчку кода можно прочитать, как 'увеличить значение переменной `a` на 3'. В последующей части пособия будет использоваться исключительно короткая запись.

## Префиксные и постфиксные инкременты и декременты

Предположим, что нам нужно увеличить или уменьшить значение переменной на единицу. Мы уже знаем как минимум два способа сделать это:

---

```
a = a + 1;
a += 1;
```

---

Выполнить увеличение переменной можно и посредством инкремента:

---

```
a++; // постфиксный инкремент
++a; // префиксный инкремент
```

---

а вычесть единицу при помощи декремента:

---

```
a--; // постфиксный декремент
--a; // префиксный декремент
```

---

Для применения операции существует требование: операнд должен иметь числовой тип или быть указателем. Операндом может быть только выражение  $l - value$ , то есть выражение, которому соответствует объект-переменная в памяти, и значение которого может быть изменено.

Если использовать данные выражения изолированно, то проблем возникнуть не должно. Риски появляются, если нет понимания, как они работают в составе других выражений<sup>2</sup>. Рассмотрим такой пример:

---

```
int a = 4;
int b = 1 + a++;
```

---

Как же посчитать значение переменной `b`? Руководствуйтесь следующим правилом: посмотрите на инкремент или декремент. Разделите его на две части: переменную и операцию отдельно. В примере выше, сначала будет переменная, а потом операция. То есть сперва будет использована переменная `a` для выражения:

---

```
int b = 1 + a;
```

---

и только потом выполнится операция: значение переменной `a` будет увеличено. Код выше можно было переписать следующим образом:

---

```
int a = 4;
int b = 1 + a;
a += 1;
```

---

Если бы инкремент был бы префиксным:

---

```
int a = 4;
int b = 1 + ++a;
```

---

<sup>2</sup>В языке программирования *Python* операция инкремента и декремента отсутствует. Они неплохие кандидаты на источник ошибок. А если такие операции излишне используются, код читается сложнее.

тогда поведение было бы эквивалентно следующему:

---

```
int a = 4;
a += 1;
int b = 1 + a;
```

---

Ситуация с декрементом аналогична<sup>3</sup>.

### 4.1.3 Умножение

В умножении чисел существует лишь одна проблема – это проблема переполнения (когда вычисленный результат не может поместиться в переменной-результате или в процессе промежуточных вычислений). В листинге ниже

---

```
int a, b;
scanf("%d %d", &a, &b);

long long c = a * b;
```

---

имеется проблема. При выполнении арифметических действий над целыми, тип которых меньше или равен `int`, промежуточные вычисления происходят в типе `int`. Если результат промежуточных вычислений окажется больше максимального значения типа, в котором ведутся вычисления, произойдёт переполнение. Предположим, что были введены значения  $a = 100000$  и  $b = 100000$ . Переведём значения в двоичную систему счисления (красным выделены 32 бита результата, допускаем в примере размер машинного слова 4 байта):

$$a = b = 100000_{10} = 1'10000110'10100000_2$$

$$a * b = (10^{10})_{10} = 10'01010100'00001011'11100100'00000000_2$$

Переменная `int` может сохранить только 32 бита. Значит, результатом вычисления будет:

$$01010100'00001011'11100100'00000000_2 = 1410065408_{10}$$

Да, он сохраняется в переменную `long long` и переменная `c` занимает 64 бита. Но туда запишется результат вычисления (который составляет 32 бит), и в двоичном представлении будет найдено

$$c = 00000000'00000000'00000000'00000000'01010100'00001011'11100100'00000000_2$$

О дополнительных уязвимостях можно прочитать в разделе о приведении типов данных (пример на странице 78).

---

<sup>3</sup>Теперь когда вы знаете, как ситуация обстоит с данными операциями, оставлю анекдот: Нелюбители языка C++ говорят, что его стоило бы назвать `++C`: сначала его нужно улучшить, а потом использовать.

#### 4.1.4 Деление и остаток от деления

Стоит остановиться на том, что существует два вида деления: вещественное и целочисленное. В первом варианте деление происходит так же, как и при помощи калькулятора. Второй вариант напоминает деление столбиком в начальной школе. И для первого и для второго способа знак операции выглядит одинаково `/`.

В языке программирования С деление будет целочисленным, если оба операнда принадлежат целочисленному типу:

---

```
int a = 10;
int b = 3;

printf("%d\n", a / b);    // 3
printf("%d\n", a / 4);    // 2
printf("%d\n", 100 / b);  // 33
printf("%d\n", 100 / 9);  // 11
```

---

Но если хотя бы один operand является вещественным, то и деление будет вещественным:

---

```
float a = 10;
float b = 3;

printf("%f\n", a / b);    // 3.333333
printf("%f\n", a / 4);    // 2.500000
printf("%f\n", 100 / b);  // 33.333333
printf("%f\n", 100. / 9); // 11.111111
```

---

Если оба операнда являются целочисленными, можно вычислить остаток от деления:

---

```
int a = 10;
int b = 6;

printf("%d\n", a % b);    // 4
printf("%d\n", a % 4);    // 2
printf("%d\n", 100 % b);  // 4
printf("%d\n", 90 % 9);  // 0
```

---

#### 4.1.5 Приоритеты арифметических операций

Приоритеты арифметических операций аналогичны тем, что приняты в математике. Приоритет операции поиска остатка от деления такой же, как и у умножения с делением (см. таблице 4.1).<sup>4</sup>

<sup>4</sup>Очерёдность операций в программировании – установленная синтаксисом конкретного языка программирования последовательность выполнения операций (или направление вычисления), реализуемая когда операции имеют одинаковый приоритет и отсутствует явное (с помощью скобок) указание на очерёдность их выполнения. Ассоциативность – свойство операций, позволяющее восстановить последовательность их выполнения при отсутствии явных указаний на очерёдность при равном приоритете; при этом различается левая ассоциативность, при которой вычисление выражения происходит слева направо, и правая ассоциативность – справа налево.

Таблица 4.1: Арифметические операции в порядке уменьшения приоритета

Операции	Ассоциативность
( )	Слева направо
+ - (унарные)	Справа налево
* / %	Слева направо
+ - (бинарные)	Слева направо

Можно изменить порядок выполнения операций, используя скобки:

---

```
int a = 7;
int b = 4;
int c = a + b * 2;      // 15
int d = (a + b) / 2;    // 5
int e = a % b + c % d; // 3 + 0 = 3
```

---

## 4.2 Побитовые операции

**Побитовые операции**<sup>5</sup> – операции, производимые над цепочками битов. Выделяют два типа побитовых операций: логические операции и побитовые сдвиги. К логическим побитовым операциям относят следующие:

1. Побитовое И (&).
2. Побитовое ИЛИ (|).
3. Побитовое НЕ (~).
4. Исключающее ИЛИ (^).

Они используют те же таблицы истинности, что и их логические эквиваленты.

Существуют следующие операции побитового сдвига:

1. Побитовый сдвиг влево (<<).
2. Побитовый сдвиг вправо (>>).

### 4.2.1 Побитовое И

Таблица истинности для логического И (таблица 4.2). Запомнить её достаточно просто: если оба операнда равняются единице – то и значение равняется единице. В противном случае – ноль.

Выполним действие над двумя однобайтовыми целыми:  $a = 42$ ,  $b = 24$ :

$$a = 00101010_2 \quad b = 00011000_2$$

Запишем нулевой бит второго числа под нулевым битом первого числа, первый бит – под первым битом и т. д., и для всех пар битов выполним операцию согласно таблице 4.2 и результат представим на рисунке 4.1.

<sup>5</sup>Рекомендуется использовать беззнаковые целые в качестве операндов для побитовых операций.

Таблица 4.2: Таблица истинности для логического И

	A	B	A and B
	0	0	0
	0	1	0
	1	0	0
	1	1	1

&	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	1	0	1	0	1	0	0	0	0	1	1	0	0	0	$a = 42$ $b = 24$
0	0	1	0	1	0	1	0											
0	0	0	1	1	0	0	0											
	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	1	0	0	0	$f = 8$								
0	0	0	0	1	0	0	0											

Рис. 4.1: Побитовое И двух чисел

С операцией побитового И (`&`) существует один олимпиадный трюк. Чтобы проверить является ли число  $x$  степенью двойки или нулём, достаточно найти результат выражения

$$x \& (x - 1)$$

Если оно равно нулю – значит,  $x$  – степень двойки или ноль. В противном случае – что-то другое.

Выполним те же самые действия в языке программирования С:

---

```
char a = 42;
char b = 24;
char f = a & b;

printf("%d", f);
```

---

и на экране увидим значение 8.

### 4.2.2 Побитовое ИЛИ

У побитового ИЛИ (`|`) результирующий бит равен единице, если хотя бы один из битов равен единице. Если оба бита равны нулю – значит и результирующий бит равен нулю. Таблица истинности представлена ниже (таблица 4.4); на рисунке 4.2 можно найти результат побитового ИЛИ чисел  $a = 42$  и  $b = 24$ .

Таблица 4.3: Таблица истинности для логического ИЛИ

	A	B	A or B
	0	0	0
	0	1	1
	1	0	1
	1	1	1

	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	1	0	1	0	1	0	0	0	0	1	1	0	0	0	a = 42 b = 24
0	0	1	0	1	0	1	0											
0	0	0	1	1	0	0	0											
	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table>	0	0	1	1	1	0	1	0	f = 58								
0	0	1	1	1	0	1	0											

Рис. 4.2: Побитовое ИЛИ двух чисел

Код примера:

---

```
char a = 42;
char b = 24;
char f = a | b;

printf("%d", f);
```

---

На экране отобразится значение 58<sup>6</sup>.

#### 4.2.3 Побитовое НЕ

Таблица 4.4: Таблица истинности для логического НЕ

A	not A
0	1
1	0

Побитовое НЕ (`~`) – единственная унарная побитовая операция. Она инвертирует (меняет 0 на 1 и 1 на 0) биты операнда. Инвертировав 'ответ на главный вопрос жизни вселенной и всего такого' (значение 42)<sup>7</sup> получим значение 213:

	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table>	0	0	1	0	1	0	1	0	a = 42
0	0	1	0	1	0	1	0			
	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	1	1	0	1	0	1	0	1	f = 213
1	1	0	1	0	1	0	1			

Рис. 4.3: Побитовое НЕ

---

```
unsigned char a = 42;
unsigned char f = ~a;

printf("%d", f); // 213
```

---

<sup>6</sup>Работает? Не трогай. (Любой программист).

<sup>7</sup>Программисты часто делают ссылку к этому моменту фильма «Автостопом по галактике». Например, задают в качестве зерна генерации случайных чисел число 42.

#### 4.2.4 Побитовое исключающее ИЛИ

Таблица истинности исключающего ИЛИ составлена по простому принципу: выражение истинно, если операнды не совпадают (таблица 4.6).

Таблица 4.5: Таблица истинности для логического исключающего ИЛИ

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Выполним вручную операцию побитового исключающего ИЛИ ( $\wedge$ ) над известными нам числами и приведём код на языке программирования С:

$\wedge$	<table border="1"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	1	0	1	0	1	0	0	0	0	1	1	0	0	0	<table border="1"> <tr><td>a = 42</td></tr> <tr><td>b = 24</td></tr> </table>	a = 42	b = 24
0	0	1	0	1	0	1	0													
0	0	0	1	1	0	0	0													
a = 42																				
b = 24																				
	<table border="1"> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> </table>	0	0	1	1	0	0	1	0	<table border="1"> <tr><td>f = 50</td></tr> </table>	f = 50									
0	0	1	1	0	0	1	0													
f = 50																				

Рис. 4.4: Побитовое исключающее ИЛИ двух чисел

---

```
char a = 42;
char b = 24;
char f = a ^ b; // 50

printf("%d", f);
```

---

#### 4.2.5 Побитовый сдвиг вправо

Побитовый сдвиг числа  $n$  на  $m$  битов вправо обозначается как  $n \gg m$ . Выполним побитовый сдвиг числа 42 на 1 бит вправо:

<table border="1"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table>	0	0	1	0	1	0	1	0	<table border="1"> <tr><td>a = 42</td></tr> </table>	a = 42
0	0	1	0	1	0	1	0			
a = 42										
<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	0	0	0	1	0	1	0	1	<table border="1"> <tr><td>f = 42 &gt;&gt; 1 = 21</td></tr> </table>	f = 42 >> 1 = 21
0	0	0	1	0	1	0	1			
f = 42 >> 1 = 21										

Рис. 4.5: Побитовый сдвиг на 1 вправо

Все биты должны сдвинуться на один вправо. Самый крайний левый бит принимает значение 0. Для закрепления выполним побитовый сдвига числа 42 на 2 вправо:

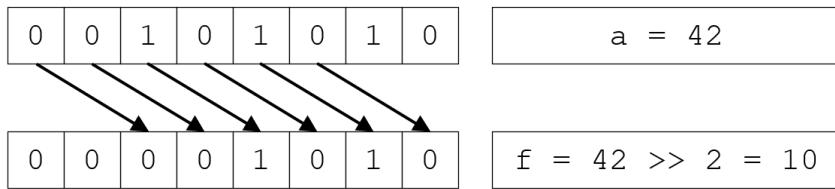


Рис. 4.6: Побитовый сдвиг на 2 вправо

Операция побитового сдвига числа  $n$  на  $m$  битов вправо эквивалентна операции целочисленного деления на  $2^m$ :

$$n \gg m \equiv n \text{ div } 2^m$$

Если комбинировать операцию побитового сдвига вправо и побитового И, можно узнать значение  $i$ -го бита числа  $n$ :

$$\text{bit} := n \gg i \& 1$$

#### 4.2.6 Побитовый сдвиг влево

Побитовый сдвиг числа  $n$  на  $m$  битов влево обозначается как  $n \ll m$ . Выполним побитовый сдвиг числа 42 на 1 бит влево:

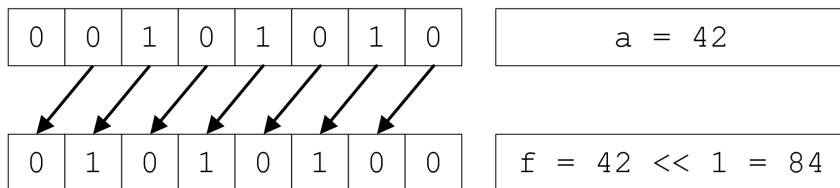


Рис. 4.7: Побитовый сдвиг на 1 влево

Все биты должны сдвинуться на один влево. Самый крайний правый бит принимает значение 0. Пример побитового сдвига числа 2 на 2 влево:

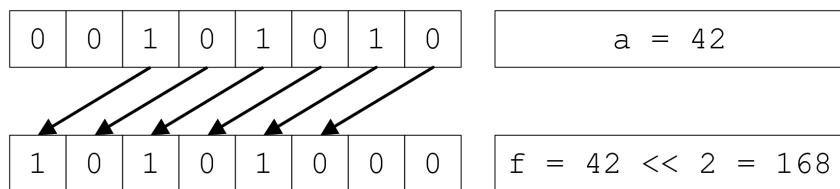


Рис. 4.8: Побитовый сдвиг на 2 влево

Операция побитового сдвига числа  $n$  на  $m$  битов влево эквивалентна операции умножения на  $2^m$ :

$$n \ll m \equiv n * 2^m$$

Если комбинировать операцию побитового сдвига влево и побитового ИЛИ, можно выставить значение  $i$ -го бита в единицу<sup>8</sup>:

$$n := n | (1 \ll i)$$

<sup>8</sup>Скобки в примере не являются обязательными.

			5    -.-					
n	1	0	0	1	1	0	1	1
$1 \ll i$	0	0	1	0	0	0	0	0
$n   1 \ll i$	1	0	1	1	1	0	1	1

Рис. 4.9: Установление  $i$ -го бита в единицу

Если комбинировать операцию побитового сдвига влево, побитового И и побитового НЕ, можно выставить значение  $i$ -го бита в ноль:

$$n := n \& \sim (1 \ll i)$$

		5    -.-						
n	1	0	1	0	1	0	1	1
$1 \ll i$	0	0	1	0	0	0	0	0
$\sim (1 \ll i)$	1	1	0	1	1	1	1	1
$n \& \sim (1 \ll i)$	1	0	0	0	1	0	1	1

Рис. 4.10: Установление  $i$ -го бита в ноль

В результате побитового сдвига вправо 'пропадают' старшие биты. Предположим, что стоит задача передвинуть данные биты в конец. Например, при сдвиге на 2 влево получалась бы следующая картина (рисунок 4.11)<sup>9</sup>:

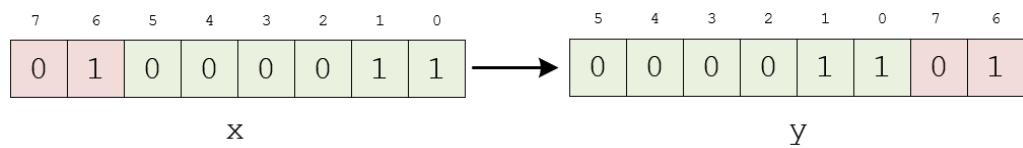


Рис. 4.11: Циклический сдвиг влево на 2

<sup>9</sup>Если хорошо понимать механику работы побитовых сдвигов влево (если сильно долго осуществлять сдвиг, рано или поздно будет получено значение 0), можно понять следующий анекдот.

Схлестнулись в схватке Добрыня Никитич и Змей Горыныч. Отрубает Добрыня Змею голову, на её месте вырастает две. Отрубает ему две головы – вырастает четыре. Отрубил в общей сложности 65535 голов и умер Змей. Потому что был шестнадцатиразрядным.

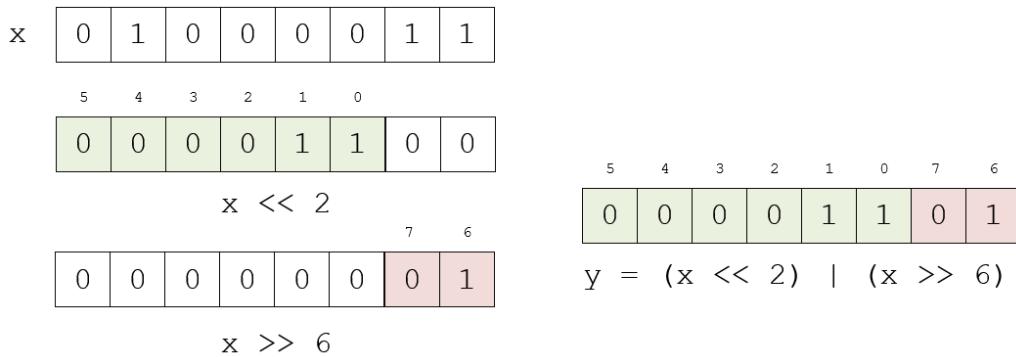


Рис. 4.12: Процесс вычисления побитового сдвига влево на 2

Если `x` имеет тип `unsigned char`:

$$x \text{ shl } j = (x \ll j) | (x \gg (8 - j))$$

Если рассматривать переменные произвольного целочисленного `unsigned` типа:

$$x \text{ shl } j = (x \ll j) | (x \gg (8 * \text{sizeof}(x) - j))$$

Циклический сдвиг вправо на 3:

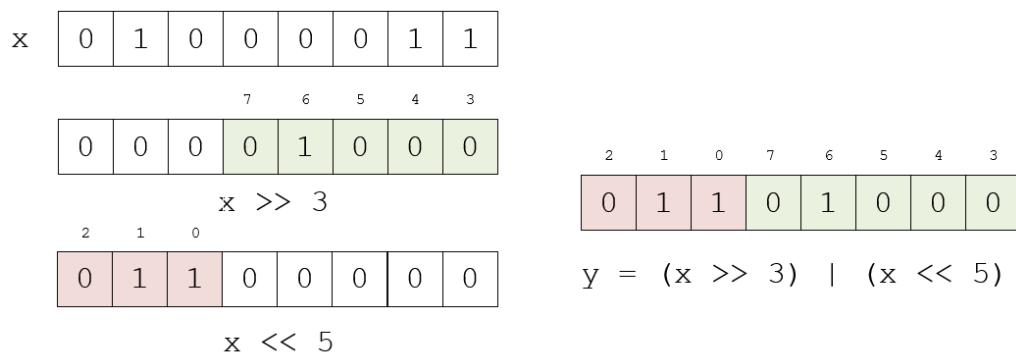


Рис. 4.13: Циклический сдвиг вправо на 3

Несложно показать, что циклический сдвиг вправо на  $j$  бит будет реализован как:

$$x \text{ shr } j = (x \gg j) | (x \ll (8 - j))$$

Если рассматривать переменные произвольного целочисленного `unsigned` типа:

$$x \text{ shr } j = (x \gg j) | (x \ll (8 * \text{sizeof}(x) - j))$$

#### 4.2.7 Приоритеты побитовых операций

Перечислим побитовые операции в порядке убывания приоритета:

Таблица 4.6: Приоритеты побитовых операций

Операция	Ассоциативность
<code>~</code>	Слева направо
<code>&lt;&lt;</code>	Слева направо
<code>&amp;</code>	Слева направо
<code>^</code>	Слева направо
<code> </code>	Слева направо

## 4.3 Логические операции и операции сравнения

Операции сравнения интуитивно понятны. К ним относятся (перечислим в порядке убывания приоритета):

- `>` (больше), `<` (меньше), `>=` (больше или равно), `<=` (меньше или равно).
- `==` (равно), `!=` (не равно).

Стоит обратить внимание на запись операции равно (два знака `=`). В качестве результата операции вернётся значение 1 (если выражение истинно) или 0 (если ложно).

Классическая ошибка начинающих программировать на С:

---

```
int a = 5;

if (a = 10)
    printf("a = 10");
else
    printf("a != 10");
```

---

Вместо сравнения со значением 10, осуществляется присваивание переменной `a` значения 10. Значения, отличные от нуля, являются истиной. Поэтому на экране можно увидеть сообщение "a = 10".

Остановимся на вопросе сравнения вещественных чисел на равенство<sup>10</sup>:

---

```
if (0.1 + 0.1 + 0.1 == 0.3)
    printf("==");
else
    printf("!=");
```

---

Программа выведет, что данные значения неравны<sup>11</sup>. Как жить с этим? – находить модуль разности между левой и правой частью, и сравнивать с некой допустимой погрешностью:

---

```
float a = 0.3;
float b = 0.1 + 0.1 + 0.1;
float eps = 0.00000001;

if (fabs(a - b) < eps)
    printf("==");
else
    printf("!=");
```

---

По правде говоря, работать со сравнением вещественных приходится редко. Но если такое случится, вы должны знать, как действовать.

В языке С определены три логических операций (перечислим их в порядке убывания приоритета):

- `!` - отрицание

<sup>10</sup>Пример в следующем листинге работает по-разному на разных ЭВМ. Так что результаты могут отличаться.

<sup>11</sup>В теории, теория и практика неразделимы. Но на практике это не так (Йоги Берра).

- `&&` - логическое И (конъюнкция)
- `||` - логическое ИЛИ (дизъюнкция)

Когда определяется значение логического выражения, вычисления выполняются по 'ленивой' схеме слева направо: если в процессе вычисления выражения уже можно понять результат, расчёты прекращаются<sup>12</sup>. Пример:

---

```
int a = 10;
int b = a > 1 || a < -1;
```

---

Будет вычислена истинность выражения  $a > 1$ . Мы уже можем сказать, что всё выражение – истина. Какой смысл считать дальше? Аналогично и с логическим И:

---

```
int a = 10;
int b = a > 10 && a < 20; // сравнение a < 20 не будет производиться,
                           // так как должно a > 10
```

---

Проверить год на високосность можно следующим образом:

---

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
    printf("високосный год");
else
    printf("невисокосный год");
```

---

Проверка на то является ли символ *c* цифрой:

---

```
int is_digit = c >= '0' && c <= '9';
```

---

Можно встретить комбинирование операций сравнения с арифметическими. Предположим, нам надо выполнить округление дроби  $a/b$  вверх:

---

```
int a = 7;
int b = 4;
int r = a / b + (a % b != 0);
```

---

Если не поставить скобки в прошлом выражении, оно будет интерпретировано как

---

```
int r = (a / b + a % b) != 0;
```

---

## 4.4 Приоритеты операций

Перечислим операции в порядке убывания приоритета. Не о всех операциях была речь, но о них будет сказано дальше. Приоритеты представлены в таблице 4.7.

---

<sup>12</sup>Если существует вероятность срабатывания ленивой схемы, рекомендуется располагать выражения исходя из следующих соображений:

1. Ставиться увеличить вероятность срабатывания 'ленивых' вычислений.
2. Располагать в правой части более нагруженные выражения (а вдруг их считать не придётся).

Таблица 4.7: Приоритеты операций в С

Приоритет	Операция	Описание	Ассоциативность
1	<code>++</code> <code>--</code> <code>()</code> <code>[]</code> <code>-&gt;</code> <code>.</code>	Постфиксный инкремент Постфиксный декремент Вызов функции и подвыражений Обращение к элементу массива Обращение к полю с разыменованием Доступ к полю структуры	Слева направо
2	<code>++</code> <code>--</code> <code>!</code> <code>~</code> <code>-</code> <code>+</code> <code>(type)</code> <code>*</code> <code>&amp;</code> <code>sizeof</code>	Префиксный инкремент Префиксный декремент Логическое отрицание Побитовое НЕ Унарный минус Унарный плюс Приведение типов Разыменование указателя Операция взятия адреса Вычисление размера	Справа налево
3	<code>*</code> <code>/</code> <code>%</code>	Умножение Деление Остаток от деления	Слева направо
4	<code>+</code> <code>-</code>	Сложение Вычитание	Слева направо
5	<code>&lt;&lt;</code> <code>&gt;&gt;</code>	Побитовый сдвиг влево Побитовый сдвиг вправо	Слева направо
6	<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code>	Меньше Меньше или равно Больше Больше или равно	Слева направо
7	<code>==</code> <code>!=</code>	Равно Не равно	Слева направо
8	<code>&amp;</code>	Побитовое И	Слева направо
9	<code>^</code>	Побитовое исключающее ИЛИ	Слева направо
10	<code> </code>	Побитовое ИЛИ	Слева направо
11	<code>&amp;&amp;</code>	Логическое И	Слева направо
12	<code>  </code>	Логическое ИЛИ	Слева направо
13	<code>?:</code>	Тернарная операция	Слева направо
14	<code>=</code> <code>op=</code>	Присваивание Корректирующее присваивание	Справа налево
15	<code>,</code>	Запятая	Слева направо

Язык программирования С не фиксирует очередьность вычисления операндов оператора (за исключением `&&`, `||`, `?:`, `,`):

```
int a = b * c + d * e
```

Не гарантируется, что `b * c` будет вычислено раньше, чем `d * e`.

Так же не определена очерёдность вычисления аргументов функций:

---

```
int a = 5;
printf("%d %d", a, ++a) // может вывести 5 6, а может и 6 6.
```

---

## 4.5 Приведение типов

Приведение типов бывает явным и неявным. Неявное приведение происходит автоматически, явное – по требованию программиста.

### 4.5.1 Неявное приведение типов

Неявное приведение типов выполняется в следующих случаях:

- после вычисления операндов бинарных арифметических, логических, битовых операций, операций сравнения, а также 2-го или 3-го операнда операции `? :`; значения операндов приводятся к одинаковому типу;
- перед выполнением присваивания;
- перед передачей аргумента функции;
- перед возвратом функцией возвращаемого значения;
- после вычисления выражений конструкций `if`, `for`, `while`, `do-while`.

Все используемые типы могут быть проранжированы:

char	short	int	long	long long	float	double	long double
unsigned char	unsigned short	unsigned int	unsigned long	unsigned long long			
Ранг 1	Ранг 2	Ранг 3	Ранг 4	Ранг 5	Ранг 6	Ранг 7	Ранг 8

В выражениях могут участвовать целочисленные операнды, чей тип меньше, чем `int`. В языке С имеется механизм целочисленного повышения: если какой-то операнд имеет тип меньший, чем `int`, выполняется целочисленное повышение<sup>13</sup>. Если `int` не способен представить значение операнда – происходит повышение до `unsigned int`<sup>14</sup>.

---

```
char c = '?';
unsigned short var = 100;

if (c < 'A')          // Символьная константа 'A' имеет тип int;
                      // Значение переменной с продвигается до int
```

---

<sup>13</sup>Объекты целочисленного типа можно преобразовать в другой более широкий целочисленный тип, то есть тип, который может представлять больший набор значений. Этот расширяющий тип преобразования называется **целочисленным повышением**.

<sup>14</sup>Если `int` и `short` имеют одинаковый размер (например, на 16-битных машинах), тогда `signed int` может быть недостаточно для представления всех значений `unsigned short`. В этом случае переменная `var` будет повышена до `unsigned int`.

```
// для выполнения сравнения.

var = var + 1; // До сложения, значение переменной var будет продвинуто
// до int или unsigned int.
```

Для арифметических операций верно следующее:

- Если какой-либо из операндов имеет тип с плавающей запятой, то операнд с более низким рангом преобразования преобразуется в тип с таким же рангом, что и другой operand.
- Если оба операнда являются целыми числами, целочисленное продвижение сначала выполняется для обоих operandов. Если после целочисленного продвижения operandы по-прежнему имеют разные типы, преобразование продолжается следующим образом:
  - Если один operand имеет беззнаковый тип T, ранг преобразования которого не меньше, чем у другого типа operand, то другой operand преобразуется в беззнаковый тип T.
  - Если один operand имеет знаковый тип T, ранг преобразования которого выше, чем у другого типа operand, тогда другой operand преобразуется в тип T только в том случае, если тип T может представлять все значения своего предыдущего типа. В противном случае оба operandы преобразуются в тип без знака, соответствующий типу со знаком T.

Рассмотрим правила на примерах. Неустаревающая классика:

---

```
unsigned a = 1;
if (a > -1)
    printf("a > -1");
else
    printf("?!");
```

---

Здесь переменная `a`, которая имеет тип `int`, была преобразована к типу `unsigned int`<sup>15</sup>. И тогда будут сравниваться не 1 и -1, а 1 и 4294967295:

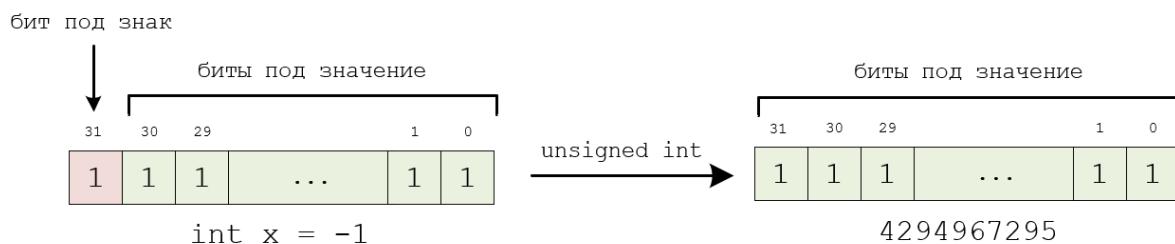


Рис. 4.14: Преобразование из `int` в `unsigned int`

Так как условие ложно – будет выполнена ветка `else`.

<sup>15</sup>Существуют разные мнения, по поводу использования `unsigned`. Большинство программистов сходятся на том, что не так уж и велик выигрыш от одного бита (если перевести его под хранение значения). Поэтому нечастой практикой является использование беззнаковых типов.

Второй случай:

---

```
int a = 10;
float k = 2;
int b = 1.24 + a * k;
```

---

В вычислении значения переменной `b` участвуют три операнда: `1.24` типа `double`, `a` типа `int` и `k` типа `float`.

Если следовать приоритету операций, сперва выполнится умножение. Умножаются две переменных различных типов. Тогда после приведения переменная `a` будет иметь тип `float`. Произойдёт перемножение двух вещественных, и тип результата произведения - `float` (20.0).

Последнее действие для вычисления выражения – поиск суммы. Складываются `double` и `float`. Второй operand будет преобразован к `double`. Результат вычисления будет иметь тот же самый тип (значение выражения: 21.24).

Перед присваиванием будет произведено ещё одно приведение типов (`int` не может вместить в себя вещественное число). У результата будет отброшена вещественная часть (ответ: 21).

Ещё один пример:

---

```
int i = -1;
unsigned limit = 200U;
long n = 30L;

if (i > limit)    // (int > unsigned int) -> (unsigned int > unsigned int) ->
                   // (-1 > 200) -> (4294967295 > 200) -> 1

x = limit * n; // вариант будет выбран в зависимости от архитектуры
                // вариант 1 - если long может уместить значения unsigned
                //      int : (unsigned * long) -> (long * long)
                // вариант 2 - если long не может уместить значения
                //      unsigned int : (unsigned * long) -> (unsigned long *
                //      unsigned long)
```

---

Предположим, что нам нужно решить квадратное уравнение. Имеется ограничение: коэффициенты уравнения `a`, `b`, `c` должны иметь тип `int`. Возможное решение выглядит следующим образом:

---

**Листинг 5** Решение квадратного уравнения
 

---

```
#include <stdio.h>
#include <math.h>

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

    long long D = (long long)b*b - (long long)4*a*c;
    if (D > 0) {
        float sqrtD = sqrt(D);
        float x1 = (-b + sqrtD) / (2 * a);
        float x2 = (-b - sqrtD) / (2 * a);

        printf("%f %f", x1, x2);
    } else if (D == 0) {
        float x = -b / (2.0 * a);

        printf("%f", x);
    } else
        printf("No roots");

    return 0;
}
```

---

Приведения:

---

```
long long D = (long long)b*b - (long long)4*a*c;
```

---

выполнены для того, чтобы бороться с переполнением в процессе выполнения арифметических операций. Если `a`, `b`, `c` были бы переменными типа `long long` явные приведения не потребовались бы, так как расчёты проводились в типе `long long`.

Проанализируем строку 14 из прошлого примера, только вместо переменных и литералов будем указывать типы. На каждой последующей строчке укажем как меняется ситуация с типами данных по ходу вычисления выражения:

---

```
float x1 = (-b + sqrtD) / (2 * a);
// (-int + float) / (int * int) -> (-float + float) / int ->
// float / int -> float / float -> float
```

---

Проанализируем и другие строки таким образом:

---

```
float x = -b / (2.0 * a);
// -int / (double * int) -> -int / (double * double) ->
// -int / double -> -double / double -> double
```

---

Когда значение будет вычислено, выполнится приведение типов при присваивании.

Рассмотрим ещё один исключительно учебный пример:

---

```
char ch;
int i;
float f;
double d;

int result = (ch / i) + (f * d) - (f + i);
// int = (char / int) + (float * double) - (float + int);
// int = (int / int) + (double * double) - (float + float);
// int = int + double - float;
// int = double + double - float;
// int = double - float;
// int = double - double;
// int = double;
```

---

За счёт последнего приведения типов вещественная часть будет отброшена.

О неявных приведениях, происходящих при работе с функциями, подробно описан в главе Рекурсия.

#### 4.5.2 Явное приведение типов

Для любого выражения можно явно указать преобразование типа, используя унарный оператор, называемый **приведением**. Конструкция вида

---

```
// (<имя типа>) <выражение>
```

---

приводит выражение к указанному в скобках типу. Решить проблему с переполнением на странице 66 в одном из прошлых случаев можно так:

---

```
int a, b;
scanf("%d %d", &a, &b);

long long c = (long long)a * b;
```

---

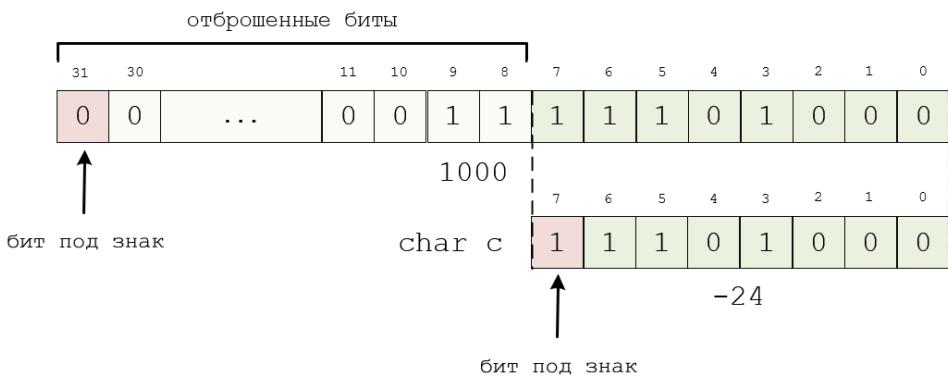
#### 4.5.3 Последствия при приведении типов

Если целое большего размера приводится к целому меньшего размера, то старшие байты отбрасываются:

---

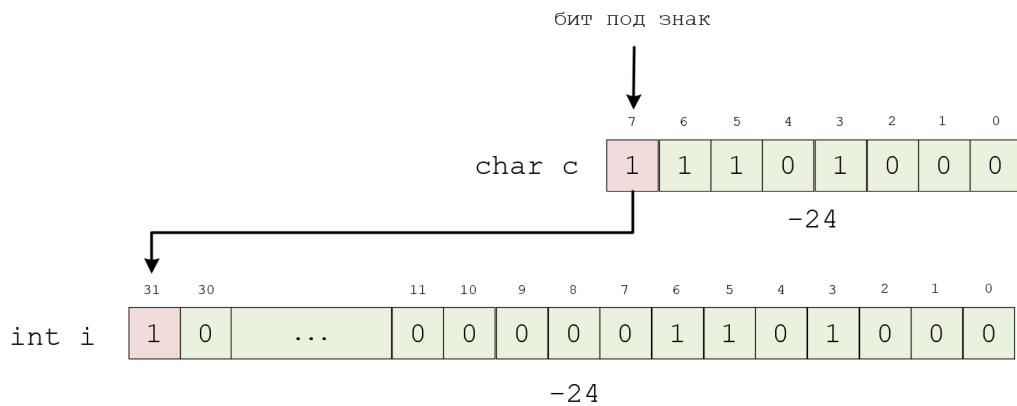
```
char c = 1000; // c = -24
```

---

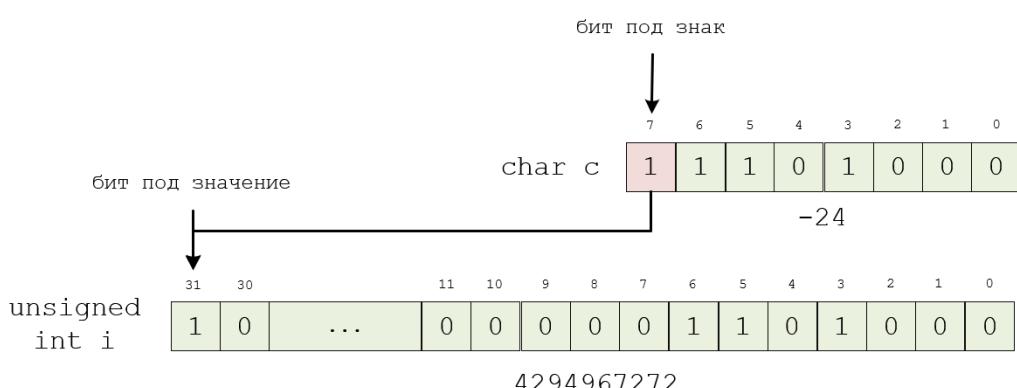


Целое меньшего размера преобразуется к целому типу большего размера, не изменяя своего значения с сохранением знака:

```
char c = -24;  
int i = c;
```



```
char c = -24;  
unsigned int i = c;
```



Вещественное значение преобразуется к целому отбрасыванием знаков после точки, если такое значение может быть присвоено целому:

```
// во всех случаях отбрасывается вещественная часть  
int i1 = 100.25; // 100
```

---

```
int i2 = 1.99; // 1
int i3 = 300.00; // 300
```

---

Если вещественное значение не может быть присвоено целому, то результат не определен:

---

```
unsigned int a = -1000.1234;
printf("%u", a); // на моей машине получено значение 0
```

---

При приведении целого типа к вещественному целое заменяется одним из ближайших к этому целому вещественным:

---

```
float f = 123456789;
printf("%f", f); // 123456792.000000
```

---

Вещественное большей точности преобразуется к вещественному меньшей точности приближенно, если оно может быть представлено в типе с меньшей точностью:

---

```
float f = 1000.1234; // 1000.123413
float f2 = 1000.1234f; // 1000.123413
double lf = 1000.1234; // 1000.123400
```

---

Приведение вещественного числа меньшей точности к вещественному большей точности, происходит без искажений:

---

```
float f = 1234.012f; // f = 1234.011963
double lf = f; // lf = 1234.011963
```

---

## Резюме

- Операция – это некоторое действие, которое выполняется над операндом.
- В зависимости от количества operandов, операции в С делятся на унарные, бинарные, тернарные.
- Арифметические операции и их приоритеты аналогичны тем, что приняты в математике.
- Побитовые операции делятся на два типа: логические побитовые операции и побитовые сдвиги.
- Логические побитовые операции используют те же таблицы истинности, что и их логические эквиваленты.
- Операции сравнения нам интуитивно понятны. Результатом этих операций может быть значение 1 (если выражение истинно) или 0 (если ложно).
- Определение значения логического выражения выполняется по 'ленивой' схеме слева направо: если в процессе вычисления выражения уже можно понять результат, расчёты прекращаются.
- С не фиксирует очередность вычисления operandов оператора и аргументов функций.
- Приведение типов бывает явным и неявным. Неявное приведение происходит автоматически, явное – по требованию программиста.
- При написании кода необходимо тщательно присматриваться к типу переменной и потенциальным значениям, получаемых в ходе вычислений, чтобы избежать возможных ошибок.

## Контрольные вопросы

1. Определения «Операция», «Операнд». Классификация операций по количеству operandов. Определение «Выражение». Что определяет выражение?
2. Арифметические операции. Приоритет арифметических операций. Переполнение целочисленных типов при выполнении арифметических операций.
3. Префиксные и постфиксные инкременты и декременты. Приведите примеры, в которых использование префиксной и постфиксной формы дают различные результаты.
4. Операция деления и остатка от деления в языке программирования С.
5. Определение «Побитовые операции». Классификация побитовых операций и их приоритеты.
6. Побитовое И. Проверка числа  $x$  на чётность / степень двойки с использованием &.
7. Побитовое ИЛИ, побитовое исключающее ИЛИ, побитовое НЕ.

8. Побитовые сдвиги влево и вправо.
9. Установка  $i$ -го бита в значения 0 и 1. Инвертирование  $i$ -го бита.
10. Реализация циклического сдвига влево и вправо.
11. Логические операции. Приоритеты логических операций.
12. Операции сравнения. Приоритеты операций сравнения. Сравнение вещественных operandов.
13. Ленивая схема вычислений. Рекомендации по использованию ленивой схемы.
14. Приоритеты операций. Определение «ассоциативность».
15. Приведения типов. Явные и неявные приведения типов. В каких случаях происходит неявное приведение типов?
16. Последствия при приведении типов.

# Глава 5

## Разветвляющиеся алгоритмы

Операторы являются основными строительными блоками программы. **Программа** – это последовательность операторов. Оператор представляет собой завершенную инструкцию для компьютера. В языке программирования С операторы распознаются по наличию ; в конце.

Как говорилось ранее, количество алгоритмов, которые имели бы линейную структуру невелико. Больший интерес будут представлять алгоритмы разветвляющейся структуры.

**Разветвляющимися алгоритмами** являются алгоритмы, в которых имеются шаги, которые выполняются или не выполняются в зависимости от каких-либо условий.

Для организации развлок в языке С используются условный оператор `if` и оператор множественного ветвления `switch`.

### 5.1 Условный оператор `if`

Знакомство с условным оператором `if` проще начать с решения задач.

#### 5.1.1 Поиск модуля числа

##### Поиск модуля числа

С клавиатуры вводится число  $a$ . Необходимо вывести на экран модуль числа  $a$  (Изменение считанного с клавиатуры значения для данной задачи допустимо).

Из школьного курса математики известно, что значение модуля всегда положительно. Если было введено положительное значение – его оставляем без изменения. Если было введено отрицательное значение – необходимо поменять его знак. Решение описано на рисунке 5.1.

Можем заметить наличие блока 'решение' в котором указывается логическое выражение. **Логическое выражение** – выражение, результатом вычисления которого является значение 'истина' или 'ложь'.

При проектировании алгоритма важно уделить внимание следующему аспекту: для блока 'решение' ветка '+' не может быть пустой. Конечно же вы можете сделать иначе, но это создаст трудности при переводе блок-схемы в код.

Каким же образом кодируется данная конструкция? Шаги следующие:

1. Указывается ключевое слово `if`, за которым следует логическое выражение в круглых скобках.

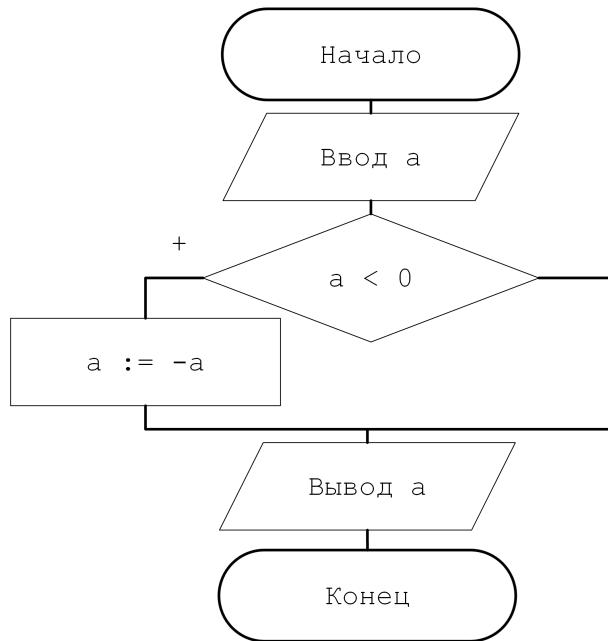


Рис. 5.1: Блок-схема поиска значения модуля числа

2. Ставятся фигурные скобки.

3. В фигурных скобках одна за другой перечисляются операторы по ветке '+'.  
Если вы умеете исполнять алгоритмы в словесном виде, получится следующее:

---

```

if (a < 0) {
    a = -a;
}
  
```

---

По правде говоря, для одного оператора скобки не нужны. А если дописать код, получим:

---

```

#include <stdio.h>

int main() {
    int a;
    scanf("%d", &a);

    if (a < 0)
        a = -a;

    printf("%d", a);

    return 0;
}
  
```

---

Общая форма оператора if:

---

```

if (<логическое выражение>)
    <оператор>
  
```

---

Если <логическое выражение> является истинным, то <оператор> выполняется. <Оператор> может быть как одиночным, так и составным<sup>1</sup>:

---

```
if (a < 0)
    a = -a;
if (b > 0) {
    a = b * 2;
    b = a + 3;
}
```

---

Вся управляющая структура с оператором `if` считается одним оператором.

Отступы в языке С не являются обязательными, однако они зрительно выделяют операторы, выполнение которых зависит от условия проверки.

Одной из распространённых ошибок является пропуск фигурных скобок в составном операторе:

---

```
if (a < 0)
    a = -a;
printf("%d", a);
```

---

Автор программы (судя по отступам) хотел в случае отрицательного значения переменной `a` поменять знак и вывести результат на экран. Но данный фрагмент будет интерпретирован так:

---

```
if (a < 0)
    a = -a;
printf("%d", a);
```

---

то есть значение переменной `a` будет выведено в любом случае. Не забывайте ставить фигурные скобки.

### 5.1.2 Поиск максимального значения из двух

#### Поиск максимального значения из двух

С клавиатуры вводятся значения переменных  $a$  и  $b$ . Необходимо найти максимальное значение (если значения равны, в качестве максимума может выступать любое из значений).

Опишем алгоритм при помощи блок-схемы (рисунок 5.2). В отличие от прошлой задачи видим, что и по второй ветке имеются операции. Необходимо будет сделать чуть больше действий для перевода конструкции в код:

1. Указывается ключевое слово `if`, за которым следует логическое выражение в круглых скобках.
2. Ставятся фигурные скобки.
3. В фигурных скобках одна за другой перечисляются инструкции по ветке '+'.

<sup>1</sup>Составной оператор – это два или большее количество операторов, сгруппированных вместе путём помещения их в фигурные скобки; его также называют **блоком**.

4. После второй закрывающей скобки пишется инструкция `else`, после которой ставятся фигурные скобки.
5. В фигурных скобках одна за другой перечисляются инструкции по ветке '-'.

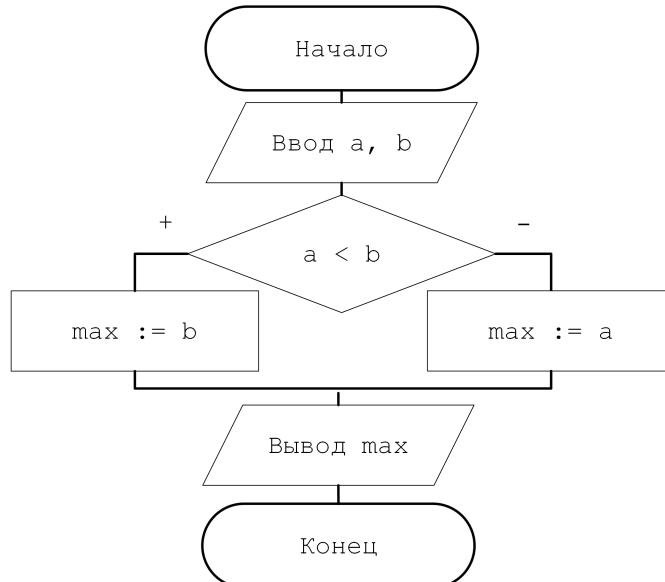


Рис. 5.2: Блок-схема к задаче о максимальном значении из двух

Оставлю несколько замечаний:

- Если говорить о блок-схеме, обозначение '-' не всегда ставится, лишь бы был '+'.
- Иногда не ставят обозначение '+', если занята только одна ветка.
- В качестве ветки '+' по умолчанию принимается левая ветка.

Получаем следующий код (стоит обратить внимание, что ввод данных, обработка, и вывод разделены пустой строкой – это хорошая практика):

---

#### Листинг 6 Поиск максимального значения из двух

---

```

#include <stdio.h>

int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    // фигурные скобки необязательны
    int max;
    if (a < b)
        max = b;
    else
        max = a;

    printf("%d", max);

    return 0;
}
  
```

---

Общая форма оператора `if-else`:

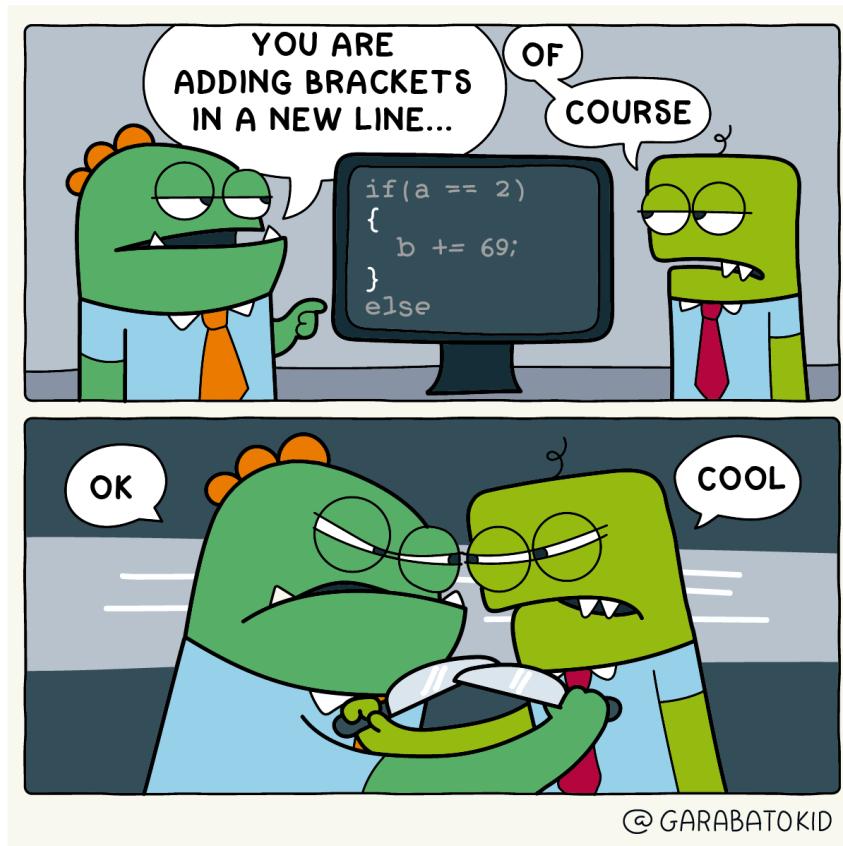
---

```
// if (<логическое выражение>)
//     <оператор1>
// else
//     <оператор2>
```

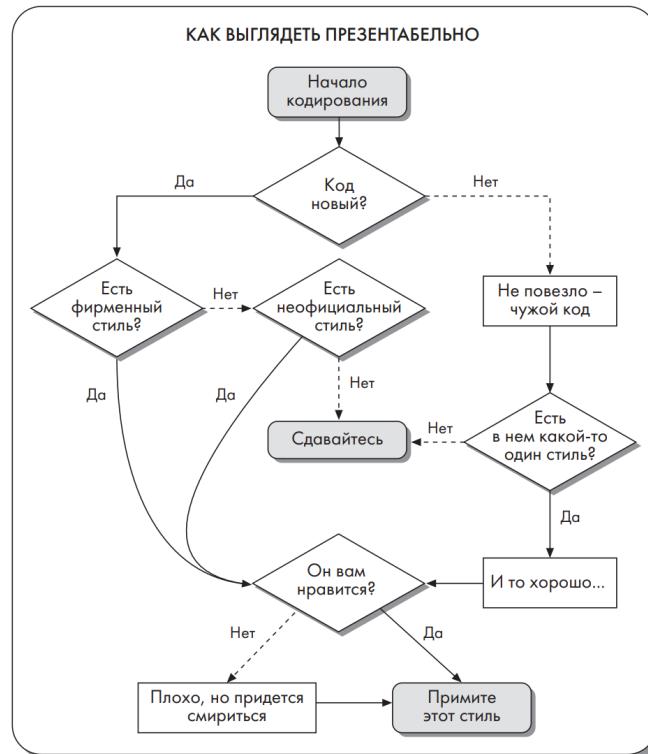
---

Данная конструкция выполняется по следующим правилам:

1. Вычисляется значение `<логического выражения>`.
2. Если оно является истиной, выполняется `<оператор1>`.
3. В противном случае выполняется `<оператор2>`.



Уже прошла не одна война о том, каким образом расставлять скобки и вообще какого стиля придерживаться, приведу рекомендацию Питера Гудлифа из книги "Ремесло программиста":



### 5.1.3 Поиск максимального значения из трёх

#### Поиск максимального значения из трёх

С клавиатуры вводятся значения переменных  $a$ ,  $b$ ,  $c$ . Необходимо найти максимальное значение из трех.

Одна из моих любимых задач. Когда студенты её решают, они показывают мне несколько вариантов решений. Рассмотрим их.

Первый вариант таков (рисунок 5.3):

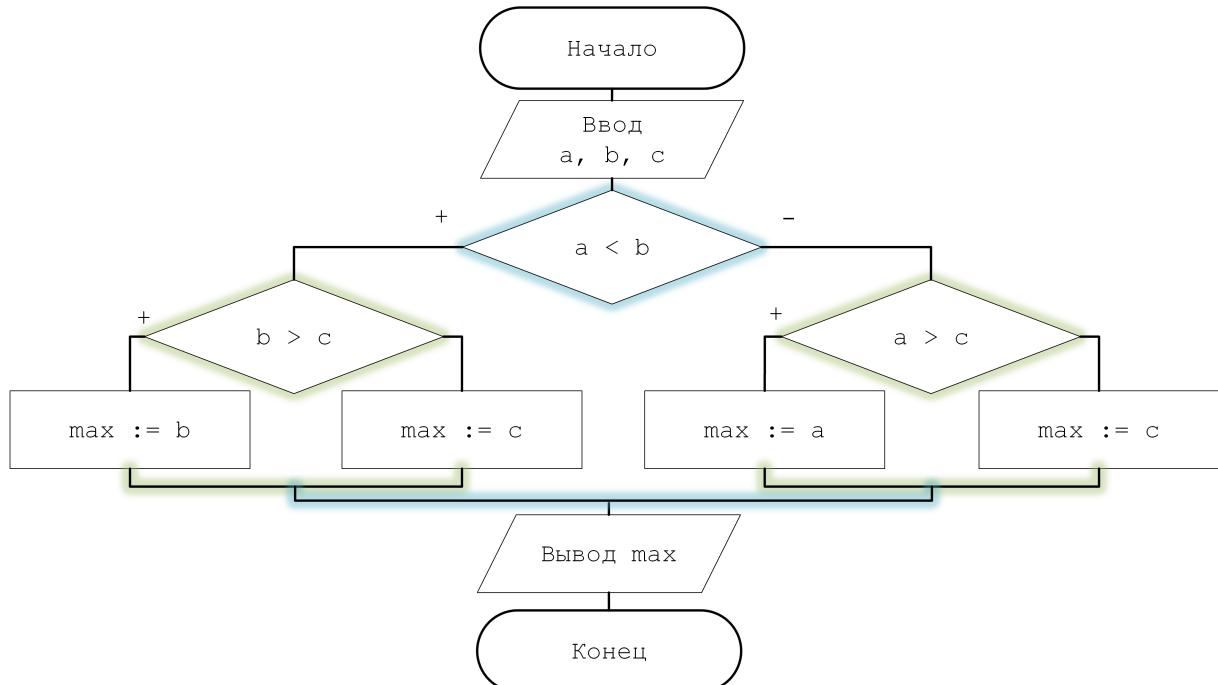


Рис. 5.3: Блок-схема для решения задачи поиска максимума из двух

Главная его проблема – наличие большого уровня вложенности. Представьте, что вам потребуется найти максимум из четырёх значений. Уровень вложенности бы увеличился, что является определенной проблемой. Но если мы имеем такую вложенную конструкцию, будет полезным понять, каким образом 'закрываются' блоки 'решение': вначале закроятся внутренние (вложенные) блоки, и только потом – внешние.

На самом деле, такая вложенность в некоторых задачах может быть даже лучшим вариантом. Например: заданы координаты  $x$  и  $y$  точки на плоскости. Гарантируется, что каждая из координат не равна нулю. Определить номер четверти, в которой оказалась точка. Попробуйте выполнить её решение самостоятельно.

Код решения прошлой задачи представлен на листинге ниже:

---

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

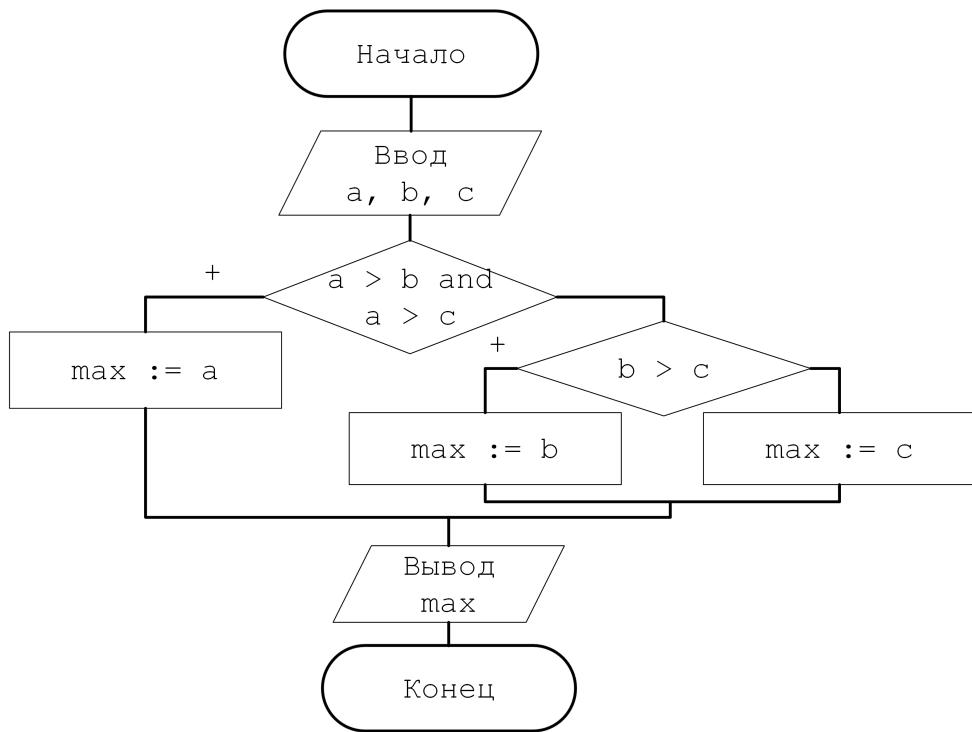
    int max;
    if (a < b) {
        if (b > c)
            max = b;
        else
            max = c;
    } else {
        if (a > c)
            max = a;
        else
            max = c;
    }

    printf("%d", max);

    return 0;
}
```

---

В следующем варианте условие из крайнего левого блока 'перешло' в первый блок 'решение' (это избавляет нас от положения, что в примере до дважды встречалось  $\max = c$ ), но проблема с расширением никуда не исчезнет:



Код программы:

---

```

#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

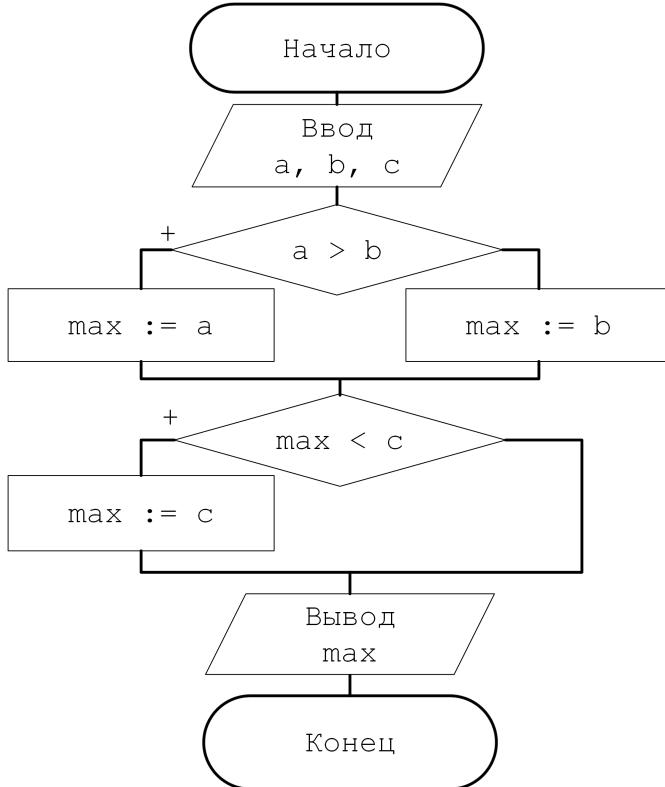
    int max;
    if (a > b && a > c)
        max = a;
    else {
        if (b > c)
            max = b;
        else
            max = c;
    }

    printf("%d", max);

    return 0;
}
  
```

---

Третий вариант решения:



выглядит заметно лучше. Блоков и вложенности стало меньше. Меньше блоков – меньше кода. Меньше кода – меньше ошибок. Меньше ошибок – меньше работы. Быстрее закончится работа – больше свободного времени<sup>2</sup>.

---

```

#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

    int max;
    if (a > b)
        max = a;
    else
        max = b;

    if (max < c)
        max = c;

    printf("%d", max);

    return 0;
}
  
```

---

Иногда нам будут попадаться такие конструкции, когда значение переменной зависит от значения логического выражения:

<sup>2</sup>Есть два подхода к программированию. Первый – сделать программу настолько простой, чтобы в ней очевидно не было ошибок. А второй – сделать её настолько сложной, чтобы в ней не было очевидных ошибок (Тони Хоар)

---

```
if (a > b)
    max = a;
else
    max = b;
```

---

которые лучше закодировать при помощи тернарной операции<sup>3</sup>:

---

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

---

Используйте тернарную операцию для таких случаев.

Тернарная операция принимает три операнда:

---

```
<логическое выражение> ? <выражение1> : <выражение2>;
```

---

1. <логическое условие> (его можно не заключать в скобки), за которым следует знак вопроса ?.
2. <выражение1>, вычисленное значение которого будет возвращено, если условие истинно. Далее ставится двоеточие :.
3. <выражение2>, вычисленное значение которого будет возвращено, если условие ложно.

Опишем последнее решение. Его код и блок-схема (рисунок 5.4) представлены ниже.

### Листинг 7 Поиск максимального значения из трёх

---



---

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

    int max = a;
    if (max < b)
        max = b;
    if (max < c)
        max = c;

    printf("%d", max);

    return 0;
}
```

---

<sup>3</sup>Было время, когда продуктивность программиста определялась количеством написанных строк кода. Билл Гейтс как-то сказал: "Измерять продуктивность программирования подсчетом строк кода — это так же, как оценивать постройку самолета по его весу".

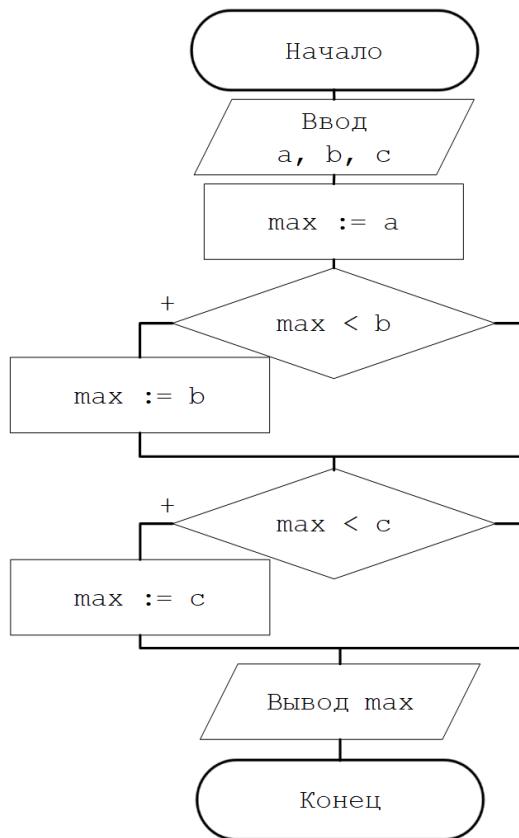


Рис. 5.4: Блок схема алгоритма поиска максимального значения из трёх

#### 5.1.4 Упорядочивание двух чисел

##### Упорядочивание двух чисел

Даны переменные  $a$  и  $b$ . Необходимо упорядочить их значения по неубыванию.

Код функции `main` представлен ниже (блок-схема на рисунке 5.5):

---

##### Листинг 8 Упорядочивание двух чисел

---

```

#include <stdio.h>

int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    if (a > b) {
        int t = a;
        a = b;
        b = t;
    }

    printf("%d %d", a, b);

    return 0;
}
  
```

---

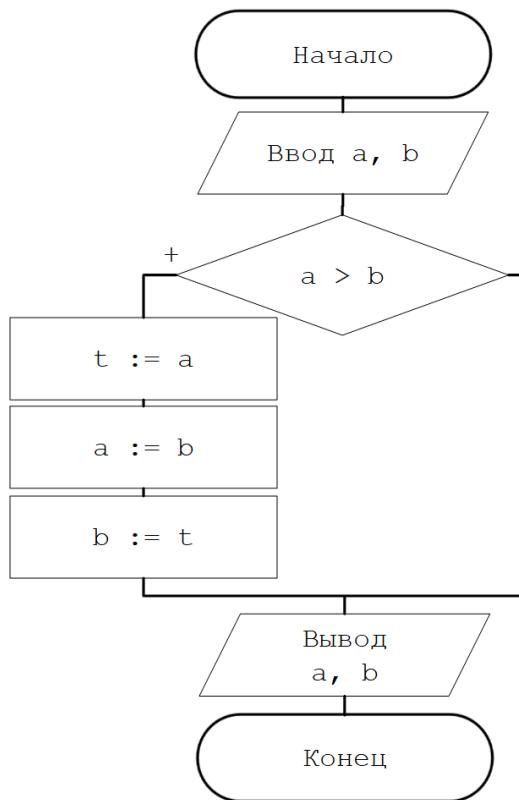


Рис. 5.5: Блок-схема алгоритма упорядочивания двух чисел

### 5.1.5 Решение линейного уравнения

#### Решение линейного уравнения

Решить уравнение  $ax + b = 0$ .

Блок-схема представлена на странице 16, код – на странице ??.

## 5.2 Цепочки операторов *if-else-if*

### 5.2.1 Задача о социальной поддержке

#### Задача о социальной поддержке

Согласно законопроекту в стране  $N$  вводятся следующие меры поддержки в виде единовременной выплаты:

- Для лиц младше 18 лет – 15000 руб.
- Для тех, кому больше 18, но не исполнилось 60 лет – 10000 руб.
- От 60 до 70 лет – 20000 руб.
- От 70 и более – 30000 руб.

Напишите программу, которая по возрасту определяет, какую сумму необходимо выплатить.

Можно построить следующую блок-схему:

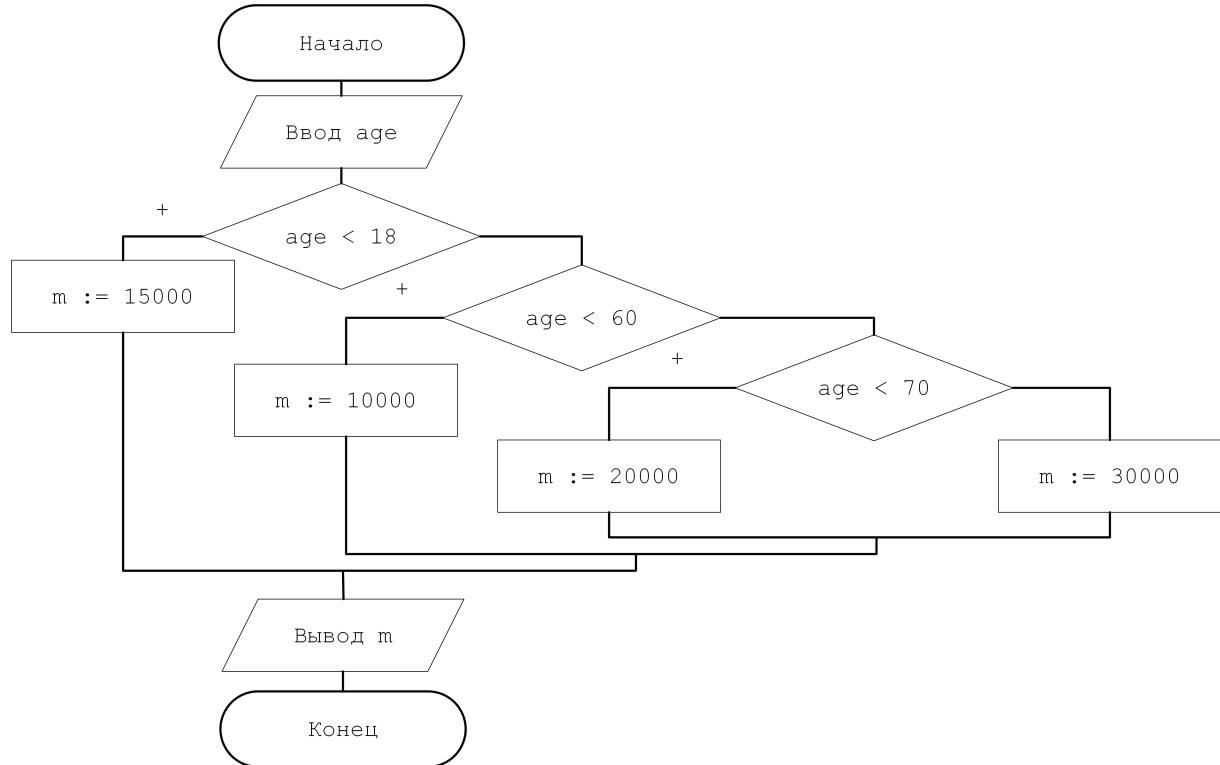


Рис. 5.6: Блок-схема алгоритма решения задачи о социальной поддержке

И это действительно похоже на правду. Рассмотрим нюанс, который касается непосредственно кодирования. Исходя из того, что было изучено, код для центральной части блок-схемы может быть написан так:

---

```

if (age < 18)
    m = 15000;
else {
    if (age < 60)
        m = 10000;
    else {
        if (age < 70)
            m = 20000;
        else
            m = 30000;
    }
}
  
```

---

Вспомним, что конструкция `if-else` является оператором. И мы могли бы не писать скобки на 6 и 11 строке и получить:

---

```

if (age < 18)
    m = 15000;
else {
    if (age < 60)
        m = 10000;
    else
        if (age < 70)
  
```

---

```

    m = 20000;
else
    m = 30000;
}

```

---

Но если следовать такой логике, можно убрать скобки на 3 и 11 строке:

```

if (age < 18)
    m = 15000;
else
    if (age < 60)
        m = 10000;
    else
        if (age < 70)
            m = 20000;
        else
            m = 30000;
}

```

---

Однако в таком виде код пишут редко (если кто-то и пишет). Вырабатывается привычка использовать подход:

```

if (age < 18)
    m = 15000;
else if (age < 60)
    m = 10000;
else if (age < 70)
    m = 20000;
else
    m = 30000;

```

---

Такой код заметно легче читается. Проверяется, меньше ли возраст 18. Если меньше – переменной `m` присваивается значение 15000 и все дальнейшие проверки заканчиваются. Но если проверки не закончились, проверяется меньше ли возраст 60 лет. Если меньше – переменной `m` присваивается значение 10000 и проверки так же закончатся и т. д.

Момент по отступам. Предположим, был написан следующий код:

```

if (points > 6)
    if (points < 12)
        printf("Количество очков > 6 и < 12")
else
    printf("Количество очков <= 6")

```

---

Если бы отступы играли важную роль в интерпретации конструкций (как в языке программирования *Python*), то проблем бы не возникло. Однако в С имеется правило: `else` относится к последнему `if`. Данная конструкция будет интерпретирована как следующая:

```

if (points > 6)
    if (points < 12)
        printf("Количество очков > 6 и < 12")

```

---

```

else
    printf("Количество очков <= 6")

```

и это явно не то, что хотел сделать автор (судя по сообщениям). Исправить положение могли бы фигурные скобки:

```

if (points > 6) {
    if (points < 12)
        printf("Количество очков > 6 и < 12")
} else
    printf("Количество очков <= 6")

```

## 5.2.2 Решение квадратного уравнения

### Решение квадратного уравнения

Решить квадратное уравнение  $ax^2 + bx + c = 0$  ( $a \neq 0$ ).

При знакомстве с решением обратите внимание на то, что корень из дискриминанта вычисляется один раз. Решение будем осуществлять через дискриминант. Блок-схема и код:

```

#include <stdio.h>
#include <math.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    long long a, b, c;
    scanf("%lld %lld %lld", &a, &b, &c);

    long long D = b*b - 4*a*c;
    if (D > 0) {
        // дополнительно выделена переменная, чтобы дважды не считать
        // квадратный корень из D
        float sqrtD = sqrtl(D);

        printf("%f %f", (-b + sqrtD) / (2*a), (-b - sqrtD) / (2*a));
    } else if (D == 0) {
        printf("%f", -b / (2.0*a));
    } else {
        printf("Действительных корней нет");
    }

    return 0;
}

```

Данное решение не является единственным и не претендует быть таковым.

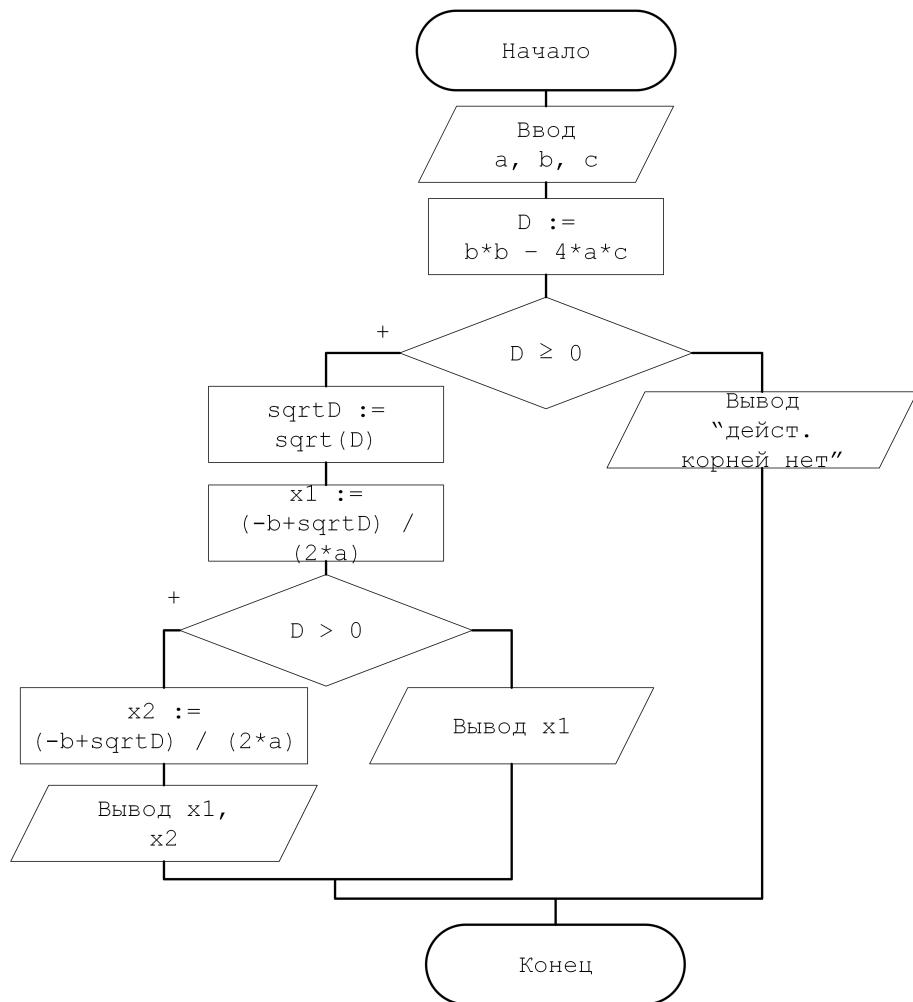


Рис. 5.7: Блок-схема алгоритма решения квадратного уравнения

### 5.3 Оператор *switch*

Структура оператора *switch*:

---

```

// switch (<целочисленное выражение>) {
//     case <константное выражение1>:
//         <операторы>
//     case <константное выражение2>:
//         <операторы>
//     ...
//     case <константное выражениеН>:
//         <операторы>
//     default:
//         <операторы>
// }

```

---

Оператор *switch* работает следующим образом:

1. Вычисляется значение <целочисленного выражения>.
2. Среди <константных выражений<sup>4</sup> ищется вычисленный результат.

<sup>4</sup>Выражение называется **константным**, если оперирует только константами.

3. Если результат найден – выполняются все **<операторы>** ниже (в том числе и те, которые находятся и по другим веткам **case**). Если результат не найден – выполняются операторы по ветке **default**.

4. Выполнение операторов заканчивается в двух случаях:

- Закончились операторы.
- Был встречен оператор **break**.

Если последним оператором в ветке **case** не будет **break** – выполняются операторы и в последующих ветках. Например, если в задаче о зимнем месяце (страница 103) убрать все **break** и ввести значение 1, то вы вдруг обнаружите, что месяц с таким номером является январём, февралём, декабрем, и вообще не является зимним. Будьте внимательны.

Если по какой-то причине отсутствие **break** в ветке является желательным – оставляйте комментарий, почему оператора **break** нет.

Стоит сказать, что ветка **default** не является обязательной, и не должна находиться в конце (но лучше помещать её в конец).

Отмечу вопрос производительности: если вы чувствуете, что можно использовать оператор **switch**, используйте его вместо цепочки операторов **if-else-if...**, так как при большом количестве разветвлений он работает быстрее.

### 5.3.1 Задача о зимнем месяце

#### Задача о зимнем месяце

Вводится номер зимнего месяца. Необходимо вывести на экран его название. Если номер месяца не является зимним – вывести сообщение.

Обозначение оператора **switch** (при условии, что если в конце каждой группы операторов имеется **break**) представлено на рисунке 5.8.

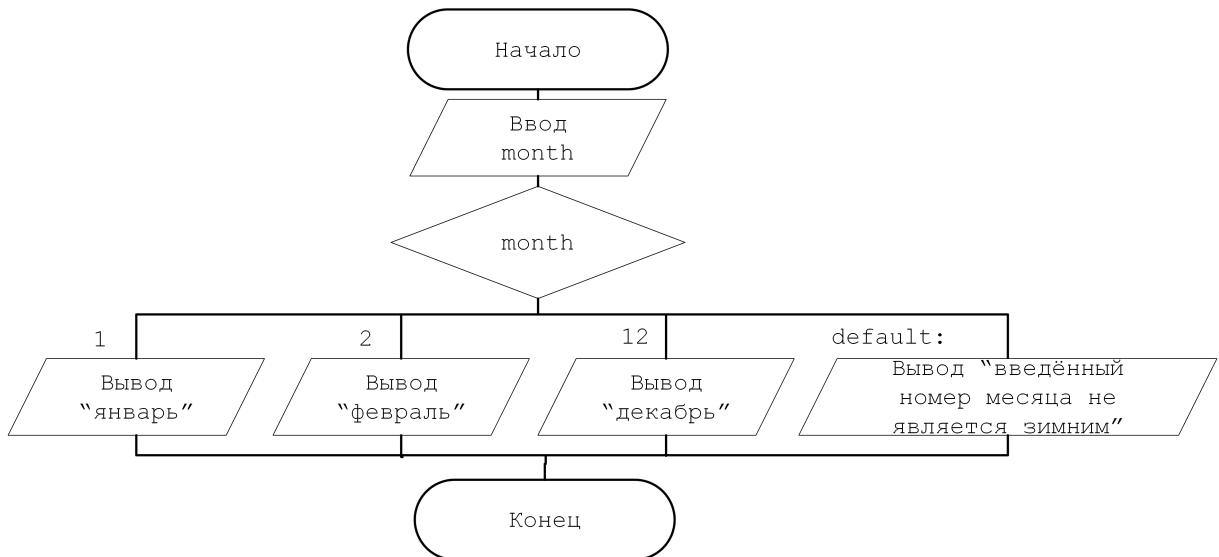


Рис. 5.8: Блок-схема решения задачи о зимнем месяце

---

```
#include <stdio.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    int month;
    scanf("%d", &month);

    switch (month) {
        case 1:
            printf("Январь");
            break;
        case 2:
            printf("Февраль");
            break;
        case 12:
            printf("Декабрь");
            break;
        default:
            printf("Введенный номер месяца не является зимним");
    }

    return 0;
}
```

---

## Резюме

- Разветвляющиеся алгоритмы – это алгоритмы, в которых имеются шаги, которые выполняются или не выполняются в зависимости от каких-либо условий.
- Для организации разветвок в языке С используются условный оператор `if` и оператор множественного ветвления `switch`.
- Условный оператор `if` работает следующим образом: если логическое выражение является истинным, то оператор выполняется.
- Оператора `if-else`:

---

```
// if (<логическое выражение>)
//     <оператор1>
// else
//     <оператор2>
```

---

Данная конструкция выполняется по следующим правилам:

1. Вычисляется значение `<логического выражения>`.
  2. Если оно является истиной, выполняется `<оператор1>`.
  3. В противном случае выполняется `<оператор2>`.
- Если значение переменной зависит от значения логического выражения, эту конструкцию лучше закодировать при помощи тернарного оператора.
  - Оператор `switch`:

---

```
// switch (<целочисленное выражение>) {
//     case <константное выражение1>:
//         <операторы>
//     case <константное выражение2>:
//         <операторы>
//     ...
//     case <константное выражениеN>:
//         <операторы>
//     default:
//         <операторы>
// }
```

---

работает следующим образом:

1. Вычисляется значение `<целочисленного выражения>`.
2. Среди `<константных выражений>` ищется вычисленный результат.
3. Если результат найден – выполняются все `<операторы>` ниже (в том числе и те, которые находятся и по другим веткам `case`). Если результат не найден – выполняются операторы по ветке `default`.
4. Выполнение операторов заканчивается в двух случаях:
  - Закончились операторы.
  - Был встречен оператор `break`.

## Термины и определения

- **Константное выражение** – это выражение, оперирующее только константами.
- **Логическое выражение** – это выражение, результатом вычисления которого является значение 'истина' или 'ложь'.
- **Разветвляющиеся алгоритмы** – это алгоритмы, в которых имеются шаги, которые выполняются или не выполняются в зависимости от каких-либо условий.
- **Составной оператор** – это два или большее количество операторов, сгруппированных вместе путём помещения их в фигурные скобки; его также называют блоком.

## Контрольные вопросы

1. Какие алгоритмы называются разветвляющимися?
2. Какие имеются требования к тестовым данным для тестирования программ с ветвлением?
3. Что представляет собой логическое выражение?
4. Как организовать бинарное ветвление?
5. Как организовать множественное ветвление?
6. В каком случае лучше использовать тернарный оператор?
7. К какому `if` относится `else`?
8. Напишите программу, определяющую является ли четырехзначное число палиндромом.
9. Напишите программу, определяющую максимальную цифру в записи четырехзначного числа.
10. Напишите программу, определяющую является ли введённое число полным квадратом.
11. Напишите программу, определяющую является ли четырехугольник, заданный координатами своих вершин, квадратом.

# Глава 6

## Циклические алгоритмы

Существует группа алгоритмов, в которых некоторая часть операций выполняется многократно. Такие алгоритмы называются **циклическими**. В языке программирования С существуют два цикла с предусловием (`for`, `while`) и один цикл с постусловием (`do-while`).

### 6.1 Цикл `for`

Конструкция цикла `for`:

```
// for (<выражение1>; <выражение2>; <выражение3>)  
//     <оператор>
```

<выражение1> называют **инициализирующим**. Как правило, в нём объявляется переменная цикла. <выражение2> – **условием возобновления цикла**. <Выражение3> – **корректирующее** (оно часто используется для изменения значения счётчика цикла). Любое из трёх выражений может отсутствовать, однако опускать ; нельзя. Оператор цикла часто именуют **телом цикла**.

В языке программирования Паскаль имеется цикл `for`, который является циклом с фиксированным числом повторений. Однако в языке С цикл `for` в общем случае к нему отнесён быть не может. Его компоненты на блок-схеме:

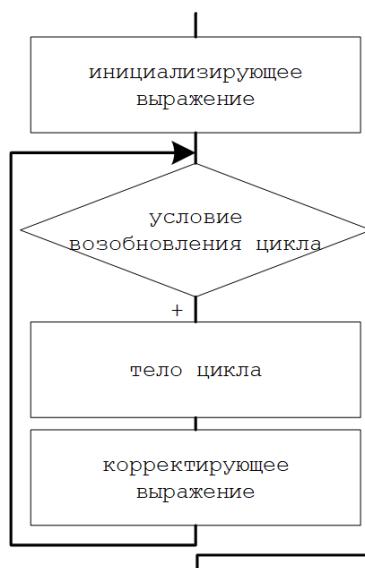


Рис. 6.1: Цикл `for` с нефиксированным числом повторений

Цикл `for` работает так:

1. Вычисляется значение `<инициализирующего выражения>`.
2. Проверяется истинность `<условия возобновления цикла>`.
3. Если оно истинно, выполняется `<оператор>` и `<корректирующее выражение>`.  
Если условие возобновления ложно, конец цикла.

Если по заголовку цикла можно сказать, что цикл выполнится заданное количество раз, то следует использовать следующее обозначение посредством блок-схем (рисунок 6.2):

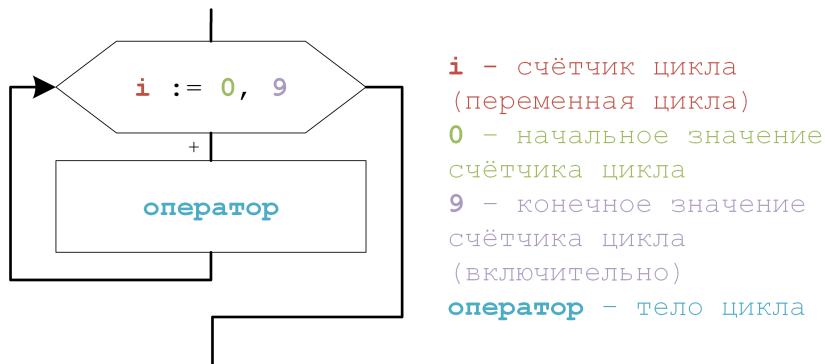


Рис. 6.2: Обозначения цикла *for* (вариант: цикл с фиксированным числом повторений)

Данный фрагмент выведет значения от 0 до 10 (не включая 10):

---

```

#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++)
        printf("%d\n", i);

    return 0;
}
  
```

---

Часто корректирующее выражение влияет на счётчик цикла (или переменную цикла; в нашем случае – `i`) посредством его увеличения или уменьшения на какое-то значение. Такое значение именуют **шагом цикла**. Для прошлого случая корректно сказать, что шаг цикла равен единице.

Если шаг цикла отличен от единицы, следует использовать вариант с рисунка 6.3:

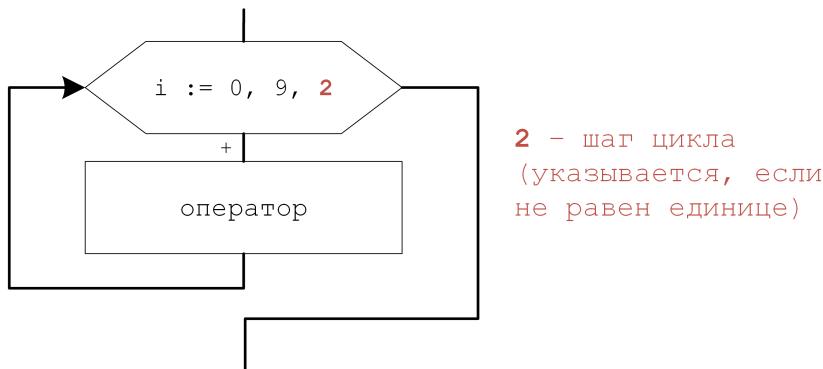


Рис. 6.3: Обозначения цикла *for* (вариант: цикл с фиксированным числом повторений; шаг отличен от +1)

Если заголовок цикла содержит несколько более сложное выражение (или определение количества итераций является проблемным) используйте пример с рисунка 6.1. Обратите внимание, что тогда будет использоваться блок 'решение'.

Если в заголовке цикла нужно инициализировать или корректировать несколько переменных, надо использовать операцию запятая (,)<sup>1</sup>. Пара выражений, разделенных запятой, вычисляется слева направо. Фрагмент

```
for (int i = 0, j = 10; i <= j; i++, j--)
    printf("%d %d\n", i, j);
```

выведет всевозможные способы представления числа 10 в виде суммы двух неотрицательных чисел. Дополнительно опишем блок-схему к данной задаче:

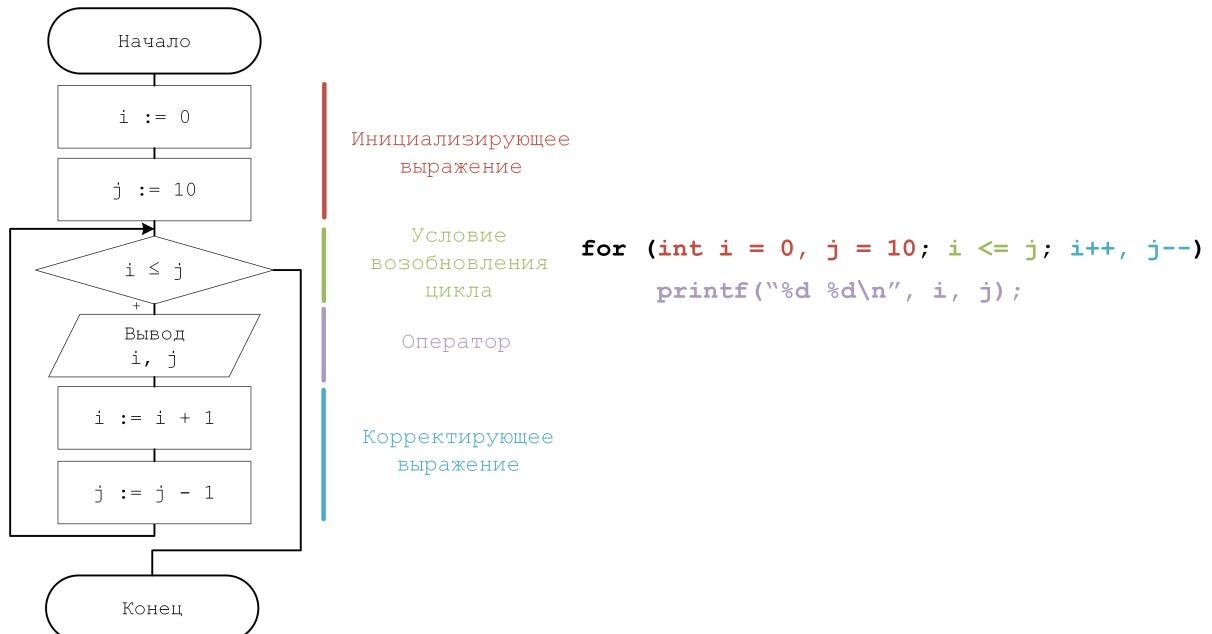


Рис. 6.4: Цикл *for* с нефиксированным числом повторений

Немного об области видимости переменных. Переменные, создаваемые в заголовке или теле цикла, видны исключительно в цикле. По окончанию работы `for` переменные уничтожаются:

<sup>1</sup>Запятые, разделяющие аргументы функции, переменные в объявлениях и пр. не являются операторами-запятыми и не обеспечивают вычислений слева направо

---

```

for (int i = 0; i < 10; i++) {
    int x = 10;
    // ...
}
// переменные x и i невидны здесь; будет ошибкой использовать их тут

```

---

Если до цикла существовала переменная с таким же именем, она будет 'скрыта' созданной переменной в заголовке или в теле:

---

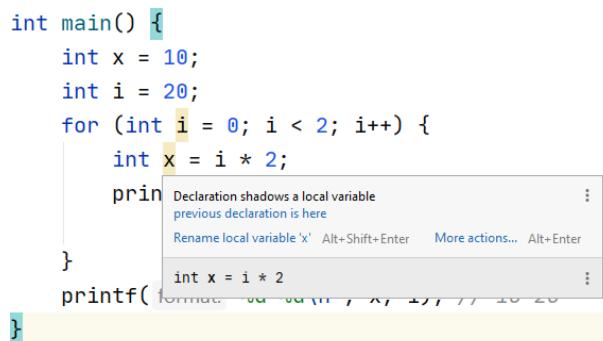
```

int x = 10;
int i = 20;
for (int i = 0; i < 2; i++) {
    int x = i * 2;           // это уже другие x и i (не те, что были до цикла)
    printf("%d %d\n", x, i); // 0 0
                           // 2 1
}
                           // x и i до цикла
printf("%d %d\n", x, i);   // 10 20

```

---

Современные *IDE* способны выдавать предупреждения для таких случаев:



```

int main() {
    int x = 10;
    int i = 20;
    for (int i = 0; i < 2; i++) {
        int x = i * 2;
        printf("%d %d\n", x, i);
    }
    printf("%d %d\n", x, i);
}

```

Несмотря на такую гибкость цикла `for` старайтесь не нагружать его заголовок вычислением всего, чего возможно. Там должны встречаться инструкции исключительно для управления циклом.

Использование цикла `for` заметно увеличивает ваши возможности при решении задач. Рассмотрим некоторые из них.

### 6.1.1 Вычисление суммы последовательности

#### Вычисление суммы последовательности

С клавиатуры вводится  $n$  целых чисел. Необходимо найти сумму введённых чисел.

Опишем решение при помощи блок-схемы (рисунок 6.5):

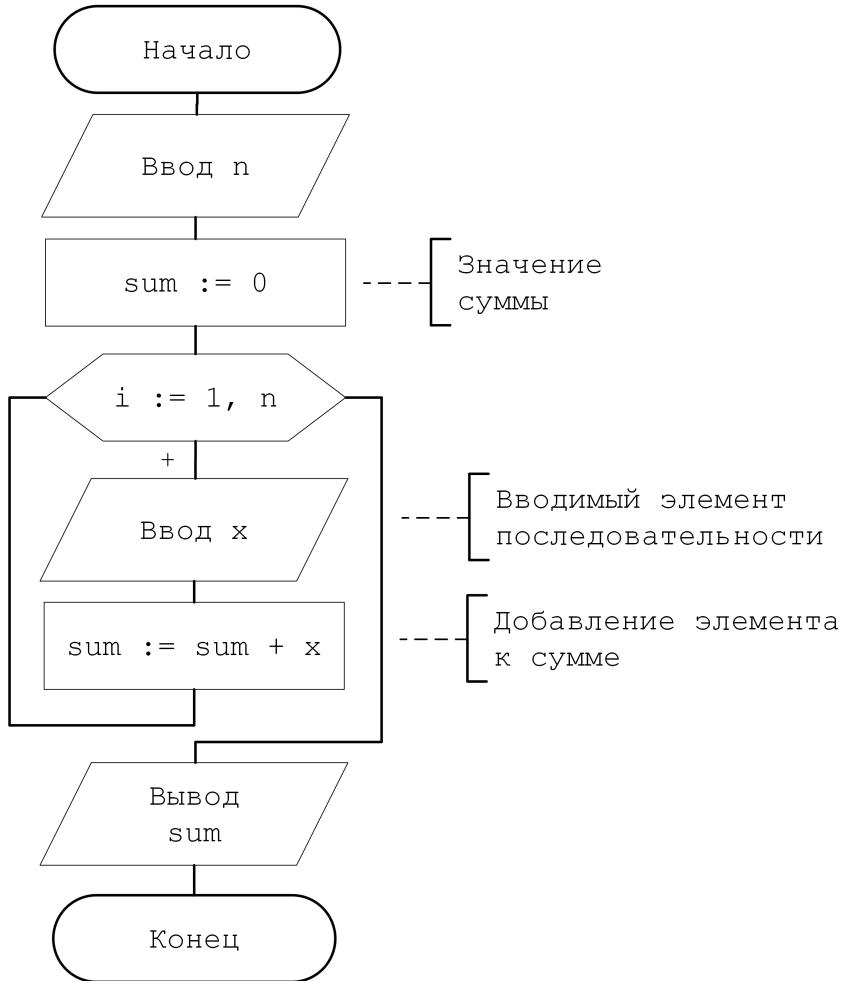


Рис. 6.5: Блок-схема решения задачи вычисления суммы последовательности

Код функции `main`:

```

#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    long long sum = 0;
    for (int i = 1; i <= n; i++) {
        int x;
        scanf("%d", &x);

        sum += x;
    }
}
  
```

```

    printf("%lld", sum);

    return 0;
}

```

Обратите внимание на то, что переменная `s` имеет тип `long long` (потенциальное значение суммы может быть большим). Решение о применении данного типа лежит на программисте.

### 6.1.2 Поиск значений выражений

#### Поиск значений выражений

С клавиатуры вводится значение  $n$  ( $n > 0$ ). Вычислить:

- $n!$

- $\prod_{i=1}^n \left(1 + \frac{1}{i^2}\right) = \left(1 + \frac{1}{1^2}\right) \left(1 + \frac{1}{2^2}\right) \cdots \left(1 + \frac{1}{n^2}\right)$

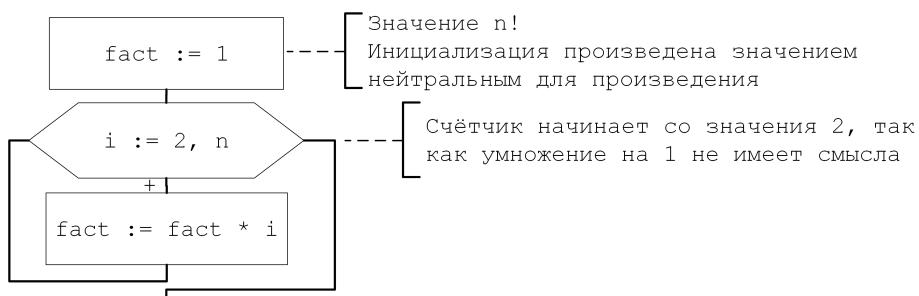
- $\sum_{i=1}^n \sin^i(x) = \sin(x) + \sin^2(x) + \dots + \sin^n(x)$

- $\sum_{i=0}^n \frac{1}{\prod_{j=0}^i (a+j)} = \frac{1}{a} + \frac{1}{a(a+1)} + \dots + \frac{1}{a(a+1)\dots(a+n)}$

- $\sum_{i=1}^n \frac{x^i}{i!} = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$

Опишем последовательно решения всех задач. Код и блок-схемы будут сфокусированы на инициализации переменных и самом цикле.

Вычисление  $n!$ :




---

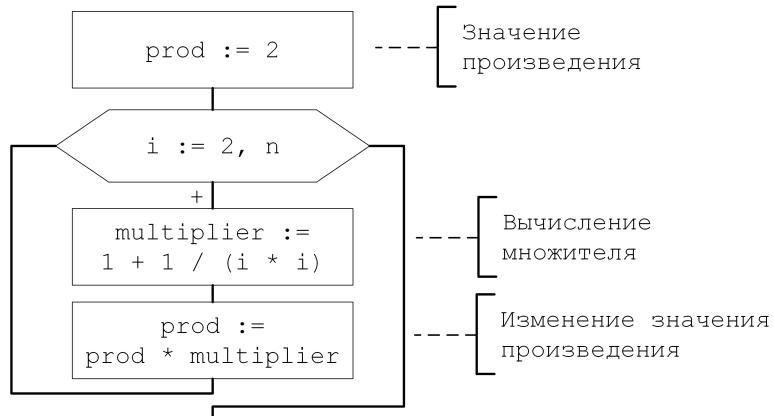
```

long long fact = 1;
for (int i = 2; i <= n; i++)
    fact *= i;

```

---

Вычисление  $\prod_{i=1}^n \left(1 + \frac{1}{i^2}\right) = \left(1 + \frac{1}{1^2}\right) \left(1 + \frac{1}{2^2}\right) \cdots \left(1 + \frac{1}{n^2}\right)$ :




---

```

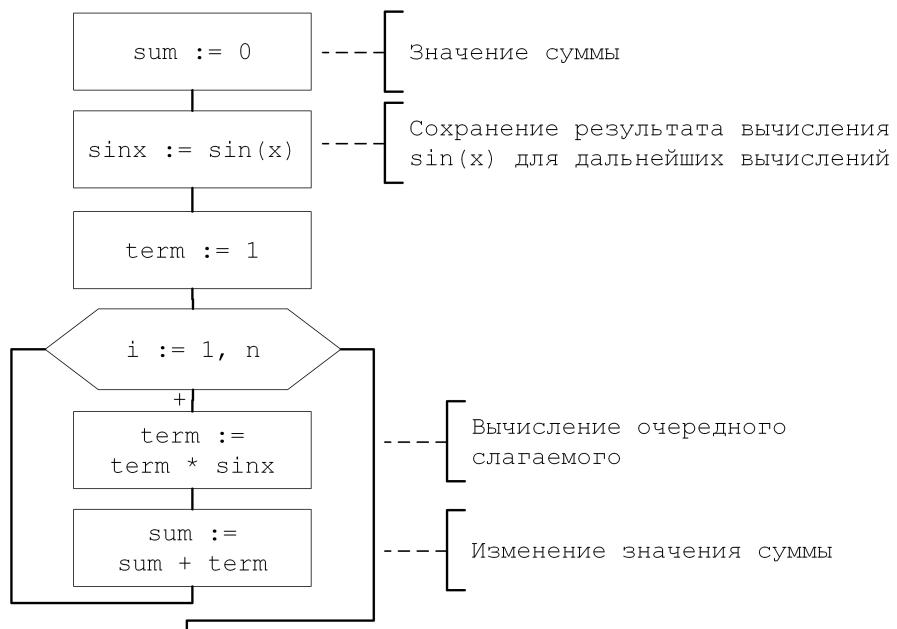
double prod = 2; // значение произведения будет вещественным
for (int i = 2; i <= n; i++) {
    double multiplier = 1 + 1.0 / i / i;
    // или double multiplier = 1 + 1.0 / ((long long) i * i)
    // приведение к long long - борьба с переполнением
    prod *= multiplier;
}
  
```

---

Вычисление  $\sum_{i=1}^n \sin^i(x) = \sin(x) + \sin^2(x) + \dots + \sin^n(x)$ . Несложно показать, что

$$\sin^i(x) = \sin^{i-1}(x) * \sin(x)$$

Организуем вычисления так, чтобы  $\sin(x)$  считался единожды. Переменная `term` (от англ. слагаемое) при  $i$ -ой итерации вычисляет значение  $\sin^i(x)$ :



---

```

double sum = 0;
double sinx = sin(x);
double term = 1;
for (int i = 1; i <= n; i++) {
    term *= sinx;
    sum += term;
}

```

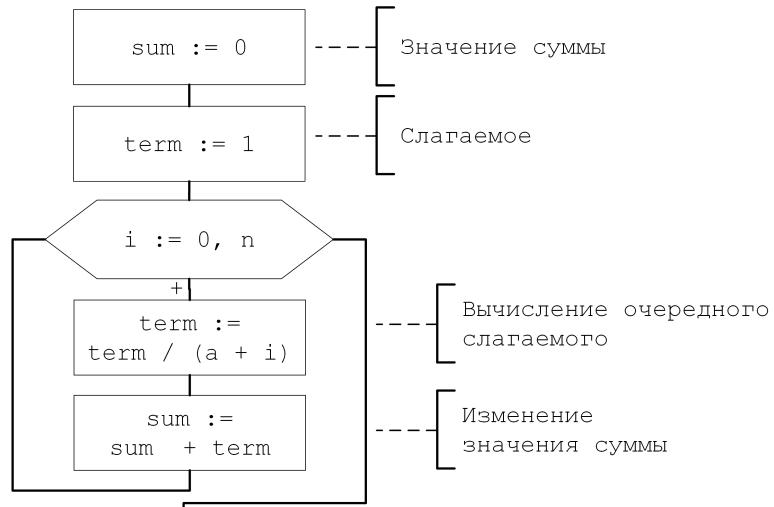
---

Вычисление значения выражения

$$\sum_{i=0}^n \frac{1}{\prod_{j=0}^i (a+j)} = \frac{1}{a} + \frac{1}{a(a+1)} + \cdots + \frac{1}{a(a+1)\dots(a+n)}$$

Идея аналогична прошлой задаче. Каждое последующее слагаемое может быть вычислено через предыдущее:

$$term_0 = \frac{1}{a} \quad term_i = \frac{term_{i-1}}{a+i}$$




---

```

double term = 1.0 / a; // значение произведения будет вещественным,
                      // деление должно быть вещественным
double sum = term;
for (int i = 1; i <= n; i++) {
    term = term / (a + i);
    sum += term;
}

```

---

Вычисление  $\sum_{i=1}^n \frac{x^i}{i!} = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$ . Очевидно, что

$$term_0 = 1 \quad term_i = term_{i-1} * \frac{x}{i}$$

```
double term = 1;
double r = 0;
for (int i = 1; i <= n; i++) {
    term = term / i * x;
    r += term;
}
```

### 6.1.3 Поиск максимума вводимой последовательности

#### Поиск максимума вводимой последовательности

С клавиатуры вводится  $n$  ( $n > 0$ ) целых чисел. Необходимо найти максимальное из введенных чисел.

Если вы ещё не забыли задачу про поиск максимума из трёх значений (блок-схема на странице 97), то помните о наличии в той задаче повторяющихся фрагментов блок-схем. Если мы вынесем данные фрагменты в цикл, то получим идею для решения задачи. Блок-схема (рисунок 6.6), код функции `main`<sup>2</sup>:

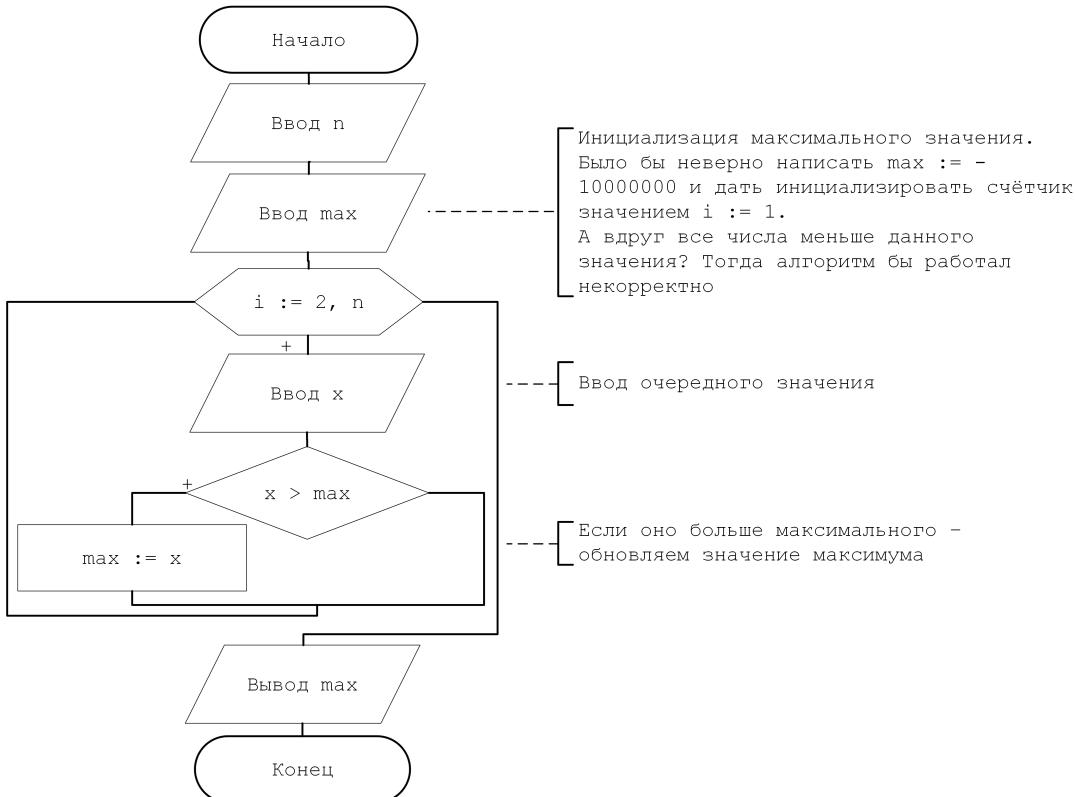


Рис. 6.6: Блок-схема для решения задачи поиска максимума

```

#include <stdio.h>

int main() {
    int n, max;
    scanf("%d %d", &n, &max);

    for (int i = 2; i <= n; i++) {
        int x;
        scanf("%d", &x);

        if (x > max)
            max = x;
    }
}

```

<sup>2</sup>Существуют разные мнения касаемо того, стоит ли объявлять переменные внутри цикла. Источники, изученные мной, говорят, что компилятор проведёт оптимизацию в таких случаях. Объявлять простые типы безопасно в контексте производительности.

```

    }

    printf("%d", max);

    return 0;
}

```

#### 6.1.4 Поиск последнего нечетного элемента

##### Поиск последнего нечетного элемента

С клавиатуры вводится  $n$  целых чисел, которые нельзя сохранить во внутренней памяти устройства. Необходимо найти последнее вхождение нечетного элемента.

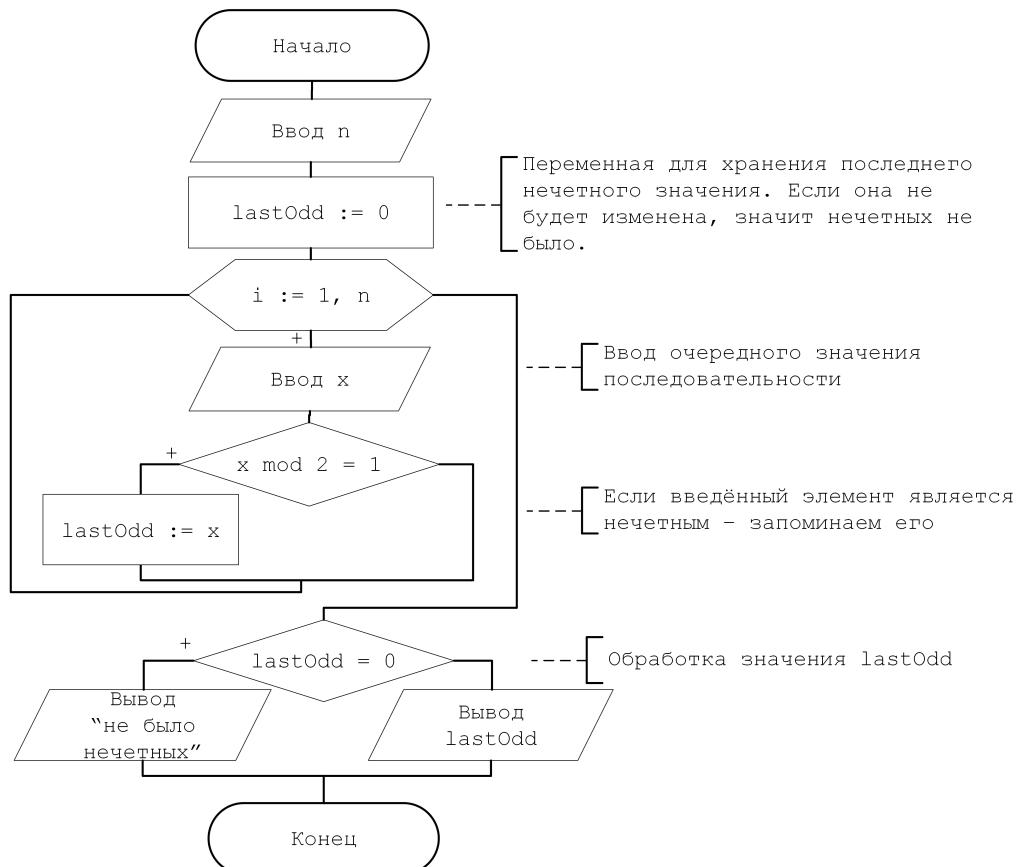


Рис. 6.7: Блок-схема для решения задачи последнего нечетного элемента

Идея решения заключается в использовании переменной `lastOdd`. Если мы встретили нечетный элемент – сохраним его. Полученное значение и будет ответом. Но не должны забывать, что нечетных может и не быть<sup>3</sup>. И мы должны быть способны как-то сигнализировать об этом. В данном случае решение было найдено за счёт инициализации переменной `lastOdd` нулём. Если по окончанию обработки последовательности значение не изменится, значит и нечетных не было.

<sup>3</sup>Опытный разработчик всегда посмотрит направо и налево, даже если переходит улицу с односторонним движением.

Можно дополнить решение приёмом: чтобы значение 0 не воспринималось магической константой, опишем макрос

---

```
#define NO_ODD_VALUES 0
```

---

который говорит о значении для того случая, когда нечетных элементов не будет.

---

```
#include <stdio.h>
#include <windows.h>

#define NO_ODD_VALUES 0

int main() {
    int n;
    scanf("%d", &n);

    int lastOdd = NO_ODD_VALUES;
    for (int i = 1; i <= n; i++) {
        int x;
        scanf("%d", &x);

        if (x % 2 == 1)
            lastOdd = x;
    }

    if (lastOdd == NO_ODD_VALUES)
        printf("No odd values");
    else
        printf("%d", lastOdd);

    return 0;
}
```

---

### 6.1.5 Поиска максимального значения последовательности и их количества

**Поиск количества максимальных значений последовательности**

С клавиатуры вводится  $n$  целых чисел. Необходимо найти максимальное значение в последовательности и их количество.

В этой задаче на этапе чтения значений нужно поддерживать значения максимального элемента и количества встреченных значений, равных максимальному. Можно сформулировать две основных идеи:

- Если находится число, которое больше текущего максимума – надо обновить максимум и установить счетчик в единицу.
- Если встретили число, равное максимальному – просто увеличиваем значение счетчика.

Блок-схема представлена на рисунке 6.8.

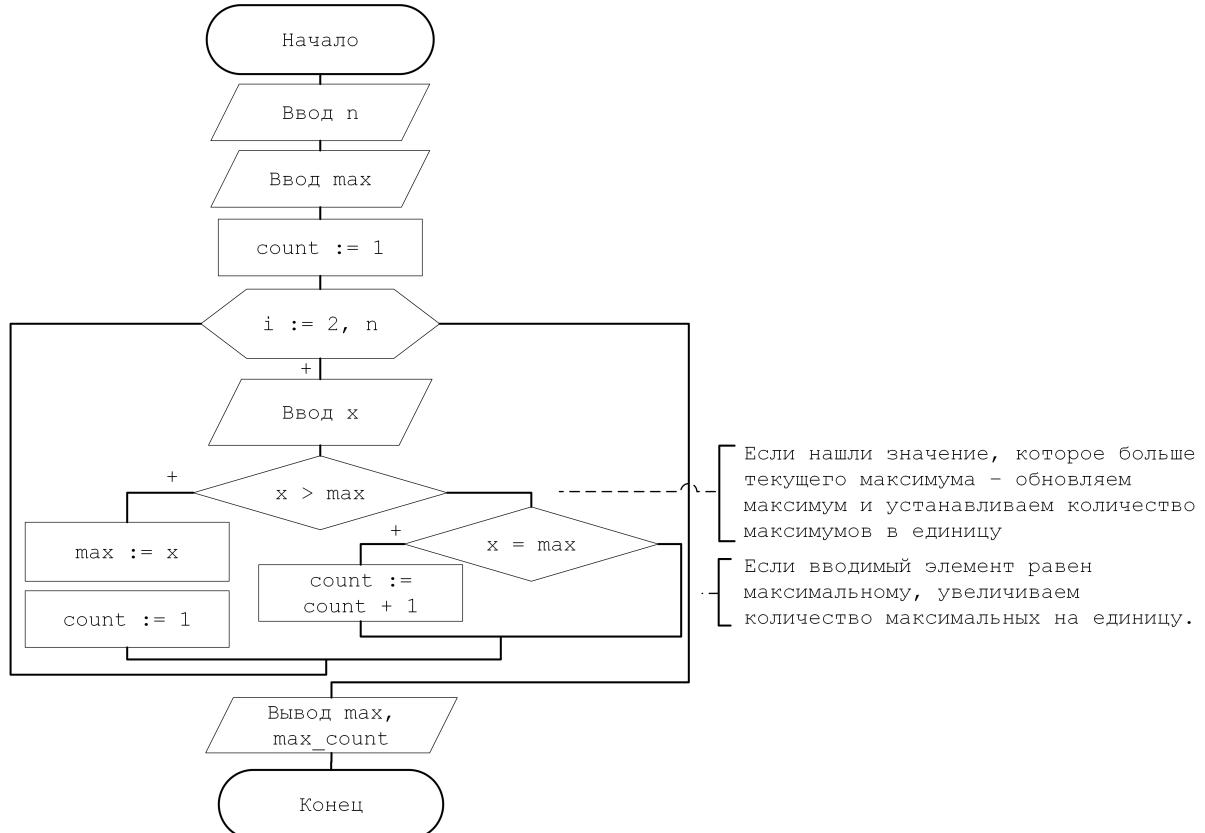


Рис. 6.8: Блок-схема решения задачи поиска максимального значения последовательности и их количества

---

```

#include <stdio.h>

int main() {
    int n, max;
    scanf("%d %d", &n, &max);

    int count = 1;
    for (int i = 2; i <= n; i++) {
        int x;
        scanf("%d", &x);

        if (x > max) {
            count = 1;
            max = x;
        } else if (x == max)
            count = count + 1;
    }

    printf("%d %d", max, count);

    return 0;
}

```

---

## 6.2 Цикл *while*

Цикл `while` является циклом с предусловием:

---

```
// while (<логическое выражение>
//       <оператор>
```

---

Его работа состоит в следующем:

1. Проверяется истинность `<логического выражения>`;
2. Если выражение истинно – выполняется `<оператор>`, иначе – конец цикла.

Таким образом, цикл будет выполняться до тех пор, пока выражение будет истинным, в том числе и ни разу, если выражение при первой проверке ложно:

---

```
int i = 5;
while (i > 10)
    i--;
```

---

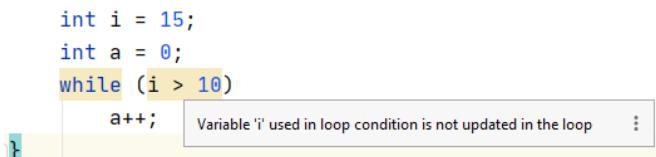
Цикл `while` может стать бесконечным, если не влиять на переменные, которые указаны в логическом выражении:

---

```
int i = 15;
int a = 0;
while (i > 10)
    a++;
```

---

Современные среды разработки способны предупреждать о проблеме:



```
int i = 15;
int a = 0;
while (i > 10)
    a++;
```

Variable 'i' used in loop condition is not updated in the loop

Обозначение цикла `while` на блок-схемах представлено на рисунке 6.9:



Рис. 6.9: Обозначение цикла *while*

### 6.2.1 Подсчёт количества цифр в числе

#### Подсчёт количества цифр в числе

Найти количество цифр в числе  $x$ .

Известно, что любое число содержит хотя бы одну цифру. Можно делить число на 10 и подсчитывать цифры до тех пор, пока исходное число будет больше 9:

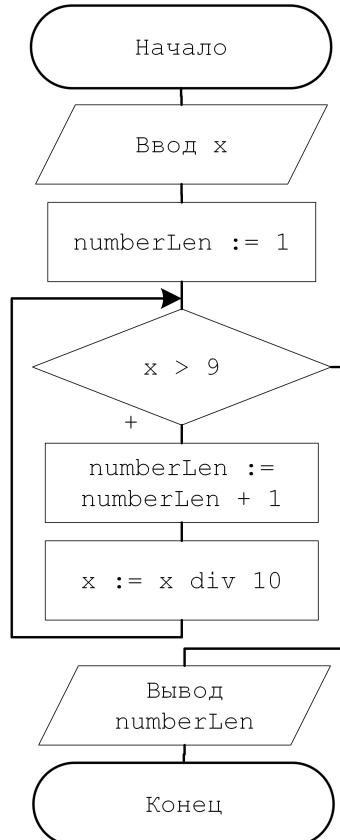


Рис. 6.10: Блок-схема для решения задачи подсчёта количества цифр в числе

---

```

#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    int numberLen = 1;
    while (x > 9) {
        numberLen++;
        x /= 10;
    }

    printf("%d", numberLen);

    return 0;
}

```

---

### 6.2.2 Подсчёт количества единиц в двоичном представлении числа $x$

Подсчёт количества единиц в двоичном представлении числа  $x$

Найти количество единиц в двоичной записи числа  $x$ .

Идея алгоритма совпадает с переводом чисел из десятичной системы в двоичную, только нам не нужно производить разворот бит:

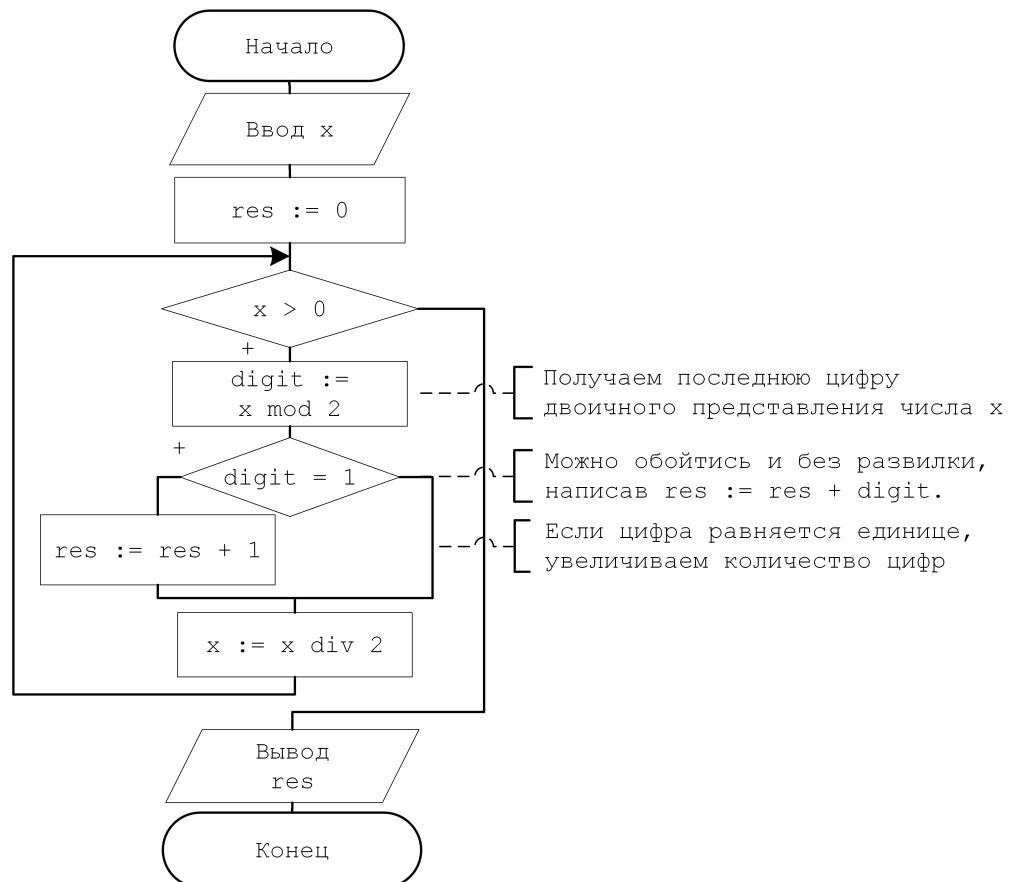


Рис. 6.11: Блок-схема для решения задачи подсчёта количества единиц в двоичной записи числа  $x$

```

#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    int res = 0;
    while (x > 0) {
        int isOne = x % 2 == 1;
        res += isOne;
        x /= 2;
    }
}
  
```

```

    printf("%d", res);

    return 0;
}

```

### 6.2.3 Изменение порядка цифр в числе на обратный

#### Изменение порядка цифр в числе на обратный

Дано число  $x$ . Требуется изменить в нём порядок цифр на обратный.

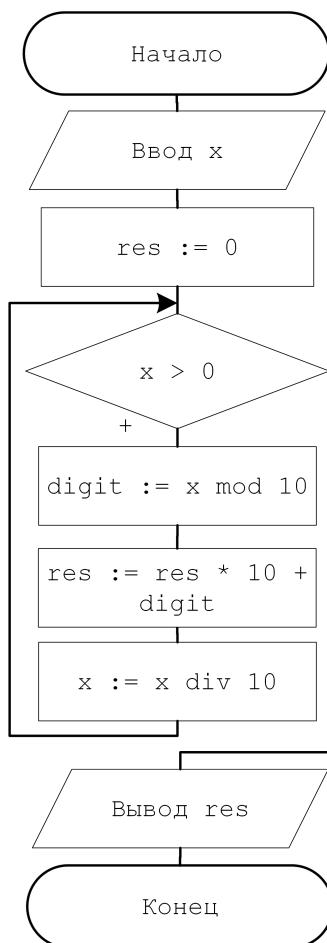


Рис. 6.12: Блок схема решения задачи изменения порядка цифр

```

#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    int res = 0;
    while (x > 0) {
        int digit = x % 10;

```

```

    res = res * 10 + digit;
    x /= 10;
}

printf("%d", res);

return 0;
}

```

#### 6.2.4 Подсчёт количества положительных и отрицательных чисел последовательности

##### Подсчёт количества положительных и отрицательных чисел последовательности

С клавиатуры вводятся числа. Признак конца ввода – 0. Найти  $c_1$  - количество положительных чисел и  $c_2$  - количество отрицательных чисел.

Блок-схема решения задачи на рисунке 6.13.

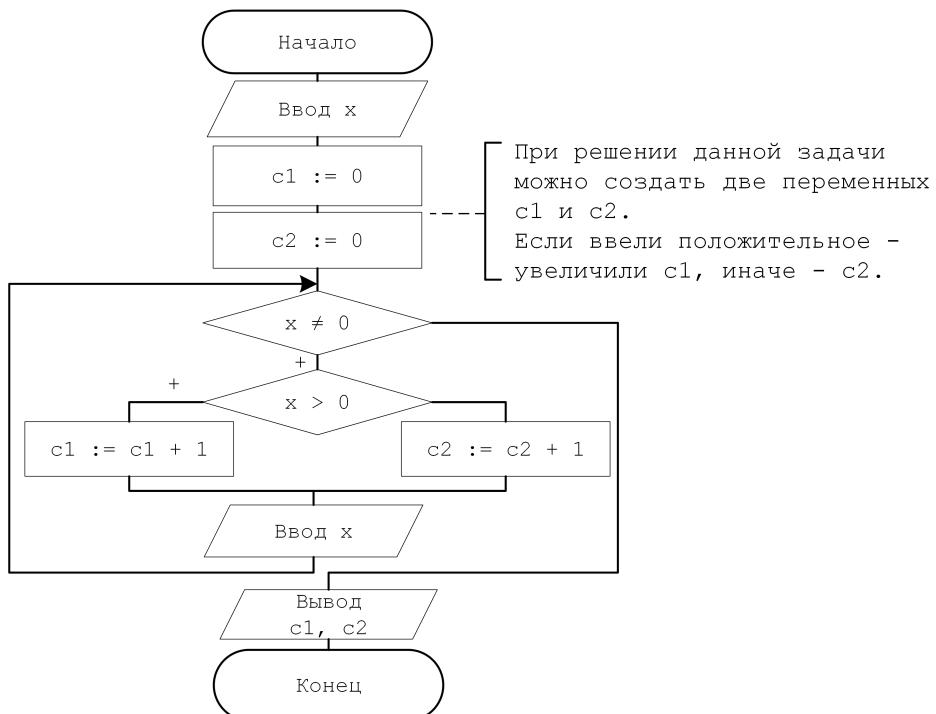


Рис. 6.13: Блок-схема к задаче о подсчёте положительных и отрицательных чисел последовательности

```

#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

```

```

int c1 = 0;
int c2 = 0;
while (x != 0) {
    if (x > 0)
        c1++;
    else
        c2++;

    scanf("%d", &x);
}

printf("c1 = %d, c2 = %d", c1, c2);

return 0;
}

```

---

### 6.2.5 Первый элемент рекуррентно заданной последовательности больше $x$

**Первый элемент рекуррентно заданной последовательности больше  $x$**

Пусть  $a_0 = 1$ ,  $a_k = k^2 a_{k-1}$ ,  $k = 1, 2, \dots, n$ . Дано число  $x$ . Получить первый элемент последовательности, большие  $x$ .

```

#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    long long a = 1;
    int k = 1;
    while (a <= x) {
        a = a * k * k;
        k++;
    }

    printf("%lld", a);

    return 0;
}

```

---

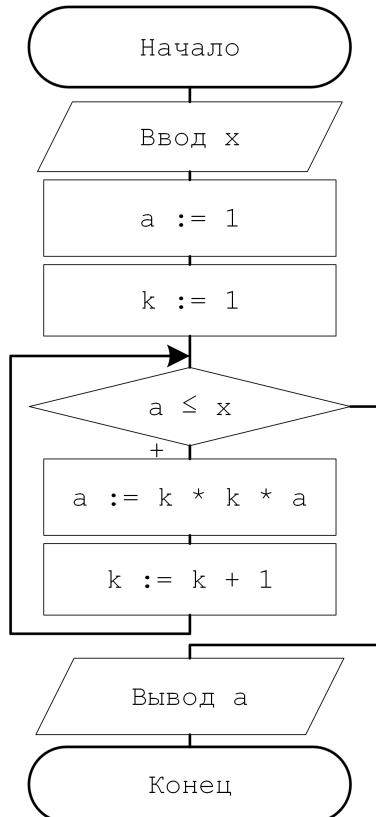


Рис. 6.14: Блок схема для поиска первого элемента рекуррентно заданной последовательности больше  $x$

### 6.2.6 Поиск суммы последних $m$ цифр числа $n$

#### Поиск суммы последних $m$ цифр числа $n$

Даны натуральные числа  $n$ ,  $m$ . Получить сумму  $m$  последних цифр числа  $n$ . Будем исходить из предположения, что  $m$  может быть больше, чем количество цифр, тогда результат - сумма цифр числа  $n$ .

---

```

#include <stdio.h>

int main() {
    int n, m;
    scanf("%d %d", &n, &m);

    int sum = 0;
    while (n != 0 && m != 0) {
        sum += n % 10;
        m--;
        n /= 10;
    }

    printf("%d", sum);

    return 0;
}
  
```

---

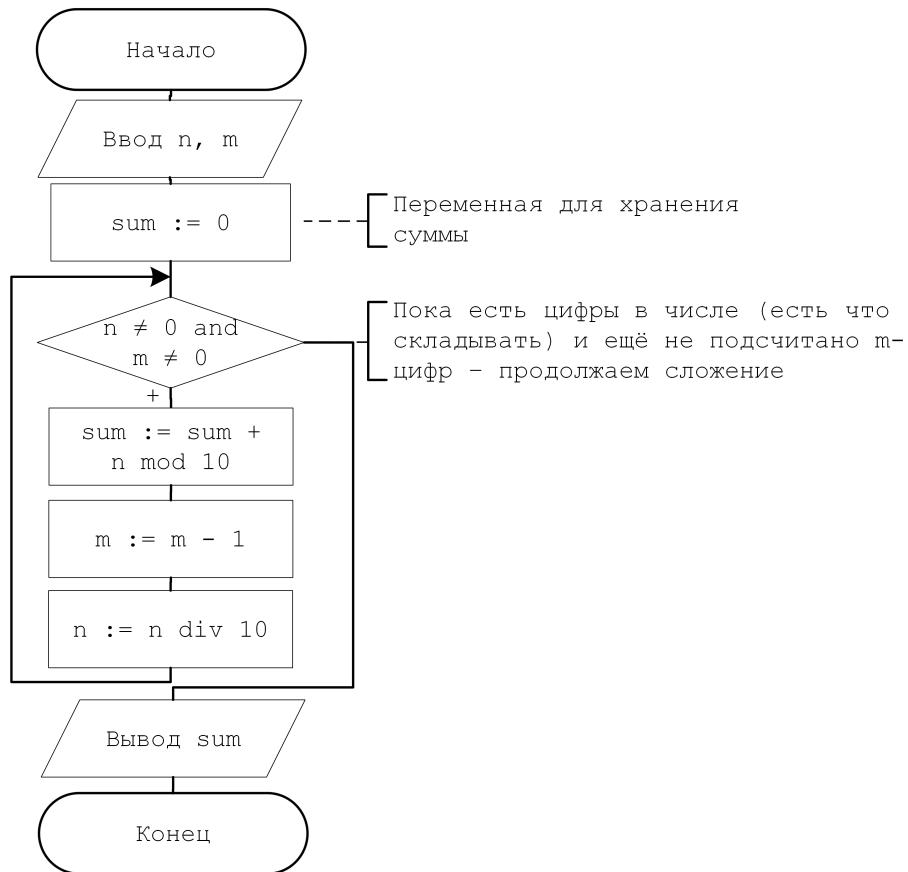


Рис. 6.15: Блок схема решения задачи о поиске суммы последних цифр

### 6.2.7 Получение средней цифры числа, если количество цифр в числе нечетно

#### Получение средней цифры числа, если количество цифр в числе нечетно

С клавиатуры вводится число  $x$ . Необходимо вывести среднюю (по позиции) цифру числа  $x$ , если количество цифр нечетно, в противном случае – -1.

Опишем алгоритм в словесно-формульном виде:

1. Создадим копию исходного значения.
2. Постепенно будем уменьшать число в 10 раз и подсчитывать количество цифр.
3. Создадим коэффициент  $k$ , при помощи которого будет доставаться нечетная цифра:

$x$	$k$
1	1
123	10
12345	100
1234567	1000

4. Используя значение  $k$  можно легко найти среднюю цифру по формуле:

$$\text{digit} = x \text{ div } k \text{ mod } 10$$

---

```
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    int copyX = x;
    int nDigits = 1;
    int k = 1;
    while (copyX > 9) {
        nDigits++;
        if (nDigits % 2)
            k *= 10;
        copyX /= 10;
    }

    int digit = nDigits % 2 ? x / k % 10 : -1;
    printf("%d", digit);

    return 0;
}
```

---

## 6.2.8 Проверка числа на простоту

### Проверка числа на простоту

С клавиатуры вводится число  $n$ . Необходимо проверить, является ли  $n$  простым<sup>a</sup>.

<sup>a</sup>Число является простым, если не имеет делителей кроме себя и единицы. Единица к простым числам не относится.

Решение можно осуществить так:

---

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    int d = 2;
    while (d < n && n % d != 0)
        d++;

    if (d == n)
        printf("Yes");
    else
        printf("No");

    return 0;
}
```

---

Для небольших чисел алгоритм будет работать, но с ростом проверяемого значения время работы сильно увеличится. Попробуем применить некоторые простые оптимизации.

Можем воспользоваться правилом: если число  $n$  имеет хотя бы один нетривиальный (не считая единицы и самого числа) делитель, значение минимального из нетривиальных не будет превышать  $\sqrt{n}$ .

Это легко показать на примере. Предположим, мы ищем делители числа  $n = 16$ . Первый нетривиальный делитель -  $d = 2$ . Но если  $d$  является делителем  $n$  то и  $n/d = 16/2 = 8$  является делителем  $n$ . Следующий нетривиальный делитель  $n = 4$ . По другую сторону будет найден делитель  $n/4 = 4$ . Продолжая, встретим делитель 8. Но он был уже найден ранее. Осуществлять проверку значений больше  $\sqrt{n}$  не имеет смысла.

Воспользуемся данным фактом:

---

```
#include <stdio.h>
#include <math.h>

int main() {
    int x;
    scanf("%d", &x);

    int d = 2;
    int max_d = sqrt(x);
```

---

```

    while (d <= max_d && x % d != 0)
        d++;

    if (d == max_d + 1 && x != 1)
        printf("Yes");
    else
        printf("No");

    return 0;
}

```

---

Можно выполнить ещё оптимизацию и сделать шаг, равный двум:

---

#### Листинг 9 Проверка числа на простоту

---

```

#include <stdio.h>
#include <math.h>

int main() {
    int x;
    scanf("%d", &x);

    int d = 3;
    int maxPotentialDivider = sqrt(x);
    int isPrime = !(x == 1 || x % 2 == 0 && x != 2);
    while (d <= maxPotentialDivider && isPrime) {
        isPrime = x % d;
        d += 2;
    }

    if (isPrime)
        printf("Yes");
    else
        printf("No");

    return 0;
}

```

---

Опишу немного момента оценки сложности алгоритмов. Сам процесс вычислений, который организован в компьютере, достаточно сложный. Мы абстрагируемся<sup>4</sup> от деталей и оперируем моделями вычислений. Когда говорится об оптимизации программы, имеется ввиду оптимизация по какой-то модели.

Мы будем подсчитывать число операций. Рассмотрим какой-нибудь простой пример: с клавиатуры вводится  $n$  чисел, необходимо найти максимум. Укажем количество операций, которое выполняются в той или иной строке:

---

<sup>4</sup> Абстрагироваться – мысленно отвлекаться от тех или иных сторон, свойств или связей предметов и явлений с целью выделения существенных и закономерных их признаков

---

```

#include <stdio.h>

int main() {
    int n, max;
    scanf("%d %d", &n, &max); // 2 операции

    int i = 2;           // 1 операция
    while (i <= n) {    // 1 операция выполнится n раз,
                        // для значений от 2 до n+1
        int x;
        scanf("%d", &x); // 1 операция n-1 раз

        if (x > max)    // 1 операция n-1 раз
            max = x;     // 1 операция, которая в худшем случае
                            // выполнится n-1 раз
        i++;             // 1 операция, n-1 раз
    }

    printf("%d", max); // 1 операция

    return 0;
}

```

---

Итого на весь алгоритм мы тратим:

$$T(n) = 4 + n + (n - 1) * 4 = 5n \text{ операций}$$

Для каких-нибудь других алгоритмов мы бы могли получить  $10n + 3$  операций. Но когда говорят о производительности, нас интересует, как решение будет справляться с большими объемами данных (например, при большом  $n$ ). Если  $n$  будет очень большим, тройка 'меркнет' на фоне  $10n$ . При больших  $n$  и значение 10 может не иметь много смысла. Да и кто говорит, что мы считаем операции правильно? Мы используем модель, а модель упрощена. Там могло происходить не 10 операций, а 12, например. С другой стороны, в значении  $n$  мы более-менее можем быть уверены. Тогда говорят, что порядок функции временной сложности в худшем случае составляет  $O(n)$ .

Если бы количество операций составляло

$$T(n) = 4n^2 + n + 42$$

то порядок функции временной сложности:  $O(n^2)$ .

Порядок функции временной сложности в худшем случае обозначают  $O(g(x))$ . Задают и порядок функции для нижней границы (наилучшего случая), который обозначают  $\Omega(g(x))$ <sup>5</sup>. Если порядки функций временной сложности для наихудшего и наилучшего случая совпадают  $O(g(x)) = \Omega(g(x))$ , говорят, что функция имеет асимптотическую сложность  $\Theta(g(x))$ <sup>6</sup>

Несколько жизненных примеров:

- «почистить ковёр пылесосом» требует времени, линейно зависящее от его площади  $\Theta(S)$ , то есть на ковёре, площадь которого больше в два раза, уйдет в два

---

<sup>5</sup> $\Omega$  - 'омега' (произношение).

<sup>6</sup> $\Theta$  - 'тета' (произношение).

раза больше времени. Соответственно, при увеличении площади ковра в сто тысяч раз объём работы увеличивается строго пропорционально в сто тысяч раз, и т. п.

- «найти имя в телефонной книге» требует всего лишь времени, логарифмически зависящего от количества записей ( $O(\log_2(n))$ ), так как, открыв книгу примерно в середине, мы уменьшаем размер «оставшейся проблемы» вдвое (за счет сортировки имен по алфавиту). Таким образом, в книге объёмом в 1000 страниц любое имя находится не больше, чем за  $\log_2 1000 \approx 10$  раз (открываний книги). При увеличении объёма страниц до ста тысяч проблема все ещё решается за  $\log_2 100000 \approx 17$  заходов.
- «найти имя в телефонной книге (электронной)» может осуществляться за один запрос  $\Theta(1)$ .

Выполним небольшое сравнение данных алгоритмов для худшего случая: число  $x$  является простым. В качестве большого простого возьмём 1000000007. Получились следующие результаты:

Оптимизация	Порядок ФВС	Время
Без оптимизаций	$O(n)$	1.654800
Шаг = 2	$O(n)$	1.1381
Поиск делителей до корня	$O(\sqrt{n})$	0.00005231
Поиск делителей до корня с шагом 2	$O(\sqrt{n})$	0.00003304

### 6.2.9 Поиск второго максимального значения

#### Поиск второго максимального значения

С клавиатуры вводятся числа. Признак конца ввода – 0. Если бы все введённые числа были упорядочены по невозрастанию, что какое бы значение стояло на втором месте (на первом месте стоит максимум, на втором – значение не большое максимума).

Решение более сложных задач стоит начать с грамотного подбора тестовых данных. Тестовые случаи:

Тестовые данные	Ожидаемый результат	Пояснение
0	"Пустая последовательность"	В последовательности может не быть элементов.
2 0	"Длина последовательности меньше двух"	Может не быть двух элементов
1 2 0	1	Простой тест без повторяющихся значений
2 2 0	2	Простой тест с повторяющимися значениями
1 2 3 3 0	3	Максимальные значения обновляются.

Блок-схема на рисунке 6.16.

---

```
#include <stdio.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    int max1;
    scanf("%d", &max1);

    if (max1 != 0) {
        int max2;
        scanf("%d", &max2);

        if (max2 != 0) {
            if (max1 < max2) {
                int t = max1;
                max1 = max2;
                max2 = t;
            }

            int x;
            scanf("%d", &x);

            while (x != 0) {
                if (x >= max1) {
                    max2 = max1;
                    max1 = x;
                } else if (x > max2)
                    max2 = x;

                scanf("%d", &x);
            }

            printf("max2 = %d", max2);
        } else
            printf("Длина последовательности меньше двух");
    } else
        printf("Пустая последовательность");

    return 0;
}
```

---

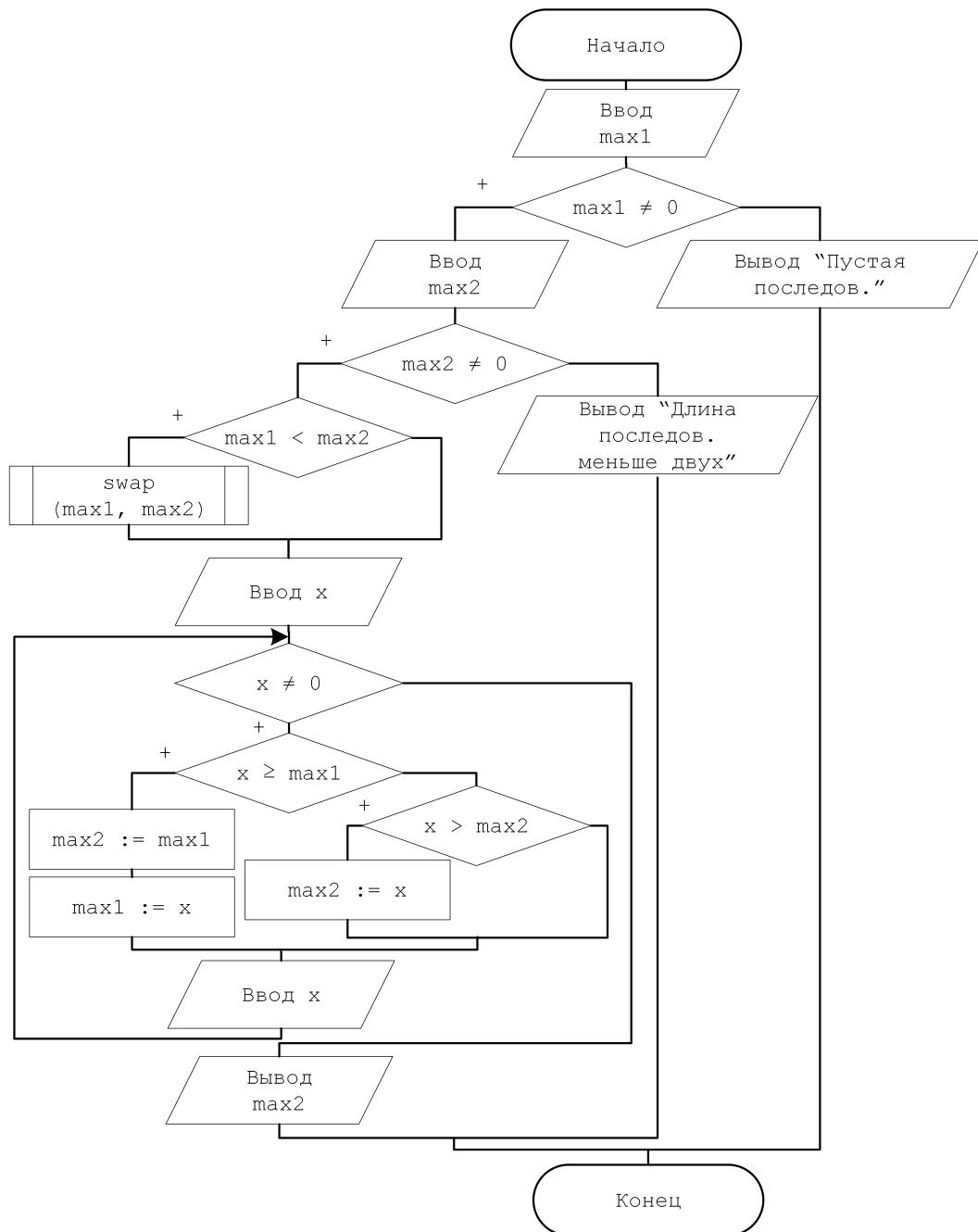


Рис. 6.16: Блок-схема задачи о поиске второго максимума

### 6.2.10 Максимальное количество подряд идущих положительных значений

**Максимальное количество подряд идущих положительных значений**

С клавиатуры вводятся числа. Признак конца ввода – 0. Необходимо найти подпоследовательность максимальной длины, в которой все элементы положительны. В качестве ответа требуется указать длину подпоследовательности.

Тестовые данные:

Тестовые данные	Ожидаемый результат	Пояснение
0	0	Нет ни одного элемента в последовательности
1 0	1	Один положительный элемент в последовательности
-1 0	0	Нет положительных, но есть отрицательные
1 -1 1 1 1 -1 0	3	Несколько последовательностей из положительных, проверка того, что длина обновится

---

```
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    int maxLen = 0;
    int currentLen = 0;
    while (x != 0) {
        if (x > 0) {
            // если число положительное, увеличиваем текущую длину
            currentLen++;
            // проверяем, длиннее ли текущая цепочка максимальной
            // если да - обновляем максимум
            if (currentLen > maxLen)
                maxLen = currentLen;
        } else
            // иначе скидываем счётчик положительных значений.
            currentLen = 0;

        scanf("%d", &x);
    }

    printf("%d", maxLen);

    return 0;
}
```

---

### 6.2.11 Максимальное количество знакочередующихся элементов последовательности

#### Максимальное количество знакочередующихся элементов последовательности

С клавиатуры вводятся натуральные числа. Признак конца ввода – 0. Необходимо найти подпоследовательность максимальной длины, в которой наблюдается знакочередование. В качестве ответа требуется указать длину подпоследовательности.

Проверить знакочередование можно на основании следующего свойства:

$$a_{i-1} * a_i < 0$$

---

```
#include <stdio.h>

int main() {
    int last;
    scanf("%d", &last);

    int maxLen = 0;
    // инициализация выполнена нулём, элементов может не быть
    int currentLen = 0;
    if (last != 0) {
        int cur;
        scanf("%d", &cur);

        // один элемент сам по себе образует знакочередующуюся
        // последовательность длины 1; именно по этой причине, когда
        // знакочередование заканчивается, сброс идёт в единицу
        currentLen += 1;
        while (cur != 0) {
            if (last * cur < 0)
                currentLen++;
            else {
                if (currentLen > maxLen)
                    maxLen = currentLen;
                currentLen = 1;
            }
            last = cur;

            scanf("%d", &cur);
        }
    }

    if (currentLen > maxLen)
        maxLen = currentLen;

    printf("%d", maxLen);

    return 0;
}
```

---

## 6.2.12 Подпоследовательность с максимальной суммой

### Подпоследовательность с максимальной суммой

С клавиатуры вводятся натуральные числа (гарантируется наличие одного положительного). Признак конца ввода – 0. Необходимо найти подпоследовательность с максимальной суммой. В качестве ответа требуется указать максимальное значение суммы.

Идея решения изображена ниже.

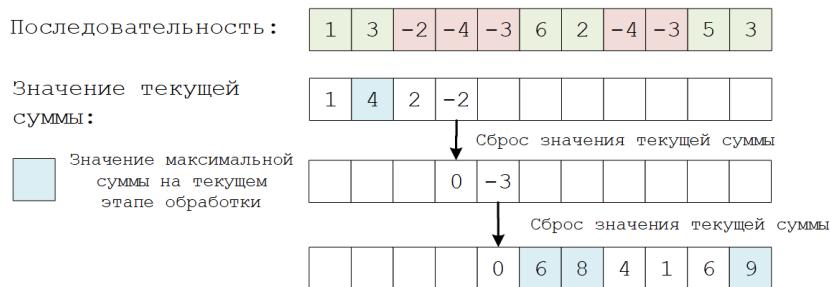


Рис. 6.17: Изменения значений *curSum* и *maxSum*

Создаётся переменная, которая хранит максимальное значение суммы, а также текущей суммы. Если текущая сумма больше максимума, происходит обновление результата.

---

### Листинг 10 Алгоритм Джая Кадана

---

```
#include <stdio.h>

int main() {
    long long curSum; // значение текущей суммы
    scanf("%lld", &curSum);

    long long maxSum = curSum; // максимальная сумма подпоследовательности
    if (maxSum != 0) {
        int x;
        scanf("%d", &x);

        while (x != 0) {
            curSum += x; // добавляем элемент к последовательности
            if (curSum > maxSum)
                maxSum = curSum; // обновление максимума
            if (curSum < 0)
                curSum = 0; // если значение суммы стало отрицательным,
                           // сбрасываем сумму в ноль, начинаем
                           // новую последовательность
            scanf("%d", &x);
        }
    }
    printf("%lld", maxSum);

    return 0;
}
```

---

### 6.2.13 Поиск элемента находящегося перед первым четным

#### Поиск элемента находящегося перед первым четным

С клавиатуры вводятся натуральные числа. Признак конца ввода – 0. Необходимо найти элемент, находящийся перед первым четным. Даже если ответ был найден, продолжайте ввод до нуля.

Блок-схема решения задачи представлена на рисунке 6.18. Подумайте, какие случаи возможны. Предложу свои:

Тестовые данные	Ожидаемый результат	Пояснение
0	"Пустая последовательность"	В последовательности может не быть элементов.
2 0	"Четное число первое в последовательности"	Перед четным элементом может ничего не быть.
1 2 0	1	Простой тест, при котором имеется ответ.
1 2 3 4 0	1	Важно убедиться, что выведется число перед первым четным (а не перед последним).

Код решения задачи:

```
#include <stdio.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    int x;
    scanf("%d", &x);

    int r = -1;
    int last = 0;
    while (x != 0) {
        if (r == -1 && x % 2 == 0)
            r = last;
        last = x;

        scanf("%d", &x);
    }

    if (last == 0)
        printf("Последовательность пуста");
    else if (r == 0)
        printf("Четное число - первое в последовательности");
    else if (r == -1)
        printf("Четных чисел не было");
```

```

else
    printf("%d", r);

return 0;
}

```

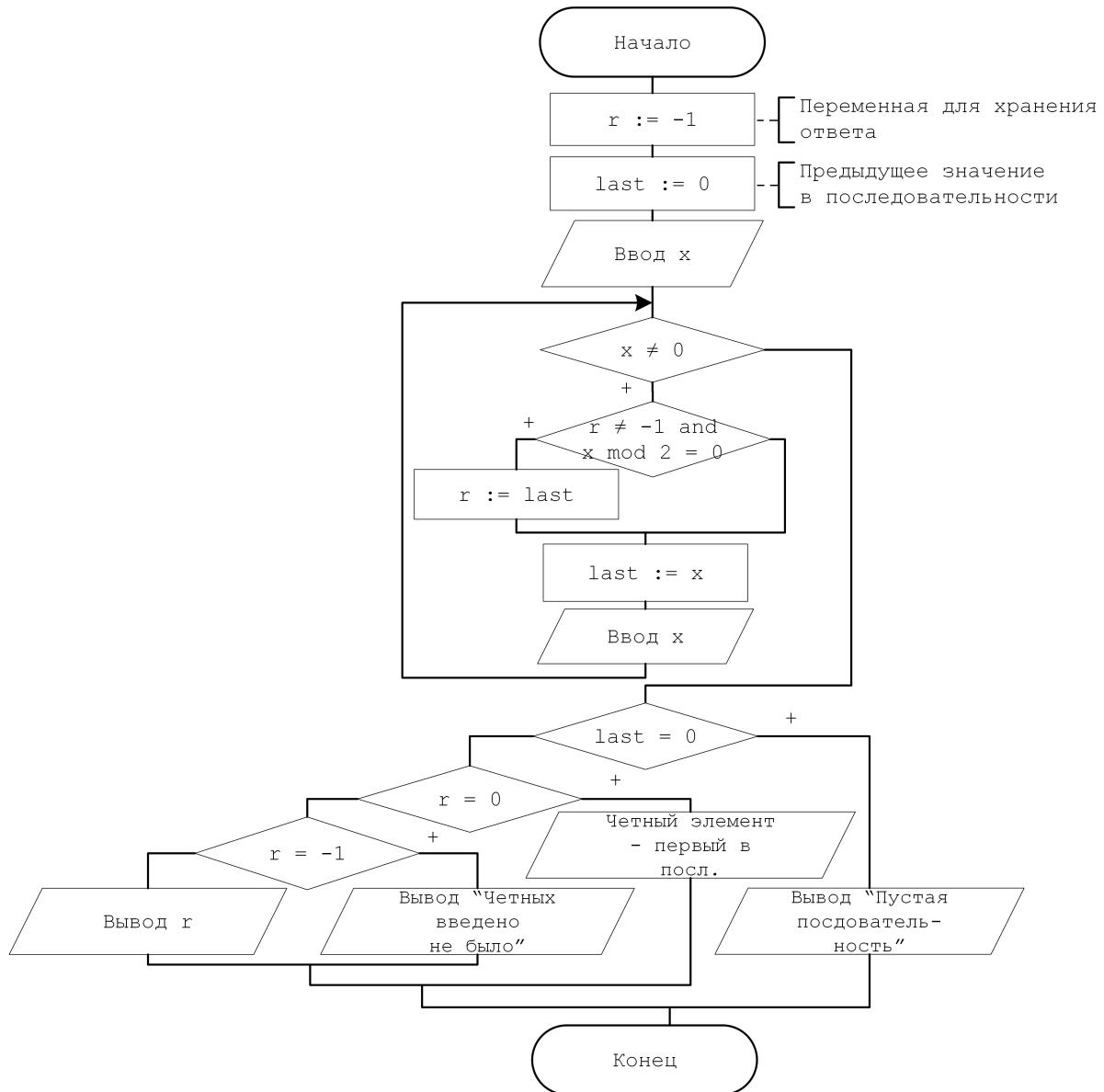


Рис. 6.18: Блок-схема к задаче о поиске числа перед первым чётным

#### 6.2.14 Поиск элемента, находящегося после последнего минимального

**Поиск элемента находящегося после последнего минимального**

С клавиатуры вводятся положительные числа. Признак конца ввода – 0. Необходимо найти элемент, находящийся после последнего минимального.

В данном решении создана переменная `needToSaveNext`, значение которой определяет, записываем ли мы текущее вводимое число в качестве ответа. Если `needToSaveNext = 1`, значение записывается, иначе – не записывается. Рассмотрим тесты:

Тестовые данные	Ожидаемый результат	Пояснение
0	"Пустая последовательность"	В последовательности может не быть элементов.
2 0	"Минимальное число первое в последовательности"	После минимального элемента элементов может не быть.
1 2 0	2	Простой тест, при котором имеется ответ.
1 2 3 0	2	Проверяем, что будет записано значение 2, а не 3.
1 2 1 10 0	10	Проверяем, что будет найдено значение после последнего минимума.

```
#include <stdio.h>
#include <windows.h>

int main() {
    int x;
    scanf("%d", &x);

    int needToSaveNext = 0;
    int min = x;
    int next = -1;
    while (x != 0) {
        if (x <= min) {
            min = x;
            needToSaveNext = 1;
        } else if (needToSaveNext) {
            next = x;
            needToSaveNext = 0;
        }
        scanf("%d", &x);
    }

    if (needToSaveNext == 1)
        printf("the minimum is last");
    else if (next == -1)
        printf("empty sequence");
    else
        printf("%d", next);

    return 0;
}
```

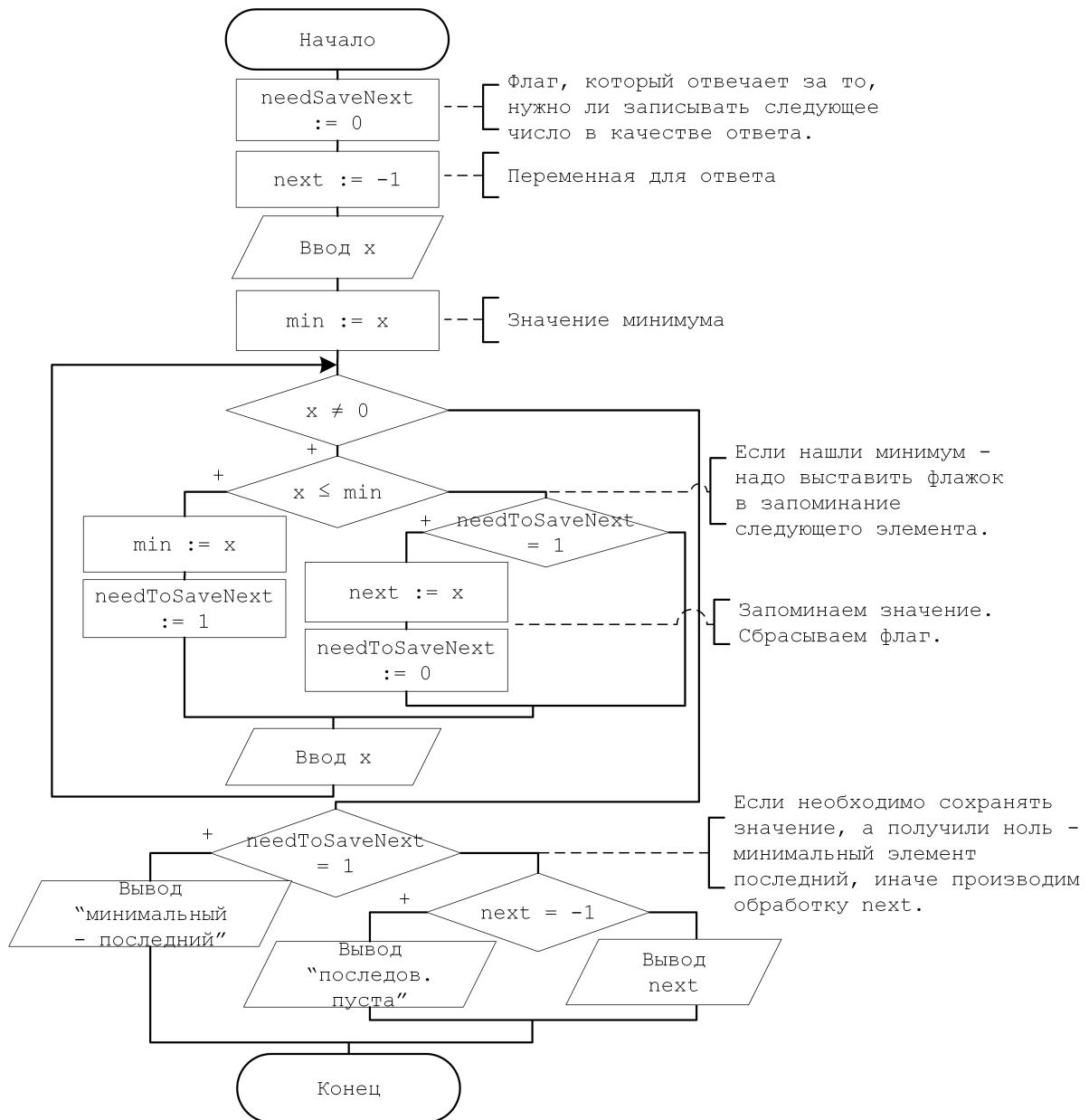


Рис. 6.19: Блок-схема к задаче о поиске элемента после последнего минимального

## 6.3 Цикл *do-while*

Цикл *do-while* описывается конструкцией:

---

```
// do
//     <оператор>
// while (<логическое выражение>)
```

---

и является циклом с постусловием. Тело данного цикла (*<оператор>*) выполнится хотя бы раз. Затем будет выполнена проверка истинности *<логического выражения>*. Если выражение истинно, будет выполнен переход к следующей итерации цикла, в противном случае – конец цикла.

Обозначение на блок-схемах представлено на рисунке 6.20.

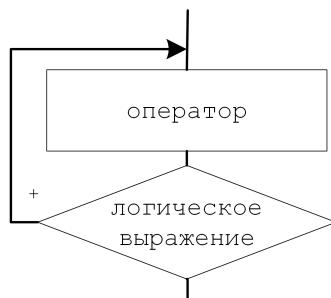


Рис. 6.20: Обозначение цикла *do – while*

### 6.3.1 Поиск количества цифр в числе

#### Поиск количества цифр в числе

С клавиатуры вводится число *x*. Найти *n* – количество цифр в записи числа *x*.

Блок-схема алгоритма представлена на рисунке 6.21. Код решения задачи:

---

```
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);

    int nDigits = 0;
    do {
        x /= 10;
        nDigits += 1;
    } while (x != 0);

    printf("%d", nDigits);

    return 0;
}
```

---

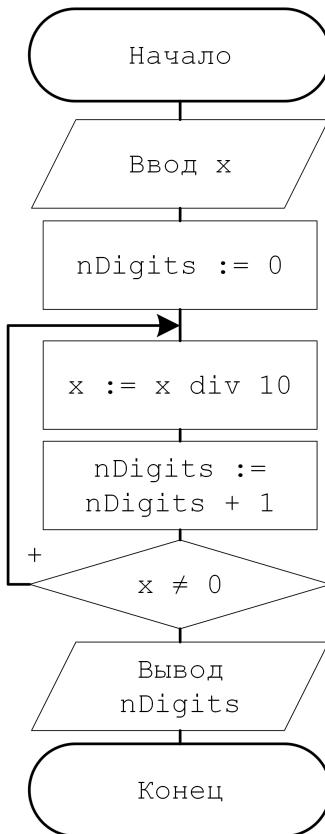


Рис. 6.21: Блок-схема решения задачи о количестве цифр

## 6.4 Выбор подходящего цикла

При наличии трех конструкций циклов возникает вопрос: какой выбрать? Исследователи в области вычислительной техники считают циклы с предусловием более удачными по следующим причинам:

1. В большинстве случаев условие проверяется до выполнения тела цикла.
2. Программа проще для восприятия.
3. В циклах с постусловием тело цикла выполнится хотя бы раз. Существует много задач, в которых тело не должно выполниться ни разу при определенных исходных данных.

Решили, что в основном будем работать с циклами с предусловием. Какой же выбрать: `while` или `for`? Частично это дело вкуса: что можно сделать при помощи одного цикла – можно сделать и при помощи другого. Вот пример цикла `for`

---

```
// for ( ; <условие возобновления цикла> ; )
```

---

который идентичен циклу `while`:

---

```
// while (<условие возобновления цикла>)
```

---

Или пример цикла `while`:

---

```
// <инициализирующее выражение>
// while (<условие возобновления цикла>) {
//     <тело цикла>
//     <корректирующее выражение>
// }
```

---

который работает почти так же, как цикл `for`<sup>7</sup>. Поэтому какого-то универсального правила разграничения нет.

## 6.5 Операторы *break*, *goto*, *continue*

### 6.5.1 *break*

Иногда бывает удобным выйти из цикла не по результату проверки, а каким-то другим способом. Такую возможность для циклов `for`, `while` и `do-while`, а также для переключателя `switch` предоставляет инструкция `break`. Эта инструкция вызывает немедленный выход из самого внутреннего из объемлющих ее циклов или переключателей:

---

```
int a = 10;
while (a > 0) {
    ...

    int x;
    scanf("%d", &x);

    if (x % 2 == 0)
        break;
    ...
    a--;
}
```

---

Если на какой-то итерации<sup>8</sup> будет введено четное значение, выполнится выход из цикла. Обозначение:

---

<sup>7</sup>Существует два незначительных различия:

- Разное поведение с оператором `continue`.
- Если инициализирующее выражение содержит объявление переменной, то в случае с циклом `while` та переменная будет видна после цикла, в отличие от цикла `for`.

---

<sup>8</sup>Итерация – в узком смысле – один шаг итерационного, циклического процесса

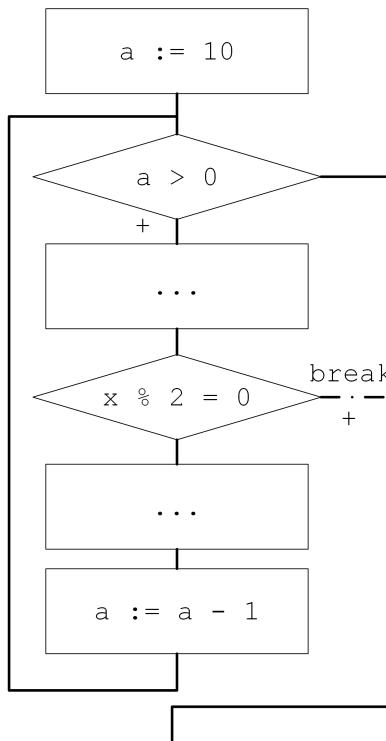


Рис. 6.22: Обозначение оператора *break* на блок-схеме

Оператор `break` позволяет выполнить выход из бесконечного цикла.

### 6.5.2 *continue*

Инструкция `continue` вынуждает ближайший объемлющий ее цикл (`for`, `while` или `do-while`) начать следующий шаг итерации. Для `while` и `do-while` это означает немедленный переход к проверке условия, а для `for` – к приращению шага.

---

```

#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        int x;
        scanf("%d", &x);

        if (x % 2 == 0)
            continue;
        // ... обработка x
    }

    return 0;
}
  
```

---

Однако конструкция выше может быть записана и без использования `continue`:

---

```

#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
  
```

---

```

int x;
scanf("%d", &x);

if (x % 2 != 0) {
    // ... обработка x
}
}

return 0;
}

```

---

Но в таком случае несколько увеличивается уровень вложенности (что не всегда является желательным).

### 6.5.3 goto

Посредством инструкции `goto` можно выполнить переход к метке. Метка имеет вид обычного имени переменной, за которым следует двоеточие `:`. На метку можно перейти с помощью `goto` из любого места данной функции, т. е. метка видима на протяжении всей функции.

Раньше `goto` крайне широко использовался в языках низкого уровня для организации управляющих конструкций. Код со сложной, запутанной `goto` логикой именуемой *спагетти код*:



Несмотря на порицание в сообществе программистов можно найти ей правильное применение. Например, если нужно выйти из вложенных циклов:

```

for (i = 0 ; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i] == b[j])
            goto found;
/* нет одинаковых элементов */

found:
/* обнаружено совпадение: a[i] == b[j] */

```

---

Если `a[i] == b[j]` программа выполнит переход к метке `found`.

Программу нахождения совпадающих элементов можно написать и без `goto`, правда, заплатив за это дополнительными проверками и еще одной переменной:

---

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;

if (found)
    /* обнаружено совпадение */
    ...
else
    /* нет одинаковых элементов */
    ...
```

---

## Резюме

- Цикл `for` работает следующим образом:
  1. Вычисляется значение `<инициализирующего выражения>`.
  2. Проверяется истинность `условия возобновления цикла`.
  3. Если оно истинно – выполняется `<оператор>` и `<корректирующее выражение>`. Если `<условие возобновления>` ложно – конец цикла.
- По окончанию работы `for` переменные, создаваемые в заголовке или теле цикла, уничтожаются.
- Цикл `while` работает следующим образом:
  1. Проверяется `<истинность логического выражения>`;
  2. Если выражение истинно, выполняется `<оператор>`, иначе – конец цикла.
- Цикл `while` может стать бесконечным, если не влиять на переменные, которые указаны в логическом выражении.
- Цикл `do-while` работает следующим образом:
  1. Тело данного цикла (`<оператор>`) выполнится хотя бы раз.
  2. Затем будет выполнена проверка истинности `<логического выражения>`. Если выражение истинно, будет выполнен переход к следующей итерации цикла, в противном случае – конец цикла.
- Универсального правила для выбора одного из трех конструкций цикла нет.
- Инструкция `break` вызывает немедленный выход из самого внутреннего из объемлющих ее циклов или переключателей.
- Инструкция `continue` вынуждает ближайший объемлющий ее цикл начать следующую итерацию.
- Инструкция `goto` выполняет переход к метке.

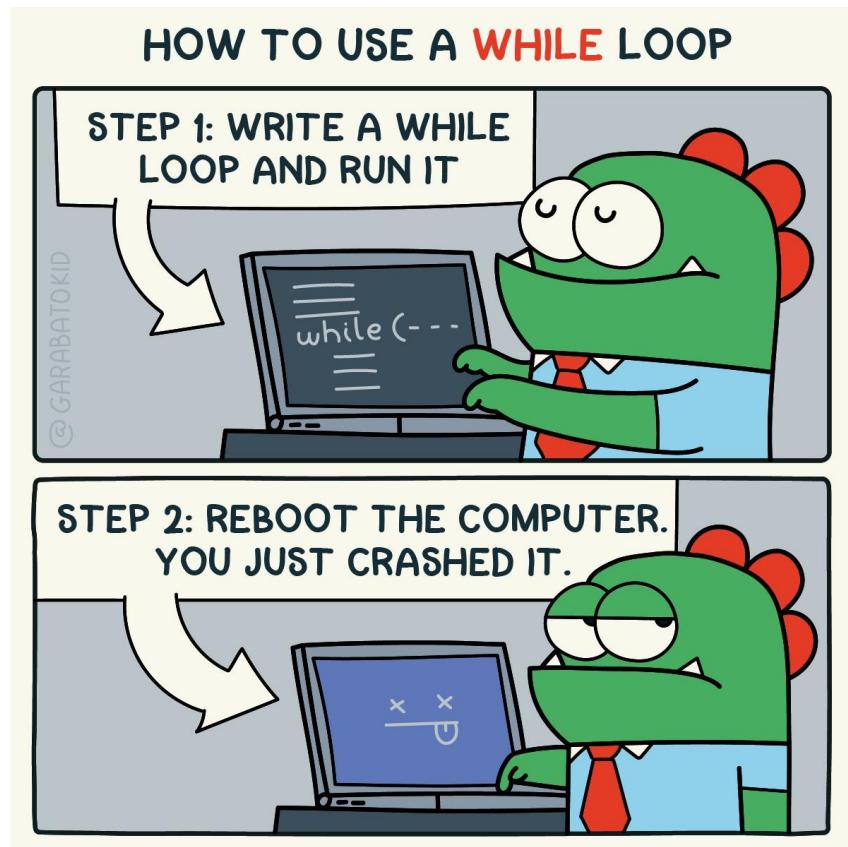
## Термины и определения

- **Циклический алгоритм** – алгоритм, в которых некоторая часть операций выполняется многократно.
- **Итерация** – в узком смысле – один шаг итерационного, циклического процесса

## Контрольные вопросы

1. Какие алгоритмы называются циклическими?
2. Перечислите все конструкции циклов. Опишите принцип их работы.
3. В чем заключается разница между циклом `while` и циклом `do-while`?

4. В каком случае цикл `while` может стать бесконечным?



5. Сколько раз обязательно выполнится тело цикла `do-while`?
6. Инструкции `break` и `goto`. Когда имеет смысл их применять?
7. Как заменить цикл `for` 'эквивалентным' через цикл `while`.
8. Инструкция `continue`. В чём разница применения `continue` для цикла `for` и `while`?
9. Напишите программу, определяющую количество четных цифр в числе.
10. Напишите программу, находящую последнее вхождение четного элемента в последовательности длины `n`.

# Глава 7

## Функции

**Функция** – это самодостаточная единица кода программы, спроектированная для выполнения отдельной задачи. Почему мы должны использовать функции:

- Избавляют от необходимости многократного написания одного и того же кода. При дублировании кода вы нарушаете принцип *DRY* (*Don't repeat yourself*)<sup>1</sup>. Написанные функции могут использоваться и в других проектах.
- Даже если задача решается всего лишь один раз в единственной программе, использование функции имеет смысл, т.к. это делает программу более модульной, таким образом улучшая ее читабельность и упрощая внесение изменений либо исправлений.
- Небольшие фрагменты кода, выполняющие функциональность проще протестировать, как вручную, так и автоматизировано. Проверить, как работает часть чего-нибудь легче, чем всё целое.
- Если появится более удачное решение задачи, которое возлагалось на функцию, модернизация затронет лишь небольшой фрагмент. Это улучшит и работу тех функций, которые использовали данную функцию.
- Для того, чтобы некто смог использовать реализованную другим функцию, ему даже необязательно знать, как она работает. Достаточно ответить на вопрос, что делает функция и какие параметры необходимы для её работы (например, вы можете легко использовать функции ввода/вывода без понимания механизмов работы).

В каждой программе на С/С++ должна присутствовать функция `main`, которая получает управление при запуске программы. Все остальные функции, необходимые для решения задачи, вызываются из `main`. Они обмениваются информацией с помощью параметров, которые получают при вызове, и возвращаемых значений.

Функции используются для того, чтобы организовать программу в виде совокупности небольших и не зависящих друг от друга частей.

---

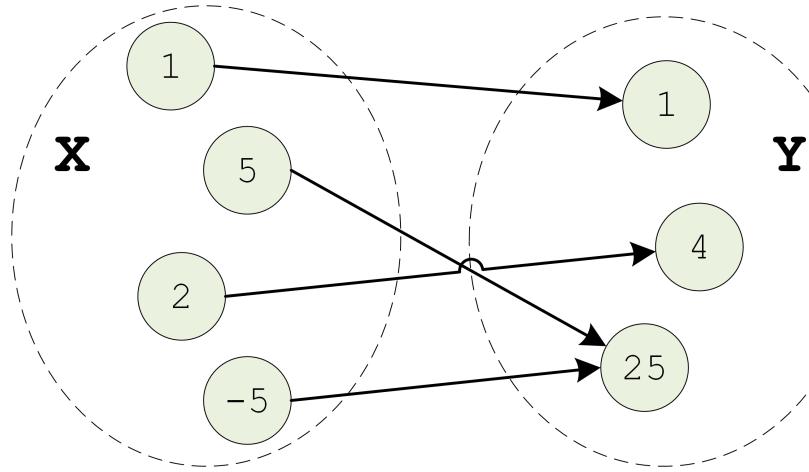
<sup>1</sup>Нарушения принципа *DRY* называют *WET* — «*Write Everything Twice*» (рус. Пиши всё дважды) или «*We enjoy typing*» (рус. Нам нравится печатать). Это игра английских слов «*dry*» (рус. сухой) и «*wet*» (рус. влажный).

## 7.1 Функции в математике

Давайте подумаем, что представляют собой функции в математике? Зачем мы их используем? Вспомним что-нибудь простое:

$$y = f(x) = x^2$$

**Функция** (отображение, оператор, преобразование) в математике — соответствие между элементами двух множеств — правило, по которому каждому элементу первого множества соответствует один и только один элемент второго множества.



Например, в данном случае значению функции в точке  $x = 5$  соответствует

$$f(5) = 5^2 = 25$$

Сколько бы мы раз не считали  $f(5)$  значение будет одним и тем же при неизменных параметрах.

Параметров может быть больше, чем один, например:

$$f(x, y) = x^2 + y^2$$

$$f(1, 4) = 1^2 + 4^2 = 17$$

Результат вычисления одной функции может быть использован для вычисления другой функции:

$$f(x, y) = x^2 = y^2 \quad g(x) = 2x$$

$$f(1, g(10)) = f(1, 20) = 1^2 + 20^2 = 401$$

Функции могут иметь разные имена, разное количество параметров и выполнять куда более сложные действия:

$$P_a(ns, k, p) = \pi_{ka} q^{ns} \sum_{m=1}^{\left[\frac{ns}{k}\right]} m \prod_{i=1}^m \frac{p}{q^k} \left( \frac{ns - mk}{i} + 1 \right)$$

Последняя функция находит некоторую вероятность  $P_a$  (вещественное число), принимает 3 параметра  $ns, k, p$  и показывает как вычисляется её значение

$$\pi_{ka} q^{ns} \sum_{m=1}^{\left[\frac{ns}{k}\right]} m \prod_{i=1}^m \frac{p}{q^k} \left( \frac{ns - mk}{i} + 1 \right).$$

## 7.2 Структурное программирование. Метод пошаговой детализации

Рассмотрим несколько задач и попробуем на них понять, зачем же нам нужны функции? А если ещё точнее, откуда появилась потребность функций в программировании? Рассмотрим несколько задач.

### 7.2.1 Задача о кирпиче

#### Задача о кирпиче

Имеется кирпич размера  $a \times b \times c$ . Имеется прямоугольное отверстие в стене размера  $h \times w$ . Можно ли через отверстие 'протиснуть' кирпич?

Будем разрабатывать алгоритм методом пошаговой детализации. Если бы стороны отверстия и ребра кирпича были упорядочены, то для решения задачи достаточно сравнить меньшую сторону с меньшим ребром и большую сторону со средним ребром. Таким образом, на первом этапе детализации выделяем две подзадачи:

- Упорядочение пары чисел по неубыванию
- Упорядочение тройки чисел по неубыванию

Опишем алгоритм в терминах выделенных подзадач (рисунок 7.1).

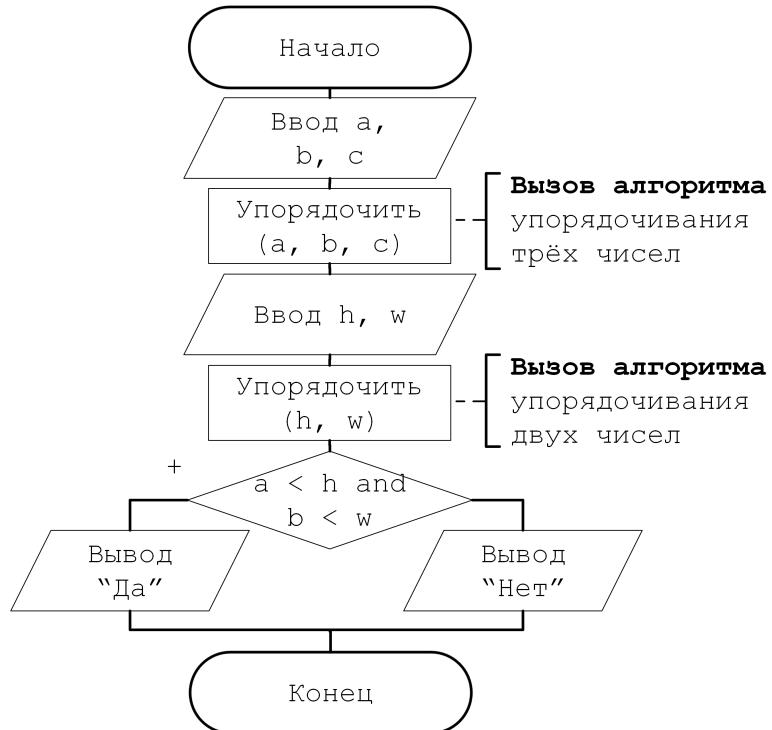


Рис. 7.1: Блок-схема алгоритма в укрупненных блоках

Блоки, в которых сформулированы выделенные подзадачи, будем называть укрупненными блоками. Они должны содержать имена переменных, значения которых являются исходными данными для решаемой подзадачи (**входные параметры**), и имена переменных, в которые будут записаны результаты (**выходные параметры**). Например, чтобы упорядочить три числа, в качестве входных данных для

алгоритма будут выступать три числа  $a, b, c$  (их называют входными параметрами). После упорядочивания мы получим, что

$$a \leq b \leq c$$

т.е. после процедуры упорядочивания нам важны обновленные значения  $a, b, c$  (данные переменные являются выходными параметрами). Каждая переменная может быть как входным параметром, так и выходным. Если значения переменных нужны для решения задачи – это входной параметр. Если значения нужны для того, чтобы хранить результат действия – выходной параметр. Если для первого и второго – и выходным, и выходным параметром. В рассматриваемом примере в обоих укрупненных блоках параметры являются как входными, так и выходными.

Теперь алгоритм решения каждой подзадачи опишем как самостоятельный алгоритм. Алгоритмы упорядочивания последовательностей называют **сортовками**. Назовем алгоритм сортировки пары *sort2*. Решение задачи сортировки пары заключается в обмене значениями переменных  $a$  и  $b$ , если  $a > b$ .

Решим задачу сортировки двух чисел как самостоятельную. И выполним определенный переход к следующей блок-схеме:

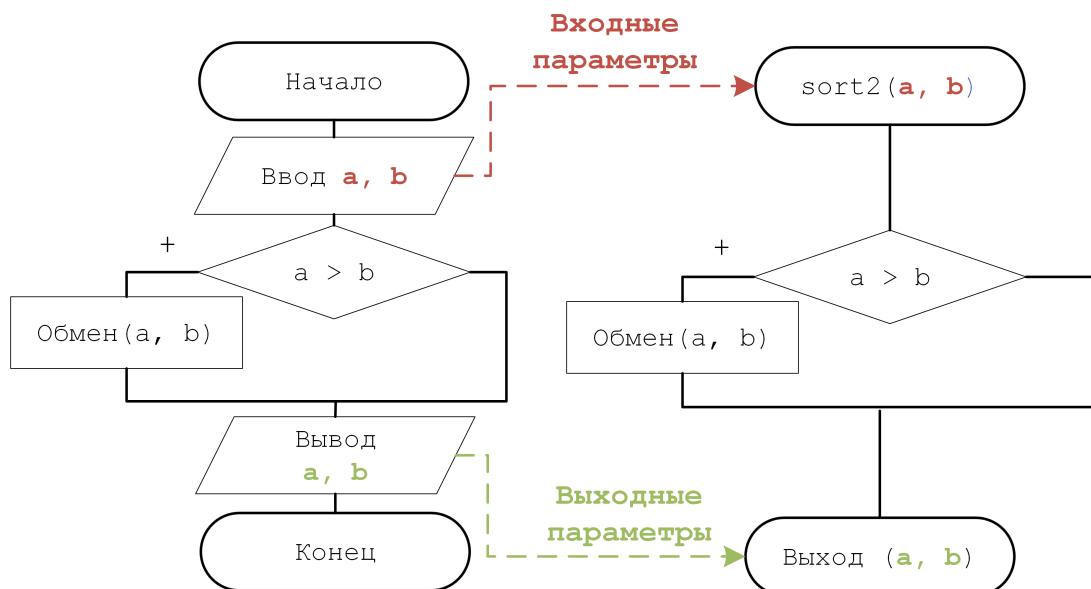


Рис. 7.2: Переход от самостоятельной задачи к функции на блок-схеме

Но в блок-схеме для решения задачи упорядочивания содержится задача обмена значений двух переменных. Опишем её таким же образом:

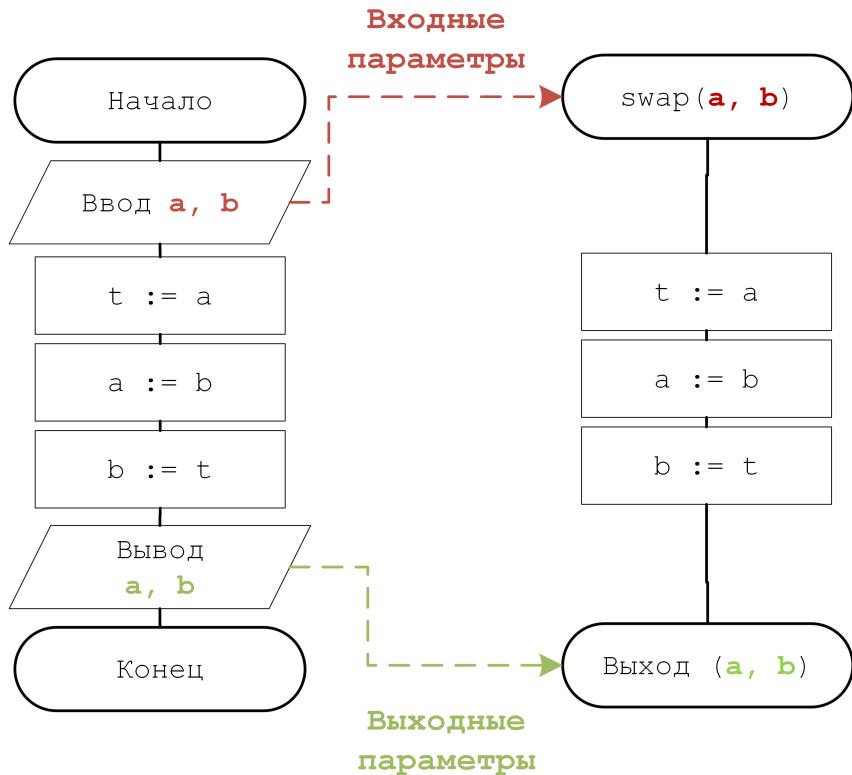


Рис. 7.3: Переход от самостоятельной задачи к функции на блок-схеме

И переменная  $a$  и переменная  $b$  являются входными и выходными данными к задаче.

Мы научились упорядочивать пару чисел. Теперь осталось решить вопрос с тройками. Для этого достаточно выполнять 3 упорядочивания пар:

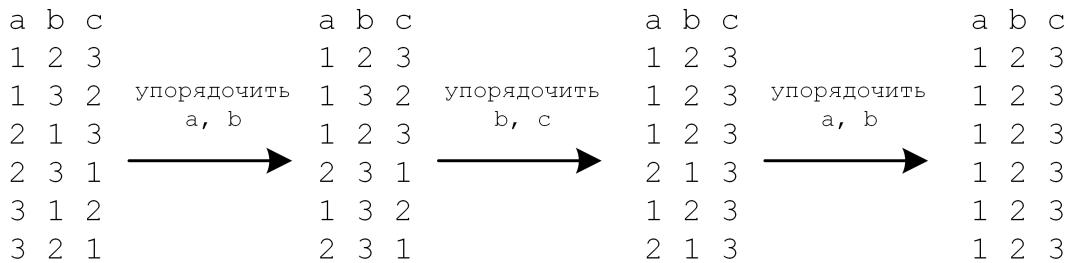


Рис. 7.4: Переход от самостоятельной задачи к функции на блок-схеме

Опишем алгоритм при помощи укрупненных блоков и при помощи блоков предопределенный процесс на рисунке 7.5. Для её решения можно использовать подзадачу сортировки пары  $sort2$ . Назовем подзадачу сортировки тройки  $sort3$ .  $a$ ,  $b$ ,  $c$  являются как входными, так и выходными параметрами.

Параметры, которые используются при описании алгоритма, называются **формальными**, а параметры, которые указываются при вызове алгоритма, называются **фактическими**; см рис. 7.7.

Описание решения задачи при помощи блоков предопределенный процесс на рисунке 7.6.

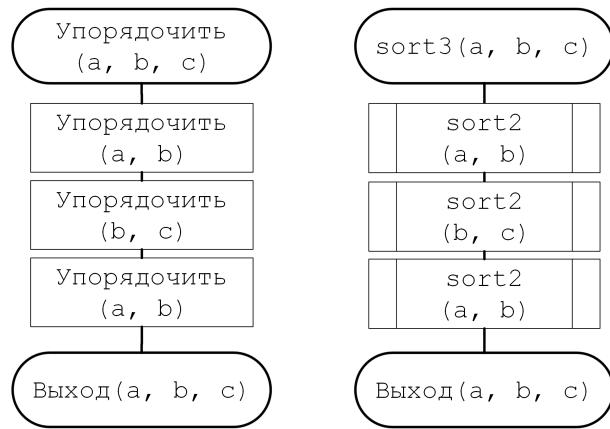


Рис. 7.5: Блок-схемы в укрупненных блоках и при помощи блоков предопределенный процесс

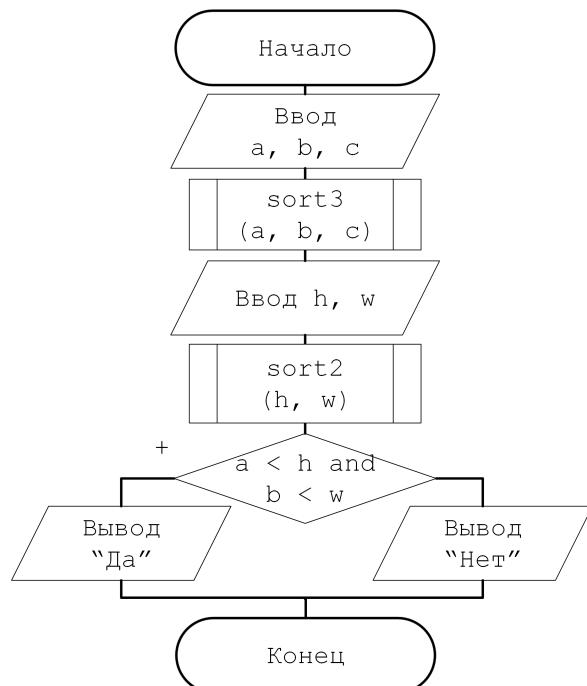


Рис. 7.6: Блок-схема решения задачи при помощи блоков предопределенный процесс

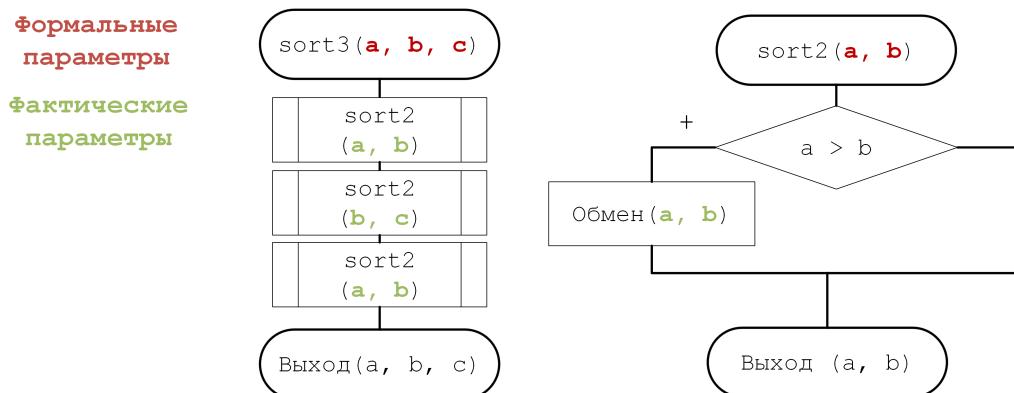


Рис. 7.7: Формальные и фактические параметры

## 7.2.2 Поиск максимального положительного значения из двух чисел

### Поиск максимального положительного значения из двух чисел

С клавиатуры вводятся два числа. Необходимо найти максимальное положительное число, или вывести значение 0, если оба числа не являются положительными.

Функции относятся к типу **функций с возвращаемым значением**, если возвращаемый ей результат должен быть подставлен в точку вызова. Например, функции в задаче 1.2.1 таковыми не являлись. В данном случае это не так. Посмотрите пример на рисунке 7.8, на котором описана решаемая задача при помощи блоков предопределенный процесс.

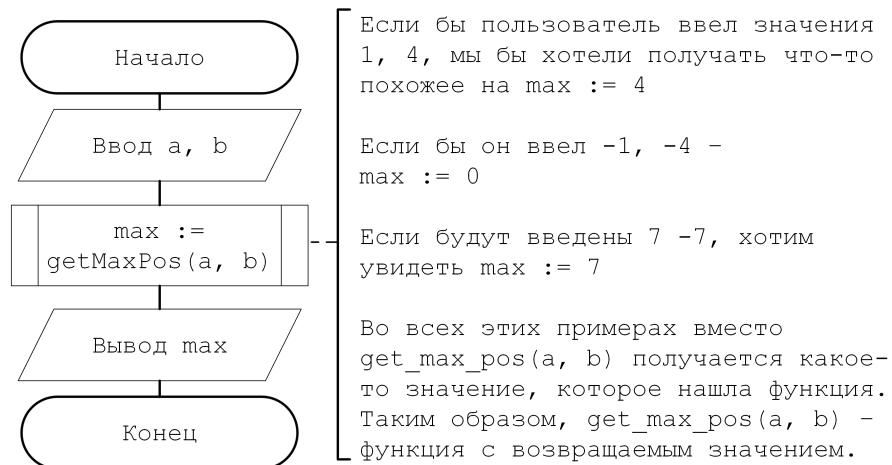


Рис. 7.8: Блок-схема задачи о максимальном положительном значении из двух

Опишем блок-схему для функции `getMaxPos`. В ней нет ничего необычного, кроме слова возврат (этой идеи пока что достаточно):

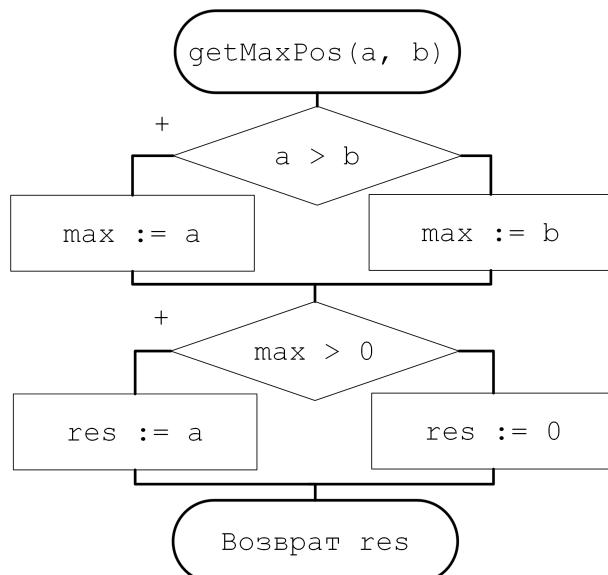


Рис. 7.9: Пример блок-схемы функции с возвращаемым значением

### 7.2.3 Задача о треугольнике

#### Задача о треугольнике

С клавиатуры вводятся координаты точек  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ . Определить тип треугольника: остроугольный, прямоугольный или тупоугольный?

Если задача кажется большой, начните решение с выделения подзадач. Не нужно лететь с низкого старта к персональной машине, потратьте время на размышления. Можем выделить следующие нетривиальные подзадачи:

- Поиск расстояния между парой точек (*getDistance*)
- Упорядочивание длин трёх сторон (*sort3*)
- Определение типа треугольника

Опишем алгоритм решения задачи при помощи блоков предопределенный процесс:

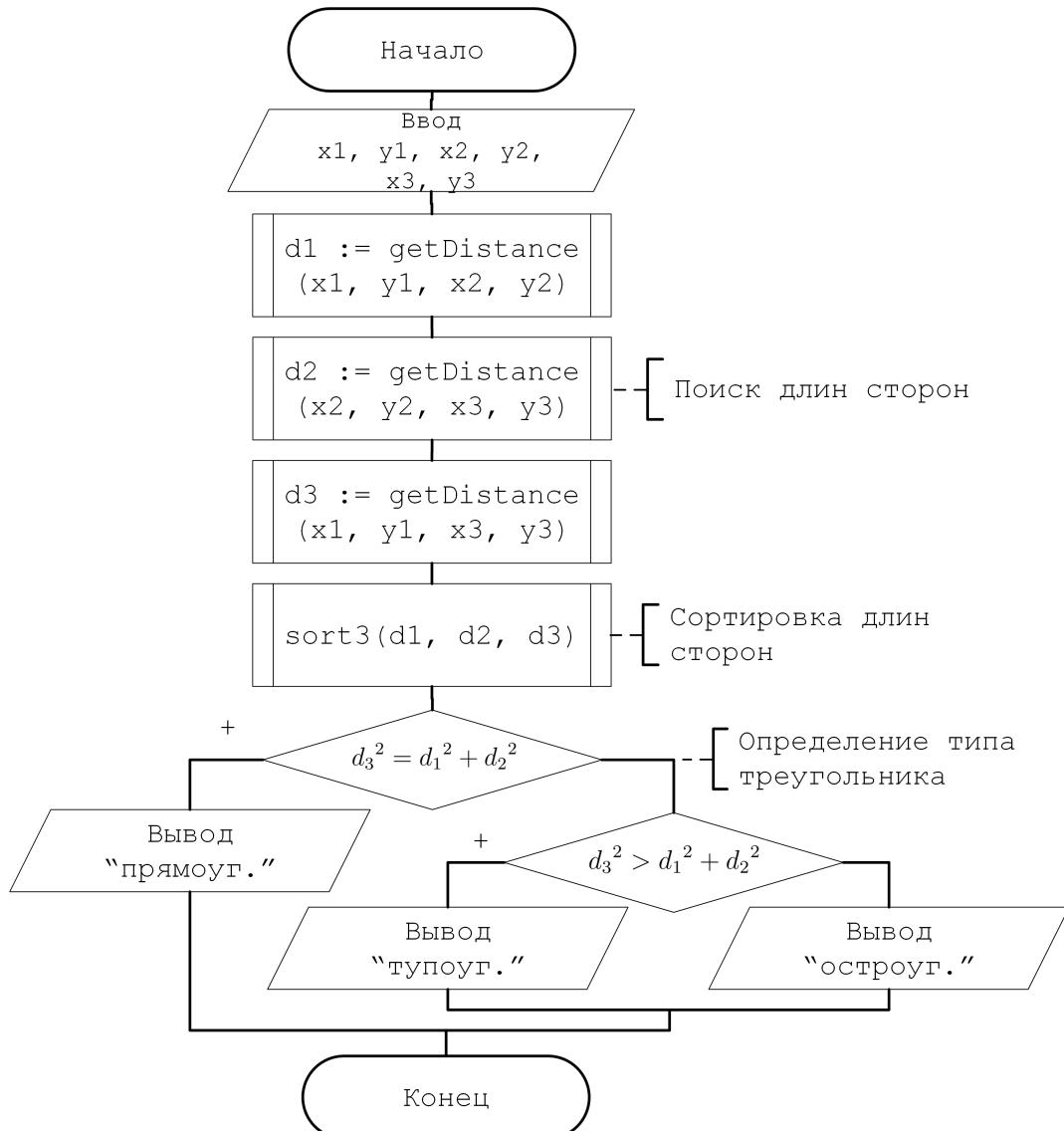


Рис. 7.10: Блок-схема с использованием блоков предопределенный процесс

Опишем алгоритм для вычисления расстояния между двумя точками:

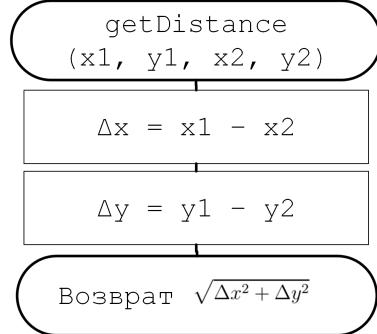
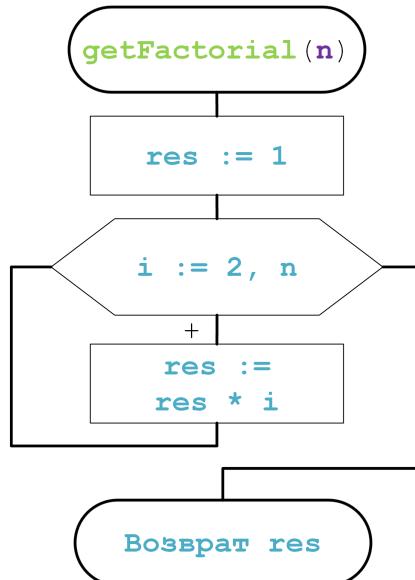


Рис. 7.11: Блок-схема функции *distance*

### 7.3 Определение функции

Опишем при помощи блок-схемы алгоритм, который находил бы значение факториала. Рассмотрим функцию `getFactorial`:



Заголовок функции / объявление функции / прототип функции		
Тип возвращаемого значения	Имя функции	Список параметров
<pre>long long getFactorial(int n) {     long long res = 1;     for (int i = 2; i &lt;= n; i++)         res *= i;     return res; }</pre>		

Рис. 7.12: Компоненты функций

Определение функции состоит из следующих частей:

---

```
// <тип возвращаемого значения> <имя функции>([<список параметров>]) {
//     <тело функции>
// }
```

---

Из данного определения мы можем вычленить:

- тип возвращаемого значения: `long long int`,
- имя функции: `getFactorial`,
- список параметров: одна переменная `n` типа `int`,
- тело функции (последовательность действий, которые осуществляет функция): несколько операторов, вычисляющие факториал числа.

Тип возвращаемого значения, имя функции, список параметров и тело составляют **определение функции**. Тип возвращаемого значения, имя функции, список параметров (необязательно с указанием имени формальных параметров) составляют **прототип функции / объявление функции**. Для функции `getFactorial` прототипами могут быть:

---

```
long long getFactorial(int)
long long getFactorial(int n)
```

---

В объявлении функции описывается её интерфейс. Он содержит все данные о том, какую информацию должна получать функция (список параметров), её имя и тип возвращаемого значения. Для прототипа неважно, что же именно функция делает. Заголовок нужен компилятору, чтобы он знал о существовании такого объекта.

Классическим примером в книгах о программировании является дверь. У двери есть ручка, которая управляет замком. Чтобы её открыть, вам не надо знать тонкости устройства механизмов, физику, содержание романа 'Война и мир' или что-то ещё. Вам просто известно, что нужно потянуть ручку вниз, и дверь откроется. Ручка двери – интерфейс, а как там это всё работает – реализация.

Что мы можем сказать о функции `getFactorial` только по её заголовку? Это нечто, что возвращает значение типа `long long`, которое (если верить названию) является факториалом числа `n`. Мы даём функции какое-то целое, а она уже неким магическим образом находит факториал.

Некоторые функции написаны до нас, например, `scanf`. Нас особо не интересовало как именно там всё работает (реализация). Достаточно было сведений об её интерфейсе: передал управляющую строку, адреса переменных. И всё считалось.

При помощи объявления компилятор в принципе узнаёт о том, что такая функция есть. Где-то есть.

## 7.4 Формальные и фактические параметры функции

Функцию можно рассматривать как операцию, определенную пользователем. Выполнение операции заключается в применении действий над операндами. Параметры функции (в такой аналогии) являются операндами.

**Параметры функции (формальные параметры функции)** — это переменные, создаваемые в объявлении функции:

---

```
// a, b -- формальные параметры
void funcName(int a, int b);
```

---

Имена формальных параметров не могут повторяться.

Если функция не содержит параметров, круглые скобки не могут быть опущены:

---

```
int funcName();
```

---

Если список параметров содержит несколько переменных одного и того же типа, нельзя осуществлять такое объединение:

---

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

---

каждый параметр должен быть описан отдельно:

---

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

---

**Аргумент (фактический параметр)** — это значение, которое передаётся в функцию при её вызове:

---

```
funcName(1, 3); // Аргументы: 1 и 3.
funcName(x, 7); // Аргументы: x и 7.
```

---

Аргументы функции должны быть перечислены через запятую.

Для того чтобы вызвать функцию, необходимо использовать оператор вызова ():

---

```
// <имя функции>(<список аргументов>)
```

---

Даже если функция не содержит формальных параметров, оператор вызова всё равно должен быть указан:

---

```
// <имя функции>()
```

---

## 7.5 Действия, производимые при вызове функции

Выполнение программы начинается с функции `main`. При достижении точки вызова функции `getFactorial` выполнение функции `main` приостанавливается. Происходит процесс передачи аргументов в функцию `print`: все параметры функции создаются как переменные, а значения аргументов копируются в переменные-параметры. Например, при вызове (строка 14):

---

```
#include <stdio.h>

long long getFactorial(int n) {
    long long res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;

    return res;
}

int main() {
    int n;
    scanf("%d", &n);

    printf("%lld", getFactorial(n));

    return 0;
}
```

---

в функции `getFactorial` создаётся переменная `n` и при выполнении тела вернёт в точку вызова `n!`. По окончанию работы `getFactorial` работа функции `main` возобновляется.

В процессе компиляции, компилятор проверяет возможность преобразования аргументов функции к типам формальных параметров. Рассмотрим на примере:

---

```
int sumAbs(int a, int b) {
    if (a < 0)
        a = -a;
    if (b < 0)
        b = -b;

    return a + b;
}

int main() {
    int a = sumAbs(1, 1);
    int b = sumAbs(1.3, -5.99);

    printf("%d, %d\n", a, b);

    return 0;
}
```

---

При вычислении значения `a` (строка 10) никаких преобразований выполнено не будет, в отличие от получения значения переменной `b` (строка 11). При вызове функции `sumAbs` переменные-параметры `a` и `b` функции `sumAbs` получат значения 1 и -5 соответственно, и будет возвращено значение 6.

Современные среды разработки выдают предупреждения о таких неявных преобразованиях:

Если мы попробуем выполнить вызов:

```

11 ► int main() {
12     int a = sumAbs( a: 1, b: 1);
13     int b = sumAbs( a: 1.3, b: -5.99);
14
15     printf(_Format: "%d, %d\n", a, b);
16
17     return 0;
18 }
```

---

```
int a = sumAbs("hello", "world");
```

---

получим ошибку, так как аргументы не могут быть преобразованы к типу формальных параметров.

Если передать меньше/больше аргументов в функцию, будет получена ошибка компиляции:

---

```
sum(1);
sum(1, 2, 3);
```

---

Ещё раз отметим, C/C++ являются языками со строгой типизацией. Компилятор проверяет соответствие типов при каждом вызове:

- Если тип аргумента может быть приведен к типу формального параметра, выполняется преобразование типов.
- Если преобразование невозможно, количество параметров при вызове и объявлении не совпадают, выдаётся ошибка компиляции.

Чтобы компилятор мог выполнить такие проверки, объявление функции должно предшествовать её вызову. Допустимы два варианта:

---

```

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

int main() {
    int a = sum(1, 1);
    int b = sum(1.3, -5.99);

    printf("%d, %d", a, b);

    return 0;
}
```

---

или

---

```

int sum(int a, int b);
// или int sum(int, int);

int main() {
    int a = sum(1, 1);
    int b = sum(1.3, -5.99);
```

---

```

    printf("%d, %d", a, b);

    return 0;
}

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

```

---

## 7.6 Возвращаемое значение функции

Результатом работы функции может быть некоторое значение, которое называют **возвращаемым значением функции**:

```

int abs(int x) {
    if (x < 0)
        x = -x;

    return x;
}

```

---

Из определения функции видно, что она возвращает значение типа `int`. Для осуществления возврата используется ключевое слово `return` за которым следует выражение. Значение выражения и является возвращаемым значением функции, которое передаётся в точку вызова:

```

int abs(int x) {
    if (x < 0)
        x = -x;

    return x;
}

int main() {
    int x = -3;
    int ax = abs(x); // ax = 3

    return 0;
}

```

---

В качестве типа возвращаемого значения может выступать как встроенный тип (такие как `int`, `double` и т.д.), так и пользовательский тип (например, `struct point`) и указатели на них (`int*`, `struct point*`). Важно отметить, что нельзя не указать тип возвращаемого значения:

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

---

Если функция не возвращает значение, то в качестве типа указывается `void`:

---

```
void print(int a, int b);
```

---

Массив не может быть типом возвращаемого значения. Ошибочно будет написать:

---

```
int[10] funcName();
```

---

Но можно вернуть указатель на необходимую область памяти:

---

```
int* funcName();
```

---

Возврат значения осуществляется одной из двух инструкций:

---

```
return;  
return expression;
```

---

Первый вариант используется для возвращения переменной типа `void`, второй вариант – для всех остальных случаев. Явно указывать `return`; не требуется. Данная строчка может быть опущена.

При достижении инструкции `return` – оставшаяся часть функции игнорируется и происходит возврат значения:

---

```
int hasOdd(const int *a, const int n) {  
    // a - указатель на массив из n элементов  
    for (int i = 0; i < n; i++)  
        if (a[i] % 2)  
            return 1;  
  
    return 0;  
}  
  
int main() {  
    int a[] = {1, 2, 3, 4, 5};  
    printf("%d", has_odd(a, 5));  
  
    return 0;  
}
```

---

Если тип возвращаемого значения не соответствует указанному в объявлении, происходит неявное преобразование типов:

---

```
int trunc(double a) {  
    return a;  
}  
  
int main() {  
    printf("%d", trunc(4.24)); // 4  
  
    return 0;  
}
```

---

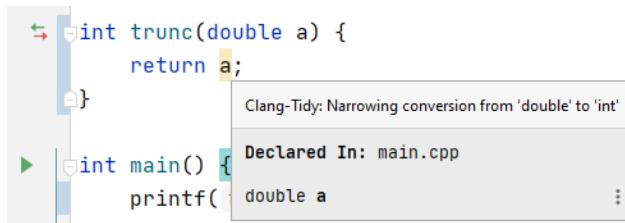


Рис. 7.13: Предупреждение *IDE* о неявном преобразовании при возврате значения

Среды выдают предупреждения при неявных преобразованиях (рисунок 18.15). Если же приведение невозможно, возникнет ошибка компиляции. В показанных примерах вы можете заметить, что возвращаемое значение функции используется в качестве аргумента для другой функции.

По умолчанию возвращаемое значение будет передано по значению (т. е. скопировано).

Нельзя вернуть указатель на начальный элемент массива, если он был создан внутри функции, так как все переменные по окончанию работы функции уничтожаются. Следующий код ошибочен (поведение неопределено):

```
int* initializationA() {
    int a[10];
    for (int i = 0; i < 10; i++)
        a[i] = i;

    return a; // массив a создан внутри функции getA.
              // нельзя вернуть ссылку на начальный элемент.
}

int main() {
    int *a = initializationA();

    return 0;
}
```

Так как поведение неопределено, не гарантируется, что будет получено то, что требуется. Есть даже какая-то вероятность того, что это сработает. Но не надо делать так.

Среда разработки *CLion* выдаёт следующее предупреждение:

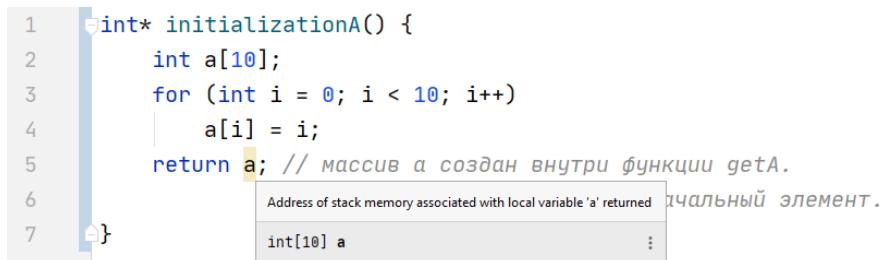


Рис. 7.14: Предупреждение *IDE* для попытки вернуть локальную переменную *a*, которая является созданным в функции массивом

Возможное решение проблемы можно осуществить через динамические массивы (что будет описано позже) или следующим образом:

---

```

void initializationA(int *a, const int n) {
    for (int i = 0; i < 10; i++)
        a[i] = i;
}

int main() {
    int a[10];

    // чтобы передать массив в функцию,
    // передаётся указатель на нулевой элемент массива
    initializationA(a, 10);

    return 0;
}

```

---

## 7.7 Передача аргументов

В языке программирования передача аргументов в функцию может осуществляться только по значению (переменная `a` получит значение 1, переменная `b` - значение 7):

---

```

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

int main() {
    int v1 = 1;
    int v2 = 7;
    int s = sum(v1, v2);

    printf("%d", s);

    return 0;
}

```

---

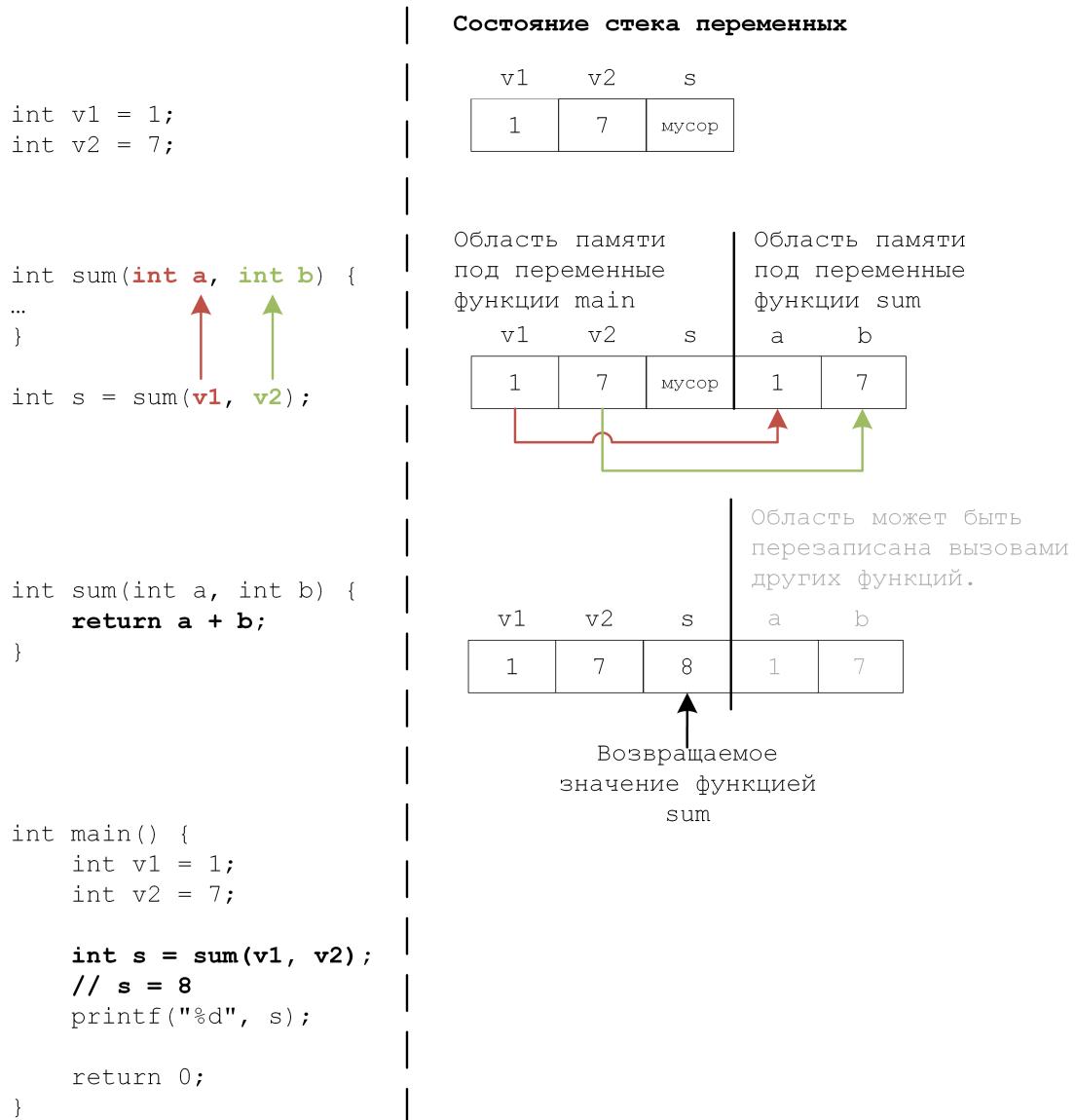


Рис. 7.15: Передача аргументов в функцию и возврат значения из функции

При этом способе передачи функция `sum` не получает доступа к переменным `v1` и `v2` а оперирует значениями локальных переменных `a` и `b`. По окончанию работы функция осуществляет возврат значения в точку вызова. Созданные на стеке функцией `sum` переменные `a` и `b` остаются там же. Но будут затёрты при первом же вызове функции из `main`. Рассмотрим такой пример:

```
void reset(int a) {
    a = 0;
}

int main() {
    int a = 42;
    reset(a);
    printf("%d", a);

    return 0;
}
```

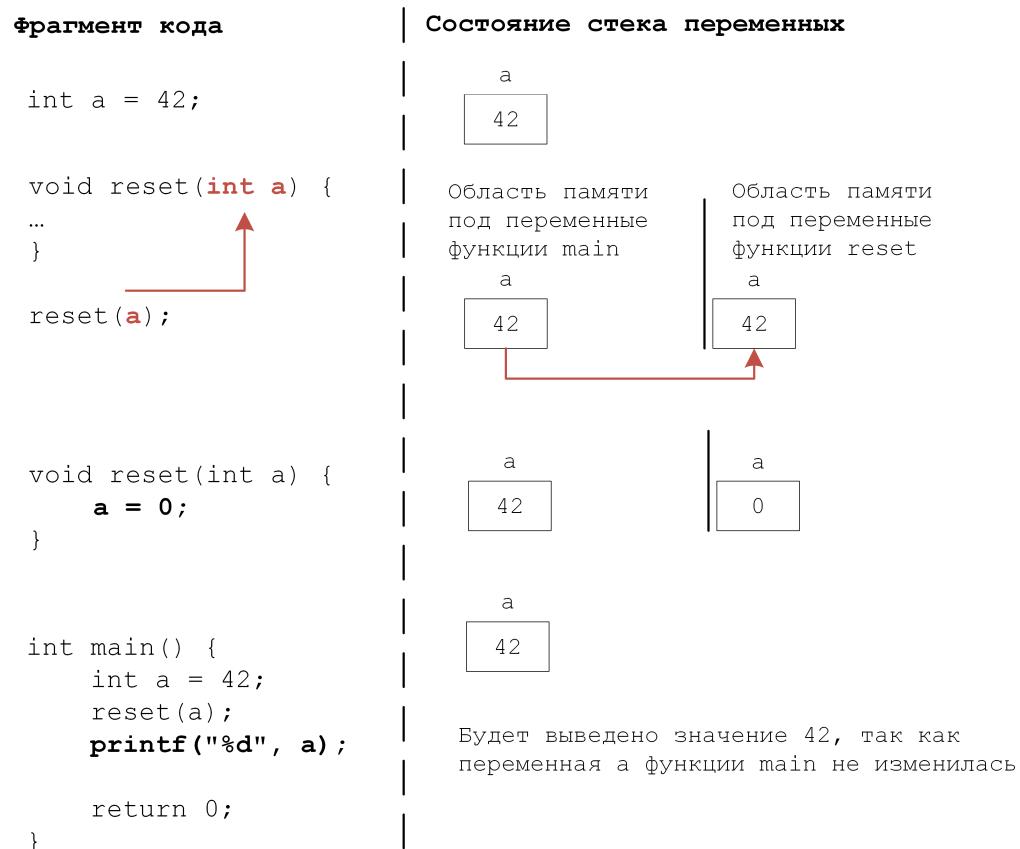


Рис. 7.16: Передача значения в функцию

В строке 8 будет выведено значение 42. Функция `reset` при своём вызове создаёт локальную переменную `a`, которая уничтожается после вызова функции. Данный способ передачи удобен, если передаваемый объект не является большим, так как при вызове функции происходит копирование. В примере выше, переменная `a` функции `main` копируется в переменную `a` функции `reset`. Это две разные переменные `a`, несвязанные между собой.

Иногда требуется, чтобы функции модифицировали переданные аргументы. Опишем функцию обмена двух значений. Вариант:

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

int main() {
    int a = 1;
    int b = 2;
    printf("Before: a = %d, b = %d\n", a, b);

    swap(a, b);
    printf("After: a = %d, b = %d\n", a, b);

    return 0;
}
```

не работает (передача осуществляется по значению):

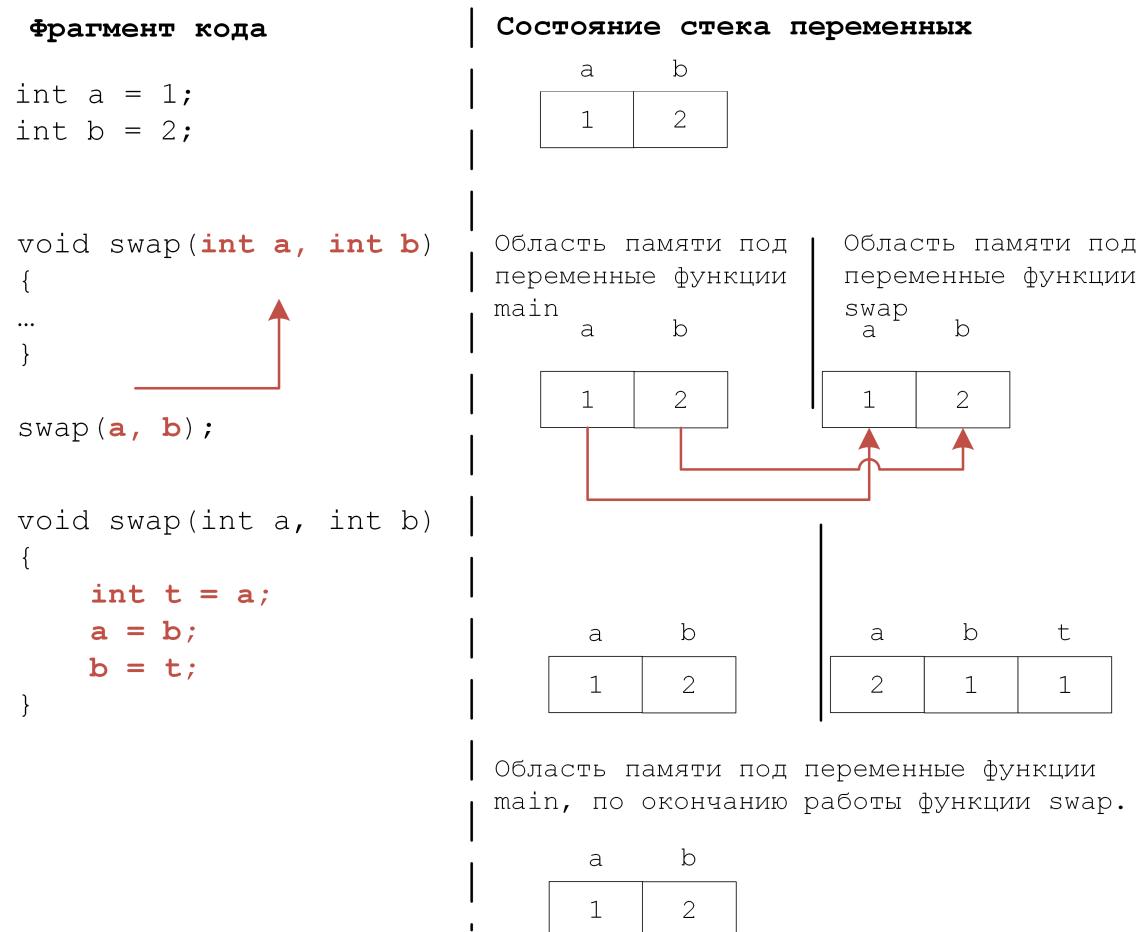


Рис. 7.17: При передаче параметров таким образом исходные данные не будут поменяны

Для того, чтобы обмен всё-таки осуществился, необходимо передать в функцию адреса изменяемых данных (а в функции в качестве формальных параметров используются указатели):

---

```

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

int main() {
    int a = 1;
    int b = 2;
    printf("Before: a = %d, b = %d\n", a, b);

    swap(&a, &b);
    printf("After: a = %d, b = %d\n", a, b);

    return 0;
}

```

---

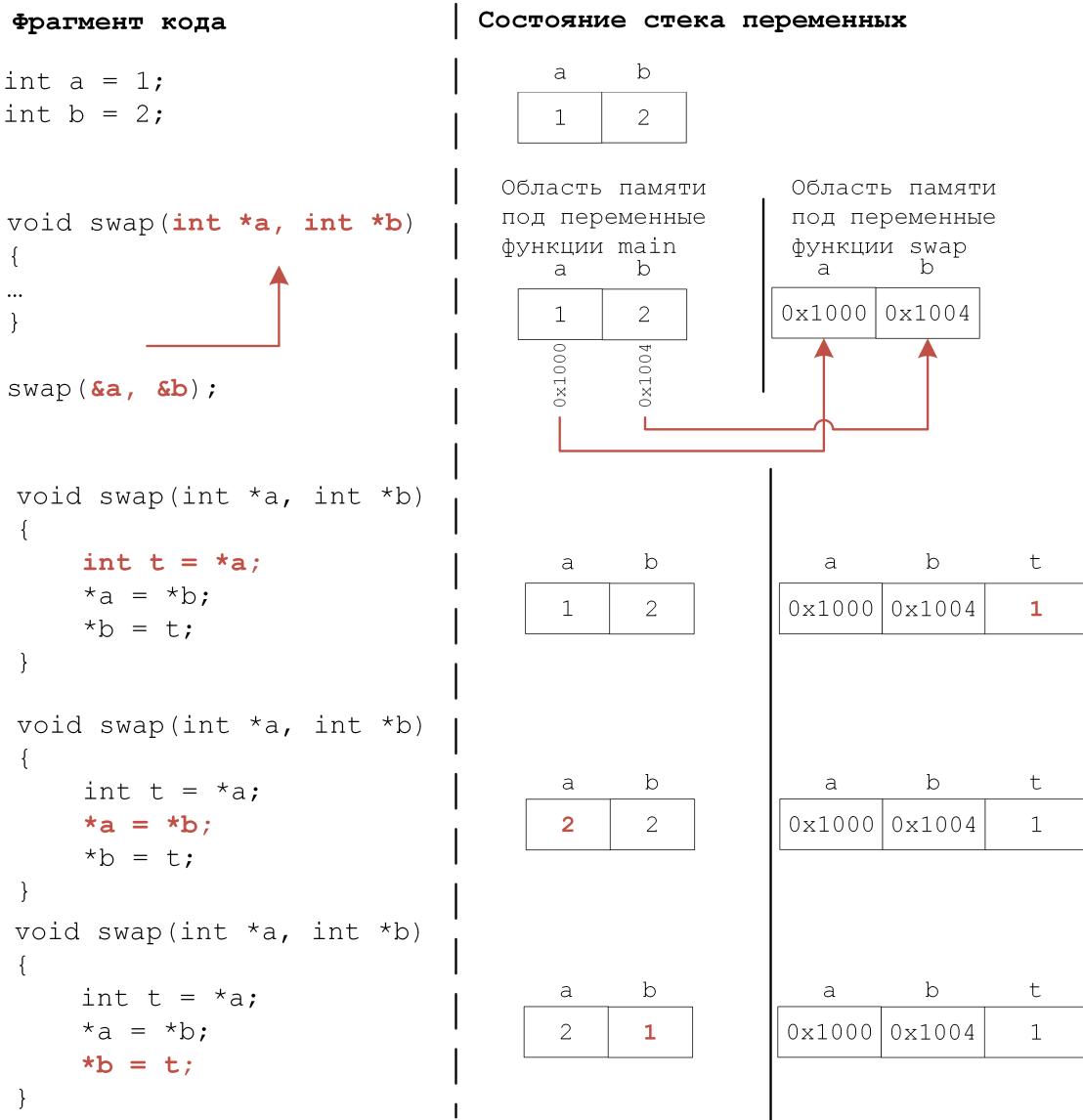


Рис. 7.18: Передача аргументов в функцию при помощи указателей

Передача адреса в функцию позволяет воздействовать на фактические параметры. Таким же образом выполняется передача больших объектов: передают не сам объект, а указатель на него. Однако возникают ситуации, когда нам нужна быстрая передача (идея с указателями) но мы бы не хотели изменять переданное значение. Для этого используется квалификатор `const`:

```
void someFunc(const int *a) {
    printf("%d", *a);
    // *a = 4; - ошибка, a - указатель на константное целое
    // значение по адресу a не может быть изменено
}
```

## 7.8 Решения задач при помощи нерекурсивных функций

Попробуем ещё раз пройтись по последовательности действий при решении задач:

- Задача разбивается на подзадачи до тех пор, пока каждая из подзадач не станет элементарной.
- Выполняется решение каждой из подзадач как самостоятельной, только предполагается, что данные в некоторых подзадачах будут поступать из других задач.
- При реализации всех компонент, исходный код должен выполнять поставленную задачу.

Переведём все блок-схемы с использованием блоком 'Предопределенный процесс' из задачи о кирпиче в код. Реализовывать продукт будем снизу-вверх. Сначала решим самые простые подзадачи и постепенно будем усложнять.

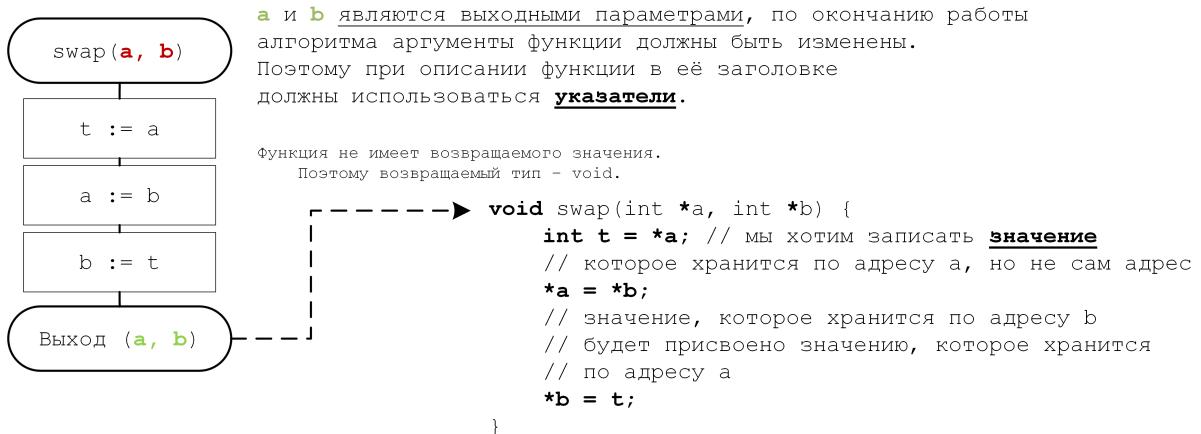


Рис. 7.19: Реализация функции *swap*

По началу это может показаться сложным. Могу предложить следующую идею (она звучит не особо профессионально), но позволяет ускорить процесс написания первых функций:

1. Напишите простую функцию без указателей.
2. Определитесь с выходными параметрами и добавьте в заголовок указатели на выходные параметры.
3. Если в функции где-то использовалась переменная, которая является выходной, перед ней надо поставить операцию косвенного доступа (разыменования). Последовательность действий представлена на рисунке 7.20.

Функция упорядочивания `sort2` использовала функцию обмена значений (рисунок 7.21), `sort3` - рисунок 7.22, код функции `main` - рисунок 7.23.

---

Программы без ошибок можно писать двумя способами. Но почему-то работает третий.

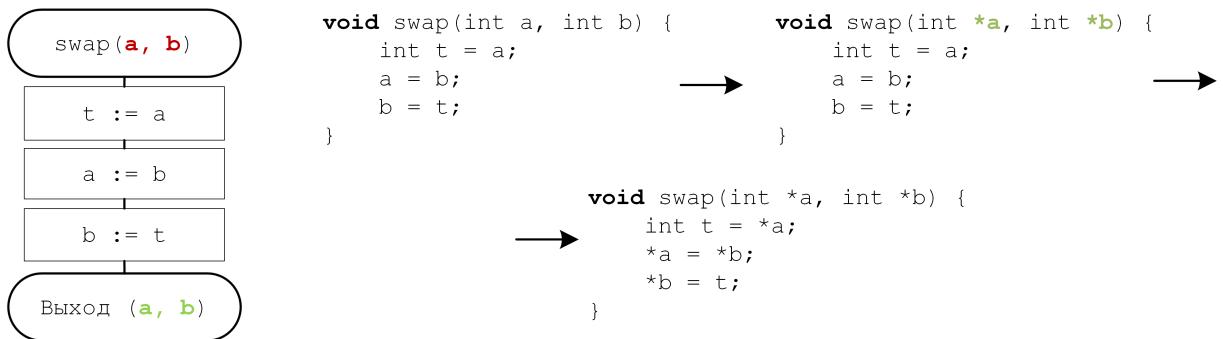


Рис. 7.20: Процесс написания функции

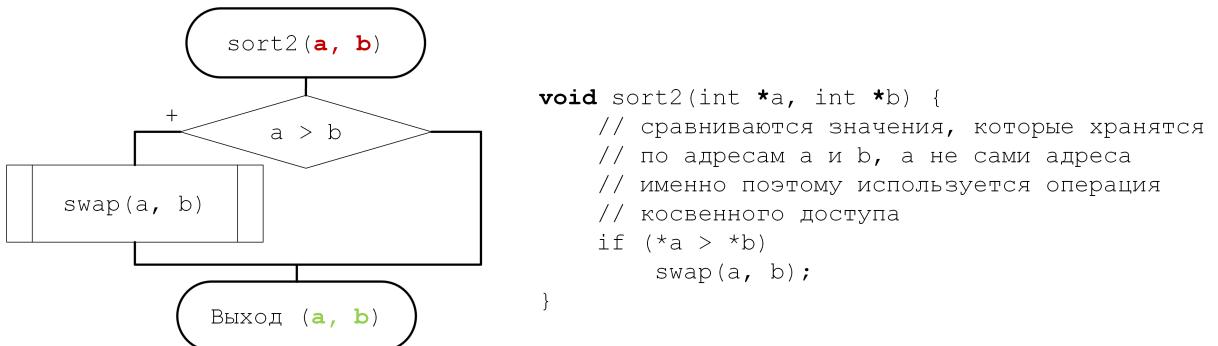


Рис. 7.21: Реализация функции sort2. В блок-схеме видно, что вызывается функция swap

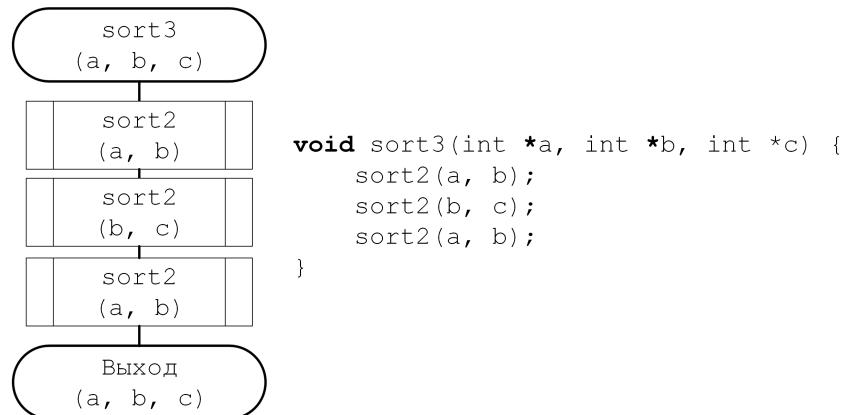
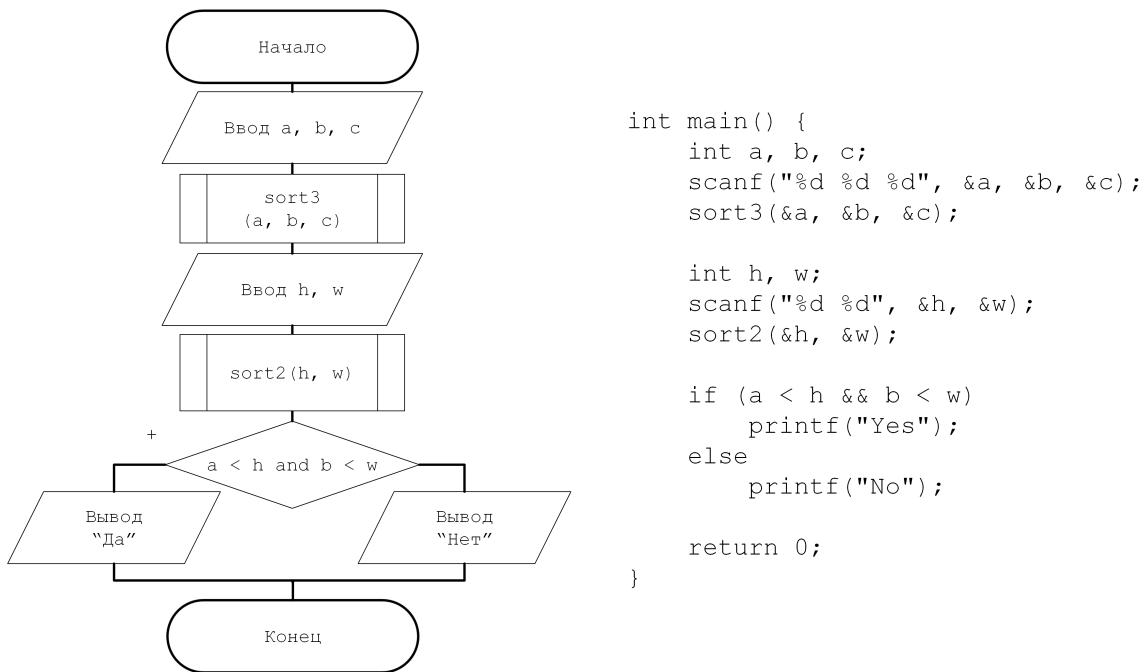


Рис. 7.22: Реализация функции sort3

Рис. 7.23: Реализация функции *main*

Исходный код:

---

```

#include <stdio.h>

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

void sort2(int *a, int *b) {
    if (*a > *b)
        swap(a, b);
}

void sort3(int *a, int *b, int *c) {
    sort2(a, b);
    sort2(b, c);
    sort2(a, b);
}

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);
    sort3(&a, &b, &c);

    int h, w;
    scanf("%d %d", &h, &w);
    sort2(&h, &w);

    if (a < h && b < w)
        printf("Yes");
    else
        printf("No");
}

```

---

```

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);
    sort3(&a, &b, &c);

    int h, w;
    scanf("%d %d", &h, &w);
    sort2(&h, &w);

    if (a < h && b < w)
        printf("Yes");
    else
        printf("No");

    return 0;
}

```

```

    else
        printf("No");

    return 0;
}

```

Код решения задачи о типе треугольника:

```

#include <stdio.h>
#include <math.h>
#include <windows.h>

// обмен значений по адресам a и b
void swap(double *a, double *b) {
    double t = *a;
    *a = *b;
    *b = t;
}

// упорядочение значений по адресам a и b
void sort2(double *a, double *b) {
    if (*a > *b)
        swap(a, b);
}

// упорядочение значений по адресам a, b, c
void sort3(double *a, double *b, double *c) {
    sort2(a, b);
    sort2(b, c);
    sort2(a, b);
}

// возвращает расстояние между точками (x1, y1), (x2, y2)
double getDistance(int x1, int y1, int x2, int y2) {
    int deltaX = x1 - x2;
    int deltaY = y1 - y2;
    return sqrt(deltaX * deltaX + deltaY * deltaY);
}

int main() {
    SetConsoleOutputCP(CP_UTF8);

    // ввод исходных данных
    int x1, y1, x2, y2, x3, y3;
    scanf("%d %d %d %d %d %d", &x1, &y1, &x2, &y2, &x3, &y3);

    // поиск длин сторон и упорядочение
    double d1 = getDistance(x1, y1, x2, y2);
    double d2 = getDistance(x1, y1, x3, y3);
    double d3 = getDistance(x2, y2, x3, y3);
    sort3(&d1, &d2, &d3);

    // определение типа треугольника

```

```

    double bigSideSquare = d3 * d3;
    double sumOfSquaresOfSides = d1 * d1 + d2 * d2;

    // специфика сравнения вещественных на равенство
    if (fabs(bigSideSquare - sumOfSquaresOfSides) < 1e-12)
        printf("Прямоугольный");
    else if (bigSideSquare > sumOfSquaresOfSides)
        printf("Тупоугольный");
    else
        printf("Остроугольный");

    return 0;
}

```

### 7.8.1 Алгоритмы возвведения в степень

#### Алгоритмы возвведения в степень

Необходимо вычислить значение выражения:

$$a^3 + b^5$$

для произвольных  $a$  и  $b$ .

Решим задачу 'в лоб' при помощи циклов:

```

#include <stdio.h>

int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    long long resA = 1;
    for (int i = 1; i <= 3; i++)
        resA *= a;

    long long resB = 1;
    for (int i = 1; i <= 5; i++)
        resB *= b;

    printf("%lld", resA + resB);

    return 0;
}

```

Было бы удобно для данной задачи выделить функцию:

```

long long int power(int a, int n) {
    // res инициализируется значением 1, чтобы корректно обрабатывать  $a^0$ 
    long long res = 1;
    for (int i = 1; i <= n; i++)

```

```

    res *= a;

    return res;
}

```

---

которая возводит число `a` в степень `n`. Тогда код `main` не содержит дублирования и сократится до

```

#include <stdio.h>

int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    printf("%lld", power(a, 3) + power(b, 5));

    return 0;
}

```

---

Разработанная функция может быть повторно использована в других приложениях.

Вы провели тестирование и поняли, что скорости работы недостаточно. И решили применить ряд оптимизаций. Если дублирование – ваш путь, тогда оптимизировать придётся в нескольких участках приложения. При выделенной функции достаточно изменить только её.

Быстрый алгоритм возведения в степень легче объяснить на примере: предположим, что  $a$  возводится в 43 степень. Результат может быть вычислен так:

$$a^{43} = a^{32} * a^8 * a^2 * a \quad 43_{10} = 101011_2 = 32 + 8 + 2 + 1$$

Создадим вспомогательную переменную `x`, которая на  $i$ -ой итерации будет представлять  $a^{2^i}$ :

Номер итерации	Значение $x$
0	$a$
1	$a^2$
2	$a^4$
3	$a^8$
...	...
$i$	$a^{2^i}$

Используя полученные сведения, напишем код:

---

**Листинг 11** Алгоритм быстрого возведения в степень
 

---

```

long long int power(int a, int n) {
    long long x = a;
    long long res = 1;
    while (n != 0) {
        if (n & 1)      // или n \% 2
            res *= x;
        n >>= 1;        // или n /= 2;
        x = x * x;
    }

    return res;
}
  
```

---

Количество операций прямо пропорционально  $\log_2 n$ .

### 7.8.2 Алгоритм Евклида

**Алгоритм Евклида**

Выполнить поиск наибольшего общего делителя двух целых чисел.

В словесном виде алгоритм описывается так:

1. Из большего числа вычитаем меньшее.
2. Если получается 0, то значит, что числа равны друг другу и являются НОД (следует выйти из цикла).
3. Если результат вычитания не равен 0, то большее число заменяем на результат вычитания.
4. Переходим к пункту 1.

Реализация<sup>2</sup>: Алгоритм может быть записан эффективнее:

1. Большее число делим на меньшее.
2. Если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла).
3. Если есть остаток, то большее число заменяем на остаток от деления.
4. Переходим к пункту 1.

---

<sup>2</sup> Greatest common divisor - наибольший общий делитель (англ.)

---

**Листинг 12** Алгоритм Евклида
 

---

```

int gcd(int a, int b) {
    while (a != 0 && b != 0)
        if (a > b)
            a = a % b;
        else
            b = b % a;

    return a + b;
}
  
```

---

### 7.8.3 Вывод числа в 16-ой системе счисления.

**Вывод числа в 16-ой системе счисления**

Вывести число  $x$  в 16-ой системе счисления без использования функции *printf* со спецификатором преобразования `%o`.

Рассмотрим произвольное число  $x$ . Определимся сначала с количеством цифр в 16-ричном представлении. Оно равняется количеству групп цифр из четырёх в значащей части числа в двоичной системе счисления:

$$42_{10} = 10'1010_2 = 2A_{16} \quad 415_{10} = 1'1001'1111_2 = 19F_{16}$$

На следующем же этапе последовательно будем получать цифру за цифрой:

x:	<table border="1" style="display: inline-table; vertical-align: middle; border-collapse: collapse;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr></table>																	<table border="1" style="display: inline-table; vertical-align: middle; border-collapse: collapse;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr></table>																
x >> 8 & 15:	<table border="1" style="display: inline-table; vertical-align: middle; border-collapse: collapse;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr></table>																	0 0 0 1 1																
x >> 4 & 15:	<table border="1" style="display: inline-table; vertical-align: middle; border-collapse: collapse;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr></table>																	1 0 0 1 1																
x & 15:	<table border="1" style="display: inline-table; vertical-align: middle; border-collapse: collapse;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr></table>																	1 1 1 1 F																

Рис. 7.24: Последовательность получения числа в 16-ричном представлении

Опишем функцию, возвращающую количество цифр в 16-ричном представлении числа:

**Листинг 13** Вывод чисел в шестнадцатеричной системе счисления (итеративно)

---

```

#define BINARY_DIGITS_IN_ONE_HEX_DIGIT 4U // U - литерал беззнакового типа

// возвращает количество цифр в 16-ом представлении числа x
int countHexDigits(unsigned int x) {
    int count = 0;
    while (x != 0) {
        x >>= BINARY_DIGITS_IN_ONE_HEX_DIGIT;
        count++;
    }
    return count;
}

// вывод 16-ричного представления числа x
void printHex(unsigned int x) {
    int nHexDigits = countHexDigits(x);
    int shift = (nHexDigits - 1) * BINARY_DIGITS_IN_ONE_HEX_DIGIT;
    while (shift >= 0) {
        int hexDigit = x >> shift & 15;
        if (hexDigit < 10)
            printf("%d", hexDigit);
        else
            printf("%c", (hexDigit - 10 + 'A'));
        shift -= BINARY_DIGITS_IN_ONE_HEX_DIGIT;
    }
}

```

---

Задача вывода числа в 16-ричном представлении может быть решена рекурсивно:

**Листинг 14** Вывод чисел в шестнадцатеричной системе счисления (рекурсивно)

---

```

void _printHex(unsigned int x) {
    int hexDigit = x % 16;
    if (x != 0) {
        _printHex(x >> 4U);
        if (hexDigit < 10)
            printf("%d", hexDigit);
        else
            printf("%c", (hexDigit - 10 + 'A'));
    }
}

void printHex(unsigned int x) {
    if (x == 0)
        printf("0");
    else
        _printHex(x);
}

```

---

## 7.8.4 Задача о счастливом билете.

### Задача о счастливом билете

Вводится номер некоторого билета  $x$ , состоящее из  $n$  цифр ( $n$  четно,  $x \leq 10^{18}$ ). Необходимо проверить, является ли он счастливым. Назовём билет счастливым, если сумма первой половины цифр билета равна сумме второй половины билета.

Например,  $14420083 \rightarrow 1442|0083 \rightarrow 1 + 4 + 4 + 2 = 0 + 0 + 8 + 3 \rightarrow 11 = 11$  (счастливый)

Решение задачи можно разбить на два этапа:

1. Подсчёт количества цифр.
2. Получение суммы половины цифр числа.

Интересна реализация функции `getSumLastNDigits`. Мало того, что она вычисляет сумму половины цифр числа, находящегося по адресу `x`, так она и изменяет это значение. Данную функцию можно отнести к функциям с побочными эффектами.

**Побочный эффект функции** – возможность в процессе выполнения своих вычислений: читать и модифицировать значения глобальных переменных или аргументов, осуществлять операции ввода-вывода и т. п. Если вызвать функцию с побочным эффектом дважды с одним и тем же набором значений входных аргументов, может случиться так, что в качестве результата будут возвращены разные значения. Такие функции называются недетерминированными функциями с побочными эффектами.

```
#include <stdio.h>

// возвращает количество цифр в записи числа x
int countDigits(long long x) {
    int count = 1;
    while (x > 9) {
        count++;
        x /= 10;
    }
    return count;
}

// возвращает сумму последних n цифр числа x
// уменьшает значение x на 10 в степени n раз
int getSumLastNDigits(long long *x, int n) {
    // так как мы хотим, чтобы переданное значение в функцию изменилось,
    // в качестве фактического аргумента передаётся адрес переменной;
    // указатель - переменная, которая может хранить адрес другой переменной
    // поэтому в качестве фактического параметра функции использован указатель
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *x % 10;
        *x /= 10;
    }
    return sum;
}
```

```

}

// возвращает значение 'истина', если сумма первых
// n/2 цифр числа x равна сумме последних n/2 цифр числа x,
// где n - количество цифр в числе x
int isLuckyNumber(long long x) {
    int halfNumberLen = countDigits(x) / 2;
    // мы хотим, чтобы значение переменной x после выполнения функции
    // getSumLastNDigits изменилось, поэтому мы передаём адрес переменной
    int secondHalfSum = getSumLastNDigits(&x, halfNumberLen);
    int firstHalfSum = getSumLastNDigits(&x, halfNumberLen);
    return firstHalfSum == secondHalfSum;
}

int main() {
    long long x;
    scanf("%lld", &x);

    printf("%d", isLuckyNumber(x));

    return 0;
}

```

---

### 7.8.5 Задача о разбиении числа.

#### Задача о разбиении числа

Вводится число  $x$  ( $x > 0$ ). Необходимо узнать, может ли оно быть разбито таким образом, чтобы сумма цифр одной части равнялась сумме цифр другой? Если да - вывести полученные значения.

Примеры чисел и их разбиений:

- 453213 → 45|3213
- 78744 → 78|744
- 7978 → решений нет

Решение задачи основано на вычислении суммы цифр числа  $x$ . Будем последовательно вычислять сумму левой и правой части:

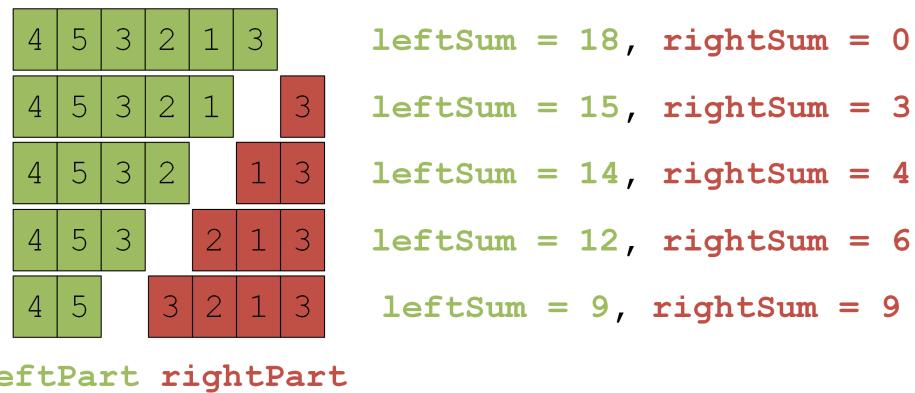


Рис. 7.25: Последовательное вычисление сумм. Каждая последующая строка может быть вычислена через предыдущую

```
#include <stdio.h>

// возвращает сумму цифр числа x
int getSumOfDigits(long long x) {
    int sum = 0;
    while (x > 0) {
        sum += x % 10;
        x /= 10;
    }

    return sum;
}

// возвращает значение 'истина', если число x может быть разделено
// на две части, суммы которых равны. Записывает по адресу a
// первую часть числа, записывает по адресу b - вторую часть
// числа. Если разбиение невозможно, возвращает 'ложь'.
int getSumLastNDigits(long long x, long long *a, long long *b) {
    int leftSum = getSumOfDigits(x);
    int rightSum = 0;
    long long y = 1;
    long long leftPart = x;
    long long secondPart = 0;
    do {
        int digit = leftPart % 10;
        secondPart += y * digit;
        y *= 10;
        rightSum += digit;
        leftSum -= digit;
        leftPart /= 10;
    } while (leftSum > rightSum);

    if (leftSum == rightSum) {
        *a = leftPart;
        *b = secondPart;
        return 1;
    } else
}
```

```

    return 0;
}

```

## 7.8.6 Задача о совершенных числах.

### Задача о совершенных числах

Совершенным числом называется число, равное сумме своих делителей, меньших его самого. Например,  $28 = 1 + 2 + 4 + 7 + 14$ .

Определите, является ли данное натуральное число совершенным. Найдите выведите все совершенные числа от  $a$  до  $b$  или сообщение, что таких чисел нет.

```

#include <stdio.h>
#include <math.h>

// возвращает значение 'истина' если число x является совершенным
// иначе - 'ложь'.
int isPerfectNumber(long long x) {
    const int minPossibleDivider = 2;
    const long long upperBound = sqrtl(x);
    long long sumOfDividers = 1;
    for (int possibleDivider = minPossibleDivider;
        possibleDivider <= upperBound; possibleDivider++) {
        if (x % possibleDivider == 0) {
            sumOfDividers += possibleDivider;
            sumOfDividers += x / possibleDivider;
        }
    }
    if (upperBound * upperBound == x)
        sumOfDividers -= upperBound;

    return x == sumOfDividers;
}

void printPerfectNumbersReport(int lowerBound, int upperBound) {
    int hasPerfectNumber = 0;
    for (int number = lowerBound; number <= upperBound; number++) {
        if (isPerfectNumber(number)) {
            if (hasPerfectNumber == 0) {
                printf("perfect numbers: ");
                hasPerfectNumber = 1;
            }
            printf("%d ", number);
        }
    }
    if (!hasPerfectNumber)
        printf("No perfect numbers");
}

int main() {

```

```

int lowerBound, upperBound;
scanf("%d %d", &lowerBound, &upperBound);

printPerfectNumbersReport(lowerBound, upperBound);

return 0;
}

```

---

### 7.8.7 Быки и коровы.

#### Быки и коровы

Опишем правила игры "Быки и коровы"<sup>a</sup>. Компьютер загадывает четырёхзначное число  $x$  (возможно, с ведущим нулём), в котором все цифры различны. Игрок пытается отгадать исходное число. На каждом ходе он высказывает некоторое предполагаемое число  $y$  (без повторяющихся цифр).

Обозначим за  $a$  – число отгаданных цифр, стоящих на своих местах (число быков) и число  $b$  – отгаданных цифр, стоящих не на своих местах (число коров). Цель игрока состоит в том, чтобы за минимальное количество ходов получить 4-х быков, т. е. угадать загаданное число. После каждой попытки компьютер выдаёт количество быков и коров. Ознакомьтесь с примерами:

$$x = 1234. \quad y = 1356 \rightarrow a = 1 \quad b = 1$$

$$x = 1234. \quad y = 2143 \rightarrow a = 0 \quad b = 4$$

$$x = 0234. \quad y = 1243 \rightarrow a = 1 \quad b = 2$$

Требуется реализовать консольное приложение-игру, в котором соблюдались указанные правила.

<sup>a</sup>Для решения данной задачи вам потребуются массивы. Вернитесь к ней, когда приобретёте навыки работы с ними.

Выделим следующие подзадачи:

- Генерация начального значения.
- Проверка на то, что число не содержит повторяющихся цифр и является  $N$ -значным (возможно с ведущим нулём).
  - Получение массива цифр из числа  $x$ .
  - Проверка на то, что все значения в массиве уникальны.
- Подсчёт количества быков.
- Подсчёт количества коров.

Последовательно опишем процесс реализации приложения. Подключим требуемые библиотеки. Дополнительно оставим возможность переключать количество быков (разрядность загадываемых чисел):

Программист – это не тот, кто пишет программы, а тот чьи программы работают.

---

```
#include <stdio.h> // ввод / вывод
#include <stdlib.h> // генерация случайных значений
#include <time.h> // из библиотеки потребуется функция time
// которая позволит задать случайное зерно генерации

// требуемое количество 'быков' в игре
#define NEED_BULLS_TOTAL 4

int main() {
    srand(time(0)); // устанавливаем зерно генерации
}

}
```

---

Для начала было бы неплохо описать функцию генерации значения. Компьютер загадывает число, которое соответствует правилам. В общем случае он будет загадывать какое-то случайное число, и мы должны выполнять перегенерацию до тех пор, пока оно не будет соответствовать правилам игры.

Функция генерации

---

```
int generateStartValue()
```

---

будет функцией с возвращаемым значением типа `int`. В точку вызова

---

```
int answer = generateStartValue();
```

---

в процессе выполнения программы подставится вычисленное значение. Например, 3476:

---

```
int answer = 3476;
```

---

При реализации генератора, очень удобно использовать цикл `do-while`, так как тело цикла выполнится хотя бы раз: компьютер сгенерирует значение, если оно подходит – заканчиваем процесс генерации, если нет – проверяем следующее значение. Нужно обеспечить, чтобы в числе было не более `NEED_BULLS_TOTAL` цифр. Можно вычислить остаток от деления на  $x = 10^{\text{NEED\_BULLS\_TOTAL}}$ . Например, если генератор выдаст значение 123456 и требуемое количество быков равно 4, получим 3456.

---

```
// возвращает случайное значение, удовлетворяющее условию игры
int generateStartValue() {
    // переменная имеет модификатор const, так как не изменяется
    // в процессе работы функции
    const long long maxValueBound = intPow(10, NEED_BULLS_TOTAL);
    int generatedValue;
    do {
        generatedValue = rand() % maxValueBound;
    } while (!isCorrectNumber(generatedValue));

    return generatedValue;
}
```

---

Функция содержит вызов `isCorrectNumber`, которая проверяет удовлетворение числа правилам игры. Как она это делает? – пока что неважно. Нам пока достаточно, что эта функция возвращала 'истину' или 'ложь' (подходит или не подходит). Так же должна вычисляться 10 в степени `NEED_BULLS_TOTAL`. Предположим, что и функция возведения недоступна и придётся её реализовывать<sup>3</sup>.

Подловим себя на мысли о том, как ведётся разработка приложения. В процессе написания кода последовательно выделяются функции, которых нет в языке. Продолжаем процесс до тех пор, пока приложение не будет реализовано.

Опишем функцию возведения в степень. Самую простую<sup>4</sup>:

---

```
// возвращает значение a в степени n
long long intPow(const int a, const int n) {
    long long res = 1;
    for (int i = 1; i <= n; i++)
        res *= a;

    return res;
}
```

---

Тип возвращаемого значения – `long long`, чтобы не допускать переполнения. Счётчик цикла `i` начинает с единицы для корректной обработки случая  $a^0$ .

Автоматическое тестирование нам пока недоступно, но доступно тестирование ручное. Написали часть функционала? – выполните его проверку.

Функция проверки на корректность состоит из двух частей:

- Проверка количества цифр.
- Проверка множества цифр на уникальность.

---

```
// возвращает значение 'истина', если число x является допустимым
// согласно правилам игры, иначе - ложь
int isCorrectNumber(const int x) {
    // если число больше или равно 10NEED_BULLS_TOTAL
    // будет возвращено значение 0
    const long long maxValueBound = intPow(10, NEED_BULLS_TOTAL);
    if (x / maxValueBound)
        return 0;

    // получаем множество цифр
    int digits[NEED_BULLS_TOTAL];
    getDigitSet(x, digits, NEED_BULLS_TOTAL);
    // проверяем уникальность
    return isUniqueArray(digits, NEED_BULLS_TOTAL);
}
```

---

<sup>3</sup>В работе над реальными проектами не изобретайте велосипед.

<sup>4</sup>Не старайтесь оптимизировать сразу. Лучше в готовом решении найти критичные по производительности места и оптимизировать только их.

Функция для получения последовательности цифр из числа:

---

```
// записывает по адресу digitSet цифры числа nDigits цифр числа x.
// если nDigits больше количества цифр в x - записывает в оставшиеся
// элементы массива значения 0
void getDigitSet(long long x, int *digitSet, const size_t nDigits) {
    for (int i = nDigits - 1; i >= 0; i--) {
        int digit = x % 10;
        digitSet[i] = digit;
        x /= 10;
    }
}
```

---

Функция принимает указатель на массив длины  $nDigits$ . В процессе обработки числа она производит его заполнение:

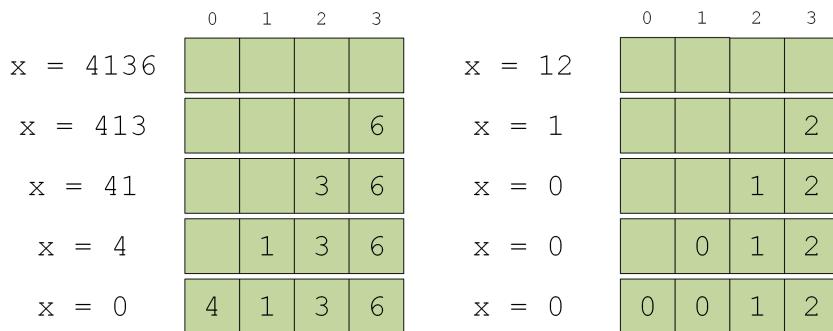


Рис. 7.26: Процесс заполнения массива  $digitSet$

Функция `getDigitSet` может быть использована и потом при подсчёте быков и коров. Поэтому решение выполнено таким образом. Можно было реализовать множество на массиве и функция проверки примет вид:

---

```
int isCorrectNumber(const int x) {
    int a[10] = {0};
    for (int i = 0; i < NEED_BULLS_TOTAL; i++) {
        int digit = x % 10;
        if (a[digit]++)
            return 0;
    }

    return 1;
}
```

---

Идея функции для проверки массива на уникальность элементов следующая: будем попарно сравнивать элементы. Если найдены равные, можно прекратить поиск и вернуть значение 0. Если одинаковых пар не нашли, возвращаем 1.

---

**Листинг 15** Проверка на уникальность элементов массива  $O(n^2)$ 


---

```
// возвращает значение 'истина', если все элементы массива a размера n
// являются уникальными иначе - 'ложь'
int isUniqueArray(const int *a, const size_t n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (a[i] == a[j])
                return 0;

    return 1;
}
```

---

Для массива из четырёх значений порядок сравнения таков:

0	1	2	3
i	j		
i		j	
i			j
	i	j	
	i		j
		i	j

Рис. 7.27: Порядок сравнения элементов массива в проверке уникальности значений

Если оценить код, написанный сверху, можно получить осознание, что задача генерации числа, удовлетворяющему правилам, решена. Найдём количество быков и коров:

```
// записывает по адресу nBulls количество быков,
// а по адресу nCows - количество коров при условии, что загадано
// число answer и выполнено предположение в виде числа hypothesis
void getNAnimals(const int answer, const int hypothesis,
                  int *nBulls, int *nCows) {
    int answerDigitSet[NEED_BULLS_TOTAL];
    getDigitSet(answer, answerDigitSet, NEED_BULLS_TOTAL);

    int hypothesisDigitSet[NEED_BULLS_TOTAL];
    getDigitSet(hypothesis, hypothesisDigitSet, NEED_BULLS_TOTAL);

    *nBulls = 0;
    *nCows = 0;
    for (int i = 0; i < NEED_BULLS_TOTAL; i++)
        for (int j = 0; j < NEED_BULLS_TOTAL; ++j) {
            if (hypothesisDigitSet[i] == answerDigitSet[j]) {
                if (i == j)
```

---

```

        *nBulls += 1;
    else
        *nCows += 1;
}
}
}

```

---

Заметьте, в данную функцию передаются указатели на количество коров и количество быков, так как и количество коров и количество быков являются выходными параметрами функции.

Логика приложения:

```

int main() {
    srand(time(0));

    int answer = generateStartValue();
    // при необходимости скрыть сгенерированное значение
    // следующую строку нужно закомментировать;
    // будет выведено NEED\_BULLS\_TOTAL-цифр, если в числе answer цифр
    // → меньше
    // выводятся ведущие нули до длины NEED\_BULLS\_TOTAL
    printf("generated value: %0*d\n", NEED_BULLS_TOTAL, answer);

    int hypothesis;
    while (1) {
        scanf("%d", &hypothesis);
        if (!isCorrectNumber(hypothesis)) {
            printf("No correct number\n");
        } else {
            int nBulls, nCows;
            getNAnimals(answer, hypothesis, &nBulls, &nCows);
            if (nBulls == NEED_BULLS_TOTAL) {
                printf("win!");
                break;
            } else
                printf("nCows = %d, nBulls = %d\n", nCows, nBulls);
        }
    }

    return 0;
}

```

---

Разместим функции в файле в следующем порядке:

```

#include <stdio.h> // ввод / вывод
#include <stdlib.h> // генерация случайных значений
#include <time.h> // из библиотеки потребуется функция time
// которая позволит задать случайное зерно генерации

// требуемое количество 'быков' в игре
#define NEED_BULLS_TOTAL 4

```

```

// заголовки функций
int generateStartValue();

int isCorrectNumber(const int x);

int isUniqueArray(const int *a, const size_t n);

void getDigitSet(long long x, int *digitSet, const size_t nDigits);

long long intPow(const int a, const int n);

void getNAnimals(const int answer, const int hypothesis, int *nBulls, int
→ *nCows);

int main() {
    //
}

// реализация функции getDigitSet
void getDigitSet(long long x, int *digitSet, const size_t nDigits) {
    //
}

// реализация функции getNAnimals
void getNAnimals(const int answer, const int hypothesis,
                 int *nBulls, int *nCows) {
    //
}

// ...

// ...

```

## Резюме

- Функция — это самодостаточная единица кода программы, спроектированная для выполнения отдельной задачи.
- В каждой программе на C/C++ должна присутствовать функция `main`, которая получает управление при запуске программы.
- Использование функций необходимо для организации программы в виде совокупности небольших и не зависящих друг от друга частей.
- Определение функции состоит из типа возвращаемого значения, имени функции, списка параметров и тела.
- Прототип функции (объявление функции) состоит из типа возвращаемого значения, имени функции, списка параметров.
- Функции относятся к типу функций с возвращаемым значением, если возвращаемый ей результат должен быть подставлен в точку вызова.

- Обычные переменные в функции C уничтожаются, как только завершается вызов функции.
- Аргументы, передаваемые в функцию, не будут изменены, так как передаются по значению. Чтобы изменить параметры, необходимо передавать их адрес.

## Термины и определения

- **Аргумент (фактический параметр)** – это значение, которое передаётся в функцию при её вызове.
- **Возвращаемое значение функции** – некоторое значение, которое будет подставлено в точку вызова функции.
- **Входные параметры** – параметры функции, значения которых являются исходными данными для решаемой подзадачи.
- **Выходные параметры** – параметры функции, в которые будут записаны результаты для решаемой подзадачи.
- **Параметры функции (формальный параметр)** – это переменные, описываемые в объявлении функции и создаваемые при её вызове.
- **Побочный эффект функции** – это возможность в процессе выполнения своих вычислений: читать и модифицировать значения глобальных переменных или аргументов, осуществлять операции ввода-вывода и т. п.
- **Функция** – это самодостаточная единица кода программы, спроектированная для выполнения отдельной задачи.
- **Функция в математике** – это соответствие между элементами двух множеств — правило, по которому каждому элементу первого множества соответствует один и только один элемент второго множества.

## Контрольные вопросы

1. В каких случаях целесообразно использовать функции?
2. Принципы *DRY* и *WET*.
3. В чём разница между объявлением и определением функции?
4. Какие параметры называются фактическими, а какие формальными?
5. Может ли в результате вызова функции измениться значение фактического параметра функции?
6. Действия, производимые при вызове функций.
7. Для чего используется квалификатор `const` для формальных параметров функции?
8. В чём заключается побочный эффект функции?
9. Как изменить аргументы, передаваемые в функцию?

# Глава 8

## Рекурсия

В программировании **рекурсия**<sup>1</sup> – вызов функции из неё же самой, непосредственно (**простая рекурсия**) или через другие функции (**сложная или косвенная рекурсия**), например, функция *A* вызывает функцию *B*, а функция *B* – функцию *A*. Количество вложенных вызовов функции или процедуры называется **глубиной рекурсии**. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

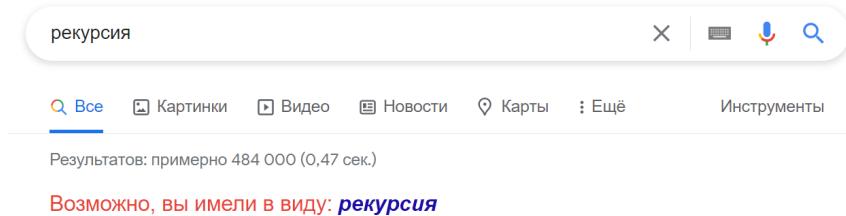


Рис. 8.1: Разработчики поисковых сервисов не могли не пошутить об этом

Структурно рекурсивная функция на верхнем уровне всегда представляет собой команду ветвления (выбор из двух или более альтернатив в зависимости от условий, которое в данном случае уместно назвать «условием прекращения рекурсии»), имеющую две или более альтернативные ветви, из которых хотя бы одна является рекурсивной и хотя бы одна – терминальной. Пример рекурсивной функции в общем виде:

```
//function f(x):
//    if <условие прекращения рекурсии>:
//        //...
//    [else if //...
//        //...
//    ] else:
//        //...
```

<sup>1</sup>Перед тем, как понять рекурсию, нужно понять рекурсию.

Если вы ждете гостей и вдруг заметили на своем костюме пятно, не огорчайтесь. Это поправимо. Например, пятна от растительного масла легко выводятся бензином. Пятна от бензина легко снимаются раствором щелочи. Пятна от щелочи исчезают от уксусной эссенции. Следы от уксусной эссенции надо потереть подсолнечным маслом. Ну, а как выводить пятна от подсолнечного масла, вы уже знаете...

Так как в условии прекращения рекурсии имеется инструкция возврата в случае бинарного ветвления можно заметить два равнозначных варианта:

---

```
//function f(x):
//    if <условие прекращения рекурсии>:
//        // ...
//        return [<возвращаемое значение>];
//    else:
//        <оператор рекурсивной ветви>
```

---

или

---

```
//function f(x):
//    if <условие прекращения рекурсии >:
//        // ...
//        return [<возвращаемое значение >];
//    else:
//        <оператор рекурсивной ветви>
```

---

Рекурсивная ветвь выполняется, когда условие прекращения рекурсии ложно, и содержит хотя бы один рекурсивный вызов – прямой или опосредованный вызов функцией самой себя. Терминальная ветвь выполняется, когда условие прекращения рекурсии истинно; она возвращает некоторое значение, не выполняя рекурсивного вызова. Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

## 8.1 Виды рекурсии

- **Линейная рекурсия** – рекурсивная функция вызывает себя единожды. Существует два типа линейно-рекурсивных функций, но мы будем относить к ним только те, которые каким-либо образом перед возвратом значения обрабатывают результат рекурсивного вызова. Пример: вычисление факториала:

$$F(n) = \begin{cases} 1 & \text{если } n = 0, \\ n * F(n - 1) & \text{если } n > 0. \end{cases}$$

- **Хвостовая рекурсия** – тип линейной рекурсии, при котором рекурсивный вызов функций – последняя инструкция в функции. Результат рекурсивного вызова никак не обрабатывается вызывающей функцией и передаётся выше по цепочке. Интерпретаторы и компиляторы функциональных языков программирования, поддерживающие оптимизацию кода (исходного или исполняемого), автоматически преобразуют хвостовую рекурсию к итерации, благодаря чему обеспечивается выполнение алгоритмов с хвостовой рекурсией в ограниченном объёме памяти.

$$F(n, a, b) = \begin{cases} b & \text{если } n = 1, \\ F(n - 1, a + b, a) & \text{если } n > 1. \end{cases}$$

- **Параллельная рекурсия** – функция вызывает саму себя в нескольких местах. Например,  $n$ -е число Фибоначчи может быть найдено так:

$$F(n) = \begin{cases} 1 & \text{если } n = 1 \text{ или } n = 2, \\ \left[ F\left(\frac{n}{2} + 1\right) \right]^2 - \left[ F\left(\frac{n}{2} - 1\right) \right]^2 & \text{если } n > 2 \text{ и } n \text{ чётно,} \\ \left[ F\left(\frac{n+1}{2}\right) \right]^2 - \left[ F\left(\frac{n-1}{2}\right) \right]^2 & \text{если } n > 2 \text{ и } n \text{ нечётно.} \end{cases}$$

- **Взаимная рекурсия** – вид рекурсии, когда несколько функций вызывают друг друга циклически. Например,  $f$  вызывает  $g$ ,  $g$  вызывает  $h$ , а та же в свою очередь вызывает  $f$ . В качестве взаимно рекурсивных могут выступать функции<sup>2</sup>:

$$isEven(n) = \begin{cases} true & \text{если } n = 0, \\ isOdd(n - 1) & \text{если } n > 0. \end{cases}$$

$$isOdd(n) = \begin{cases} false & \text{если } n = 0, \\ isEven(n - 1) & \text{если } n > 0. \end{cases}$$

- **Вложенная рекурсия** – вид рекурсии, при которой аргумент рекурсивной функции определяется как результат другого рекурсивного вызова:

$$F(s, n) = \begin{cases} 1 + s & \text{если } n = 1 \text{ или } n = 2, \\ F(n - 1, F(n - 2, 0)) & \text{если } n > 2. \end{cases}$$

Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов – адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Обратной стороной этого довольно простого по структуре механизма является то, что на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов.

Вопрос о желательности использования рекурсивных функций в программировании неоднозначен: с одной стороны, рекурсивная форма может быть структурно проще и нагляднее, в особенности, когда сам реализуемый алгоритм по сути рекурсивен. Кроме того, в некоторых декларативных или чисто функциональных языках (например, *Haskell*) просто нет синтаксических средств для организации циклов, и рекурсия в них – единственный доступный механизм организации повторяющихся вычислений. С другой стороны, обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии.

Теоретически, любую рекурсивную функцию можно заменить циклом и стеком. Однако такая модификация, как правило, бессмысленна, так как приводит лишь к замене автоматического сохранения контекста в стеке вызовов на ручное выполнение тех же операций с тем же или большим расходом памяти.

---

<sup>2</sup>В качестве взаиморекурсивных определений приведены следующие: операция – действие, выполняемое над операндом; операнд – объект, которым оперируют операции.

Большинство рассматриваемых в литературе задач на тему рекурсии часто являются чисто учебными, однако их решение помогает лучше адаптироваться к реальным условиям.

## 8.2 Решение задач с использованием рекурсии

### 8.2.1 Вычисление значения $n!$

#### Вычисление $n!$

С клавиатуры вводится число  $n$ . Необходимо найти  $n!$ .

Из определения:

$$F(n) = \begin{cases} 1 & \text{если } n = 0, \\ n * F(n - 1) & \text{если } n > 0. \end{cases}$$

Очевидно, что эта задача может быть решена итеративно:

---

#### Листинг 16 Вычисление значения факториала (итеративно)

---

```
long long getFactorial(int n) {
    long long res = 1;
    for (int i = 2; i <= n; i++)
        res *= i;

    return res;
}
```

---

Но следуя из определения факториала можно получить рекурсивное решение:

---

#### Листинг 17 Вычисление значения факториала (рекурсивно)

---

```
long long getFactorialR(int n) {
    if (n <= 1) // 
        return 1;
    else
        return n * getFactorialR(n - 1);
}

int main() {
    printf("%lld", getFactorialR(4))
}
```

---

В примере можем заметить ветвление в зависимости от текущего значения  $n$ .

Введём ряд определений. Порождение все новых копий рекурсивной подпрограммы до выхода на условие прекращения рекурсии называется **рекурсивным спуском**. Завершение работы рекурсивных подпрограмм, вплоть до самой первой, инициировавшей рекурсивные вызовы, называется **рекурсивным подъёмом**.

---

Рекурсия – это когда Ипполит приходит к своей Наде Шевелевой, а там по телевизору показывают не "Соломенную шляпку", а "Иронию судьбы".

На этапе рекурсивного спуска постепенно определялись выражения для вычисления  $F(i)$ , начиная с  $F(4)$  до  $F(1)$ . На этапе рекурсивного подъема последовательно вычисляются  $F(2)$ ,  $F(3)$  и  $F(4)$  и получается значение 24.

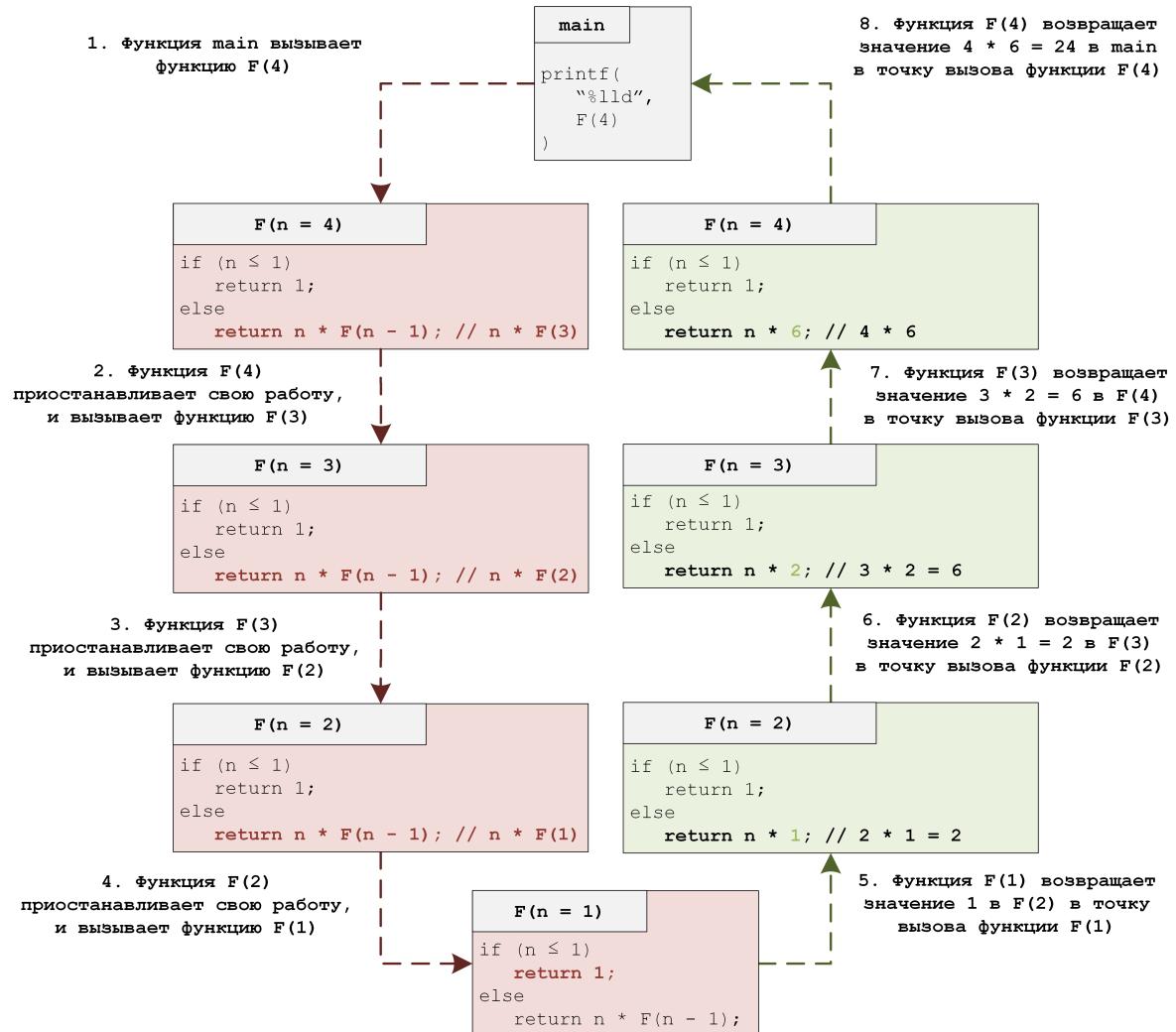


Рис. 8.2: Процесс вычисления значения 4!

## 8.2.2 Вывод двоичного представления числа

### Выход двоичного представления числа

С клавиатуры вводится натуральное число  $x$ . Необходимо вывести двоичное представление числа  $x$ .

На этапе рекурсивного спуска, пока число не стало равно нулю, можно 'отцеплять' последний бит числа `x` и сохранять его в переменной `digit`. На этапе рекурсивного подъёма осуществляется его вывод. Оба варианта допустимы, но мы будем предпочитать первый:

```
void printBin(int x) {
    if (x == 0)
        return;
    else {
        int digit = x & 1;
```

```

    printBin(x >> 1);
    printf("%d", digit);
}
}

```

```

void printBin(int x) {
    if (x == 0)
        return;

    int digit = x & 1;
    printBin(x >> 1);
    printf("%d", digit);
}

```

Строки 2-4 относятся к рекурсивному спуску, строка 6 – к рекурсивному подъему. Отразим то, что происходит на схеме:

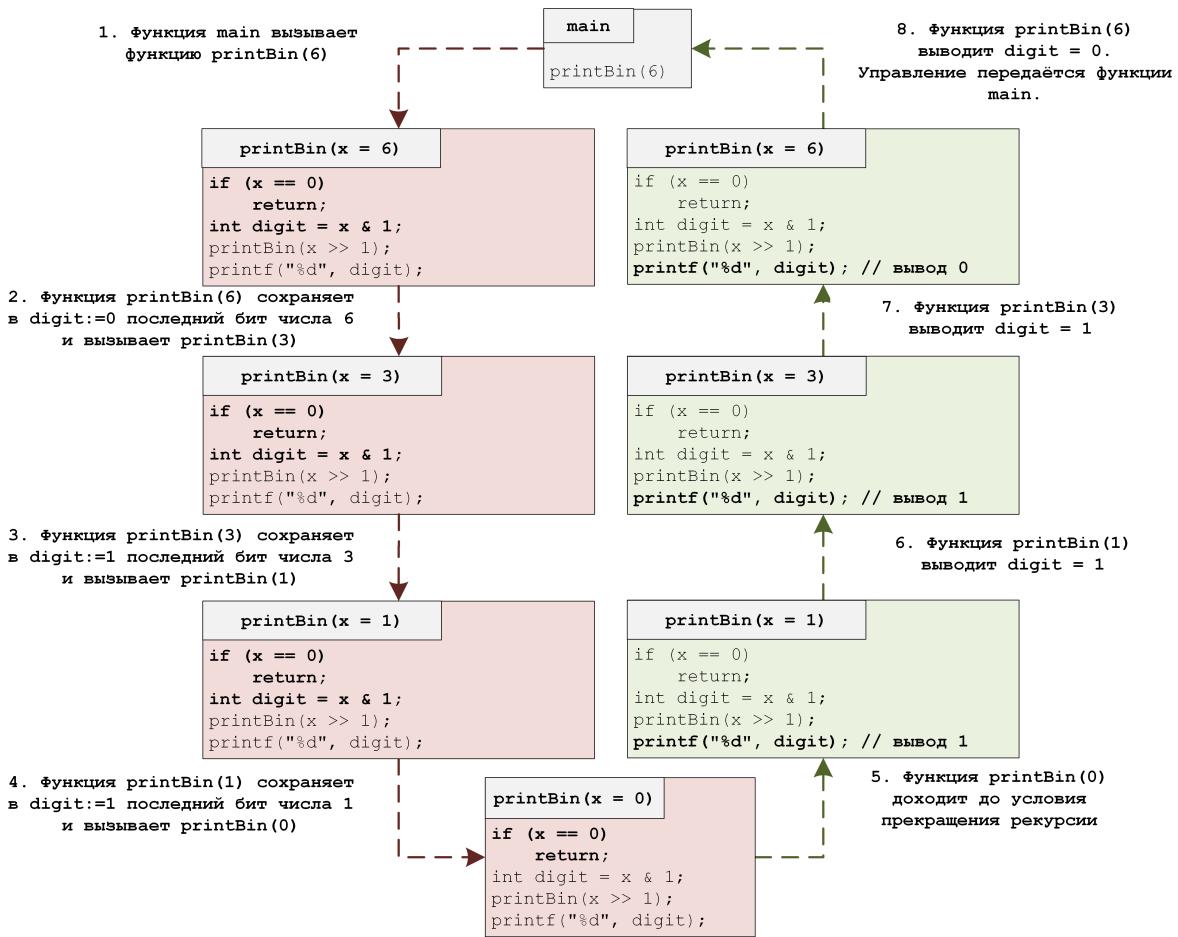


Рис. 8.3: Процесс вывода двоичного представления числа 6

### 8.2.3 Вычисление суммы массива

#### Вычисление суммы массива

Дан массив  $a$  размера  $n$ . Необходимо вычислить сумму элементов массива.

Если каким-то образом возможно выразить большую задачу через решение меньших, в реализации допускается использование рекурсии. Рассмотрим задачу вычисления суммы  $n$  чисел последовательности. Она может быть разбита несколькими способами (рисунок ??).

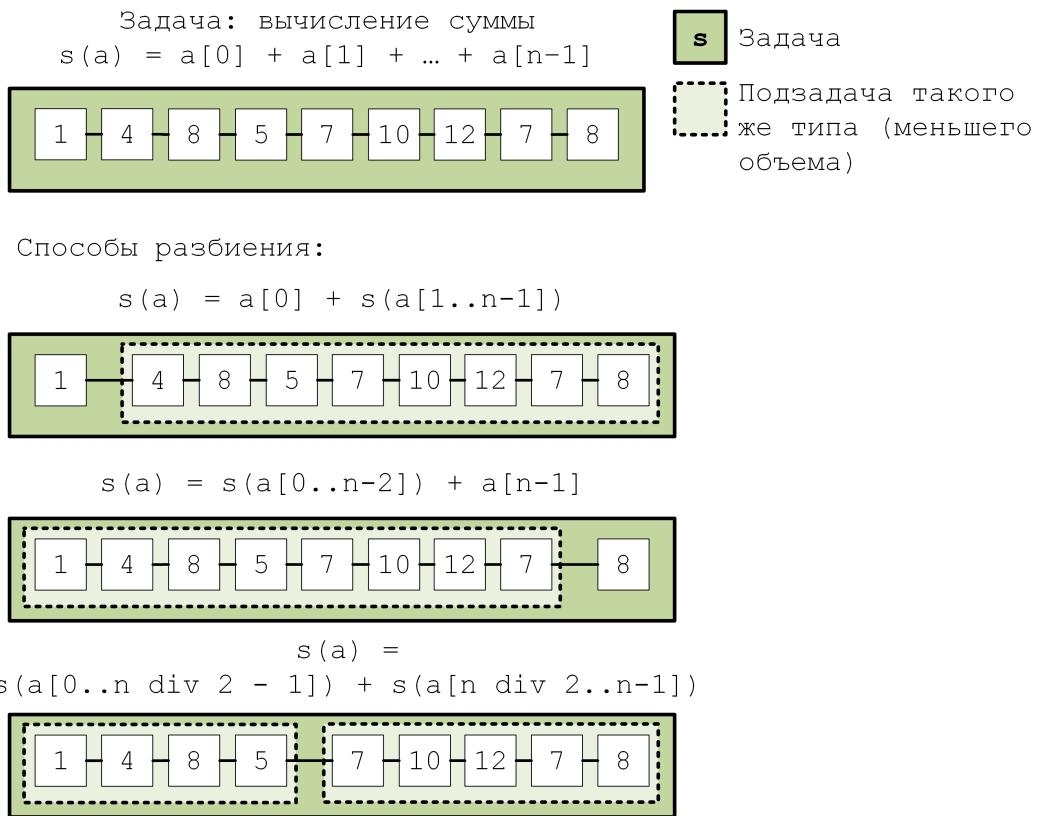


Рис. 8.4: Способы вычисления суммы последовательности

Более формально, сбор суммы с начала последовательности:

$$s(a) = \begin{cases} 0 & \text{если } n = 0, \\ a[0] + s(a[1..n-1]) & \text{если } n > 0. \end{cases}$$

С конца последовательности:

$$s(a) = \begin{cases} 0 & \text{если } n = 0, \\ s(a[0..n-2]) + a[n-1] & \text{если } n > 0. \end{cases}$$

Разбиение по середине:

$$s(a) = \begin{cases} 0 & \text{если } n = 0, \\ a[0] & \text{если } n = 1, \\ s(a[0..n \text{ div } 2 - 1]) + a[n \text{ div } 2..n-1] & \text{если } n > 1. \end{cases}$$

В последнем случае задача поделена на 2 более простых. Если не указать случай  $n = 1$  рекурсия станет бесконечной. Простой пример, на котором этом можно обнаружить – последовательность из одного элемента. Без случая  $n = 1$  она будет разбиваться на две последовательности: пустая и снова из одного элемента.

Рассмотрим вариант, когда последовательность хранится в памяти ЭВМ. Можно использовать арифметику указателей, но укажем решение без неё. Во всех примерах

`left` – левая граница (индекс) массива, для которого вычисляется сумма, `right` – правая граница массива (не включительно):

---

**Листинг 18** Вычисление суммы массива с начала последовательности (рекурсивно)

---

```
int getSum(const int *a, int left, int right) {
    int n = right - left;
    if (n == 0)
        return 0;
    else
        return a[left] + getSum(a, left + 1, right);
}

int main() {
    int a[4] = {1, 2, 3, 4};

    printf("%d", getSum(a, 0, 4));

    return 0;
}
```

---

С конца последовательности:

---

**Листинг 19** Вычисление суммы массива с конца последовательности (рекурсивно)

---

```
int getSum(const int *a, int left, int right) {
    int n = right - left;
    if (n == 0)
        return 0;
    else {
        right--;
        return getSum(a, left, right) + a[right];
    }
}
```

---

Разбиение на две подзадачи:

---

**Листинг 20** Вычисление суммы массива путём разбиения на подзадачи (рекурсивно)

---

```
int getSum(const int *a, int left, int right) {
    int n = right - left;
    if (n == 0)
        return 0;
    else if (n == 1)
        return a[left];
    else {
        int middle = (left + right) / 2;
        return getSum(a, left, middle) + getSum(a, middle, right);
    }
}
```

---

## 8.2.4 Алгоритм однопроходного удаления

### Алгоритм однопроходного удаления

Дан массив  $a$  размера  $n$ . Реализовать алгоритм однопроходного удаления.

Оставлен на самостоятельное изучение.

### Листинг 21 Алгоритм однопроходного удаления (рекурсивно)

```
void deleteIf_(int *a, int (*deleteCondition)(int),
               int *size, int iRead, int iWrite) {
    if (iRead == *size) {
        *size = iWrite;
        return;
    } else if (!deleteCondition(a[iRead])) {
        a[iWrite] = a[iRead];
        iWrite++;
    }
    deleteIf_(a, deleteCondition, size, iRead + 1, iWrite);
}

void deleteIf(int *a, int *n, int (*deleteCondition)(int)) {
    deleteIf_(a, deleteCondition, n, 0, 0);
}
```

Обратите внимание, что все переменные, которые участвовали в итеративном алгоритме вычисления 'оказались' в качестве формальных параметров функции:

## 8.2.5 Проверка массива на палиндром

### Проверка массива на палиндром

Дан массив  $a$  размера  $n$ . Необходимо проверить, является ли он палиндромом.

Опишем функцию, которая проверяет является ли последовательность палиндромом. Интуитивно очевидно, что надо сравнивать первый с последним, второй - с предпоследним. Если нашли хотя бы одно несовпадение – прерываем сравнения. Если последовательность уже вся обработана – она является палиндромом. Проверки нужно осуществлять пока  $left < right$ , где  $left$  – индекс левого сравниваемого элемента,  $right$  – правого.

### Листинг 22 Проверка последовательности на палиндром (рекурсивно)

```
static int isPalindrome_(const int *a, int left, int right) {
    if (left < right)
        return a[left] == a[right] && isPalindrome_(a, left+1, right-1);
    else
        return 1;
}
```

Основной недостаток данной функции - крайне неудобный интерфейс. Чтобы проверить, является ли массив *a* размера 10 палиндромом, необходимо выполнить вызов:

---

```
isPalindrome_(a, 0, 9);
```

---

Чтобы предоставить более удобный интерфейс, опишем функцию обёртку:

---

```
int isPalindrome(const int *a, int n) {
    return isPalindrome_(a, 0, n - 1);
}
```

---

И при необходимости проверять последовательность на палиндром, будем вызывать именно её. Важно заметить, что ключевое слово `static` для функции `isPalindrome_` делает запрет на её вызов если расположить её в библиотеке.

### 8.2.6 Сортировка выбором

#### Сортировка выбором

Дан массив *a* размера *n*. Реализовать сортировку выбором.

В качестве учебного примера опишем рекурсивную функцию сортировки выбором. Если коротко, она заключается в следующем: массив делится на две части: сортированную и несортированную. Среди элементов неотсортированной части ищется минимальный и добавляется в конец сортированного фрагмента. Это выполняется до тех пор, пока не будут отсортированы все элементы:

#### Листинг 23 Сортировка выбором (рекурсивный)

---

```
static void selectionSort(int *unsortedPart, int nUnsorted) {
    if (nUnsorted == 1) // массив из одного элемента является
                        // упорядоченным и так
        return;
    else {
        int minElementIndex = getMinIndex(unsortedPart, nUnsorted);
        swap(&unsortedPart[0], &unsortedPart[minElementIndex]);
        selectionSort(unsortedPart + 1, nUnsorted - 1);
    }
}
```

---

Преимущество данного подхода – хорошая читаемость. Если посмотреть на строчки алгоритма, там описано ровно то же самое, что и текстом до листинга.

### 8.2.7 Вычисление многочлена в точке

#### Вычисление многочлена в точке

Дан многочлен

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

и точка  $x_0$ .

Необходимо найти значение  $P(x_0)$ .

Для решения такой задачи часто применяется метод Горнера. Можно выполнить преобразование:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + a_n x)))$$

В таком случае используется всего лишь  $n$  операций умножения. Легко реализуется итеративный вариант, рекурсивный выглядит не сложнее:

**Листинг 24** Вычисление многочлена в точке / Схема Горнера (рекурсивно)

```
static long long getPolyValue_(long long *a, int x, int i) {
    if (i == 0)
        return a[0];
    else
        return a[i] + x * getPolyValue_(a, x, i - 1);
}

long long getPolyValue(long long *a, int n, int x) {
    return getPolyValue_(a, x, n);
}
```

### 8.2.8 Бинарный поиск

#### Бинарный поиск

Дан отсортированный массив  $a$  размера  $n$ . Требуется выполнить поиск элемента  $x$  и вернуть его позицию. Если элемент не найден –  $-1$ .

---

**Листинг 25** Бинарный поиск (рекурсивно)
 

---

```

static int binarySearch_(const int *a, int x, int right, int left) {
    if (left > right)
        return -1;

    int middle = (left + right) / 2;
    if (a[middle] == x)
        return middle;
    else if (a[middle] < x)
        return binarySearch_(a, x, right, middle + 1);
    else
        return binarySearch_(a, x, middle - 1, left);
}

int binarySearch(const int *a, int n, int x) {
    return binarySearch_(a, x, n - 1, 0);
}
  
```

---

Оставлю решения нескольких задач, которые помогут чуть лучше разобраться в рекурсии:

1. С клавиатуры вводятся целые числа. Признак конца ввода – ноль. Найти сумму введенных четных чисел.

```

long long getSum() {
    int x;
    scanf("%d", &x);

    if (x == 0)
        return 0;
    else if (x % 2 == 0)
        return x + getSum();
    else
        return getSum();
}
  
```

---

2. С клавиатуры вводится последовательность неотрицательных целых чисел. Вывести сначала все четные, а затем все нечетные числа в обратном порядке. Последовательность заканчивается нулем.

```

void getNumbers() {
    int number;
    scanf("%d", &number);

    if (number == 0)
        return;
    else if (number % 2 == 0) {
        printf("%d ", number);
        getNumbers();
    }
}
  
```

---

```

    } else {
        getNumbers();
        printf("%d ", number);
    }
}

```

3. Найти номер позиции последнего вхождения элемента со значением  $x$ .

```

static int linearSearchLast_(const int *a, const int n,
                           const int x, const int i) {
    if (i == -1)
        return -1;
    else if (a[i] == x)
        return i;
    else
        return linearSearchLast_(a, n, x, i - 1);
}

int linearSearchLast(const int *a, const int n, const int x) {
    return linearSearchLast_(a, n, x, n - 1);
}

```

4. Найти номер первого вхождения минимального значения в последовательность длины  $n$ .

```

static int getFirstMinPos_(const int *a, int n,
                           int i, int minPos) {
    if (i >= n)
        return minPos;
    else {
        if (a[i] < a[minPos])
            minPos = i;
        getFirstMinPos_(a, n, i + 1, minPos);
    }
}

int getFirstMinPos(const int *a, int n) {
    return getFirstMinPos_(a, n, 1, 0);
}

```

5. С клавиатуры вводится последовательность символов. Признак конца ввода – символ перехода на новую строку `\n`. Вывести цифры из введенной последовательности сначала в порядке ввода, а затем в обратном порядке.

```

void printDigits() {
    int c = getchar();
    if (c == '\n')
        return;
    else {

```

```

        if (isDigit(c))
            printf("%c", c);
        printDigits();
        if (isDigit(c))
            printf("%c", c);
    }
}

```

---

6. Даны две последовательности

$$x_1 = y_1 = 1, \quad x_i = x_{i-1} + \frac{y_{i-1}}{2}, \quad y_i = y_{i-1} + \frac{x_{i-1}}{3}$$

Вывести на экран  $n$ -е члены этих последовательностей.

```

static void printXY_(double x, double y, int n, int i) {
    if (i == n + 1)
        printf("%lf %lf", x, y);
    else {
        double lastX = x;
        x += y / 2;
        y += lastX / 3;
        printXY_(x, y, n, i + 1);
    }
}

void printXY(int n) {
    printXY_(1, 1, n, 2);
}

```

---

7. Дано натуральное число  $s$ . Определить, может ли число  $s$  быть суммой некоторого числа первых членов последовательности Фибоначчи. Последовательность Фибоначчи задается следующим образом:

$$f_1 = f_2 = 1 \quad f_i = f_{i-1} + f_{i-2} \text{ для } i > 2$$

```

static int isFibSum_(int s, int accSum,
                     int lastFib2, int lastFib1) {
    accSum += lastFib1;
    if (accSum >= s)
        return accSum == s;
    else
        return isFibSum_(s, accSum, lastFib1, lastFib1 + lastFib2);
}

int isFibSum(int s) {
    int accSum = 0;
    int lastFib1 = 1;
    int lastFib2 = 0;
}

```

```

    return isFibSum_(s, accSum, lastFib2, lastFib1);
}

```

---

8. Дан первый член арифметической прогрессии и ее разность. Вычислить  $n$ -й член прогрессии без использования операции умножения.

```

static int multiplyRecursion(int i, int n, int d) {
    if (i == n)
        return d;
    else
        return d + multiplyRecursion(i + 1, n, d);
}

int getLastElementOfAP(int a0, int n, int d) {
    return a0 + multiplyRecursion(1, n, d);
}

```

---

9. Определить количество букв в тексте, вводимом с клавиатуры. Текст заканчивается символом переноса на новую строку `\n`.

```

int countLetters() {
    int c = getchar();
    if (c == '\n')
        return 0;
    else
        return ('a' <= c && c <= 'z' || 'A' <= c && c <= 'Z') +
               countLetters();
}

```

---

10. Найти наибольший общий делитель натуральных чисел  $n$  и  $m$ .

```

int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

```

---

# Глава 9

## Работа в CLion

*IDE*<sup>1</sup> – интегрированная среда разработки (англ. *Integrated Development Environment*), которая представляет собой комплекс программ, используемый для создания программного обеспечения. Современные среды разработки снабжены рядом возможностей, облегчающим отладку и написание программ. Поговорим о некоторых из них:

1. Статический анализ кода.
2. Отладчик.
3. Автодополнение.
4. Выделение программных объектов.
5. Горячие клавиши.
6. Шаблоны кода.
7. Шаблон-окружение.

### 9.1 Статический анализ кода

*Clion* выполняет анализ<sup>2</sup> кода, который пишет программист в реальном времени, и подсвечивает красным и желтым участки, которые, по его мнению, могут содержать ошибки.

Рассмотрим пример. Пусть программист Вася написал следующую программу:

```
#include <stdio.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    char s[255];
```

<sup>1</sup>Глава написана Азаровым Александром. Стиль изложения автора сохранён, в материал внесены небольшие правки. На самом деле он был одним из первых, кто ловил баги в этой методичке. Спасибо.

<sup>2</sup>Разделяют динамический, статический и ручной анализ кода. Первые два осуществляют специальные программы, а третий человек. Динамический анализ осуществляется при выполнении кода, в то время как статический производится без исполнения программ, здесь мы говорим про последний.

```

printf("Введите ваше имя\n")
scanf("%s", s, s);

printf("Привет \%s", s)

return 0;
}

```

В его коде, так как он торопился, допущен ряд ошибок. Вася хочет сам исправить программу, и готовится листать учебники, чтобы понять как. Однако хорошая среда разработки поможет быстрее устранить ошибки и предупреждения, выводя сообщения о них во вкладке *Problems* (рисунок 9.1):

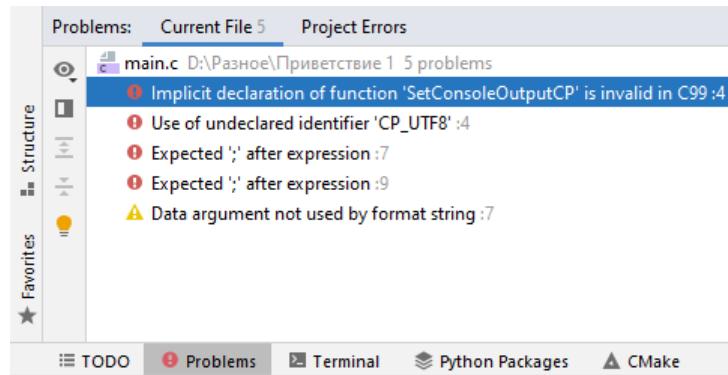


Рис. 9.1: Ошибки и предупреждения среды

Будем не слишком гуманными и озвучим ошибки Васи: забыл подключить библиотеку, не написал точки с запятыми в конце операторов, зачем-то передал `scanf` больше аргументов, чем обещал в управляющей строке.

Можно сказать, что анализ кода *IDE* проводит следующим образом: проходится сверху-вниз по тексту программы, при встрече первой ошибки, среда о ней сигнализирует, а затем продолжает поиск ошибок. Это делается для поддержания свойства, называемого полнотой анализа. У такой логики есть одна проблема – точность: Вася вспиннет, подсунув *IDE* следующий код (представляющий собой предыдущий с исправлением ошибок и удалением открывающей фигурной скобки блока `main`):

```

#include <stdio.h>
#include <windows.h>

int main()
{
    SetConsoleOutputCP(CP_UTF8);
    char s[255];
    printf("Введите ваше имя\n");
    scanf("%s", s);
    printf("Привет %s", s);
    return 0;
}

```

Здесь среда не поправит ошибку Васи, поставив фигурную скобку, из-за чего впоследствии возникнет масса 'мнимых' ошибок:

На рисунке 9.2 первая ошибка с открывающей скобкой отмечена верно, однако все следующие малость не корректны. Если Вася начнет исправлять программу с

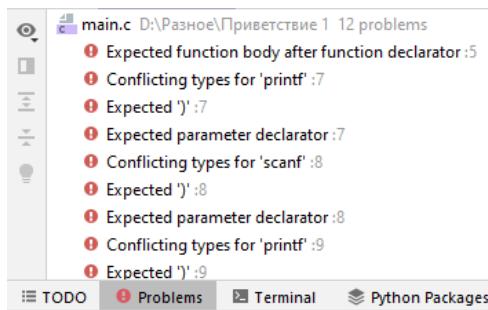


Рис. 9.2: Мнимые ошибки

них, то не уверены, что он своего добьется. Поэтому можно вывести рекомендацию начинать исправление с самой ранней ошибки.

## 9.2 Отладчик

Представим ситуацию: уже вы написали программу, она успешно исполняется, но выдает неверный ответ, а анализатор молчит. Как решить эту проблему?

Можно поискать опечатки в коде, можно добавить в программу вывод промежуточных значений, чтобы проследить ее работу, и можно воспользоваться помощью еще одной программы для поиска ошибок – отладчика<sup>34</sup>. Последний вариант (после овладения им) самый продуктивный.

Допустим, мы писали программу для вывода считанной строки (в которой записано число) в обратном порядке:

---

```
#include <stdio.h>
#include <windows.h>
#include <string.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

    char s[255];
    gets(s);

    printf("Ответ: \n");
    for (size_t i = strlen(s); i >= 0; --i) {
        printf("%c", s[i]);
    }
    printf("\n");

    return 0;
}
```

---

Запустили ее, и увидели что, что-то пошло не так (рисунок 9.3):

Скажем нашей *IDE* остановить выполнение программы до входа в цикл `for`: будем считать, что ошибка в нем. Для этого щелкнем слева от строки, перед которой нужно остановиться. Появится красный круг, изображенный на рисунке 9.4.

<sup>3</sup> Отладчик – программа позволяющая программисту контролировать выполнение кода. В современных средах она обычно встроена.

<sup>4</sup> Вариант отвлечься в пособии не рассматривается.

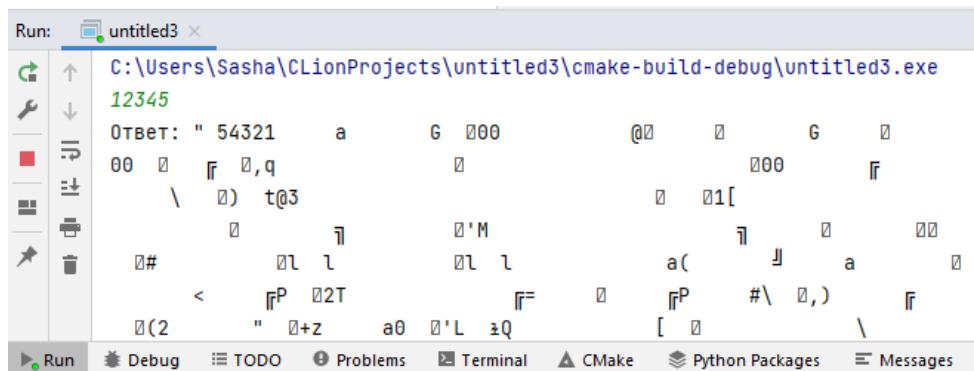


Рис. 9.3: Бесконечный цикл вывода символов

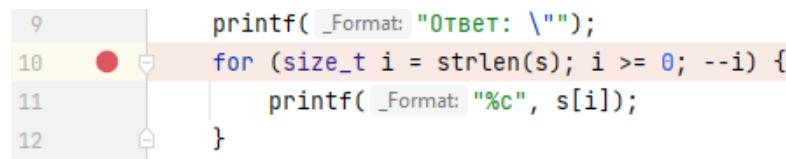


Рис. 9.4: Точка останова на 10 строке

Говорят, что в помеченной строке находится **точка останова**<sup>5</sup>.

Запустим программу в режиме отладки, нажав на зеленого жучка (Либо комбинацию *Shift + F9*)<sup>6</sup>. IDE покажет остановку следующим образом.

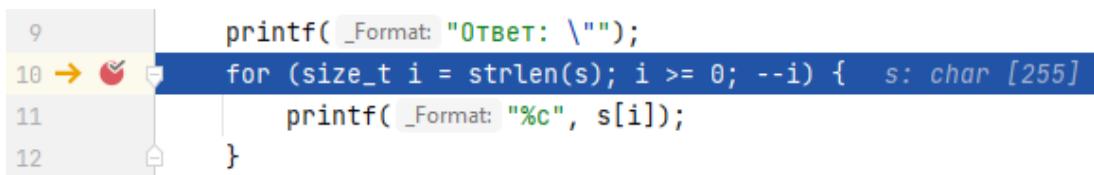


Рис. 9.5: Остановка исполнения программы на 10 строке

Порою можно ввестись в заблуждение, что строка из рисунка 9.5, помеченная желтой стрелочкой, уже исполнена. Это не совсем так, исполнен весь код до нее.

Помимо появления стрелок, на интерфейсе IDE появилось окно отладчика:

<sup>5</sup>Эти точки называют и точками остановки и точками останова. Есть мнение, что термин останов пришел к нам от необходимости различать остановку процессора и останов программы в прошлом.

<sup>6</sup>Стоит отметить, что элементы интерфейса могут отличаться. Однако комбинации клавиш остаются схожими. Поэтому сфокусируйтесь на комбинациях клавиш для того или иного действия.

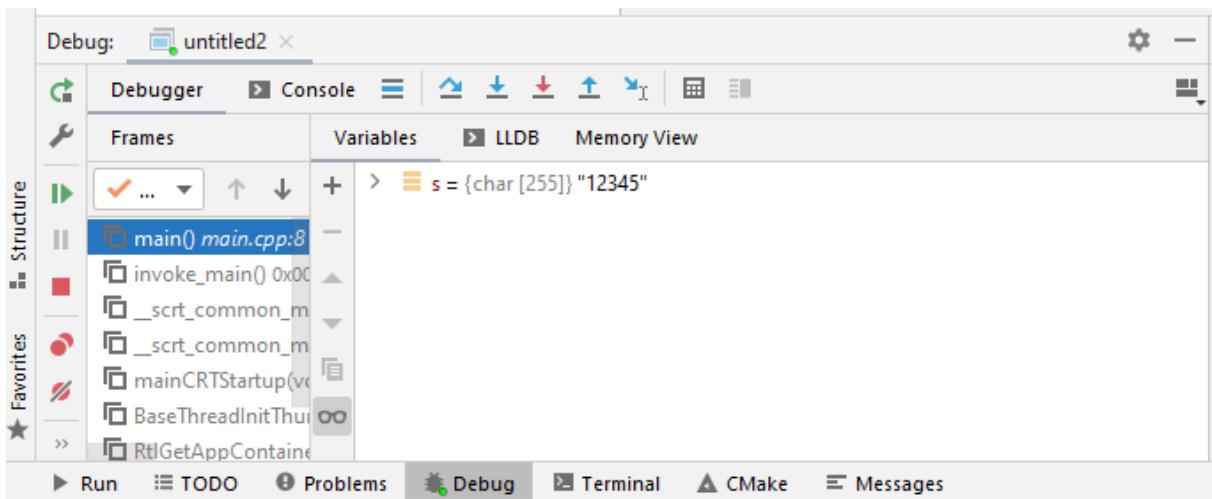


Рис. 9.6: Встроенный отладчик

На рисунке 9.6 в окне отладчика мы можем увидеть значения переменных и изменить их. Судя по значению переменной `s` строка считана корректно. Вполне вероятен сценарий, увидеть пустую строку или иероглифы.

Через отладочное окошко можно запустить исполнение помеченою желтой стрелкой строки. Продолжим искать ошибку, войдя в цикл при помощи крайней левой стрелочки панели (Вместо этого нажать `F8` на клавиатуре). Смысл данной стрелки – исполнение строки. С другими стрелками мы познакомимся чуть-чуть позже.

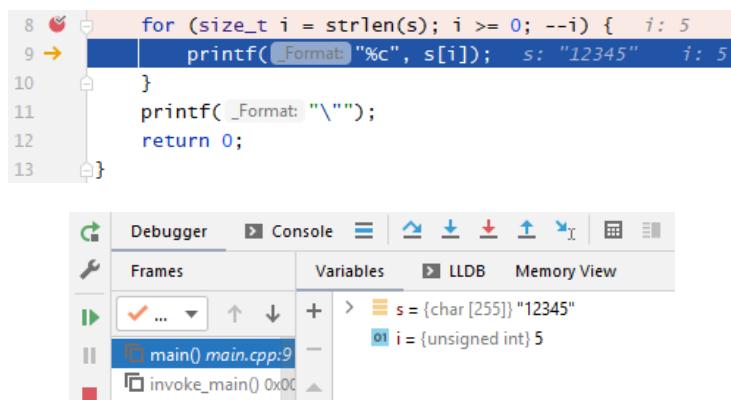


Рис. 9.7: Изменения интерфейса после шага отладки

Заметим, что на рис. 9.7 у нас появилась новая переменная – счетчик `i = 5`. Продолжим исполнять строки и следить за поведением программы.

Первая ошибка – до ожидаемого конца цикла, мы вывели 6 символов, когда в строке их 5. Нужно было выводить символы от `length(s) - 1`

Вторая ошибка: переменная счетчика после 0 совершает резкий скачок (Рисунок 9.8):

```
s = {char [255]} "12345"
i = {unsigned int} 4294967295
```

Рис. 9.8: Переполнение типа `size_t`

Такой скачок обусловлен переполнением. Так как `i` переменная беззнаковая, то после вычитания из `i` единицы, мы получили верхнюю границу типа `size_t`.

Поговорим о других способах навигации, в частности о пятой стрелке: ↵ (Alt + F9), которая исполняет код до позиции курсора. Допустим, мы отлаживаем следующий код 9.9:

```

1 #include <stdio.h>
2
3 int main() {
4     for (int i = 0; i < 10000; ++i) {
5         printf(_Format: "%i ", i);
6     }
7     return 0;
8 }
```

Рис. 9.9: Вход в цикл

И нам хочется перейти к строке с `return`. Чтобы это сделать без прохождения 10000 итераций, можно пойти двумя путями: поставить точку останова на 7 строке и нажать на значок ▶ (F9) слева от панели стрелок, либо перенести курсор на `return` и нажать на пятую стрелку. Результат будет примерно таким рис. 9.10:

```

1 #include <stdio.h>
2
3 int main() {
4     for (int i = 0; i < 10000; ++i) {
5         printf(_Format: "%i\n", i);
6     }
7     return 0;
8 }
```

Рис. 9.10: Остановка на выходе из цикла

Обратим внимание на то, что на данный момент времени в консоли выведены не все числа рис. 9.11:

```

9224
9225
9226
9227
9228
9229
9229
9230
9231
9232
9
```

Рис. 9.11: Не полный вывод

Эта странность связана с буферизацией вывода в С и особенностями встроенной консоли *Clion*: перед тем как попасть на консоль, символы скапливаются в буфере<sup>7</sup>.

<sup>7</sup>Буфер – это область памяти, используемая для временного хранения данных при вводе или выводе

Поэтому остальные числа появятся на консоли после завершения программы, при котором будет осуществлено копирование текста из буфера на консоль.

Иногда необходимо сразу видеть вывод программы, для этого можно воспользоваться функцией `fflush(stdout)` после `printf`, которая выполнит сброс буфера на консоли, либо включить внешнюю консоль в настройках. Чтобы это сделать нажмите на левой панели в дебаг меню на значок . После чего во всплывшем меню поставьте галочку:  Run in external console.

Стоит заметить, что посимвольный вывод значительно замедляет работу программы. В *IDE Microsoft Visual Studio* уходит 6 секунд на вывод чисел приведенного примера, а у *Clion* только 1 секунда. Но если мы добавим функцию сброса буфера, *Clion* также затратит 6 секунд.

Чтобы рассмотреть еще две стрелки навигации модифицируем пример. Вынесем цикл прошлой нашей программы в отдельную функцию `print_num`. А затем, запустив пошаговую отладку с первой строки функции `main`, подумаем, как посмотреть значения счетчика внутри `print_num` рис. 9.12.

```

1 #include <stdio.h>
2
3 void print_numbers(void) {
4     for (int i = 0; i < 10000; ++i) {
5         printf(_Format: "%i\n", i);
6     }
7 }
8
9 ► int main () {
10 → ⚡ print_numbers();
11     return(0);
12 }
```

Рис. 9.12: Дилемма как попасть в функцию

Если мы воспользуемся стрелкой (*F8*), чтобы сделать шаг отладки, то отладчик 'проскочит' функцию. Можно поставить точку останова внутри `print_num`, а потом запустить отладку (*F9*) или воспользоваться (*F8*). Тут нам может помочь и . Однако все такие способы достаточно неудобны. Для того, чтобы зайти в исполняемую функцию применяется стрелка (*F7*). Смысл данной стрелки – шаг в функцию.

Убедившись, что счетчик правильно прибавляет единичку, выйдем из функции. Для этого воспользуемся (*Shift + F8*). Смысл данной стрелки – выход из функции.

Изученная нами навигация позволяет достаточно быстро перемещаться по строкам кода. Вот только нам не хватает возможности прервать исполнение ровно по середине цикла. Вдруг у нас появится основания, чтобы искать ошибку именно там.

Для такого прерывания можно воспользоваться точкой останова с условием. Поставим точку останова на теле цикла и нажав правую кнопку мыши укажем условие остановки рис. 9.13:

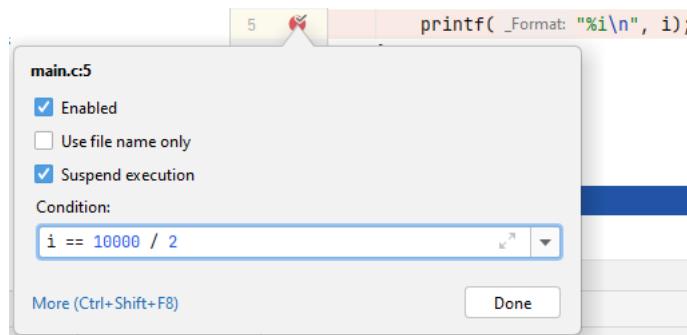


Рис. 9.13: Точка останова с условием

Запустим исполнение и увидим что-то вроде рис. 9.14:

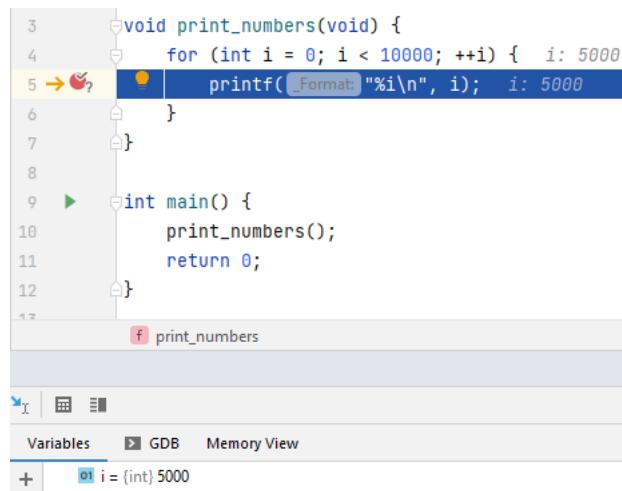


Рис. 9.14: Остановка программы при заданном значении счетчика

Представим, что нам пришлось бы щелкать все 5000 итераций и вздохнем спокойно.

На этом рассматриваемая часть арсенала программиста не заканчивается, рассмотрим еще одно орудие отладки. Допустим, мы составили рекурсивную функцию получения  $n$ -го числа Фибоначчи. И предполагаем, что для получения  $n$ -го числа нам нужно вызвать функцию около  $n$  раз. И мы хотим в этом убедиться, увидев вызовы рекурсивной функции.

Наша рекурсивная функция имеет следующий вид:

```
#include <stdio.h>
#include <windows.h>

long long body_fib(long long i1, long long i2, long long n) {
    if (n <= 0)
        return(i2);
    else
        return(body_fib(i2, i1 + i2, n - 1));
}

long long fibonacci (long long n) {
    return(body_fib(1, 1, n - 2));
}
```

```

int main() {
    SetConsoleOutputCP(CP_UTF8);

    long long n;
    printf("Введите номер числа Фибоначчи\n");
    scanf("%lli", &n);

    printf("%lli", fibonacci(n));

    return 0;
}

```

Поставим точку останова в функции `body_fib` и поставим на нее условие, если `n == 0`. Запустим в дебаг режиме. Будем искать 5 число Фибоначчи.

В дебаг окошке есть стек вызовов рис. 9.15, в котором содержатся все вызванные функции. Нажав на какую-то функцию в нем можно узнать, в какой строке выполнение этой функции было прервано вызовом другой и состояние переменных на вызове.

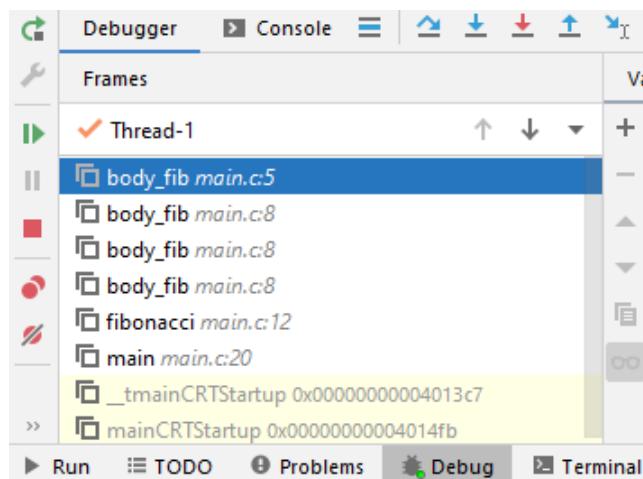


Рис. 9.15: Стек вызовов

Заметим, что в стеке вызовов видно 4 вызова `body_fib` и 1 вызов `fibonacci`. В сумме их ровно `n`. Запустим программу еще раз для определения 7 числа. Рассмотрим еще раз окно отладчика 9.16

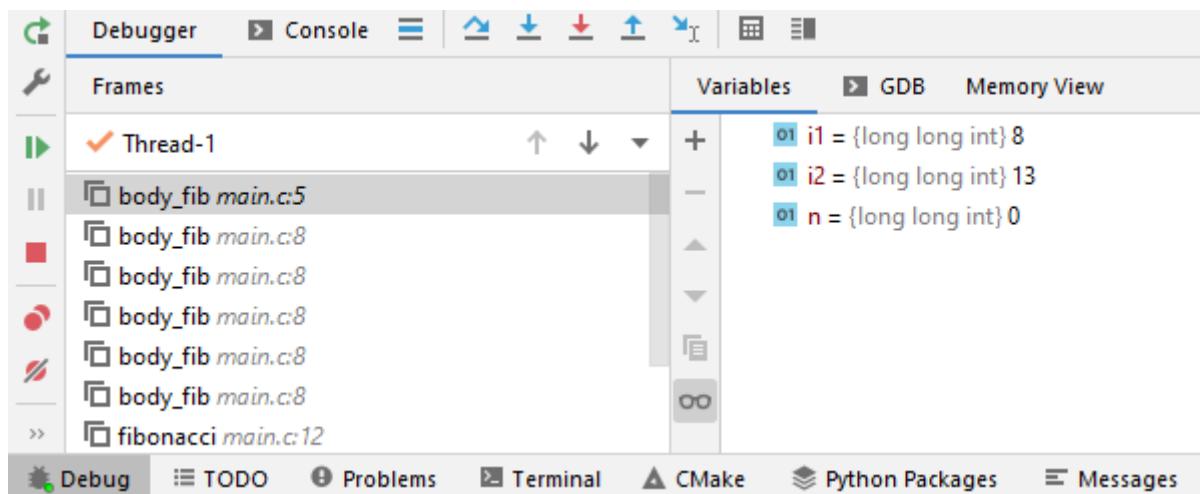


Рис. 9.16: Окно отладчика

Здесь 7 вызовов функций (Не считая `main`). Вот так мы на практике увидели, что для получения  $n$ —числа Фибоначчи потребуется  $n$  вызовов.

Пошаговая отладка позволяет, как находить ошибки в коде, так и выявлять нарушения логики алгоритма.



### 9.3 Автодополнение

*IDE* позволяют частично автоматизировать процесс подключения библиотек и обращения к программным объектам.

Допустим, мы писали какую-то программу и воспользовались функцией, библиотеку для которой не подключали рис. 9.17:

```

1 > int main() {
2   printf("Hello world");
3   return 0;
4 }

```

Рис. 9.17: Некорректная программа

Наведем курсор на красную функцию и нажмем на правую кнопку мыши, появится окно, представленное на рис. 9.18:

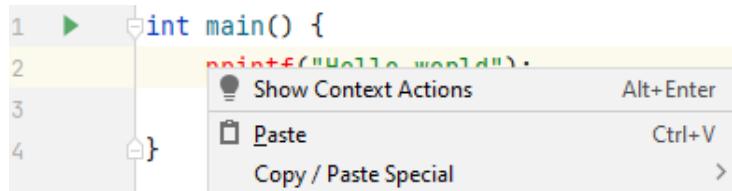


Рис. 9.18: Контекстное меню

Выберем **Show Context Actions**, и увидим предложения среды по исправлению ошибки. Остановимся на варианте **Import symbol 'printf'** и импортируем функцию `printf` из `stdio.h` **function 'printf' <stdio.h>**. Программа за нас напечатает знакомую директиву.

Если у нас имеется следующая программа:

---

```

#include <stdio.h>

int many_letters_sum(int a, int b) {
    return a + b;
}

int main() {
    int many_letters_1;
    int many_letters_2;

    return 0;
}

```

---

И мы хотим, модифицировать ее для ввода двух значений `int` и вывода их суммы, затратив минимальное количество сил. Воспользуемся автодополнением. Для этого начнем набирать функция, а потом, когда увидим ее в предложенном *Clion* списке рис. 9.19, нажмем клавишу *Tab*.

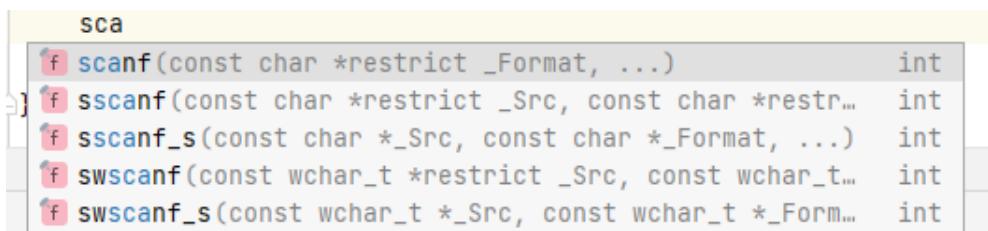


Рис. 9.19: Применение автодополнения

Учитывая, что в имени функции 5 символов. Смысла в этом видится мало. Теперь воспользуемся автодополнением для обращения к длинным переменным рис. 9.20. Когда увидим 2-ую переменную нажмем стрелку вниз и *Tab* или *Enter*:

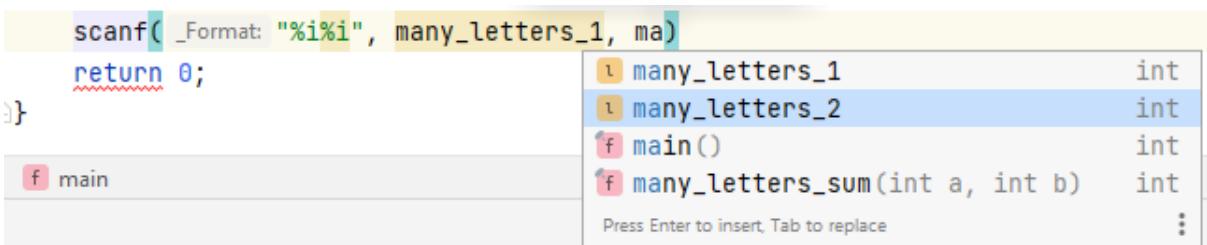


Рис. 9.20: Применение автодополнения

Далее напишем аналогично `printf`, вызвав длинную функцию рис. 9.21:

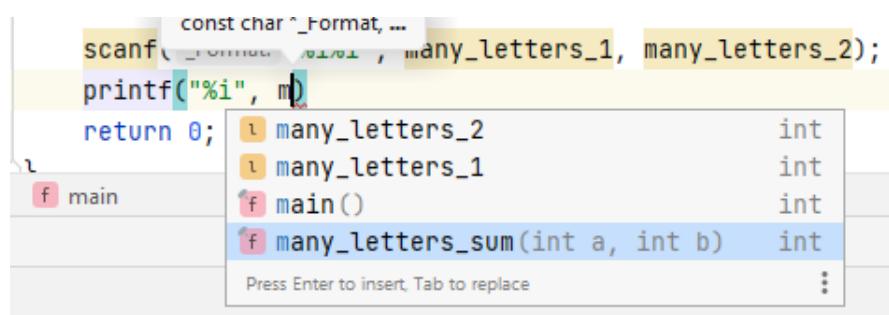


Рис. 9.21: Применение автодополнения

Не правда ли удобно?

```
scanf(_Format: "%i%i", many_letters_1, many_letters_2);
printf(_Format: "%i", many_letters_sum(many_letters_1, many_letters_2));
```

Рис. 9.22: Полученные строки

## 9.4 Выделение программных объектов

Принцип *DRY* (*Don't repeat yourself*) говорит, что код не должен дублироваться. Однако программист не всегда выделяет функции сразу, или забывает о необходимости введения дополнительной переменной. *Clion* позволяет быстро решить эту проблему.

Рассмотрим следующий код, в котором вычисляется значение функции

$$f(x, y) = 2 * x + y + |2 * x + y|$$

---

```
#include <stdio.h>
#include <math.h>

int main() {
    int x, y;
```

---

```

    scanf("%i%i", &x, &y);
    printf("%i", 2 * x + y + abs(2 * x + y));
    return 0;
}

```

---

В нем нарушен принцип *DRY*: выражение  $2 * x + y$  вычисляется дважды. Давайте выделим переменную под значение этого выражения, это должно сократить вычисления. Для этого выделим мышкой выражение  $2 * x + y$  и нажмём комбинацию клавиш *Ctrl+Alt+V*. *IDE* за нас выделит переменную и заменит выражения на обращения к ней рис. 9.23:

```

#include <stdio.h>
#include <math.h>

int main() {
    int x, y;
    scanf(_Format: "%d %d", &x, &y);
    int i = 2 * x + y;
    printf(_Format: "%d", i + abs(i));
    return 0;
}

```

Рис. 9.23: Выделение переменной

А теперь поговорим про выделение функций. Студенту Пете преподаватель задал лабораторную по сортировке выбором. Наш герой любит все делать в функции `main`, потому что так быстрее<sup>8</sup>. Вот и сейчас у него получилось:

---

```

#include <stdio.h>

int main () {
    int a[2500];
    int n;
    scanf("%i", &n);
    for (int i = 0; i < n; ++i)
        scanf("%i", &a[i]);
    int imax, t;
    for (int i = 0; i < n - 1; ++i) {
        imax = i;
        for (int j = i + 1; j < n; ++j)
            if (a[j] > a[imax])
                imax = j;
        t = a[imax];
        a[imax] = a[i];
        a[i] = t;
    }
    for (int i = 0; i < n; ++i)
        printf("%i ", a[i]);
}

```

---

Однако преподаватель принимает лабораторные, только если выделены функции. Поэтому Петя выделяет цикл ввода массива (строки 7-8) и после нажатия комбина-

<sup>8</sup>Выделять функции стоит по ходу написания проекта, но только не в самом конце.

ции *Ctrl+Alt+M* появляется окошко, в которой он указывает имя функции `read_arr` и ее расположение на *above* рис. 9.24:

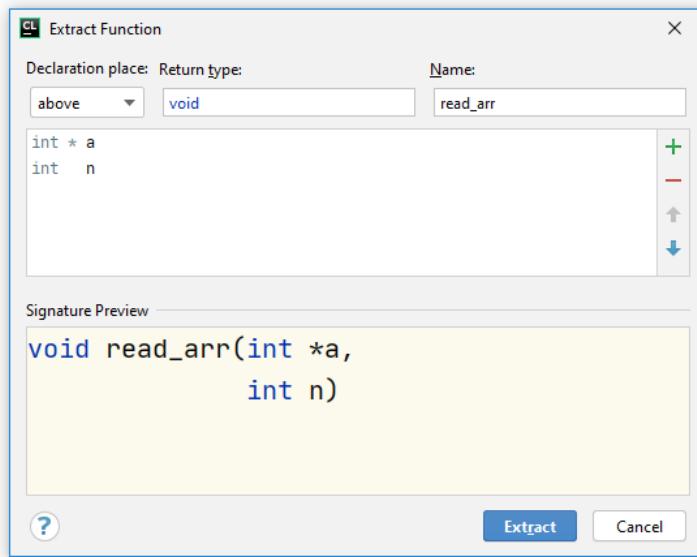


Рис. 9.24: Окно выделение функции

После нажатия кнопки *extract* Clion выделяет функцию, чем экономит драгоценное время Пети. Потом студент повторяет это действие для цикла вывода массива, и тела сортировки. В итоге Петя получает следующий код:

---

```
#include <stdio.h>

void read_arr(int *a, int n) {
    for (int i = 0; i < n; ++i)
        scanf("%i", &a[i]);
}

void sort(int *a, int n) {
    int imax, t;
    for (int i = 0; i < n - 1; ++i) {
        imax = i;
        for (int j = i + 1; j < n; ++j)
            if (a[j] > a[imax])
                imax = j;
        t = a[imax];
        a[imax] = a[i];
        a[i] = t;
    }
}

void write_arr(const int *a, int n) {
    for (int i = 0; i < n; ++i)
        printf("%i ", a[i]);
}

int main () {
    int a[2500];
```

```

int n;
scanf("%i", &n);

read_arr(a, n);
sort(a, n);
write_arr(a, n);
}

```

Помимо выделения переменных можно выделять свои типы, константы, параметры и т. д.. Но это остается уже на самостоятельное изучение. В качестве упражнения попробуйте выделить свой тип.

## 9.5 Горячие клавиши

Для ускорения работы с *IDE* можно применять ряд специальных сочетаний клавиш. Ниже приведен их перечень, овладение ими полезно для раскрытия возможностей средой. Обычно рекомендуют запоминать сочетания клавиш, постепенно вводя их в оборот.

1. *Ctrl + Space* вызов автодополнения.
2. *Ctrl + J* вызов списка доступных шаблонов кода (примеры будут описаны позже).
3. *Ctrl + Alt + T* вызов перечня шаблонов-окружений.
4. *Ctrl + W* выделение большего блока, относительно уже выделенного.
5. *Ctrl + Shift + A* поиск команды *IDE*. Применение: позволяет не пользоваться стандартной навигацией настроек, не удобной обилием кнопок.
6. *Ctrl + R* поиск и замена участков текста.

Применение представлено на рис. 9.25: переименование программный объектов в проекте.



Рис. 9.25: Поиск и переименование

7. *Ctrl + X* вырезать фрагмент.
  8. Выделение текста *Ctrl + /* закомментировать блок.
  9. *Alt + Enter* просмотр меню контекстных изменений.
- Применение представлено на рис. 9.26: вызов контекстного меню без мыши.

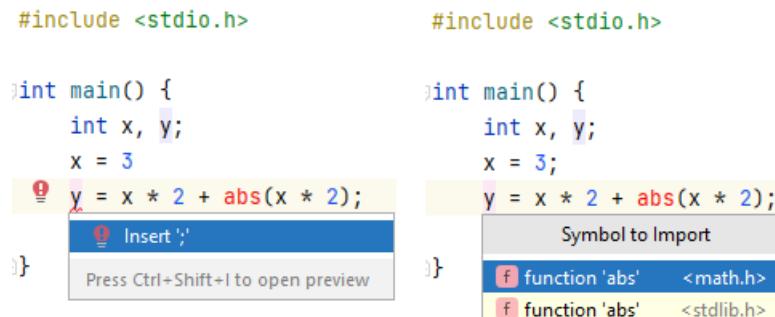


Рис. 9.26: Контекстное меню

10. *Ctrl + Shift + <Стрелка вверх или вниз>* перемещение строки.

11. *Alt + <Выделение колонки текста>* создание мульти-курсора.

Пример применения представлен на рис. 9.27: исправление опечатки, замена `getchar()` на `putchar()`.

```
int main() {
    int x, y;
    getchar('H');
    getchar('e');
    getchar('l');
    getchar('l');
    getchar('o');
    getchar(' ');
    getchar('w');
    getchar('o');
    getchar('r');
    getchar('l');
    getchar('d');
    return 0;
}
```

Рис. 9.27: Столбиковое выделение и множественный курсор

12. *Ctrl + Alt + V* выделение переменной.
13. *Ctrl + Alt + M* выделение функции.

Поддерживаются базовые сочетания клавиш.

1. *Ctrl + C* скопировать текст.

2. *Ctrl + X* вырезать текст.
3. *Ctrl + V* вставить текст.
4. *Ctrl + A* выделить текст файла.

## 9.6 Шаблоны кода

В программировании можно выделить часто повторяющиеся участки кода. Например, функции ввода и вывода массива, или заголовок цикла `for`. Для того, чтобы программист не переписывал их в каждой программе, придумали шаблоны кода.

Попробуйте в коде написать `for` и нажать *Tab* или *Enter*. При этом на экране появится заголовок цикла. Вы можете изменить имя счетчика, при этом оно поменяется не только в инициализации, но и в других разделах. Нажав 1 раз *Tab*, вы переместите курсор к логическому условию.

```
for (int i2 = 0; i2 < [i]; ++i2) {  
}
```

Рис. 9.28: Шаблон *for*

Вы воспользовались вставкой шаблона, называемого *Live Template* в *Clion*<sup>9</sup>.

Попробуем написать свой шаблон для функции ввода массива длины *n*. Для этого зайдем в меню *File*, а затем в *Settings*. Далее зайдем в *Editor*, и в *Live templates*. В появившемся оконке рис. 9.29 нажмем на "+" и *Live Template*.

---

<sup>9</sup>В *Microsoft Visual Studio Community* шаблоны кода имеют название *Snippets*. Реализованы они там посложнее, чем в *Clion*

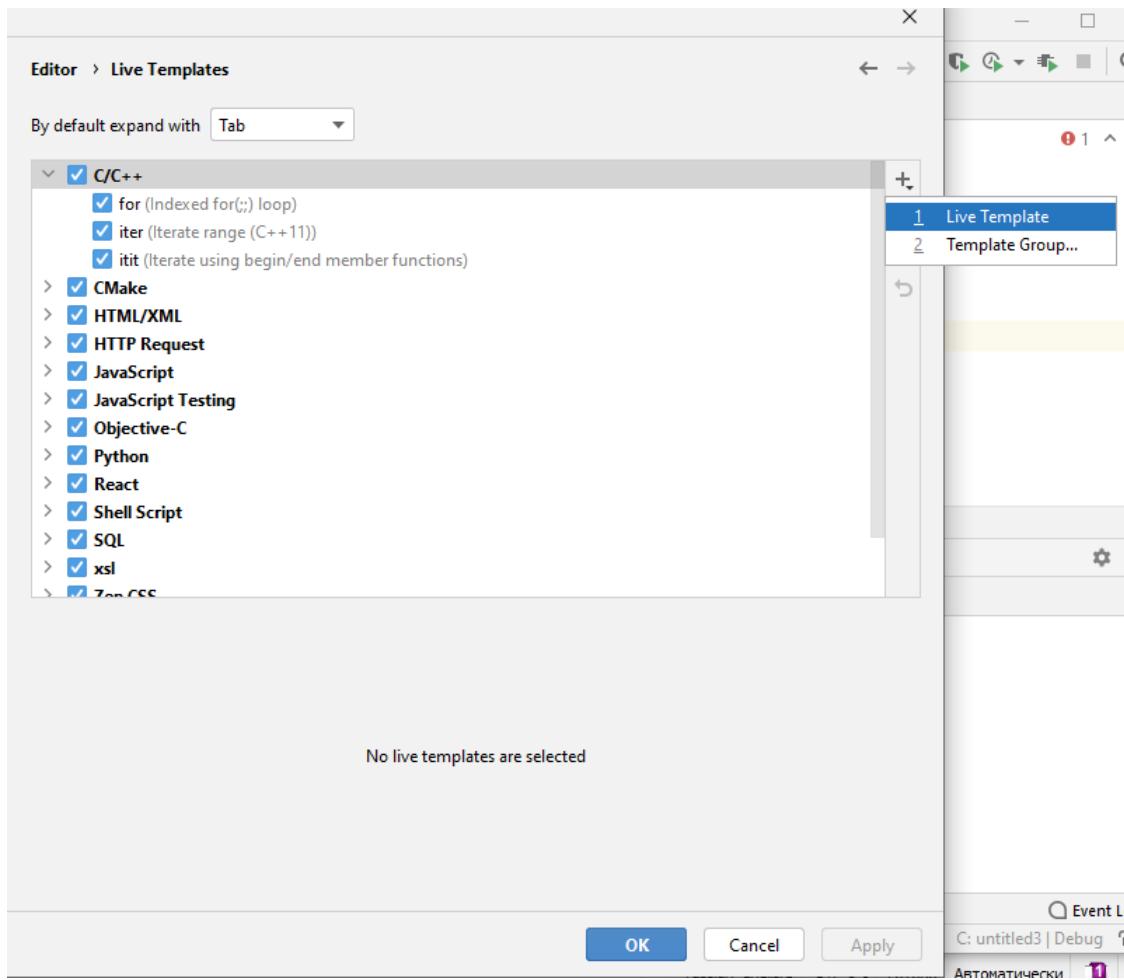


Рис. 9.29: Список шаблонов

Введем данные в появившееся окошко рис. 9.30:

- *Abbreviation* – символы, нажав после которых *Tab* мы можем вызвать шаблон;
- *Description* – описание (подсказка, которая будет выведена в процессе просмотра *Abbreviation*);
- *Template text* – непосредственно шаблон.

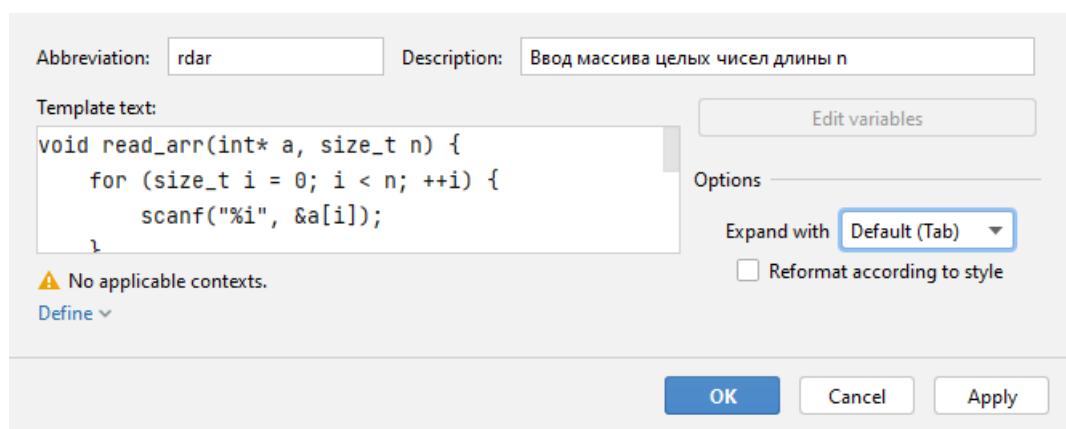


Рис. 9.30: Подготовка шаблона

Определим, программируя на каких языках можно вызвать наш шаблон. Для этого нажмем на *Define* и укажем  C.

А теперь испытаем наш шаблон. Откроем проект, напишем *rdrar* и нажмем *Tab* или *Enter*. Должна произойти вот такая вставка рис. 9.31:

```
#include <stdio.h>

void read_arr(int* a, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        scanf(_Format: "%i", &a[i]);
    }
}
```

Рис. 9.31: Применение шаблона кода

А теперь попробуем модифицировать наш шаблон. Заменим тип *int* нашего массива на *TYPE*, а после кода пропустим строку и укажем *END*, получив такой код (*Template text*):

---

```
void read_arr($TYPE$* a, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        scanf("%i", &a[i]);
    }
}
```

---

Теперь после вставки будет предложено ввести тип, а после ввода и нажатия *Tab* курсор будет перемещен в точку *END*.

## 9.7 Шаблон-окружение

На *Codeforces* часто попадаются задачки, в которых предусмотрен ввод нескольких наборов тестовых данных.

Программист Петя, с любовью к быстроте написания программ которого мы познакомились в выделении функций, знает замечательный способ ускорить написание цикла обработки тестового набора.

Способ замечательный, потому что позволяет ему за шесть нажатий на клавиатуру и одно выделение текста превратить такой код:

---

```
#include <stdio.h>

int main () {
    int n;
    scanf("%i", &n);

    printf("%i\n", n + 1);

    return 0;
}
```

---

В такой:

```
#include <stdio.h>

int main () {
    int t;
    scanf("%i", &t);

    while (t--) {
        int n;
        scanf("%i", &n);

        printf("%i\n", n + 1);
    }

    return 0;
}
```

Петя знает, что можно создать шаблон кода, который будет вставляться на выделенный фрагмент текста.

Чтобы догнать Петю наберите в среде *Clion* какой-нибудь код. Выделите его часть рис. 9.32 и нажмите *Code*, а затем *Surround With...*. В появившемся окне выберете наиболее знакомое для вас слово. В итоге ваш код должен быть обернут, как на рис. 9.32

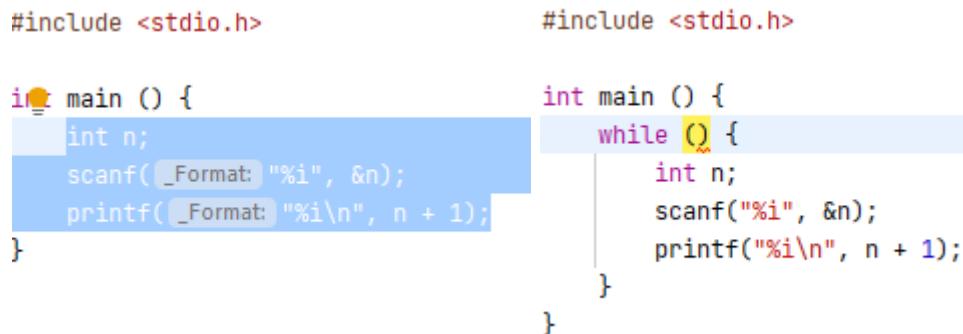


Рис. 9.32: Применение шаблона-окружения

Далее, когда мы знаем как шаблоны-обертки работают, попробуем повторить прием Пети. Создадим *Live Template* следующего вида рис. 9.33:

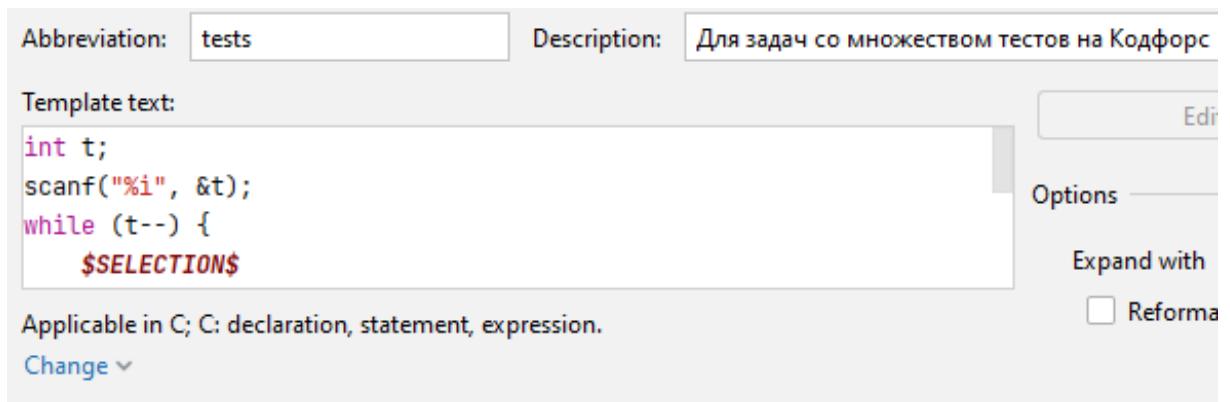


Рис. 9.33: Применение шаблона-окружения

На рис. 9.33 стоит обратить внимание на ключевое слово `$SELECTION$`. Им обозначено место куда будет помещен выделенный фрагмент. В остальном ничего не изменилось относительно обычных шаблонов кода.

Теперь выделенный код может быть обёрнут нашим окружением посредством `Ctrl + Alt + T`.

## 9.8 Вывод

Мы поговорили о полезных функциях *IDE*. Однако на этом они не заканчиваются. Уверены, что в дальнейшем вам нужно будет узнать о других возможностях среды.



В поиске новых знаний, рекомендуем обратиться к официальной страничке *Clion* в сети интернет<sup>10</sup>.

<sup>10</sup>Сайт *Clion* на английском. Будьте готовы к *google translate* и к переводу.

# Глава 10

## Алгоритмы обработки одномерных массивов

### 10.1 Алгоритмы, не зависящие от упорядоченности

#### 10.1.1 Ввод массива

##### Ввод массива

Необходимо ввести с клавиатуры массив  $a$  размера  $n$ .

Сложность по времени:  $O(n)$ .

Рассмотрим следующую задачу. Имеется массив (участок непрерывной памяти, в которой располагаются элементы одного типа). При его объявлении

```
int main() {
    int a[5];
    ...
}
```

выделяется кусок памяти из 5 элементов типа `int`, в которых хранится мусор:

Фрагмент памяти  
для 5 значений типа `int`,  
которые заполнены мусором.

`int a[5]`



Рис. 10.1: Объявление массива и состояние памяти

Ввести элементы массива можно было и так:

```
int main() {
    int a[5];
    scanf("%d %d %d %d %d", &a[0], &a[1], &a[2], &a[3], &a[4]);
    ...
}
```

но довольно очевидно, что если их количество будет велико, следует воспользоваться циклом:

---

```
int main() {
    int a[5];
    for (int i = 0; i < 5; i++)
        scanf("%d", &a[i]);
    // ...
}
```

---

Можно выделить функцию, в которой бы происходил ввод 5 элементов:

---

```
void inputArray5(int *a) {
    for (int i = 0; i < 5; i++)
        scanf("%d", &a[i]);
}

int main() {
    int a[5];
    inputArray5(a);
    // ...
}
```

---

Функция принимает адрес нулевого элемента массива (имя массива является указателем на нулевой элемент). Основной недостаток `inputArray5` заключается в том, что она подходит для ввода только 5 элементов. А если придётся ввести 2 массива: один размера 5, а другой размера 20? Хотелось бы использовать одну и ту же функцию для решения задач. Выполним модификацию: выделим параметр – количество вводимых элементов `n`. Так как в процессе работы функции мы не хотим давать возможность изменить количество элементов, объявляем константой:

---

```
void inputArray(int *a, const int n) {
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);
}
```

---

Тогда для ввода двух массивов подошел бы код:

---

```
int main() {
    int a[5];
    inputArray(a, 5);

    int b[20];
    inputArray(b, 20);
    // ...
}
```

---

Сделаем одну небольшую поправку. Имеется специальный тип `size_t`, который часто используется для указания размеров массивов и для переменных-индексов для обращения к элементам массива. При написании спецификации и кода будем использовать тип `size_t`.

Так как мы хотим запретить изменять значение `n` внутри функции, используем ключевое слово `const`.

Спецификация функции `inputArray`<sup>1</sup>:

1. Заголовок: `void inputArray(int *a, const size_t n)`.
2. Назначение: ввод массива по адресу `a` размера `n`.

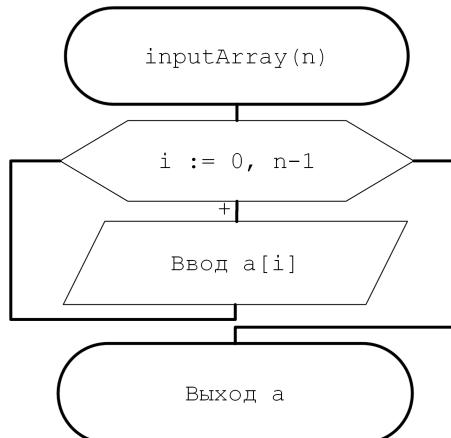


Рис. 10.2: Блок схема функции ввода массива

Код функции:

---

#### Листинг 26 Ввод массива размера $n$

---

```

void inputArray(int *a, const size_t n) {
    for (size_t i = 0; i < n; i++)
        scanf("%d", &a[i]);
}
  
```

---

Она может быть использована для ввода массива и подмассивов:

```

int main() {
    int a[10];
    inputArray(a, 10); // ввод элементов с 0 по 10

    int b[20];
    inputArray(b, 20); // ввод элементов с 0 по 20

    inputArray(&a[2], 5) // ввод элементов с индексами с 2 по 7
    inputArray(a + 2, 5) // ввод элементов с индексами с 2 по 7

    return 0;
}
  
```

---

<sup>1</sup>В первом блоке 'терминатор' указываются входные параметры. `a` – указатель на массив не является входным параметром.

Так как массив – это участок памяти, в котором располагаются элементы, а функция принимает указатель на какой-то участок памяти, можно передать адрес не только начала массива, а какой-то другой его части. Применяя арифметические операции к указателю, можно вычислять адреса соседних ячеек. Ознакомьтесь с примером:

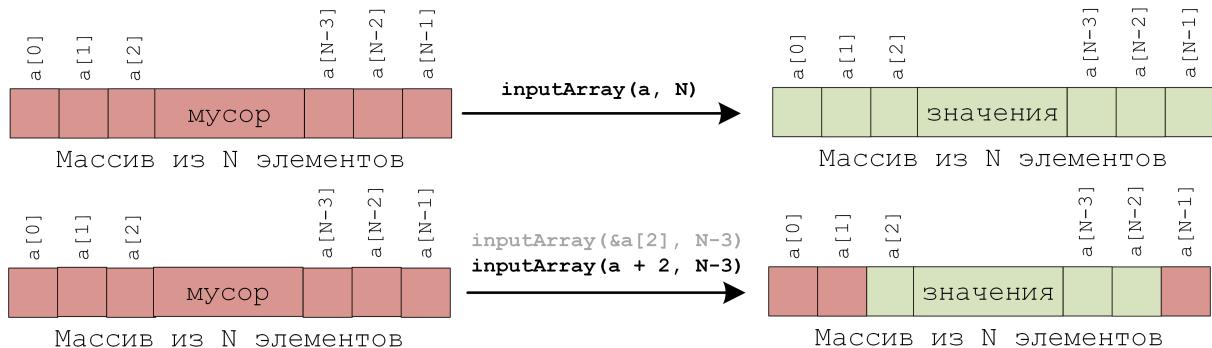


Рис. 10.3: Вызовы функции *inputArray*

Можно не писать специализированных функций, а грамотно подобрать аргументы готовой.

Иногда реализацию одной функции можно выразить через другую. Например, если бы стояла цель решить задачу ввода значений для элементов с индекса `startIndex` до `endIndex` можно было бы поступить и так (пример, очевидно, учебный):

```
void inputArrayRange(int *a, const size_t startIndex, const size_t endIndex) {
    size_t nElements = endIndex - startIndex + 1;
    inputArr(a + startIndex, nElements);
}
```

## 10.1.2 Вывод массива

### Выход массива

Необходимо вывести на экран массив  $a$  размера  $n$ .

Сложность по времени:  $O(n)$ .

Вывод массива мало чем отличается от ввода. Функция вывода не должна иметь возможность хоть каким-бы то ни было образом изменять значения в массиве через указатель. Мы можем запретить такую возможность. Для этого достаточно использовать тип `const int *` для указателя, который хранит в себе адрес начала массива.

Спецификация функции `outputArray`<sup>2</sup>:

- Заголовок: `void outputArray(const int *a, const size_t n);`
- Назначение: вывод массива по адресу `a` размера `n`.

Дополнительно выполним перенос на следующую строку<sup>3</sup>:

<sup>2</sup>Обратите внимание на оформление последнего блока блок-схемы для функции без выходных параметров.

<sup>3</sup>Момент, улучшающий чтение выводимых данных на блок-схеме отображаться не должен.

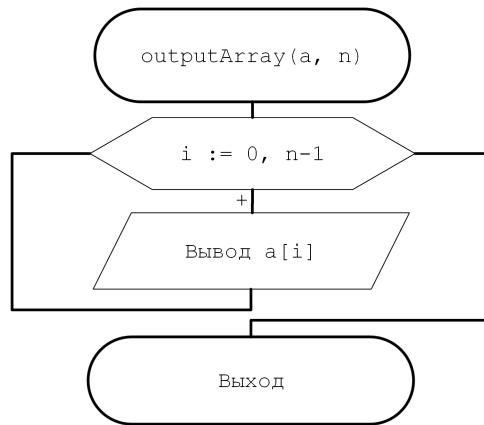


Рис. 10.4: Блок схема функции вывода массива

**Листинг 27** Вывод массива размера  $n$ 

```

void outputArray(const int *a, const size_t n) {
    for (size_t i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
  
```

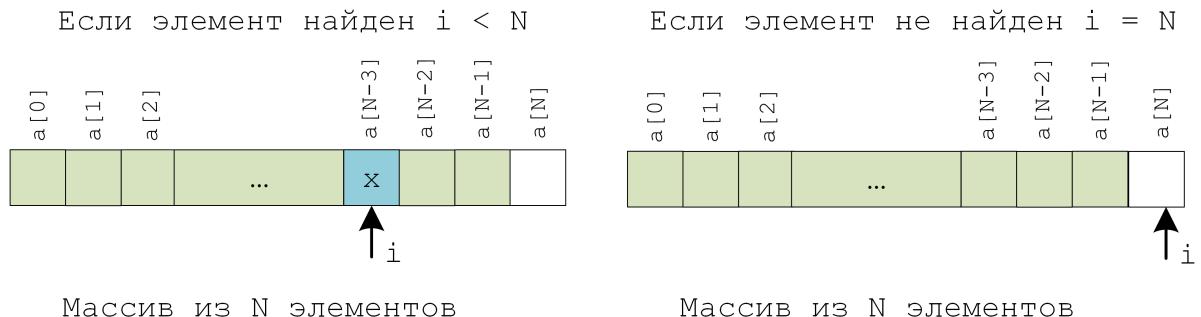
**10.1.3 Поиск позиции элемента, удовлетворяющему условию****Поиск позиции элемента, удовлетворяющему условию**

Дан массив  $a$  размера  $n$ . Необходимо найти такой минимальный  $i$ , что  $f(a[i])$  истина, или вернуть значение  $-1$ , если такого элемента нет.

Пусть в качестве  $f(x)$  выступает логическое выражение:  $x < 0$ .

Сложность по времени:  $O(n)$ .

Последовательно будем перебирать элементы. Если элемент подходит – прекращаем поиск и анализируем значение индекса, при котором закончился поиск. Возможны два случая:



Опишем два варианта решения с разным количеством инструкций возврата (блок-схемы 10.5, 10.6). В первом способе особое внимание стоит уделить очередности условий. Прежде всего надо проверить границы массива, а потом уже – значение элемента массива по индексу.

## Использование варианта

---

```
while (a[i] >= 0 && i < n)
```

---

вместо

---

```
while (i < n && a[i] >= 0)
```

---

является ошибочным, так как выход за пределы массива является неопределенным поведением. **Неопределенное поведение** означает, что результат компиляции и исполнения программы непредсказуем. Ожидание конкретного результата, в том числе аварийного завершения программы, при наличии в ней неопределенного поведения является неправильным.

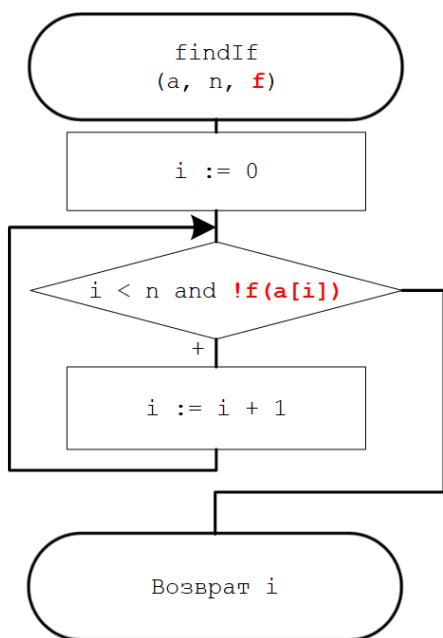
Преимущество данного решения – наличие лишь одного возврата. Минус – не самая простая логика.

---

```
size_t getFirstNegativeIndex(const int *a, const size_t n) {
    size_t i = 0;
    while (i < n && a[i] >= 0)
        i++;

    return i;
}
```

---



Для поиска позиции первого отрицательного вместо `!f(a[i])` будет написано  
 $\neg(a[i] < 0) \rightarrow a[i] \geq 0$

Для поиска первого четного:  
 $\neg(a[i] \% 2 == 0) \rightarrow a[i] \% 2$

Для поиска первого простого числа в массиве (при наличии функции `isPrime`): `!isPrime(a[i])`

Рис. 10.5: Первый способ решения задачи в общем случае. Вместо  $\neg f(a[i])$  будет подставлена требуемая операция сравнения

Во втором способе используется две инструкции возврата:

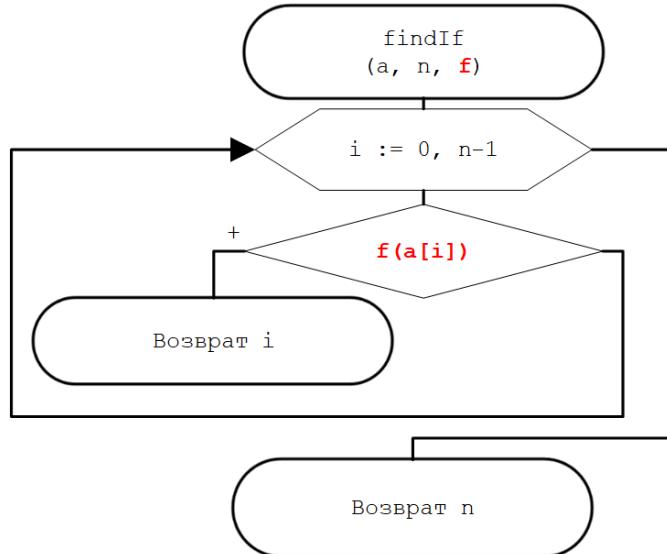


Рис. 10.6: Второй способ решения задачи. На блок-схеме он выглядит нетривиально, а вот код считается легче

---

#### Листинг 28 Поиск позиции первого отрицательного элемента в массиве

---

```

size_t getFirstNegativeIndex(const int *a, const size_t n) {
    for (size_t i = 0; i < n; i++) {
        if (a[i] < 0)
            return i;

        return n;
}
  
```

---

Спецификация функции `getFirstNegativeIndex`:

1. Заголовок: `size_t getFirstNegativeIndex(const int *a, const size_t n);`
2. Назначение: возвращает<sup>4</sup> индекс первого вхождения отрицательного значения в массиве `a` размера `n`, если элемент найден, иначе – `n`.

<sup>4</sup>Если основной задачей функции является возврат значения, рекомендуется начинать назначение со слова 'возвращает'.

### 10.1.4 Поиск количества элементов, удовлетворяющему условию

#### Поиск количества элементов, удовлетворяющему условию

Дан массив  $a$  размера  $n$ . Необходимо найти какое количество элементов массива удовлетворяют условию  $f(a[i])$ .

Пусть в качестве  $f(x)$  выступает логическое выражение:  $x$  – число, сумма цифр которого равняется 10.

Сложность по времени:  $O(n)^a$ .

<sup>a</sup>Без учета сложности алгоритма проверки логического выражения.

Выделим подзадачи:

1. Найти сумму цифр числа  $x$ .
2. Подсчёт элементов, удовлетворяющих условию.

Решение первой задачи может быть описано функцией  $getSumOfDigits$ , а решение второй – функцией  $countSumOfDigitsX$ .

Спецификация функции  $getSumOfDigits$ :

1. Заголовок: `int getSumOfDigits(long long x);`
2. Назначение: возвращает сумму цифр числа  $x$ .

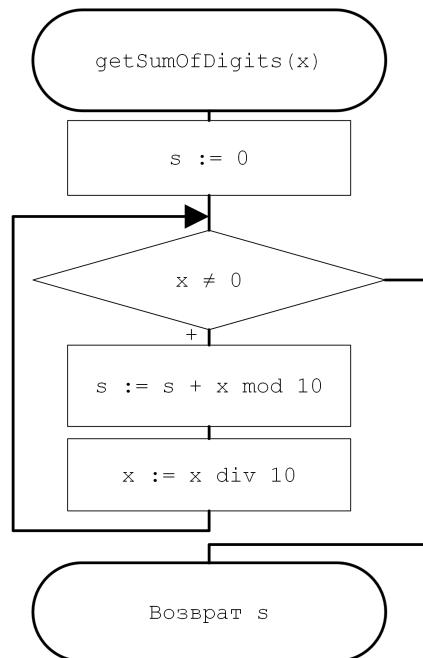


Рис. 10.7: Блок схема функции подсчёта суммы цифр

Спецификация функции  $countSumOfDigitsX$ :

1. Заголовок:

```
int countSumOfDigitsX(const int *a, const int n, const size_t digitsSum);
```

---

**Листинг 29** Подсчёт количества элементов, удовлетворяющих условию
 

---

```

int getSumOfDigits(long long x) {
    int sum = 0;
    while (x > 0) {
        sum += x % 10;
        x /= 10;
    }

    return sum;
}

int countSumOfDigitsX(const int *a, const size_t n,
                      const int digitsSum) {
    int count = 0;
    for (size_t i = 0; i < n; ++i)
        count += getSumOfDigits(a[i]) == digitsSum;

    return count;
}
  
```

---

2. Назначение: возвращает количество элементов массива `a` размера `n` сумма цифр которых равна `digitsSum`.

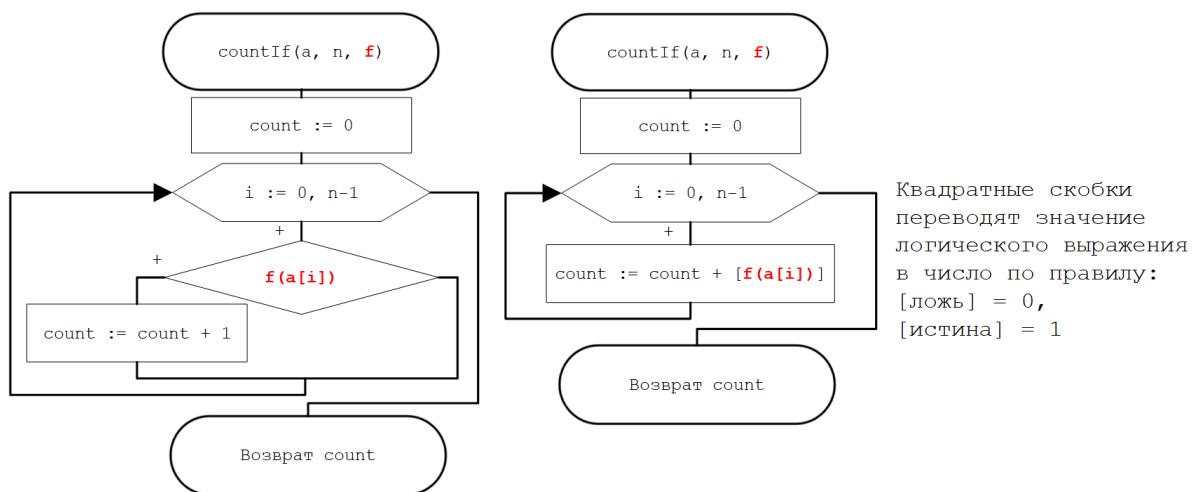


Рис. 10.8: Блок схема функции подсчёта количества элементов, удовлетворяющих условию

### 10.1.5 Поиск максимального количества подряд идущих элементов, удовлетворяющих условию

Поиск максимального количества подряд идущих элементов, удовлетворяющих условию

Дан массив  $a$  размера  $n$ . Необходимо найти какое максимальное количество подряд идущих элементов массива удовлетворяют условию  $f(a[i])$ .

Пусть в качестве  $f(x)$  выступает логическое выражение:  $x$  – четное число.

Сложность по времени:  $O(n)$ .

В решении данной задачи надо подумать, каким образом действовать. Если встречается четное число, тогда нужно увеличить значение счетчика текущего количества четных чисел на единицу. В противном случае проверим, а больше ли четных чисел накопилось, чем во всех случаях, которые встречались раньше. Если текущая подпоследовательность была длиннее – изменяем значение максимума.

Стоит отметить так же и такую ситуацию: все числа могут оказаться четными или самая длинная последовательность находится в конце (максимум не будет обновлен). Требуются определенные модификации алгоритма для обработки такого случая.

---

**Листинг 30** Поиск максимального количества подряд идущих элементов, удовлетворяющих условию.

---

```
int getMaxEvenChainLen(const int *a, const size_t n) {
    int maxLen = 0;
    int currentLen = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] % 2 == 0)
            currentLen++;
        else {
            maxLen = max(maxLen, currentLen);
            current = 0;
        }
    }
    maxLen = max(maxLen, currentLen);

    return maxLen;
}
```

---

1. Заголовок: `int getMaxEvenChainLen(const int *a, const size_t n);`
2. Назначение: возвращает максимальное количество подряд идущих четных элементов массива `a` размера `n`.

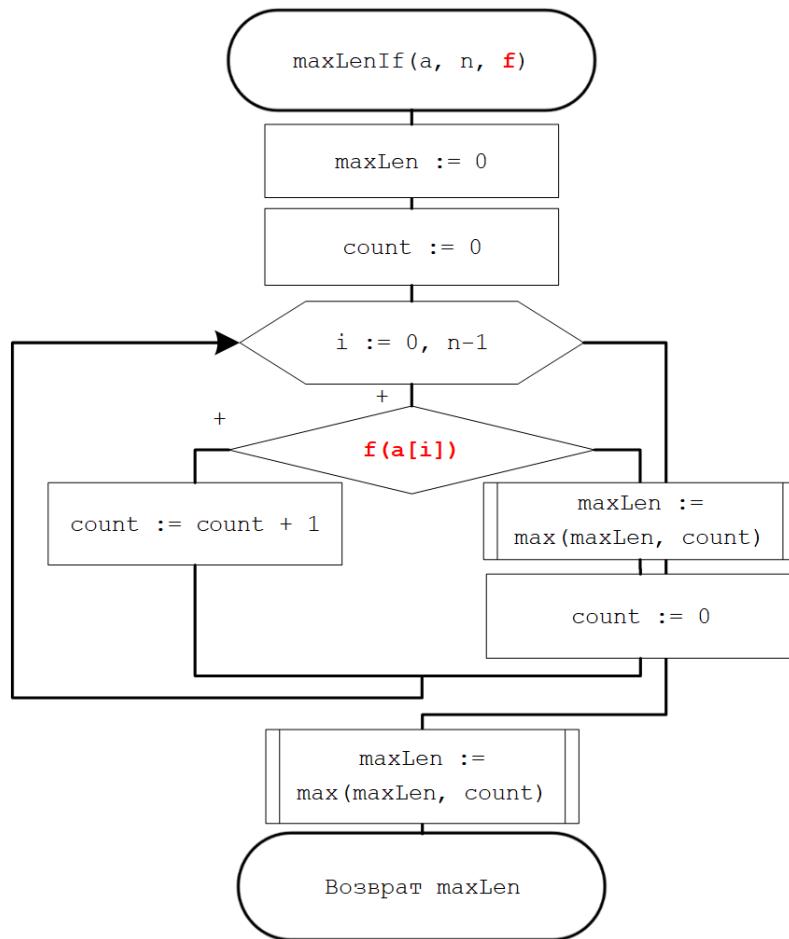


Рис. 10.9: Блок-схема алгоритма для нахождения максимального количества подряд идущих элементов, удовлетворяющих условию

### 10.1.6 Однопроходный алгоритм удаления

#### Однопроходный алгоритм удаления

Дан массив  $a$  размера  $n$ . Необходимо удалить из него все элементы, удовлетворяющие условию  $f(a[i])$ .

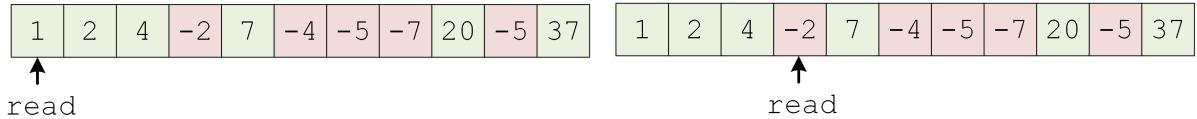
Пусть в качестве  $f(x)$  выступает логическое выражение:  $x$  – отрицательное число.

Сложность по времени:  $O(n)$ .

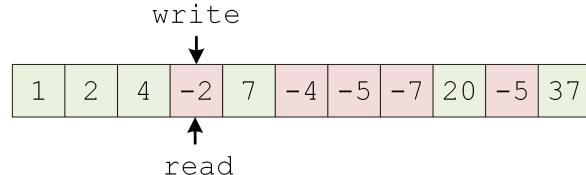
У нас имеется массив размера  $n$ , и мы планируем удалить из него некоторые элементы. Преобразованную последовательность будем получать на той же области памяти, на которой записаны исходные данные.

Идея решения такой задачи состоит в следующем: надо создать два индекса: один будет отвечать за позицию для чтения, другой – для записи. Последовательно просматриваем значения. Если его не нужно удалять – оно записывается в позицию для записи (после чего последняя смещается дальше). Если элемент надо удалять – индекс для чтения движется дальше.

1. Пропускаем элементы, которые не будут удалены:

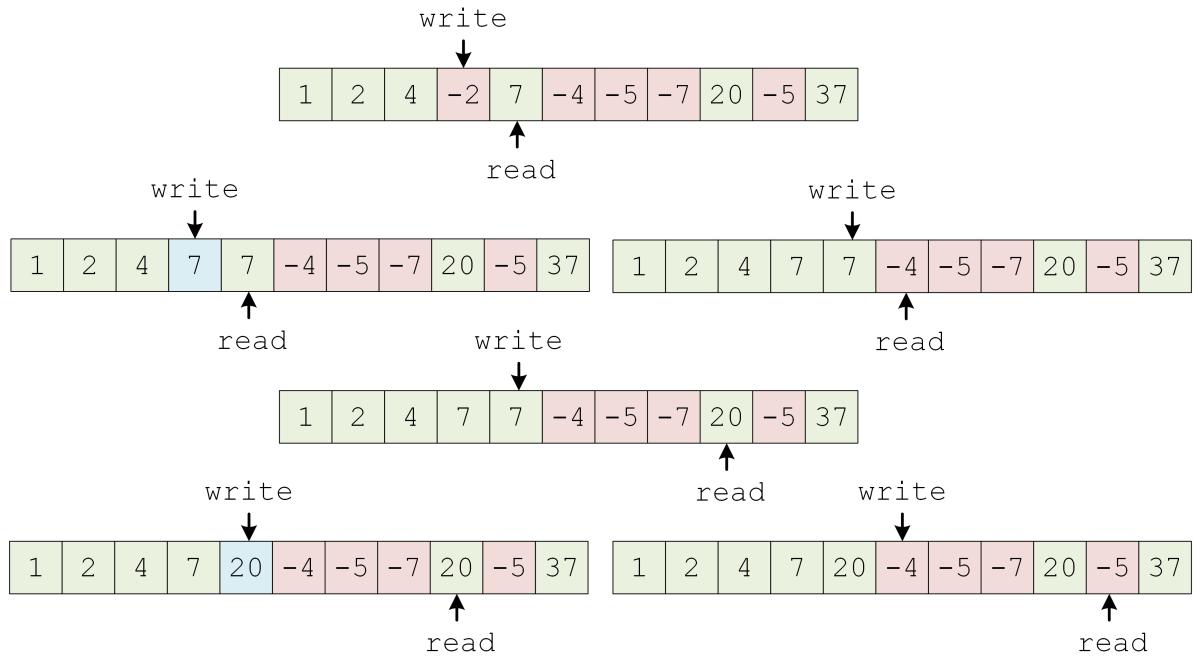


2. Если последовательность не закончилась, создаём индекс для записи:



3. Если элемент в индексе по чтению не нужно удалять – его значение сохраняется в позиции для записи. После чего индекс для записи движется дальше.

Индекс для чтения смещается дальше.



4. В конце обработки обрезаем массив:

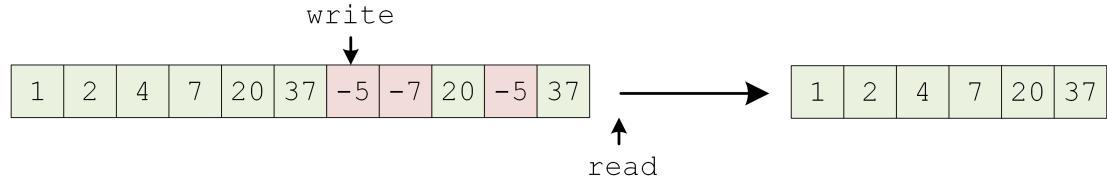


Рис. 10.10: Однопроходный алгоритм удаления

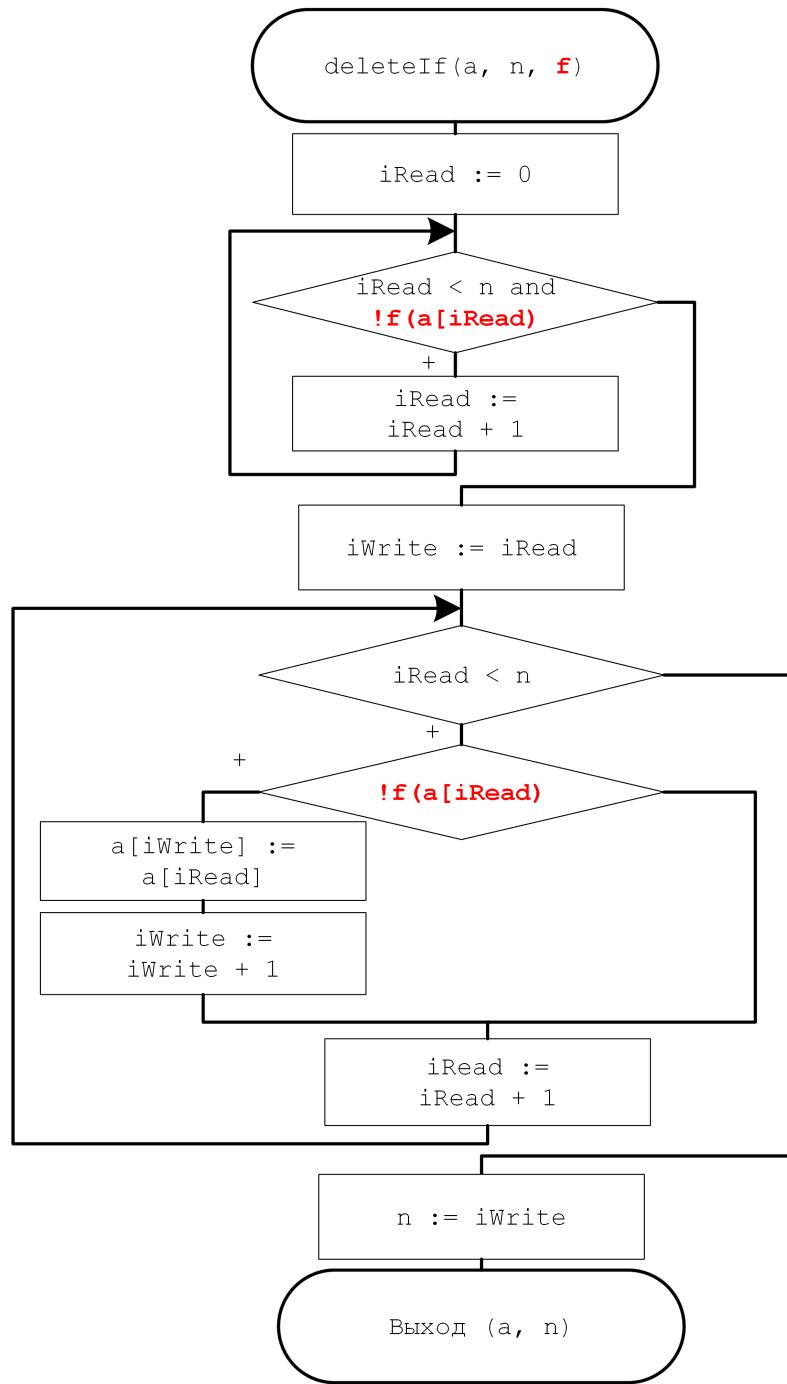


Рис. 10.11: Блок-схема однопроходного алгоритма удаления

Спецификация функции `deleteNegative`:

1. Заголовок: `void deleteNegative(int *a, size_t *n);`
2. Назначение: удаляет из массива `a` размера `n` отрицательные элементы. Сохраняет в `n` количество оставшихся элементов.

---

**Листинг 31** Алгоритм однопроходного удаления (итеративный)
 

---

```

void deleteNegative(int *a, size_t *n) {
    size_t iRead = 0;
    while (iRead < *n && a[iRead] >= 0)
        iRead++;

    size_t iWrite = iRead;
    while (iRead < *n) {
        if (a[iRead] >= 0) {
            a[iWrite] = a[iRead];
            iWrite++;
        }
        iRead++;
    }

    *n = iWrite;
}

int main() {
    int a[5] = {1, 2, -4, -3, -4};
    size_t n = sizeof(a) / sizeof(int);

    deleteNegative(a, &n);
    printf("%d", n); // n = 2
    // Функция изменила значение n, так как бы передан её адрес
    // в deleteNegative.

    // Если при помощи операции косвенного доступа происходят
    // изменения переменной n - меняется непосредственно
    // n, которая находится в функции main

    return 0;
}
  
```

---

Особое внимание следует уделить моменту, что размер массива объявлен как указатель на целое. Это сделано по причине того, что мы хотим, чтобы все изменения, которые бы произошли над размером массива в функции `deleteNegative` отразились на переменной, адрес которой передаётся в функцию. Обратитесь к примеру<sup>5</sup>

<sup>5</sup>Фрагмент `size_t n = sizeof(a) / sizeof(int)` работает только для случая, когда массив объявлен в той же функции, где и используется данный приём.

### 10.1.7 Вставка элемента с сохранением относительного порядка других элементов

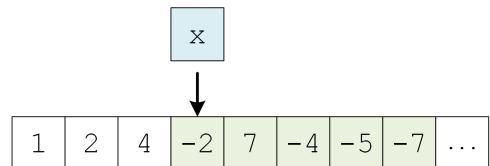
#### Вставка элемента с сохранением относительного порядка других элементов

Дан массив  $a$  размера  $n$ . Необходимо вставить элемент в позицию  $i$ . Будем считать, что исходный порядок элементов важен и не должен быть нарушен.

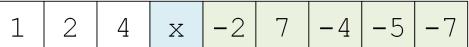
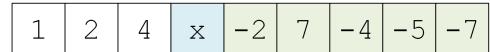
Сложность по времени:  $O(n)$ .

Идея алгоритма:

1. Смещаем все элементы с позиции вставки до последнего на 1 вправо:



2. Записываем значение



Спецификация функции *insert*<sup>6</sup>:

1. Заголовок: `void insert(int *a, int *n, const int pos, const int value);`
2. Назначение: вставляет значение `value` в позицию `pos` массива `a` размера `n`. Увеличивает количество элементов `n` на единицу.

---

#### Листинг 32 Вставка элемента с сохранением порядка элементов

---

```
void insert(int *a, size_t *n, const size_t pos, const int value) {
    for (size_t i = *n - 1; i >= pos; i--)
        a[i + 1] = a[i];
    a[pos] = value;

    (*n)++;
}
```

---

Стоит сделать оговорку, что размер массива `a` после вставки элемента станет на единицу больше. Нужно быть уверенным в том, что для него выделено достаточно памяти, чтобы не допустить запись за пределы массива. Рассмотрим пример:

---

<sup>6</sup>Корректная реализация `insert` несколько более сложная, если использовать тип `size_t`.

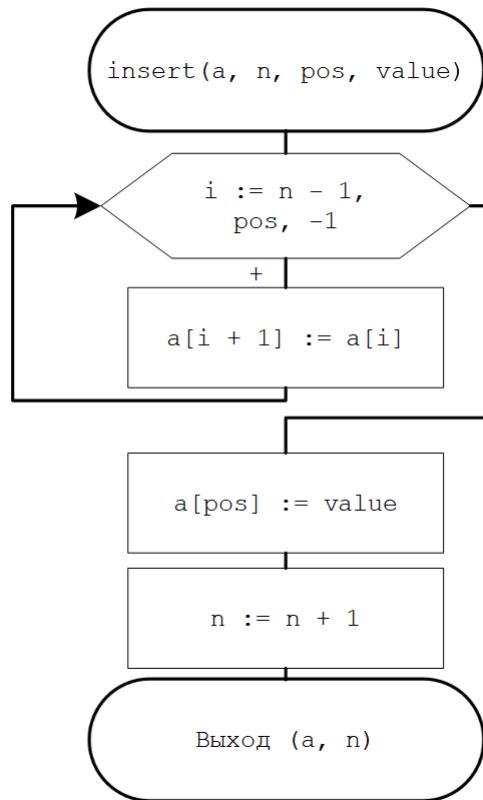


Рис. 10.12: Блок-схема алгоритма вставки элемента

---

```

int main() {
    int a[10];

    size_t size;
    scanf("%u", &size);

    inputArray(a, size);      // вводим массив размера size

    insert(a, &size, 5, 100) // осуществляем вставку элемента
                           // количество заполненных массива увеличивается
                           // на один
    return 0;
}
  
```

---

Памяти выделено под 10 элементов типа `int`. Если пользователь введёт в `inputArray` 10 элементов, то вставка одиннадцатого элемента – неопределенное поведение. Если было введено меньше 10 значений в 7 строке, вставка будет безопасной.

### 10.1.8 Удаление элемента с сохранением относительного порядка других элементов

#### Удаление элемента с сохранением относительного порядка других элементов

Дан массив  $a$  размера  $n$ . Необходимо удалить элемент на позиции  $i$ . Будем считать, что исходный порядок элементов важен и не должен быть нарушен.

Сложность по времени (худший случай):  $O(n)$ .

При удалении элемента, размер массива должен уменьшиться на единицу. Все элементы с  $\text{pos} + 1$  по  $n - 1$  позицию должны сместиться на один влево:

1	2	4	x	-2	7	-4	-5	-7
1	2	4	-2	7	-4	-5	-7	-7

Спецификация функции *deleteByPosSaveOrder*:

1. Заголовок: `void deleteByPosSaveOrder(int *a, size_t *n, const size_t pos);`
2. Назначение: удаляет элемент по индексу  $\text{pos}$  из массива  $a$  размера  $n$ . Уменьшает количество элементов  $n$  на единицу.

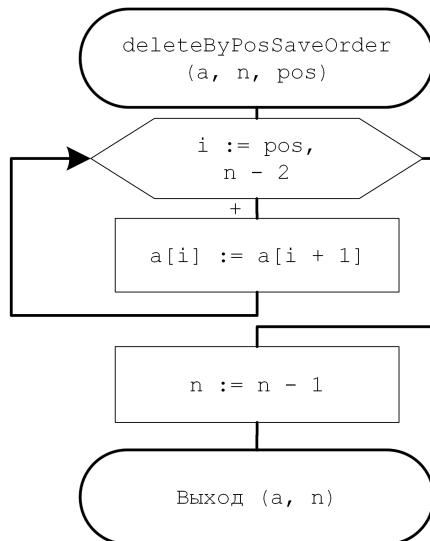


Рис. 10.13: Блок-схема удаления элемента из массива

**Листинг 33** Удаление элемента с сохранением относительного порядка других элементов

```

void deleteByPosSaveOrder(int *a, size_t *n, const size_t pos) {
    for (size_t i = pos; i < *n - 1; i++)
        a[i] = a[i + 1];

    (*n)--;
}
  
```

### 10.1.9 Обращение элементов массива

#### Обращение элементов массива

Дан массив  $a$  размера  $n$ . Необходимо обратить его элементы: обменять первый элемент с последним, второй – с предпоследним и так далее.

Сложность по времени:  $O(n)$ .

Блок-схема алгоритма на рисунке 10.14.

Приведём два решения: первый вариант будет использовать индексы, второй – через указатели:

---

#### Листинг 34 Обращение элементов массива

---

```
void reverse(int *a, const size_t n) {
    for (size_t i = 0, j = n - 1; i < j; i++, j--)
        swap(&a[i], &a[j]);
}

void reverse(int *a, const size_t n) {
    for (int *pLeft = a, *pRight = a + n - 1;
         pLeft < pRight;
         pLeft++, pRight--)
        swap(pLeft, pRight);
}
```

---

Спецификация функции *reverse*:

1. Заголовок: `void reverse(int *a, const size_t n);`
2. Назначение: обращение элементов массива `a` размера `n`.

### 10.1.10 Проверка упорядоченности массива по неубыванию

#### Проверка упорядоченности массива по неубыванию

Дан массив  $a$  размера  $n$ . Необходимо проверить, является ли он упорядоченным по неубыванию.

Сложность по времени (в худшем случае):  $O(n)$ .

Реализация алгоритма:

```
int isNonDecreasing(const int *a, const int n) {
    int i = 1;
    while (i < n && a[i - 1] <= a[i])
        i++;

    return i == n;
}
```

---

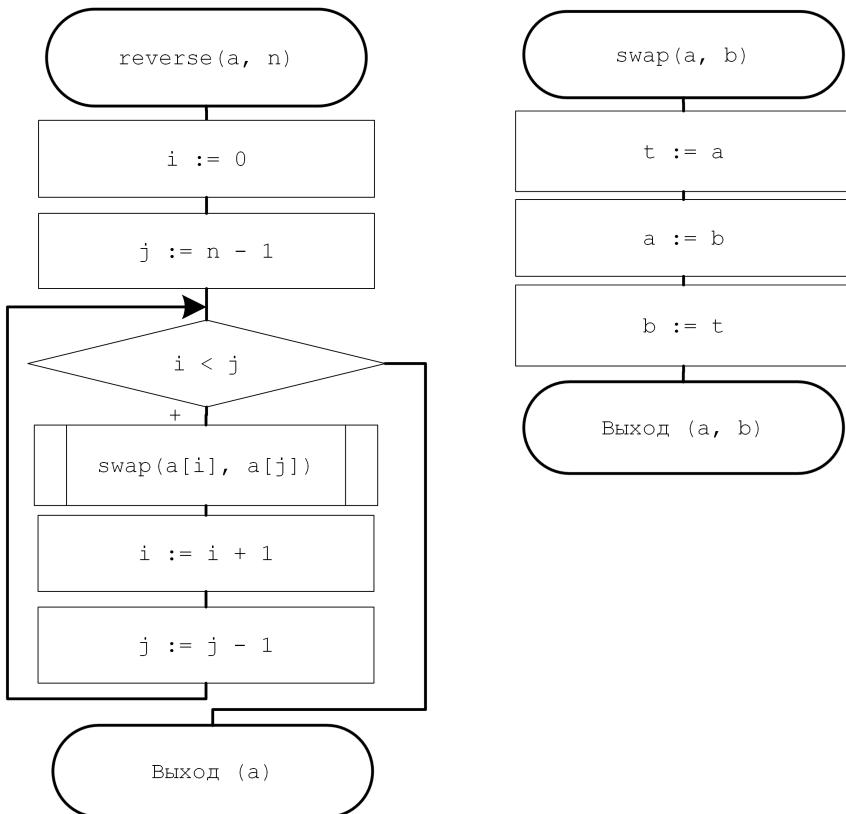


Рис. 10.14: Блок-схема обращения элементов массива и обмена двух элементов

---

### Листинг 35 Проверка упорядоченности массива по неубыванию

---

```

int isNonDecreasing(const int *a, const int n) {
    for (int i = 1; i < n && a[i - 1] <= a[i]; i++)
        if (a[i - 1] > a[i])
            return 0;

    return 1;
}

```

---

Спецификация функции *isNonDecreasing*:

1. Заголовок: `int isNonDecreasing(const int *a, const int n);`
2. Назначение: возвращает значение 'истина', если массив `a` размера `n` является упорядоченным по неубыванию, иначе – 'ложь'.

## 10.2 Неупорядоченные массивы

### 10.2.1 Добавление элемента в конец массива

#### Добавление элемента в конец массива

Дан массив  $a$  размера  $n$ . Необходимо добавить элемент  $x$  в конец массива.

Сложность по времени:  $\mathbf{O}(1)$ .

Если требуется добавить элемент в конец массива и выделенной памяти достаточно – необходимо значению по индексу  $n$  присвоить вставляемый элемент и увеличить размер массива на единицу:

---

#### Листинг 36 Добавление элемента в конец массива

---

```
void append(int *a, size_t *n, const int value) {
    a[*n] = value;
    (*n)++;
}
```

---

### 10.2.2 Удаление элемента из массива без сохранения относительного порядка других элементов

#### Удаление элемента из массива без сохранения относительного порядка других элементов

Дан массив  $a$  размера  $n$ . Необходимо удалить элемент на позиции  $i$ . Конечный порядок элементов неважен.

Сложность по времени:  $\mathbf{O}(1)$ .

Для удаления элемента из неупорядоченного массива достаточно переместить последний элемент массива на место удаляемого, и уменьшить размер массива на единицу:

---

#### Листинг 37 Удаление элемента из массива без сохранения относительного порядка других элементов

---

```
void deleteByPosUnsaveOrder(int *a, size_t *n, size_t pos) {
    a[pos] = a[*n - 1];
    (*n)--;
}
```

---

## 10.3 Одномерные массивы и работа с динамической памятью

### Стек

**Стек** – это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Работа со стековыми переменными осуществляется быстро. Но имеется ограничение на размер стека – это фиксированная величина, и превышение лимита выделенной на стеке памяти приведёт к переполнению стека.

Стек позволяет управлять памятью наиболее эффективным образом – но если вам нужно использовать динамические структуры данных, то стоит обратить внимание на кучу.

### Куча

**Куча** – это хранилище памяти, также расположеннное в ОЗУ<sup>7</sup>, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок, к ней можно обратиться из любого участка приложения. По завершению работы программы все выделенные участки памяти освобождаются.

Взаимодействие с кучей происходит при помощи указателей. Можно выделить фрагмент памяти в куче и сохранить адрес начала фрагмента, зная который, мы можем получить доступ к этому значению. В языках без сборщика мусора (C, C++) разработчику нужно вручную освобождать ресурсы, которые больше не используются. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу кучи.

В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности.

### Функции для работы с динамической памятью

При написании подавляющего количества приложений, возникает работа с динамической памятью. Например, размеры массива не всегда могут быть известны заранее. Предположим, что перед нами стоит задача: отсортировать массив из **n** элементов, где **n** заранее неизвестно. Например, оно вводится пользователем с клавиатуры.

Первое, что приходит в голову – выделить памяти с запасом. С запасом это сколько? Сто, тысяча, десятки тысяч? Пользователь может вообще ввести 10 элементов, а наша избыточная память будет простаивать. С другой стороны, там может оказаться значение большее, чем наше предположение, тогда памяти будет не хватать.

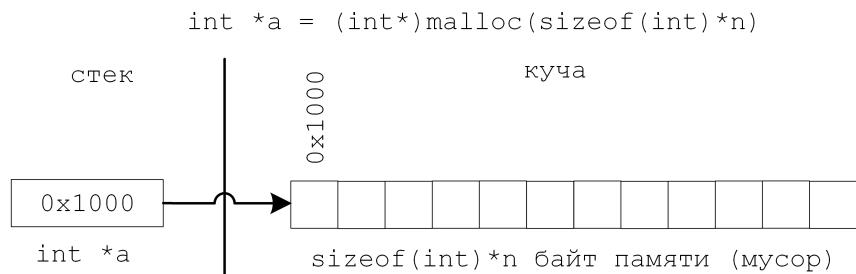
---

<sup>7</sup>**Оперативное запоминающее устройство** (ОЗУ) – это компонент, который позволяет компьютеру кратковременно хранить данные и осуществлять быстрый доступ к ним.

Если количество требуемой памяти заранее неизвестно, используются динамические массивы.

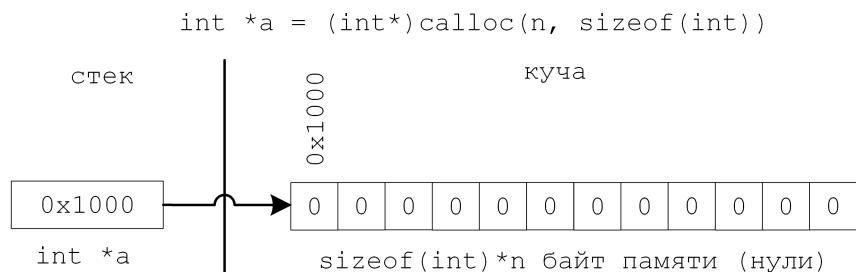
За работу с динамической памятью в языке С отвечают следующие функции, определенные в заголовочном файле `malloc.h`:

- `void* malloc(size_t sizeem)` – выделяет блок памяти, размером `sizeem` байт, и возвращает указатель на начало блока. Содержание выделенного блока памяти не инициализируется, оно остается с неопределенными значениями. Возвращаемым значением функции является нетипизированный указатель `void*`, который может быть приведен к требуемому типу указателя.



Если функции не удалось выделить блок памяти, возвращается значение `NULL`.

- `void* calloc(size_t number, size_t size)` – выделяет блок памяти для массива размером — `number` элементов, каждый из которых занимает `size` байт, и инициализирует все свои биты в нулями.



Возвращает указатель на выделенный блок памяти. Тип данных, на который ссылается указатель всегда `void*`, поэтому это тип данных может быть приведен к желаемому. Если функции не удалось выделить требуемый блок памяти, возвращается `NULL`.

- `void* realloc(void* ptrmem, size_t size)` – выполняет перераспределение памяти. Размер блока памяти, на который ссылается параметр `ptrmem` становится равным `size` байтов. Блок памяти может уменьшаться или увеличиваться в размере. Эта функция может перемещать блок памяти на новое место вместе с данными, в этом случае функция возвращает указатель на новое место в памяти. Содержание блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Отбрасываются только те данные, которые не поместились в новый блок. Если новое значение `size` больше старого, то содержимое вновь выделенной памяти будет неопределенным.

Тип данных возвращаемого значения всегда `void*`, который может быть приведен к любому другому. Если функции не удалось выделить требуемый блок

памяти, возвращается нулевой указатель `NULL`, и блок памяти, на который указывает аргумент `ptr` остается неизменным:

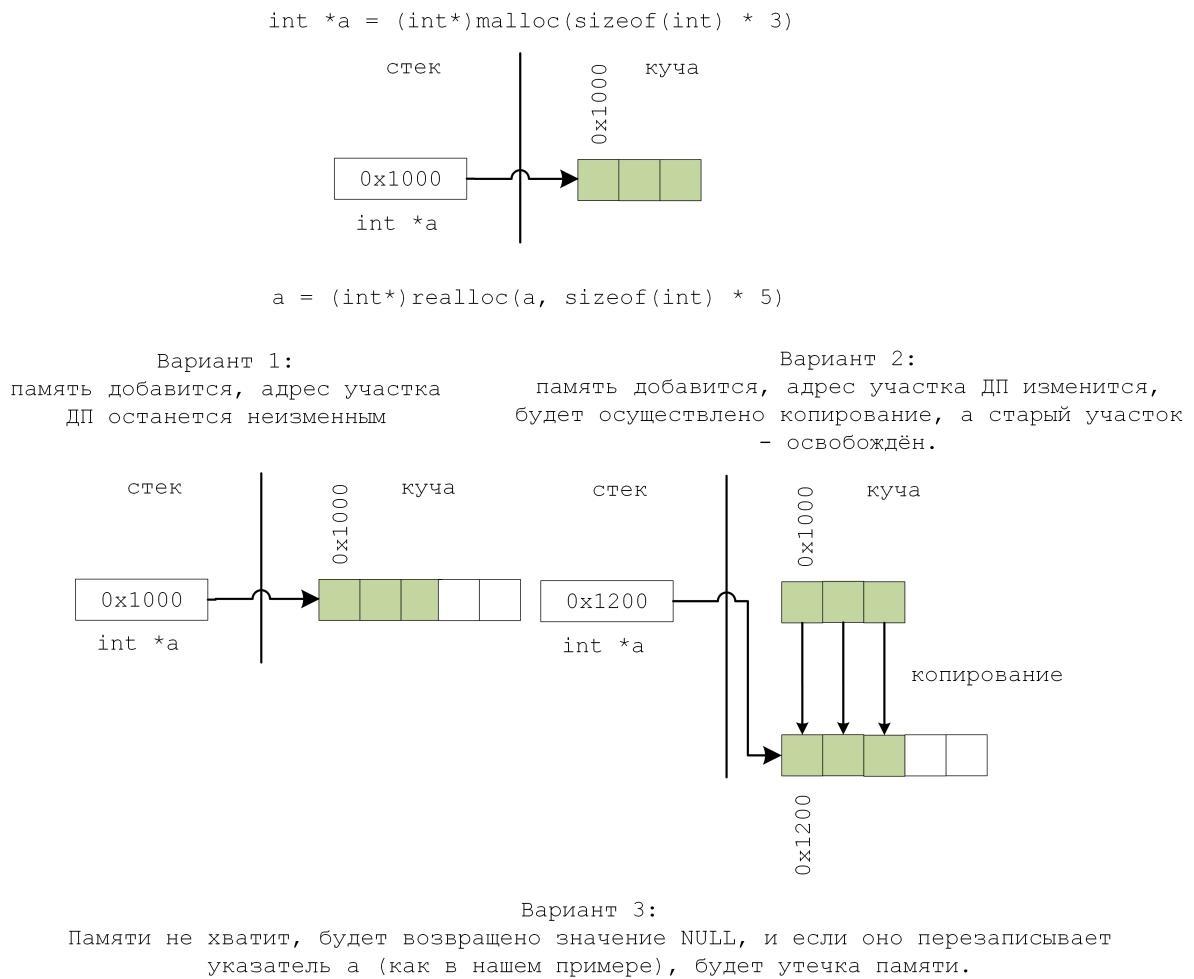


Рис. 10.15: Варианты работы `realloc`

- `void free(void *ptrmem)` - освобождает место в памяти. Блок памяти, ранее выделенный с помощью вызова `malloc`, `calloc` или `realloc` освобождается. То есть освобожденная память может дальше использоваться программами или ОС.

Рассмотрим несколько задач.

### 10.3.1 Ввод и вывод массива (вариация 1)

#### Ввод и вывод массива

С клавиатуры вводится массив из  $n$  чисел (число  $n$  вводится с клавиатуры). Необходимо вывести массив.

Сложность по времени:  $O(n)$ .

Опишем блок-схему в укрупненных блоках. Обратите особое внимание на блоки, которые отвечают за выделение или освобождение памяти:

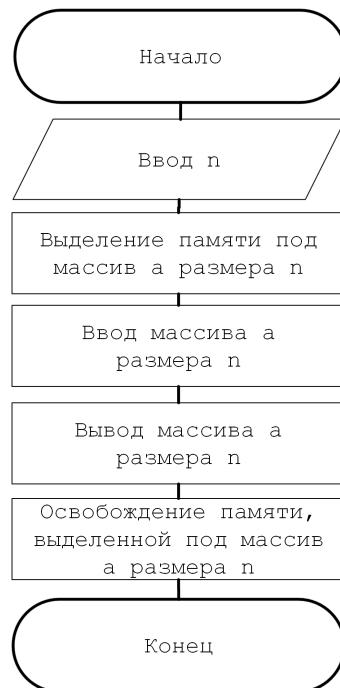


Рис. 10.16: Алгоритм в укрупненных блоках

Код для решения задачи:

---

```

#include <stdio.h>
#include <malloc.h>

void inputArray(int *a, const size_t size) {
    for (size_t i = 0; i < size; ++i)
        scanf("%d", &a[i]);
}

void outputArray(int *a, const size_t size) {
    for (size_t i = 0; i < size; ++i)
        printf("%d ", a[i]);
    printf("\n");
}

int main() {
    int n;
    scanf("%d", &n);

    // выделение памяти под n элементов типа int
    int *a = malloc(sizeof(int) * n);
    inputArray(a, n);

    outputArray(a, n);

    // освобождение памяти
    free(a);

    return 0;
}

```

---

### 10.3.2 Ввод и вывод массива (вариация 2)

#### Ввод и вывод массива

С клавиатуры вводится массив чисел. Признак конца ввода – 0. Необходимо вывести массив.

Сложность по времени:  $O(n)$ .

В данном случае момент ввода массива и выделения памяти происходят одновременно. Мы не знаем, сколько памяти выделить, а сколько потребуется мы узнаем только по окончанию этапа ввода. Опишем решение задачи блок-схемой:

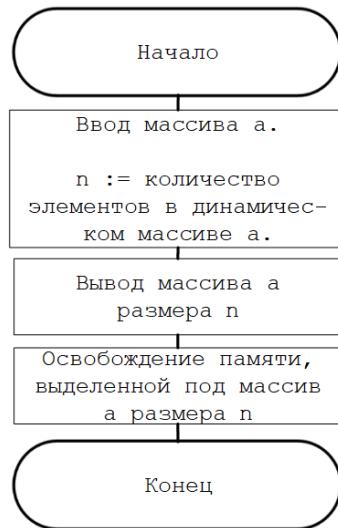


Рис. 10.17: Алгоритм в укрупненных блоках

Больший интерес для нас будет представлять функция ввода и механизм выделения памяти. Неправильным будет вызывать `realloc` при вводе элемента, например так:

---

```

void inputArray(int **a, size_t *size) {
    *a = malloc(sizeof(int));

    int i = 0;
    while (1) {
        int value;
        scanf("%d", &value);

        if (value == 0)
            break;

        *a = realloc(*a, sizeof(int) * (i + 1));
        (*a)[i++] = value;
    }

    *size = i;
}
  
```

---

Каждый раз будет происходить перевыделение памяти. Лучше сделать так:

---

**Листинг 38** Ввод элементов массива до первого нуля
 

---

```

#include <stdio.h>
#include <malloc.h>

// функция принимает указатель на указатель, так как указатель а функции main
// должен быть изменен после работы функции. По окончанию работы там должен
// храниться адрес выделенного участка памяти.
void inputArray(int **a, size_t *size) {
    // выделяем память под элемент
    int maxSize = 1;
    *a = malloc(sizeof(int) * maxSize);

    int i = 0;
    while (1) {
        // считываем значение
        int value;
        scanf("%d", &value);

        if (value == 0)
            break;

        // если выделенная память закончилась, выполняем перераспределение
        if (i == maxSize) {
            maxSize *= 2;
            *a = realloc(*a, sizeof(int) * maxSize);
        }
        // сохраняем значение в массиве
        (*a)[i++] = value;
    }

    // уменьшаем количество выделенной памяти
    *a = realloc(*a, sizeof(int) * i);
    *size = i;
}

void outputArray(int *a, const size_t size) {
    for (int i = 0; i < size; ++i)
        printf("%d ", a[i]);
    printf("\n");
}

int main() {
    size_t n;
    int *a;
    inputArray(&a, &n);

    outputArray(a, n);
    free(a);

    return 0;
}
  
```

---

## 10.4 \* Передача функций, как параметров

Опишем две функции, каждая из которых принимает один параметр типа `int` и возвращает значение 0 или 1 в зависимости от переданного значения:

---

```
int isEven(int x) {
    return x % 2 == 0;
}

int isNegative(int x) {
    return x < 0;
}
```

---

Идентификаторы `isEven` и `isNegative` - имена функций, которые подобно переменным имеют свой адрес в памяти:

---

```
printf("%p\n", isEven);      // \%p - спецификатор для вывода адреса
printf("%p\n", isNegative);
```

---

На моей машине получены адреса: 004015C0, 004015D3. Для вызова функции мы используем оператор вызова функции `()`. Если переменная хранит адрес функции, и мы применим оператор вызова, можно вызвать функцию.

Переменные, которые хранят адреса являются указателями. Объявление указателя на функцию:

---

```
// указатель на функцию, которая принимает один параметр
// типа int и возвращает значение типа int
int (*f)(int);

// указатель на функцию, которая принимает два параметра
// и возвращает значение типа float
float (*g)(float, int);
```

---

Переменным-указателям можно присваивать адреса функций:

---

```
#include <stdio.h>

int isEven(int x) {
    return x % 2 == 0;
}

int isNegative(int x) {
    return x < 0;
}

int main() {
    const int x = 10;

    int (*f)(int) = isEven;
    printf("%d\n", f(x));           // выведет 1 в обоих случаях
    printf("%d\n", isEven(x));     // варианты работают одинаково
```

---

```

    f = isNegative;
    printf("%d\n", f(x));           // выведет 0 в обоих случаях
    printf("%d\n", isNegative(x)); // варианты работают одинаково

    return 0;
}

```

---

Опишем возможную проблему при разработке приложения. Рассмотрим задачу поиска количества отрицательных чисел. Её можно было реализовать посредством функции `countNegative`:

```

int countNegative(const int *a, const size_t size) {
    int negativeCount = 0;
    for (size_t i = 0; i < size; i++)
        negativeCount += a[i] < 0;

    return negativeCount;
}

```

---

Позже была поставлена задача считать четные:

```

int countEven(const int *a, const size_t size) {
    int evenCount = 0;
    for (size_t i = 0; i < size; i++)
        negativeCount += a[i] % 2 == 0;

    return evenCount;
}

```

---

Если посмотреть на написанный код, можно заметить дублирование. Изменяется только условие, по которому мы увеличиваем значение на 1.

Решение состоит в следующем: раз можно создать указатель на функцию, то можно адрес функции передавать в другую функцию:

```

int isEven(int x) {
    return x % 2 == 0;
}

int isNegative(int x) {
    return x < 0;
}

// последний параметр функции - указатель на функцию
int countIf(const int *a, const size_t size, int (*f)(int)) {
    int counter = 0;
    for (size_t i = 0; i < size; i++)
        if (f(a[i]))
            counter++;

    return counter;
}

```

```

int main() {
    int a[5] = {1, 2, 3, 4, -5};

    // передаём адреса функций в функцию countIf
    printf("evenCount = %d\n", countIf(a, 5, isEven));
    printf("negativeCount = %d\n", countIf(a, 5, isNegative));

    return 0;
}

```

Можно писать и другие алгоритмы обобщенно. Например, однопроходный алгоритм удаления:

```

#include <stdio.h>

// функции-предикаты
int isEven(int x) {
    return x % 2 == 0;
}

int isNegative(int x) {
    return x < 0;
}

void deleteIf(int *a, size_t *n, int (*deletePredicate)(int)) {
    size_t iRead = 0;
    while (iRead < *n && !deletePredicate(a[iRead]))
        iRead++;

    size_t iWrite = iRead;
    while (iRead < *n) {
        if (!deletePredicate(a[iRead])) {
            a[iWrite] = a[iRead];
            iWrite++;
        }
        iRead++;
    }

    *n = iWrite;
}

void outputArray(const int *a, const size_t size) {
    for (size_t i = 0; i < size; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main() {
    int a[5] = {1, 2, 3, 4, -5};
    size_t n = 5;

    printf("Before: ");

```

```

    outputArray(a, n);

    deleteIf(a, &n, isNegative);

    printf("After: ");
    outputArray(a, n); // 1, 2, 3, 4

    int b[5] = {1, 2, 3, 4, -5};
    size_t m = 5;

    printf("Before: ");
    outputArray(b, m);

    deleteIf(b, &m, isEven);

    printf("After: ");
    outputArray(b, m); // 1 3 -5

    return 0;
}

```

---

Раз можно создать указатель на функцию, то можно создать и массив таких указателей:

```

int (*fs[2])(int) = {
    isEven,
    isNegative,
};

for (int i = 0; i < 2; i++)
    printf("%d\n", fs[i](10));

```

---

В качестве примера задачи, в которой передаётся функция в функцию рассмотрим вычисление значения определенного интеграла:

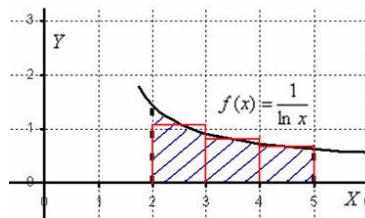
### Вычисление значения определенного интеграла

Необходимо вычислить значение определенного интеграла:

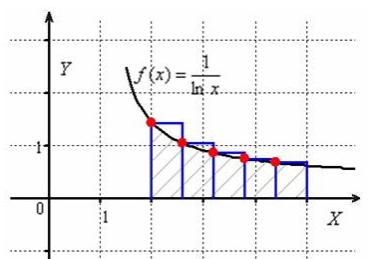
$$s = \int_a^b f(x)dx$$

Определенный интеграл от неотрицательной функции  $y = f(x)$  с геометрической точки зрения равен площади криволинейной трапеции, ограниченной сверху графиком функции  $y = f(x)$ , слева и справа – отрезками прямых  $x = a$  и  $x = b$ , снизу – отрезком оси  $Ox$ .

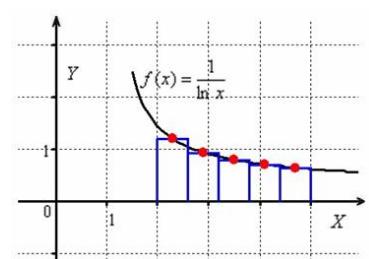
Одним из способов приближенного вычисления площади под графиком является метод прямоугольников. Отрезок интегрирования разбивается на несколько частей и строится ступенчатая фигура, которая по площади близка к искомой площади:



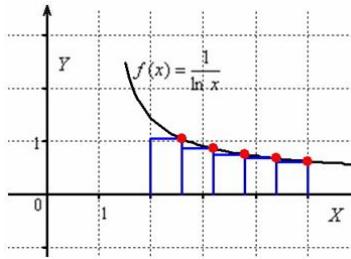
Выделяют метод левых средних и правых прямоугольников суть которых интуитивно понятна из рисунков:



(a) Левых прямоугольников



(b) Средних прямоугольников



(c) Правых прямоугольников

Рис. 10.18: Метод прямоугольника

Опишем способ вычисления площади прямоугольника в зависимости от выбранного способа. Обратите внимание на использование перечисления. Если константы связаны как-то между собой, рекомендуется собирать их значения в перечисления:

```
#include <stdio.h>
#include <stdlib.h>

// enum - перечисление - позволяет задать набор
// связанных между собой именованных констант
// обявление перечисления:
enum typeOfMethods {
    LEFT = 0,
    CENTRAL = 1,
    RIGHT = 2,
};

// f - указатель на функцию, которая возвращает значение функции f(x) в точке x
// где x имеет тип double и функция возвращает значение типа double
// n - количество прямоугольников
double getS(double a, double b, int n, double (*f)(double), enum typeOfMethods
→ method) {
    double delta = (b - a) / n;
```

```

double heightPoint;
switch (method) {
    case LEFT:
        heightPoint = a;
        break;
    case CENTRAL:
        heightPoint = (a + delta) / 2;
        break;
    case RIGHT:
        heightPoint = a + delta;
        break;
    default:
        // выводим в поток ошибок сообщение
        // об ошибке и заканчиваем работу программы
        fprintf(stderr, "incorrect typeOfMethod");
        exit(1);
}

double S = 0;
for (int i = 0; i < n; i++) {
    S += f(heightPoint)*delta;
    heightPoint += delta;
}
return S;
}

```

---

Для теста методов возьмём функцию  $f(x) = x^2$ :

```

double getX2(double x) {
    return x*x;
}

```

---

И при необходимости вычислить приближенное значение определенного интеграла тем или иным способом достаточно осуществить любой из вызовов:

```

printf("%f\n", getS(0, 2, 1000, getX2, LEFT)); // 2.662668
printf("%f\n", getS(0, 2, 1000, getX2, CENTRAL)); // 2.666666
printf("%f\n", getS(0, 2, 1000, getX2, RIGHT)); // 2.670668

```

---

Значение определенного интеграла, найденного аналитически:

$$s = \int_0^2 x^2 dx = \frac{8}{3} = 2.(6)$$

Мы не будем вдаваться в анализ методов, только отметим, что передача функции в функцию – довольно сильный приём, который позволяет писать алгоритмы обобщенно. Его не стоит недооценивать.

## 10.5 \* Массив префиксных сумм

**Массивом префиксных сумм**  $a$  называется такой массив  $b$ , что

$$b[0] = 0$$

$$b[1] = a[0]$$

$$b[2] = a[0] + a[1]$$

$$b[3] = a[0] + a[1] + a[2]$$

$$b[n+1] = a[0] + \dots + a[n] = \sum_{i=0}^n a[i]$$

Пример:

	0	1	2	3	4	5	6	
a	1	3	5	4	2	4		
b	0	1	4	9	13	15	19	

	0	1	2	3	4	5	6	
a	1	3	5	4	2	4		
b	0	1	4	9	13	15	19	

Рис. 10.19: Пример массива и префиксной суммы массива.  $i$ -ый элемент массива  $b$  определяет сумму элементов подмассива  $a[0...i - 1]$

При создании массива  $b$  особое внимание следует уделить вопросу его потенциального переполнения. Рассмотрим два соседних элемента массива  $b$ :

$$b[i] = a[0] + \dots + a[i - 1]$$

$$b[i + 1] = a[0] + \dots + a[i - 1] + a[i]$$

Несложно заметить, что  $b[i + 1]$  может быть выражен через предыдущий:

$$b[i + 1] = \left[ a[0] + \dots + a[i - 1] \right] + a[i] = b[i] + a[i]$$

При наличии массива префиксных сумм можно быстро выполнять запросы на поиск суммы на полуинтервале. Пусть дан полуинтервал  $[l, r)$  на котором нужно найти сумму. Несложно показать, что сумма на полуинтервале может быть выражена как разность двух элементов префиксного массива:

$$\sum_{i=l}^{r-1} a[i] = \left[ a[0] + \dots + a[r - 1] \right] - \left[ a[0] + \dots + a[l - 1] \right] = b[r] - b[l] \quad (10.1)$$

Например, найдём сумму на отрезке  $[2, 4]$  для массива  $a$ :

	0	1	2	3	4	5	6	
a	1	3	5	4	2	4		
b	0	1	4	9	13	15	19	

Преобразуем отрезок в полуинтервал:  $[2, 4] \rightarrow [2, 5)$ . Подставим в формулу 10.1 и получим значение суммы 11.

Мы можем вычислить сумму на отрезке, так как она выражается в виде разности двух других сумм. Префиксные массивы можно использовать для всех случаев, в которых имеется обратная операция. Для сложения – это вычитание, для исключающего или – исключающее или:

$$\bigoplus_{i=l}^{r-1} a[i] = \left[ a[0] \oplus \dots \oplus a[r-1] \right] \oplus \left[ a[0] \oplus \dots \oplus a[l-1] \right] = b[r] \oplus b[l]$$

### 10.5.1 Поиск количества подотрезков с нулевой суммой

#### Поиск количества подотрезков с нулевой суммой

С клавиатуры вводится массив из  $n$  чисел. Необходимо найти количество подотрезков с нулевой суммой.

Сложность по времени:  $O(n \log n)$ .

Если решать задачу в лоб, организовать расчеты можно так:

1. Находить суммы на подотрезках.
2. Увеличивать количество подотрезков на 1, если найдена нулевая сумма.

Код:

```
#include <stdio.h>

int main() {
    int a[] = {-1, 1, -1, 0, 0, 1};
    int n = sizeof(a) / sizeof(int);

    int count = 0;
    for (int i = 0; i < n; i++) {
        long long s = 0;
        for (int j = i; j < n; j++) {
            s += a[j];
            count += s == 0;
        }
    }

    printf("%d", count);

    return 0;
}
```

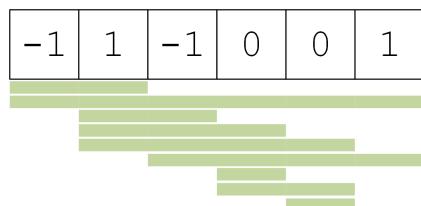


Рис. 10.20: Все нулевые подмассивы в порядке их нахождения алгоритмом.

С другой стороны можно найти префиксный массив сумм. Если в нём будут содержаться одинаковые значения, значит сумма на соответствующем подотрезке равна нулю:

	0	1	2	3	4	5	6
a	-1	1	-1	0	0	1	
b	0	-1	0	-1	-1	-1	0

Рис. 10.21: Префиксный массив сумм.

Осталось найти количество равных пар. Данная задача может решиться за  $O(n \log n)$ . После сортировки массива легко можно найти сколько раз встречался тот или иной элемент. В примере значение 0 встретилось 3 раза, значение -1 – 4 раза. Пусть  $n_x$  – количество раз, которое встречался элемент  $x$  в массиве  $a$ . Тогда количество пар:

$$n = n_{pairs_0} + n_{pairs_{-1}} = \frac{n_0(n_0 - 1)}{2} + \frac{n_{-1}(n_{-1} - 1)}{2} = 3 + 6 = 9$$

Пусть  $X$  – множество значений массива префиксных сумм, тогда количество подмассивов с нулевой суммой:

$$n = \sum_{x \in X} \frac{n_x(n_x - 1)}{2}$$

## 10.6 \* Разностные массивы

**Разностным массивом** массива  $b$  называется массив  $a$ , определяющийся следующим образом:

$$a[0] = b[1] - b[0]$$

$$a[1] = b[2] - b[1]$$

...

$$a[n - 1] = b[n] - b[n - 1]$$

Если для массива  $a$  найти префиксный массив сумм, и для результата найти разностный массив, получится массив  $a$ :

	0	1	2	3	4	5	6
a	1	3	5	4	2	4	
prefSum(a)	0	1	4	9	13	15	19
diffArr(prefSum(a))	1	3	5	4	2	4	

Рис. 10.22: Разностный массив от массива префиксных сумм  $a$  равен исходному массиву  $a$ .

По самому разностному массиву невозможно восстановить исходный массив. Если дополнить исходный массив нулям перед получением разностного, такое становится возможным:

	0	1	2	3	4	5	6
a	1	3	5	4	2	4	
a'	0	1	3	5	4	2	4
diffArr(a')	1	2	2	-1	-2	2	
prefSum(diffArr(a')) = a'	0	1	3	5	4	2	4
a	1	3	5	4	2	4	

Рис. 10.23: Добавление к массиву значения 0, перед преобразованием к разностному массиву позволяет восстановить исходный массив.

С другой стороны, разностный массив можно строить иначе, например так:

$$a[0] = b[0]$$

$$a[1] = b[1] - b[0]$$

...

$$a[n-1] = b[n-1] - b[n-2]$$

А префиксный по нему как:

$$b[0] = a[0]$$

$$b[1] = a[0] + a[1] = b[0] + a[1]$$

$$b[2] = a[0] + a[1] + a[2] = b[1] + a[2]$$

...

$$b[n-1] = a[0] + \dots + a[n-1] = b[n-2] + a[n-1]$$

Такой подход удобен, когда мы хотим получать разностные массивы, оперировать ими, и возвращать исходные.

### 10.6.1 Добавление на отрезке

Разностные массивы позволяют прибавлять значение на отрезке. Рассмотрим как изменится разностный массив, если прибавить на отрезке некоторое значение:

a	1	3	5	4	2	4	
diffArr(a')	1	2	2	-1	-2	2	

a	1	3	5+x	4+x	2+x	4	
diffArr(a')	1	2	2+x	-1	-2	2-x	

Рис. 10.24: Изменение разностного массива при добавлении значения  $x$  на отрезке.

В примере выше для отрезка с левой границей  $l = 2$  и правой границей  $r = 4$  добавляется значение 2. На разностном массиве произошли изменения для индексов  $l$  и  $r+1$  на  $+2$ .

Возможен случай, когда правая граница добавления - конец массива. Тогда изменения произойдут лишь на одной части разностного массива:

a	1	3	5	4	2	4	
diffArr(a')	1	2	2	-1	-2	2	

a	1	3	5+x	4+x	2+x	4+x	
diffArr(a')	1	2	2+x	-1	-2	2	

Рис. 10.25: Изменение разностного массива при добавлении значения для случая когда правая граница – конец массива.

### 10.6.2 Добавление арифметической прогрессии на отрезке

При добавлении арифметической прогрессии получаем следующее изменение на разностном массиве:

a	1	3	$5+x$	$4+2x$	$2+3x$	4	1
diffArr(a')	1	2	$2+x$	$-1+x$	$-2+x$	$2-3x$	$-3$
add(1, r, x)							
add(r+1, r+1, -(r - 1 + 1)*x)							

Рис. 10.26: Добавление арифметической прогрессии.

При помощи разностного массива от разностного массива можем произвести изменения разностного массива на двух отрезках: с  $l$  до  $r$  на  $+x$  и с  $r + 1$  до  $r + 1$  на  $-(r - l + 1)x$ .

## 10.7 Решения задач на одномерные массивы с использованием принципа пошаговой детализации

Рассмотрим последовательность рассуждений, которых стоит придерживаться при решении задач на одномерные массивы. Первая задача будет оформлена полностью. Во всех последующих задачах повторяющиеся функции будут опущены.

### 10.7.1 Сортировка подпоследовательности до первого вхождения нуля

#### Сортировка подпоследовательности до первого вхождения нуля.

Вводится массив целых чисел ( $N$  – константа). Отсортировать по неубыванию элементы до первого вхождения нуля. Если нуля нет, оставить массив без изменения.

##### 1. Опишем тестовые данные:<sup>8</sup>

Входные данные	Выходные данные
$N = 5$	5 4 3 2 1
5 4 3 2 1	
$N = 10$	1 2 3 0 3 2 1 0 3 2
3 2 1 0 3 2 1 0 3 2	
$N = 10$	1 2 3 4 5 7 0 1 2 3
1 3 2 4 5 7 0 1 2 3	

##### 2. Выполним выделение подзадач:<sup>9</sup>

1. Ввод массива.
2. Поиск позиции первого вхождения нуля.
3. Сортировка элементов подмассива.
  - (а) Обмен двух значений.
4. Вывод массива.

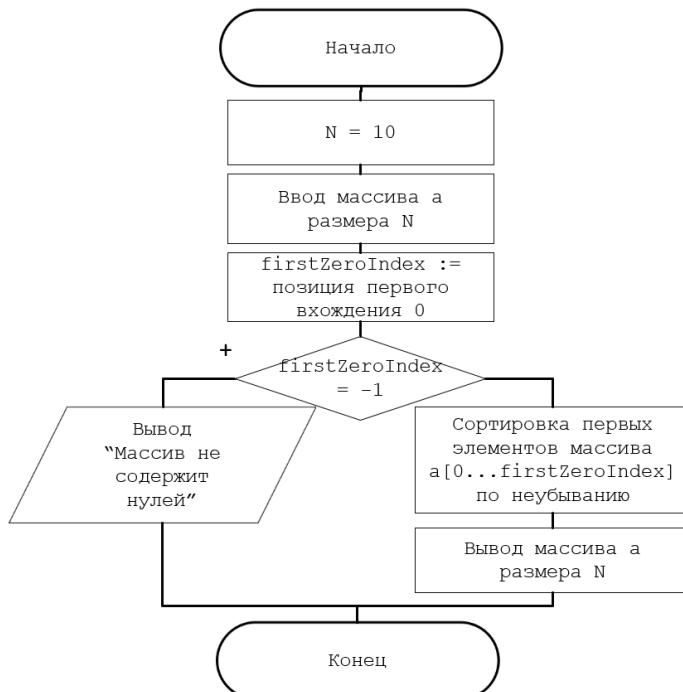
<sup>8</sup>Относительно оформления была принята следующая мера: описание спецификаций функций были упразднены из требований к решениям лабораторных работ. Ранее они содержали информацию о заголовке функции и её назначении. Прикладывалась блок-схема. Но

- Заголовок функции может быть обнаружен в коде.
- Назначение пишется в виде комментария перед функцией.
- В наше время существуют автоматизированные системы составления документации на основе заголовка функции и комментария перед ней.

Количество решаемых задач стало больше, важно расставить акценты на процессе кодирования, а не оформлении отчета.

<sup>9</sup>Подзадачи описываются без привязки к переменным. Просто подумайте, какие действия вам придётся осуществить при решении задачи.

3. Блок-схема в укрупненных блоках в терминах выделенных подзадач<sup>10</sup>:



#### 4. Текст программы:

---

```

#include <stdio.h>

// определяем именованную константу, которая отвечает за размер массива
#define N 10

// ввод элементов массива а размера n
void inputArray(int *a, const size_t size) {
    for (size_t i = 0; i < size; i++)
        scanf("%d", &a[i]);
}

// вывод элементов массива а размера n
void outputArray(const int *a, const size_t size) {
    for (size_t i = 0; i < size; i++)
        printf("%d ", a[i]);
    printf("\n");
}

// обмен значений двух переменных по адресам a и b
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
  
```

---

<sup>10</sup> Сформулированные на прошлом этапе подзадачи должны найти своё отражение в блок-схеме. Блок-схема должна исключать любую неоднозначность понимания. Она обязана быть описана таким образом, чтобы было возможным восстановить условие задачи.

```

// сортировка элементов массива a размера n
void bubbleSort(int *a, const size_t size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = size - 1; j > i; j--) {
            if (a[j - 1] > a[j]) {
                swap(a[j - 1], a[j]);
            }
        }
    }
}

// возвращает позицию первого элемента со значением value в массиве a размера
// n,
// если такой имеется, иначе -- -1.
int linearFind(const int *a, const size_t n, const int value) {
    size_t i = 0;
    while (i < n && a[i] != value)
        i++;
    if (i == n)
        i = -1;
    return i;
}

int main() {
    int a[N];
    inputArray(a, N);

    int firstZeroIndex = linearFind(a, N, 0);
    if (firstZeroIndex == -1)
        printf("No zeros");
    else {
        bubbleSort(a, firstZeroIndex);
        outputArray(a, N);
    }

    return 0;
}

```

### 10.7.2 Получение упорядоченной последовательности из неуникальных элементов

Получение упорядоченной последовательности из неуникальных элементов.

Вводится массив целых чисел ( $N$  – константа). Получить упорядоченную по возрастанию последовательность из чисел, которые встречаются в данной не менее двух раз. Вспомогательный массив использовать запрещено.

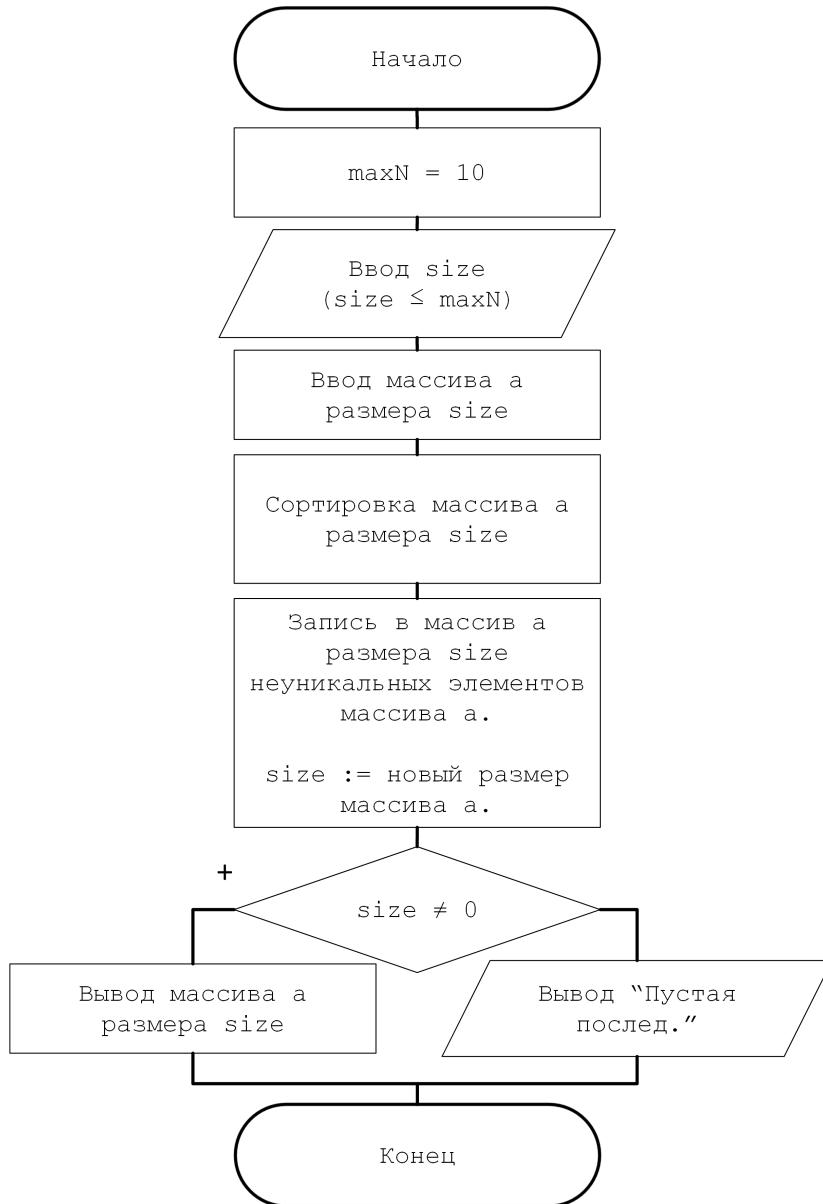
**1. Опишем тестовые данные:**

Входные данные	Выходные данные
$N = 5$	3 4
4 4 3 3 3	
$N = 10$	1 4 5
1 1 2 4 4 3 5 5 6 7	
$N = 5$	Последовательность пуста
1 2 3 4 5	

**2. Выполним выделение подзадач:**

1. Ввод массива.
2. Сортировка элементов массива.
  - (а) Обмен двух значений.
3. Запись в массив неуникальных элементов массива.
4. Вывод массива.

**3. Блок-схема в укрупненных блоках в терминах выделенных подзадач**



#### 4. Текст программы:

---

```

#include <stdio.h>

// определяем именованную константу
// которая определяет максимальный размер массива
#define N 10

// записывает по адресу a неуникальные элементы массива по адресу
// a размера size. Записывает в size количество неуникальных элементов.
void saveDuplicate(int *a, size_t *size) {
    int lastSavedValue = a[0] - 1;
    int iRec = 0;
    for (size_t i = 1; i < *size; i++) {
        if (a[i] == a[i - 1] && a[i] != lastSavedValue) {
            a[iRec++] = a[i];
            lastSavedValue = a[i];
        }
    }
}
  
```

```

    *size = iRec;
}

int main() {
    int a[N];
    size_t size;
    scanf("%u", &size);

    inputArray(a, size);

    selectionSort(a, size);
    saveDuplicate(a, &size);

    if (size != 0)
        outputArray(a, size);
    else
        printf("Empty sequence");

    return 0;
}

```

---

### 10.7.3 Сортировка с шагом

#### Сортировка с шагом

Дана последовательность целых чисел. Упорядочить члены, стоящие на четных местах по невозрастанию, а на нечетных – по неубыванию.

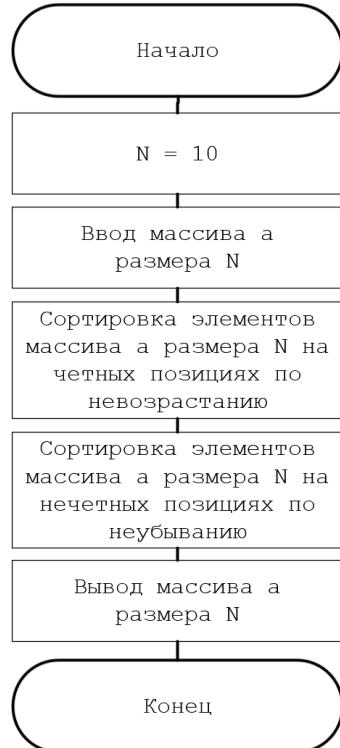
#### 1. Опишем тестовые данные:

Входные данные	Выходные данные
$N = 5$ 5 4 3 2 1	5 2 3 4 1
$N = 10$ 3 2 1 0 3 2 1 0 3 2	3 0 3 0 3 2 1 2 1 2

#### 2. Выполним выделение подзадач:

1. Ввод массива.
2. Сортировка элементов массива на четных позициях по невозрастанию.
  - (а) Обмен двух значений.
3. Сортировка элементов массива на нечетных позициях по неубыванию.
  - (а) Обмен двух значений.
4. Вывод массива.

### 3. Блок-схема в укрупненных блоках в терминах выделенных подзадач



#### 4. Текст программы:

```

#include <stdio.h>

// определяем именованную константу, которая отвечает за размер массива
#define N 10

// сортировка элементов массива a размера size с шагом step по неубыванию
void selectionSortInc(int *a, const size_t size, int step) {
    for (size_t i = 0; i < size - 1; i += step) {
        int iMin = i;
        int minValue = a[iMin];
        for (size_t j = i + step; j < size; j += step) {
            if (a[j] < minValue) {
                iMin = j;
                minValue = a[iMin];
            }
        }
        swap(&a[i], &a[iMin]);
    }
}

// сортировка элементов массива a размера size с шагом step по невозрастанию
void selectionSortDec(int *a, const size_t size, int step) {
    for (size_t i = 0; i < size - 1; i += step) {
        int iMax = i;
        int maxValue = a[iMax];
        for (size_t j = i + step; j < size; j += step) {
            if (a[j] > maxValue) {
  
```

```

        iMax = j;
        maxValue = a[iMax];
    }
}
swap(&a[i], &a[iMax]);
}

int main() {
    int a[N];
    inputArray(a, N);

    selectionSortDec(a, N, 2);
    // +1 к указателю - возвращает указатель, который указывает не на
    // нулевую ячейку массива, а на первую
    selectionSortInc(a + 1, N - 1, 2);

    outputArray(a, N);

    return 0;
}

```

---

Опишем некоторые моменты, касаемые реализации:

- Функция `selectionSortInc(int *a, const size_t size, int step)` помимо массива получает значение шага `step`. Предположим, что возникнет потребность сортировки не с шагом 2, а каким-то другим значением. Она может быть использована.
- Стоит обратить внимание на вызов функции `selectionSortInc` (строка 63). Сама функция сортирует с шагом 2 элементы, стоящие на позициях 0, 2, 4, 6.... Но мы бы хотели сортировать элементы на позициях 1, 3, 5, 7.... Можно принять такое решение: в функцию мы передаём адрес не нулевого элемента, а адрес первого, и тогда сортировка решает поставленную подзадачу (но надо учсть, что и область памяти на единицу меньше, рисунок 10.27):

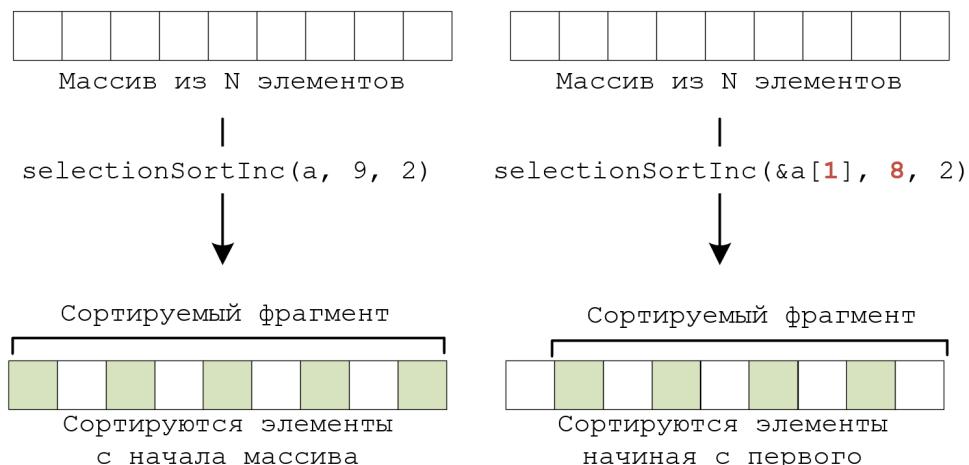


Рис. 10.27: Подбор аргументов для функции `selectionSortInc`

- `selectionSortInc(&a[1], N - 1, 2)`
- `selectionSortInc(a + 1, N - 1, 2)`

работают одинаково.

#### 10.7.4 Подсчёт количества вхождений элементов

##### Подсчёт количества вхождений элементов

Дана целочисленная последовательность размера  $n$  ( $n$  вводится с клавиатуры). Определить количество вхождений каждого числа в последовательность.

Опишем решение данной задачи с использованием динамических массивов. Идея решения состоит в следующем: отсортируем исходный массив и вычислим количество раз, которое встречался тот или иной элемент:

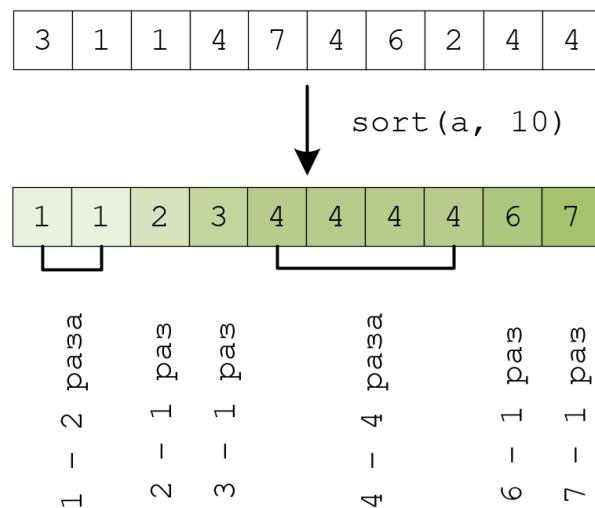


Рис. 10.28: После сортировки исходного массива довольно легко выполнить подсчёт.

##### 1. Опишем тестовые данные:

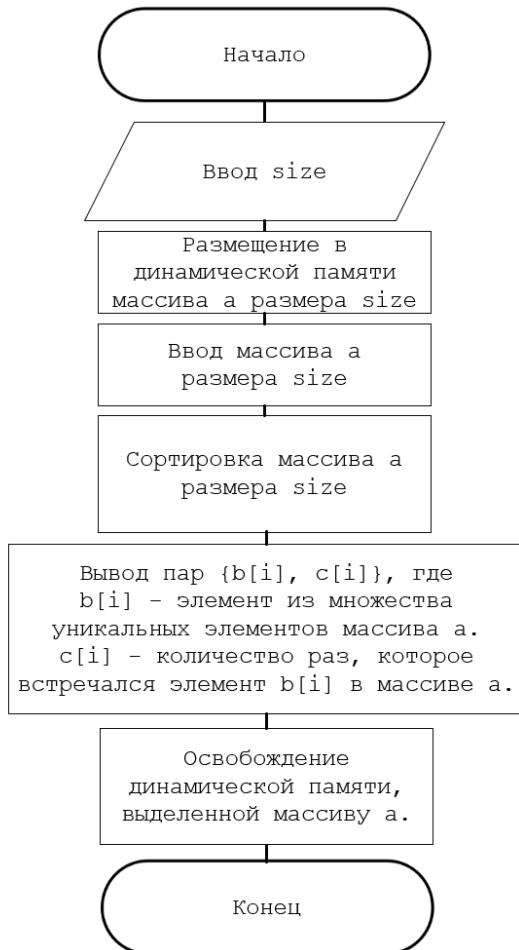
Входные данные	Выходные данные
$n = 5$	3 : 3
4 4 3 3 3	4 : 2
$n = 10$	1 : 2
1 1 2 4 4 3 5 5 6 7	2 : 1 3 : 1 4 : 2 5 : 2 6 : 1 7 : 1

##### 2. Выполним выделение подзадач:

1. Размещение в динамической памяти массива размера  $size$ .
2. Сортировка элементов массива.

- (а) Обмен двух значений.
3. Вывод пар  $\{b[i], c[i]\}$ , где  $b[i]$  - элемент из множества уникальных элементов массива  $a$ .  $c[i]$  - количество раз, которое встречался элемент  $b[i]$  в массиве  $a$ .
  4. Освобождение динамической памяти, выделенной массиву  $a$ .

### 3. Блок-схема в укрупненных блоках в терминах выделенных подзадач



#### 4. Текст программы:

```

#include <stdio.h>
#include <malloc.h>

// для каждого значения из множества элементов отсортированного массива a
// размера size выводит количество раз, сколько встречалось данное значение в
// массиве a
void outputCount(const int *a, const size_t size) {
    int i = 0;
    while (i < size) {
        int countingValue = a[i];
        int count = 0;
        while (countingValue == a[i] && i < size) {
            count++;
            i++;
        }
    }
}
  
```

```

        printf("%d: %d\n", countingValue, count);
    }

}

int main() {
    size_t size;
    scanf("%u", &size);

    int *a = (int*)malloc(sizeof(int) * size);
    inputArray(a, size);

    selectionSort(a, size);
    outputCount(a, size);

    free(a);

    return 0;
}

```

---

### 10.7.5 Сортировка точек по удаленности от другой точки

#### Сортировка точек по удаленности от другой точки

Упорядочить последовательность точек на числовой оси по неубыванию их расстояний до данной точки  $x$ .

Основная проблема данной задачи – сортировка значений. В качестве ключа сортировки будут выступать расстояние от точки  $a[i]$  до  $x$ . Пусть  $a$  – массив точек,  $distances$  – массив расстояний элементов массива  $a$  до точки  $x$ .

Предположим, имеются 4 точки:  $a = \{3, 4, 8, 10\}$ . В качестве  $x$  выступает значение 7. Вычислив массив расстояний получим:

$$distances = \{4, 3, 1, 3\}$$

Будем выполнять сортировку массива  $distances$ , но процесс обмена будем производить и на массиве  $a$  параллельно. Предположим, что используется сортировка выбором:

```

void selectionSort(int *values, int *keys, const size_t size) {
    for (int i = 0; i < size - 1; i++) {
        int minPos = i;
        for (int j = i + 1; j < size; j++)
            if (keys[j] < keys[minPos])
                minPos = j;
        swap(&keys[i], &keys[minPos]);
        swap(&values[i], &values[minPos]);
    }
}

```

---

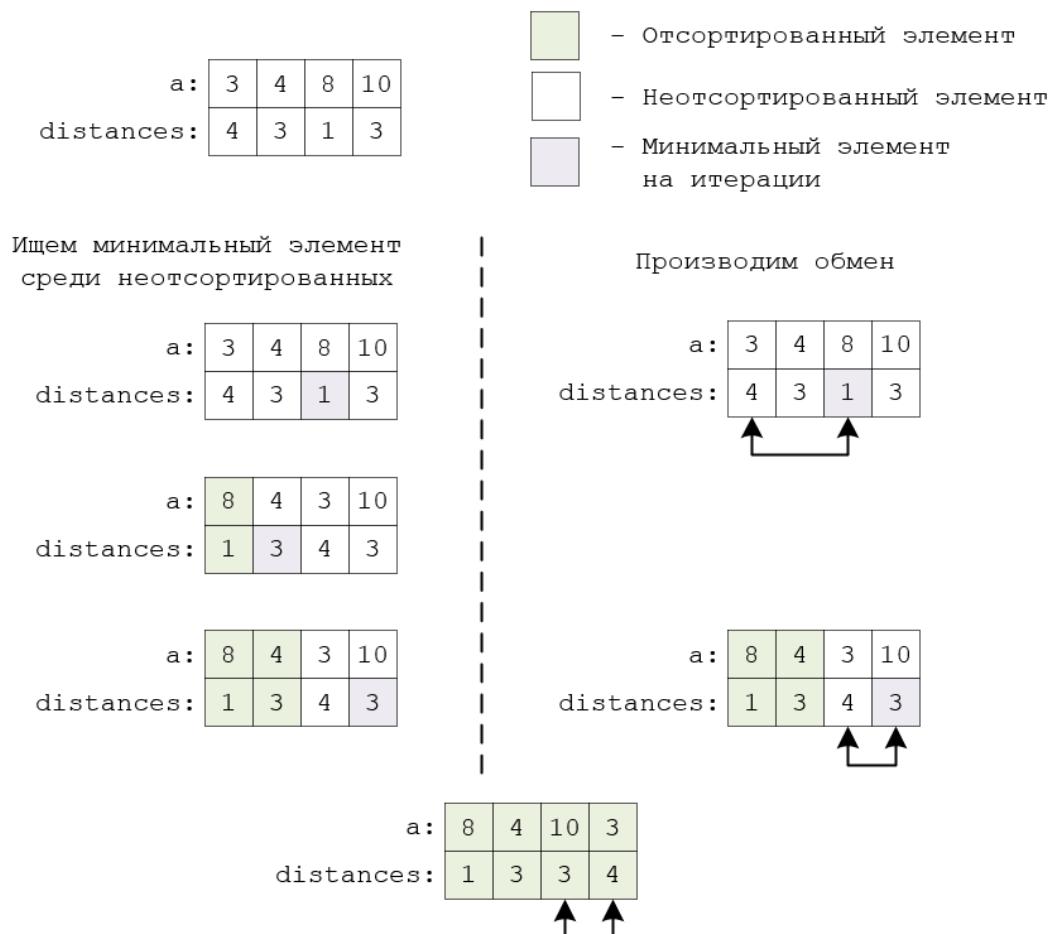


Рис. 10.29: Процесс сортировки.

### 10.7.6 Вывод чисел, непринадлежащих последовательности

#### Вывод чисел, непринадлежащих последовательности.

Даны целые числа  $a_1, a_2, \dots, a_n$ . Пусть  $\max$  – максимальное из этих чисел, а  $\min$  – минимальное. Получить в порядке возрастания все целые числа, заключенные в интервале между  $\min$  и  $\max$  данных чисел и не принадлежащие данной последовательности.

И снова решение достигается при применении сортировки. Используя отсортированный массив легко производить такой вывод, поочередно просматривая значения от  $\min + 1$  до  $\max - 1$ :

```
// выводим множество элементов, заключенных между min(a) и max(a) массива a
// размера n и непринадлежащих данному массиву
void outputSet(const int *a, const int n) {
    int min = a[0];
    int max = a[n - 1];
    for (int v = min + 1, i = 1; v <= max; v++)
        if (v == a[i])
            i++;
        else
            printf("%d ", v);
}
```

### 10.7.7 Сортировка элементов массива, выбранных по критерию

#### Сортировка элементов массива, выбранных по критерию.

Упорядочить по невозрастанию только четные числа данной целочисленной последовательности, нечетные оставить без изменения. Указание: можно использовать вспомогательный массив с номерами четных элементов.

Выполним создание вспомогательного массива, который будет содержать индексы элементов, удовлетворяющих условию:

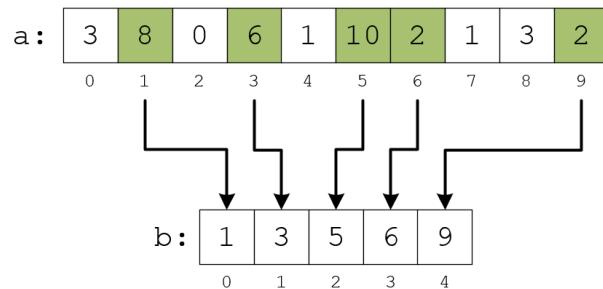
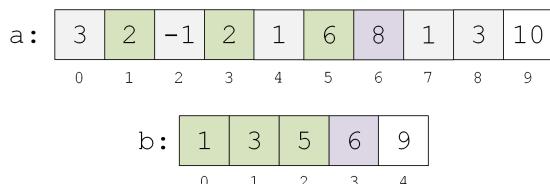
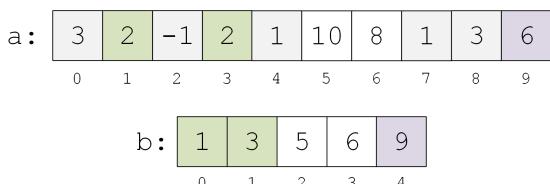
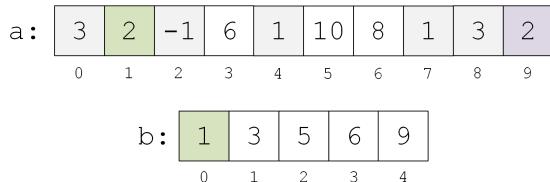
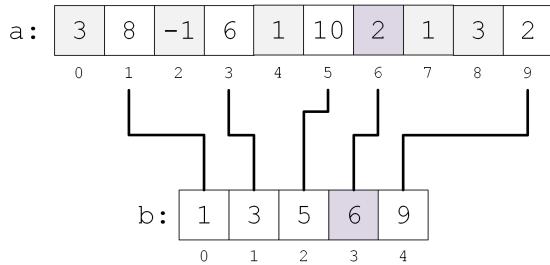


Рис. 10.30: Создание вспомогательного массива.

Осталось выполнить сортировку. Но мы будем взаимодействовать с массивом  $a$  через массив  $b$ . Опишем сортировку выбором для такого случая:

На  $i$ -ом шаге находим минимальное  $a[b[j]]$  для  $j = [0 \dots \text{size}(b[i])]$



Производим обмен  $a[b[i]]$  с  $a[b[j]]$

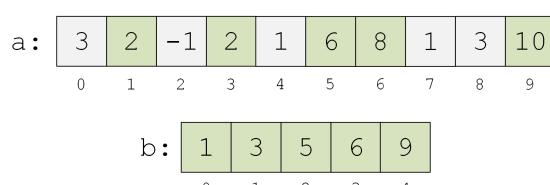
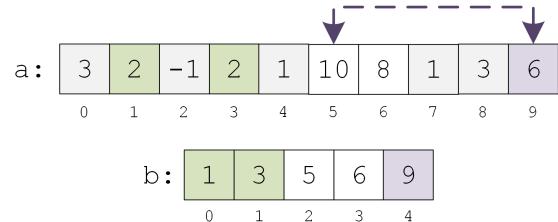
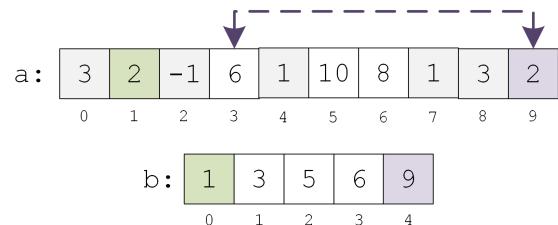
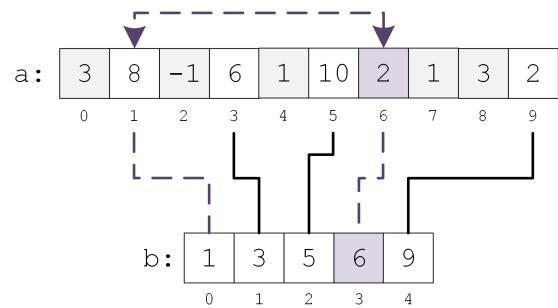


Рис. 10.31: Сортировка массива.

```
#include <stdio.h>

#define N 10

// ввод массива a размера size
void inputArray(int *a, size_t size) {
    for (size_t i = 0; i < size; i++)
        scanf("%d", &a[i]);
}
```

```

// обмен значений двух переменных по адресам a и b
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// сортировка выбором элементов массива {a[indexes[i]]} где i = [0...indexesSize)
// по неубыванию
void selectionSort(int *a, const int *indexes, const size_t indexesSize) {
    for (int i = 0; i < indexesSize - 1; i++) {
        int minPos = indexes[i];
        for (int j = i + 1; j < indexesSize; j++)
            if (a[indexes[j]] < a[minPos])
                minPos = indexes[j];
        swap(&a[indexes[i]], &a[minPos]);
    }
}

// запись по адресу indexes массива индексов четных элементов массива a размера
// size;
// indexesSize := количество четных элементов массива a размера size
void getArrayOfIndexesOfEven(const int *a, size_t size, int *indexes, size_t
    *indexesSize) {
    size_t iWrite = 0;
    for (int i = 0; i < size; i++)
        if (a[i] % 2 == 0)
            indexes[iWrite++] = i;
    *indexesSize = iWrite;
}

// вывод массива a размера size
void outputArray(const int *a, size_t size) {
    for (size_t i = 0; i < size; i++)
        printf("%d ", a[i]);
}

int main() {
    int a[N];
    inputArray(a, N);

    int indexes[N];
    size_t indexesSize;
    getArrayOfIndexesOfEven(a, N, indexes, &indexesSize);
    selectionSort(a, indexes, indexesSize);

    outputArray(a, N);

    return 0;
}

```

## Резюме

- Для работы с массивами рекомендуется выделять функции.
- В функции можно передать адрес не только начала массива, а какой-то другой его части.
- Грамотно подобранные аргументы позволяют избежать дублирования кода при реализации схожих функций.
- Для использования динамических структур требуется осуществлять работу с динамической памятью (кучей). В сравнении со стеком куча работает медленнее, поскольку переменные разбросаны по памяти, а не находятся на верхушке стека.
- Динамические массивы используются, если количество требуемой памяти заранее неизвестно.
- За работу с динамической памятью в языке С отвечают следующие функции, определенные в заголовочном файле `malloc.h`:
  - `malloc`
  - `calloc`
  - `realloc`
  - `free`
- Адрес функции можно передавать в другую функцию, тем самым избежать дублирования кода.

## Термины и определения

- **Куча** – это хранилище памяти, также расположеннное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных.
- **Неопределенное поведение** – это когда результат компиляции и исполнения программы непредсказуем.
- **Оперативное запоминающее устройство (ОЗУ)** – это компонент, который позволяет компьютеру кратковременно хранить данные и осуществлять быстрый доступ к ним.
- **Стек** – это область оперативной памяти, которая создаётся для каждого потока.

## Контрольные вопросы

1. Как описываются массивы в С?
2. Как осуществляется ввод и вывод одномерных массивов?

3. Какими способами может быть осуществлен поиск элемента в упорядоченном и неупорядоченном массиве?
4. Опишите алгоритм однопроходного алгоритма удаления из последовательности членов, удовлетворяющих заданному условию.
5. Как передавать в функцию адрес некоторой отличной от начальной части массива? Приведите пример.
6. В чем заключается принцип работы стека?
7. В чем заключается принцип работы кучи?
8. В каких случаях используются динамические массивы?
9. Какие функции отвечают в С за работу с динамической памятью? Как работают данные функции?
10. Передача функции в функцию. Для чего необходим данный прием? Приведите примеры.
11. Опишите последовательность решения задачи на одномерные массивы с использованием принципа пошаговой детализации.

# Глава 11

## Поиск

### 11.1 Быстрый линейный поиск

Версия линейного поиска в неотсортированном массиве описана на странице 233. Подход к его ускорению заключается в следующем:

- В массив дописывается элемент равный искомому. Таким образом  $x$  будет обнаружен в массиве рано или поздно. В связи с этим можно не проверять индекс на выход за пределы массива. Количество операций сравнения уменьшается примерно в 2 раза.
- Если позиция первого вхождения  $x$  равна размеру массива – элемент отсутствовал, иначе – найдена позиция.

В силу того, что порядок функции временной сложности в худшем случае и простой и ускоренной версии совпадают, ускорение кажется сомнительным. Если поиск нужно осуществлять часто, имеются более продвинутые техники.

### 11.2 Линейный поиск в отсортированном массиве

Пусть имеется отсортированный массив  $a$  размера  $n$ . Необходимо выполнить поиск некоторого элемента со значением  $x$ . Можно было бы написать следующий алгоритм линейного поиска:

---

#### Листинг 40 Линейный поиск в отсортированном массиве

---

```
size_t linearSearch(const int *a, const size_t n, int x) {
    size_t i = 0;
    while (i < n && a[i] < x)
        i++;
    return i < n;
}
```

---

Выполним поиск значения  $x = 6$  в массиве  $a$ :

---

**Листинг 39** Линейный поиск в неотсортированном массиве
 

---

```
#include <stdio.h>

size_t linearFindFast(int *a, const size_t n, const int x) {
    // на позицию i записываем искомое значение
    a[n] = x;

    size_t i = 0;
    while (a[i] != x)
        i++;

    return i;
}

int main() {
    int a[10];
    // для возможности осуществить быстрый линейный поиск
    // должно иметься место для служебного элемента
    inputArray(a, 9);
    linearFindFast(a, 9, 5);

    return 0;
}
```

---

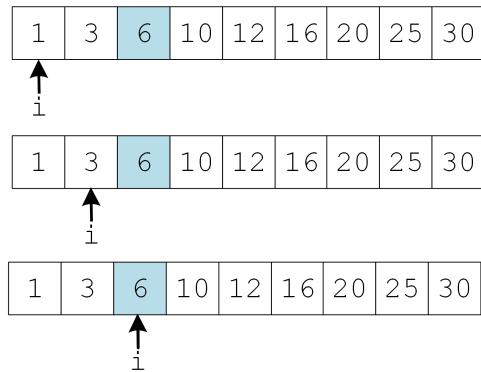


Рис. 11.1: Вариация линейного поиска для случая, когда элемент в массиве есть

Как только находим значение, которое больше или равно *x* или значение *i* становится равным *n*, поиск прекращается. В его окончании проверяется условие

---

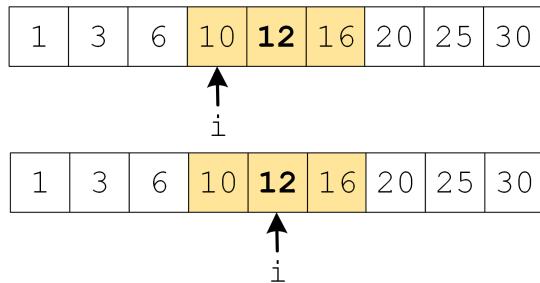
```
i < n && a[i] == x
```

---

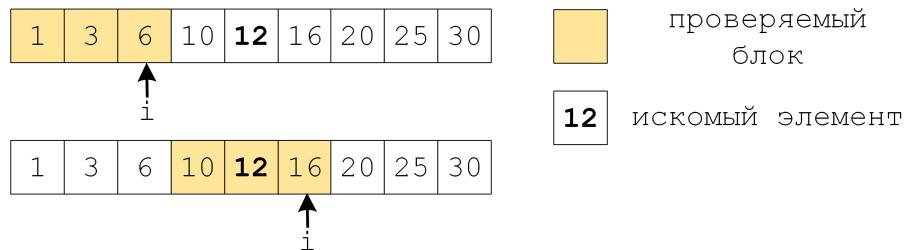
на основании которого можно понять, имеется ли элемент *x* в массиве.

### 11.3 Блочный поиск. *Sqrt*-декомпозиция

Немного усовершенствуем прошлую идею. Теперь будем проверять значения не последовательно, а с каким-то шагом. Например, возьмём шаг равный трём.



1. Определимся с блоком:



2. Проанализируем элементы блока. Если там содержится элемент – возвращаем его позицию (в противном случае, элемент не найден):

Можно сказать, что линейный поиск в отсортированном массиве, является частным случаем блочного (с размером блока 1).

Определим оптимальный размер блока. Рассмотрим худший случай, когда элемент находится в последнем блоке на последнем месте. Будем исходить из предположения, что размер массива делится на размер блока нацело.

Пусть  $block\_size$  – размер блока. Для определения блока, в котором потенциально находится элемент, в худшем случае требуется  $\frac{n}{block\_size}$  сравнений. Более того, потребуется  $block\_size$  сравнений на проверку блока. Количество сравнений – это некоторая функция, зависящая от  $block\_size$ . Общее количество сравнений:

$$n_{\text{compares}}(block\_size) = \frac{n}{block\_size} + block\_size$$

Вычислим такое значение  $block\_size$ , при котором функция  $n_{\text{compares}}(block\_size)$  принимает минимальное значение. Найдём производную:

$$n'_{\text{compares}}(block\_size) = \left( \frac{n}{block\_size} \right)' + block\_size' = -\frac{n}{block\_size^2} + 1$$

Приравняем производную к нулю:

$$n'_{\text{compares}}(block\_size) = -\frac{n}{block\_size^2} + 1 = 0 \Rightarrow block\_size = \sqrt{n}$$

Следовательно, при размноге блока  $\sqrt{n}$  достигается минимальное количество сравнений для худшего случая.

**Sqrt-декомпозиция** – это метод, или структура данных, позволяющая в режиме онлайн проводить различные операции на отрезке за  $O(\sqrt{n})$  (поиск минимума, максимума, суммы, **хог** и любых других ассоциативных операций). Существуют более эффективные методики для решения таких задач (например, деревья Фенвики, деревья отрезков, *Segment Tree Beats* и др., работающие за  $O(\log n)$ ).

Рассмотрим произвольный массив размера  $n = 14$ . Разделим его на блоки размера  $s = \sqrt{n} = 4$ . Для каждого блока вычислим значение функции:

$$b[k] = \sum_{i=k*s}^{\min(n-1, (k+1)*s-1)} a[i]$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	6	5	10	12	7	12	52	5	4	12	3	8	25	45
b	5				5				3		25			

Рассмотрим, как будет происходить вычисление минимума на отрезке. Пусть задан отрезок  $[l..r]$ . Вычислить значение минимума можно на нём, если руководствоваться следующими соображениями:

- Если в отрезок входит блок целиком, достаточно посмотреть на значение в блоке.
- Если вхождение блока в отрезок частично, вычислить операцию придётся на фрагменте блока.

В качестве отрезка возьмём следующий:  $a[2..9]$ . Тогда вместо того, чтобы искать минимум из 8 элементов, достаточно просмотреть 5 значений:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	6	5	10	12	7	12	52	5	4	12	3	8	25	45
b	5				5				3		25			

или более формально:

$$\min_{i=l}^r a[i] = \min \left( \min_{i=l}^{(k+1)*s-1} a[i], \min_{i=k+1}^{p-1} b[i], \min_{i=p*s}^r a[i] \right)$$

где  $k$  – номер первого блока,  $p$  – номер последнего блока.

---

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int min(const int a, const int b) {
    return a < b ? a : b;
}

int getMin(const int *a, int left, int right) {
    int _min = INT_MAX;
    for (int i = left; i < right; i++)
        _min = min(_min, a[i]);
}
```

```

        return _min;
    }

void get.SqrtDecomposition(const int *a, const int n, int *b) {
    const int blockSize = ceil(sqrt(n));
    const int nBlocks = ceil(n / blockSize);
    int startBlockIndex = 0;
    for (int blockIndex = 0; blockIndex < nBlocks; blockIndex++) {
        const int nElementsInBlock = min(blockSize, n - startBlockIndex);
        b[blockIndex] = getMin(a, startBlockIndex,
                               startBlockIndex + nElementsInBlock);
        startBlockIndex += blockSize;
    }
}

int getMinRequest(const int *a, int n, const int *b,
                  const int left, const int right) {
    const int blockSize = ceil(sqrt(n));
    int _min = INT_MAX;
    const int leftBlockIndex = left / blockSize;
    const int rightBlockIndex = (right + 1) / blockSize;
    if (leftBlockIndex == rightBlockIndex)
        // элементы лежат в рамках одного блока (но не занимают весь блок)
        _min = min(_min, getMin(a, left, right + 1));
    else {
        // минимум до целых блоков
        _min = min(_min, getMin(a, left, leftBlockIndex * blockSize));
        // минимум в целых блоках
        _min = min(_min, getMin(b, leftBlockIndex, rightBlockIndex));
        // минимум после целых блоков
        _min = min(_min, getMin(a, rightBlockIndex * blockSize, right + 1));
    }
    return _min;
}

```

---

При обновлении значений в массиве  $a$  требуется проверить, не изменяются ли элементы массива  $b$ . Возможно потребуется дополнительный проход по блоку для обновления элемента массива  $b$ :

```

void setValue(int *a, int n, int *b, int i, int x) {
    const int blockSize = ceil(sqrt(n));
    const int blockNumber = i / blockSize;
    if (x > a[i] && a[i] == b[blockNumber]) { // если изменяет потенциально
                                                // минимальный элемент блока
        a[i] = x;
        const int jStart = i / blockSize * blockSize;
        const int jEnd = min(jStart + blockSize, n);
        b[blockNumber] = getMin(a, jStart, jEnd);
    } else
        a[i] = x;
}

```

---

## 11.4 Бинарный поиск (вариация №1)

### Бинарный поиск

Имеется отсортированный по возрастанию массив размера  $n = 7$ :

$$a = \{2, 5, 8, 13, 21, 27, 35\}$$

Дано число  $x$ . Необходимо найти:  $i : a[i] = x$ . Если значение присутствует в массиве, функция должна вернуть индекс, иначе – значение -1.

Сложность по времени:  $O(\log n)$ .

Начнём со способа, как не нужно писать двоичный поиск: возьмём два числа  $left$  ( $left = 0$ ) и  $right$  ( $right = n - 1$ ) и будем предполагать, что искомый элемент лежит в подмассиве:

$$x \in a[left...right]$$

Ставим задачей постепенно уменьшать отрезок, но таким образом, чтобы предположение выполнялось.

Вычислим индекс<sup>1</sup>

$$middle = \left\lfloor \frac{left + right}{2} \right\rfloor = left + \left\lfloor \frac{right - left}{2} \right\rfloor$$

Если  $a[middle] > x$  – тогда нужно сдвинуть правую границу  $right = middle - 1$ . Если  $a[middle] < x$  – сдвигаем левую границу  $left = middle + 1$ . Если  $a[middle] = x$  – найдено число и можно возвращать ответ. Поиск необходимо осуществлять до тех пор, пока  $left \leq right$ .

Работа алгоритма изображена на рисунках 11.2, 11.3. Реализация поиска на C<sup>2</sup>:

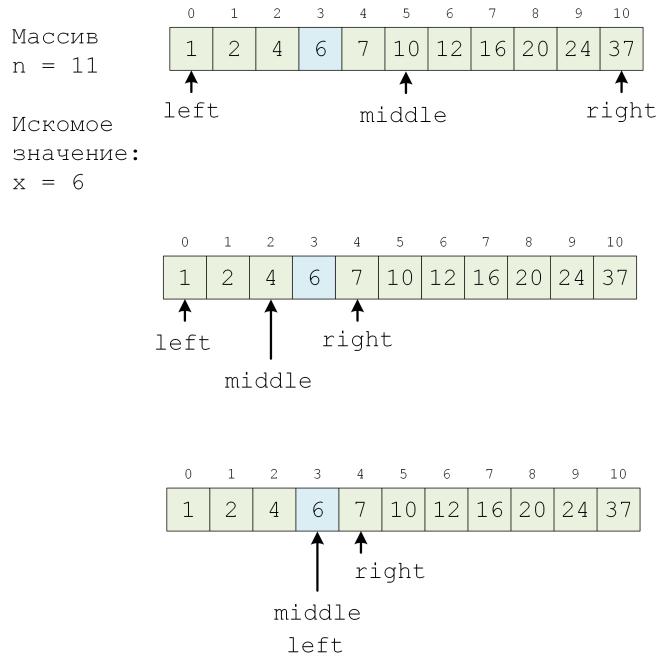
```
int binarySearch(const int *a, const int n, const int x) {
    int left = 0;
    int right = n - 1;
    while (left <= right) {
        int middle = (left + right) / 2;
        if (a[middle] < x)      // если 'истина', искомый элемент лежит правее
            left = middle + 1;
        else if (a[middle] > x) // если 'истина', искомый элемент лежит левее
            right = middle - 1;
        else
            return middle;
    }

    return -1;
}
```

Недостаток такого варианта: сложно искать более сложные случаи, речь о которых пойдёт далее.

<sup>1</sup> Второй способ вычисления индекса предостерегает от потенциального переполнения.

<sup>2</sup> В процессе написания кода, я часто рассматриваю массив из одного элемента, чтобы правильно указать условие в строке 4.



```
left = 0
right = n - 1 = 10
middle = (left + right) div 2
= 5

a[middle] > x -> right = middle - 1
= 4

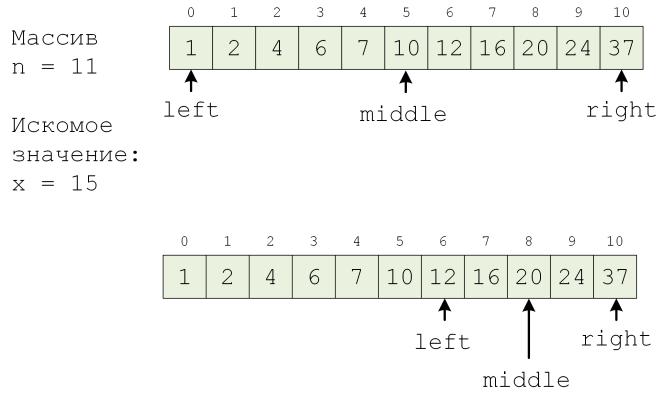
left = 0
right = 4
middle = (left + right) div 2
= 2

a[middle] < x -> left = middle + 1
= 3

left = 3
right = 4
middle = (left + right) div 2
= 3

a[middle] = x -> элемент найден
```

Рис. 11.2: Бинарный поиск (вариация №1) для случая, когда элемент имеется в массиве

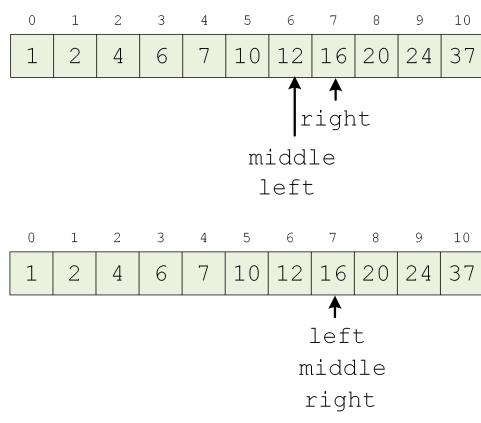


```
left = 0
right = n - 1 = 10
middle = (left + right) div 2
= 5

a[middle] < x -> left = middle + 1
= 6

left = 6
right = 10
middle = (left + right) div 2
= 8

a[middle] > x -> right = middle - 1
= 7
```



```
left = 6
right = 7
middle = (left + right) div 2
= 6

a[middle] < x -> left = middle + 1
= 7

left = 7
right = 7
middle = (left + right) div 2
= 7

a[middle] > x -> right = middle - 1
= 6
```

right < left -> конец алгоритма

Рис. 11.3: Бинарный поиск (вариация №1) для случая, когда элемента нет в массиве

## 11.5 Бинарный поиск (вариация №2)

Теперь будем осуществлять поиск из других соображений. Возьмём два числа  $left$  и  $right$  и будем так изменять правила, гарантировались следующие условия:

$$a[\text{left}] < x \quad \quad a[\text{right}] \geq x$$

который обеспечит нам поиск такого минимального  $i$ , что:

$\min i : a[i] \geq x$

Снова будем вычислять значение *middle*:

$$middle = \left\lfloor \frac{left + right}{2} \right\rfloor = left + \left\lfloor \frac{right - left}{2} \right\rfloor$$

Если  $a[middle] < x$  – тогда нужно сдвинуть левую границу в  $middle$ . Иначе в  $middle$  двигается правая граница. По окончанию поиска массив будет поделён на две части. В одной из них лежат элементы меньше  $x$ , в другой – большие или равны  $x$ . Если искомое значение найдено, то  $a[right] = x$ , если не найдено –  $a[right] \neq x$ .

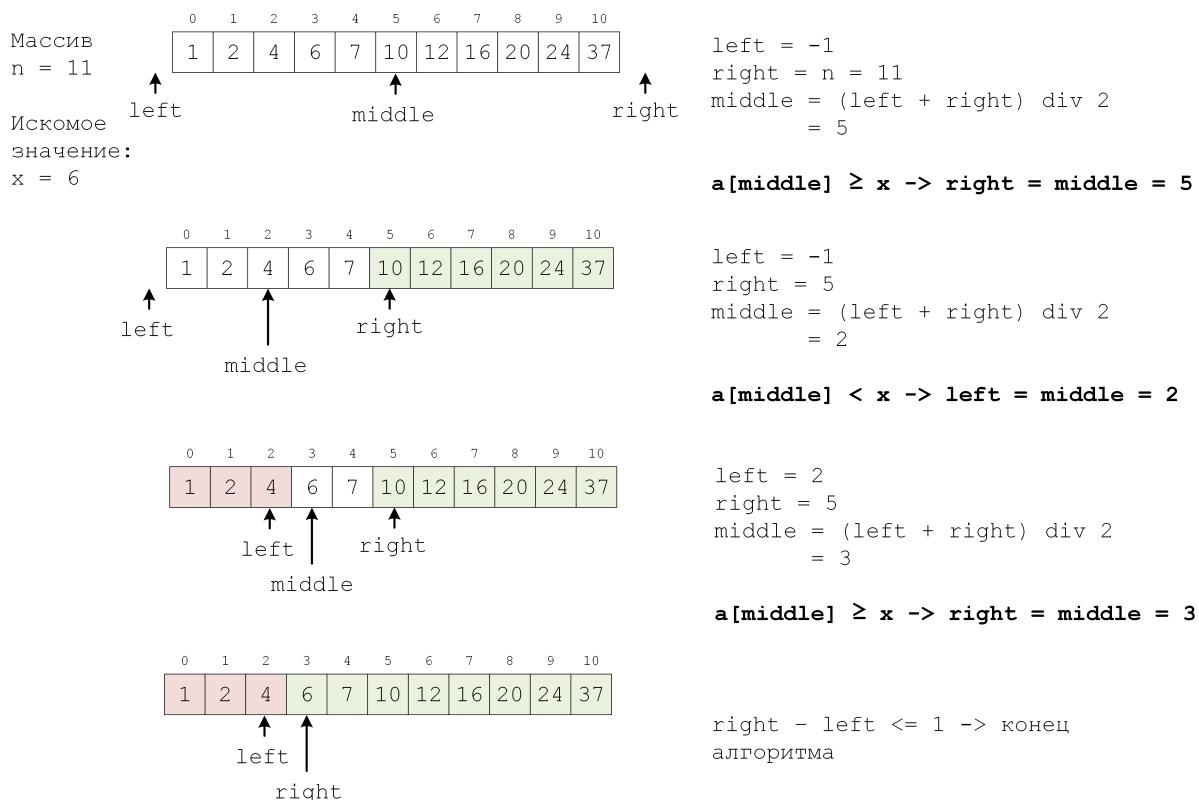


Рис. 11.4: Бинарный поиск (вариация №2) для случая, когда элемент имеется в массиве

Если бы выполнялся поиск значения 5 – конечные значения *left* и *right* совпадали бы с примером выше.

Рассмотрим случай, когда искомый элемент больше всех, которые имеются в массиве:

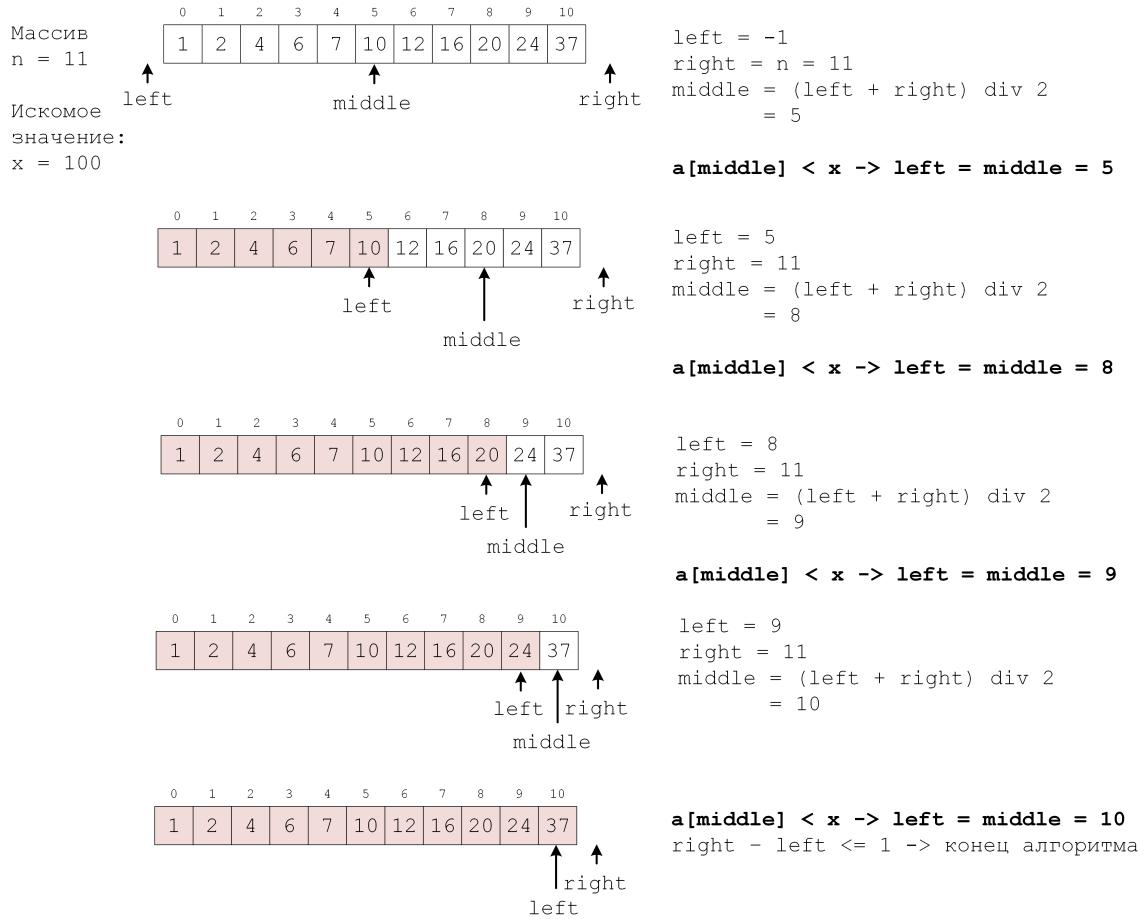


Рис. 11.5: Бинарный поиск (вариация №2) для случая, когда все элементы больше того, который ищем

Реализация поиска на C:

---

#### Листинг 41 Поиск позиции первого элемента большего или равного $x$

---

```
// возвращает позицию первого элемента большего или равного x
// (вернёт значение n если все элементы массива меньше x)
int binarySearchEqualOrMore(const int *a, const int n, const int x) {
    int left = -1;
    int right = n;
    while (right - left > 1) {
        int middle = (left + right) / 2;
        if (a[middle] < x)
            left = middle;
        else
            right = middle;
    }
    return right;
}
```

---

Для поиска

$$\max i : a[i] \leq x$$

можно воспользоваться следующим фрагментом:

---

**Листинг 42** Поиск позиции последнего элемента меньшего или равного  $x$

---

```
// возвращает позицию последнего элемента меньшего или равного x
// (вернёт -1, если все числа больше x)
int binarySearchLessOrEqual(const int *a, const int n, const int x) {
    int left = -1;
    int right = n;
    while (right - left > 1) {
        int middle = (left + right) / 2;
        if (a[middle] <= x)
            left = middle;
        else
            right = middle;
    }
    return left;
}
```

---

Если нам нужно точное совпадение, т. е. решить задачу:

$$\max i : a[i] = x$$

можно найти ответ для:

$$\max i : a[i] \leq x$$

а в конце проверить, является ли  $a[i] = x$ .

## 11.6 Бинарный поиск по критерию

Можно обобщить идею бинарного поиска:

**Бинарный поиск по критерию**

Пусть имеется некоторая последовательность  $a$  размера  $n$ , которая может быть разделена на две части посредством какого-то критерия. Причём граница разделения делит все числа одной группы от чисел другой группы.

Дан массив  $a$ , критерий – числа больше 20:

$$a = \{2, 5, 8, 13, 21, 27, 35\}$$

Необходимо найти первое число, которое бы удовлетворяло критерию.

Сложность по времени:  $O(\log n)$ .

Критерий может быть задан какой-то функцией, адрес которой передаётся в функцию поиска как фактический параметр.

Пусть  $left$  – наибольший индекс числа первой группы, а  $right$  – наименьший индекс числа второй группы. Поддерживая данную идею, можно прийти к следующей реализации:

---

**Листинг 43** Бинарный поиск по критерию

---

```
#include <stdio.h>

int isMore20(int x) {
    return x > 20;
}

// вернёт значение n, если все элементы не удовлетворяют критерию
int binarySearchCriteria(const int *a, const int n,
                         int (*fcriteria)(int)) {
    int left = -1;
    int right = n;
    while (right - left > 1) {
        int middle = (left + right) / 2;
        if (fcriteria(a[middle]))
            right = middle;
        else
            left = middle;
    }
    return right;
}

int main() {
    int a[] = {2, 5, 8, 13, 21, 27, 35};
    int n = 7;

    printf("%d", a[binarySearchCriteria(a, n, isMore20)]);
}


```

---

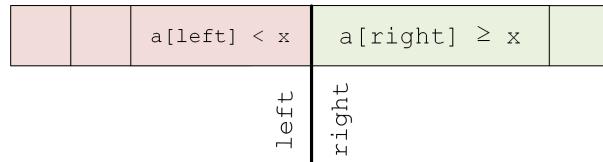
## 11.7 Решение задач на бинарный поиск

### 11.7.1 Поиск позиции первого значения равному $x$

#### Поиск позиции первого значения равному $x$

Дан отсортированный массив  $a$  размера  $n$ . Требуется написать функцию для поиска первого вхождения элемента со значением  $x$  или возвращающую размер массива, если элемент не найден.

Будем стремиться к следующему разбиению:



чтобы все элементы, имеющие индекс меньше или равный  $left$  были меньше  $x$ , а оставшиеся элементы были больше или равны  $x$ .

Реализация:

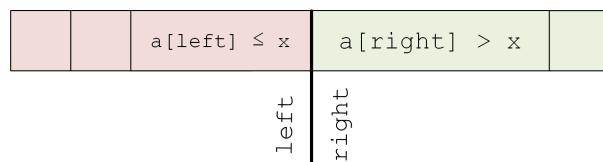
```
int binarySearchFirstEqual(const int * const a, const size_t n,
                           const int x) {
    int left = -1, right = n;
    while (right - left > 1) {
        int middle = left + (right - left) / 2;
        if (a[middle] < x)
            left = middle;
        else
            right = middle;
    }
    return right;
}
```

### 11.7.2 Поиск позиции последнего значения равному $x$

#### Поиск позиции последнего значения равному $x$

Дан отсортированный массив  $a$  размера  $n$ . Требуется написать функцию для поиска последнего вхождения элемента со значением  $x$  или возвращающую  $-1$ , если элемент не найден.

Стремимся к следующим значениям  $left$  и  $right$ :



Подход к реализации аналогичен:

```
int binarySearchLastEqual(const int * const a, const size_t n,
                           const int x) {
    int left = -1;
    int right = n;
    while (right - left > 1) {
        int middle = left + (right - left) / 2;
        if (a[middle] <= x)
            left = middle;
        else
            right = middle;
    }
    return (left != -1 && a[left] == x) ? left : -1;
}
```

### 11.7.3 Задача о верёвках

#### Задача о верёвках

Есть  $n$  ( $n \leq 10000$ ) веревочек, длины которых известны. Нужно нарезать из них  $k$  кусков одинаковой длины. Найдите максимальную длину кусков, которую можно получить.

Очевидно, что если мы будем нарезать короткие верёвочки, их будет много. Если длинные – их будет мало. Количество веревок может быть найдено по формуле:

$$n' = \sum_{i=1}^n \left\lfloor \frac{\text{length}_i}{k} \right\rfloor$$

Для подбора длины  $k$  воспользуемся бинарным поиском:

```
#include <stdio.h>
#include "libs/algorithms/array/array.h"

long long countRopes(int *ropeLengths, size_t n, double oneRopeLength) {
    long long count = 0;
    for (size_t i = 0; i < n; i++)
        count += ropeLengths[i] / oneRopeLength;
    return count;
}

int main() {
    int n, k;
    scanf("%d %d", &n, &k);

    int ropeLengths[10000];
    inputArray(ropeLengths, n);

    double left = 0;
    double right = 1e18;
    double eps = 1e-7;
```

```

        while (right - left > eps) {
            double middle = left + (right - left) / 2;
            if (countRopes(ropeLengths, n, middle) >= k)
                left = middle;
            else
                right = middle;
        }
        printf("%.6f", left);

        return 0;
    }
}

```

---

#### 11.7.4 Задача о ксероксах

##### Задача о ксероксах

Имеется документ, с которого нужно снять  $n$  копий. Дано два ксерокса, первый делает копию за время  $t_1$  секунд, второй делает копию за время  $t_2$  секунд. Снимать копию можно и с копии. Необходимо найти минимальное время, которое потребуется потратить для изготовления  $n$  экземпляров документа.

Прежде всего, найдём ксерокс, который работает быстрее. Пусть  $t_1 \leq t_2$ . Тогда количество копий  $n'$ , которое можно сделать за время  $t$  определяется по формуле:

$$n' = \left\lfloor \frac{t}{t_1} \right\rfloor + \left\lfloor \frac{t - t_1}{t_2} \right\rfloor$$

Осталось перебрать различные значения  $t$  при помощи бинарного поиска:

```

#include <stdio.h>

void order2(long long *a, long long *b) {
    if (*a > *b) {
        long long t = *a;
        *a = *b;
        *b = t;
    }
}

long long countCopies(long long firstXeroxTime,
                      long long secondXeroxTime,
                      long long sumTime) {
    order2(&firstXeroxTime, &secondXeroxTime);
    return sumTime / firstXeroxTime +
           (sumTime - firstXeroxTime) / secondXeroxTime;
}

int main() {
    long long n, x, y;
    scanf("%lld %lld %lld", &n, &x, &y);

    long long left = 0;

```

```

long long right = 1e18;
while (right - left > 1) {
    long long middle = left + (right - left) / 2;
    if (countCopies(x, y, middle) >= n)
        right = middle;
    else
        left = middle;
}

printf("%lld", right);

return 0;
}

```

---

### 11.7.5 Задача об уравнении

#### Задача об уравнении

Найдите такое число  $x$ , что  $x^2 + \sqrt{x} = C$  ( $\geq 1$ ).

Функция является монотонно возрастающей и все её значения положительны.

```

#include <stdio.h>
#include <math.h>

double f(double x) {
    return x*x + sqrt(x);
}

int main() {
    double C;
    scanf("%lf", &C);

    double left = 0;
    double right = 1e16;
    while (right - left > 1e-7) {
        double middle = left + (right - left) / 2;
        if (f(middle) > C)
            right = middle;
        else
            left = middle;
    }

    printf("%f", right);

    return 0;
}

```

---

## 11.7.6 Задача о сборе

### Задача о сборе

На числовой прямой живет  $n$  человек, каждый из которых находится в координате  $x_i$  и имеет скорость передвижения  $v_i$ . Все люди решили собраться в одной точке  $X$ . Необходимо найти минимальное время сбора.

Если  $t$  'хорошее', то за  $t$  секунд можно собраться в точке  $X$ , если  $t$  – плохое, собраться в точке  $X$  нельзя. Если  $t$  подходит, то подходят и все  $t' : t' > t$ .

Мысль следующая: для каждого человека посчитаем, в каком промежутке на числовой прямой он сможет оказаться, если будет двигаться со своей скоростью времени  $t$ . Если бы у жителей было бы бесконечное количество времени, они могли бы собраться в любой точке. Если время будет ограничено, в какой-то точке они смогут собраться с учетом ограничений, а в какой-то нет. На рисунке 11.6 представлены позиции, в которых могут оказаться жители через одну секунду движения. Отметим, что не имеется общей точки пересечения всех трёх интервалов, что означает, что за время  $t = 1$  собраться невозможно. Бинарным поиском можно достичь поиска оптимальной точки.

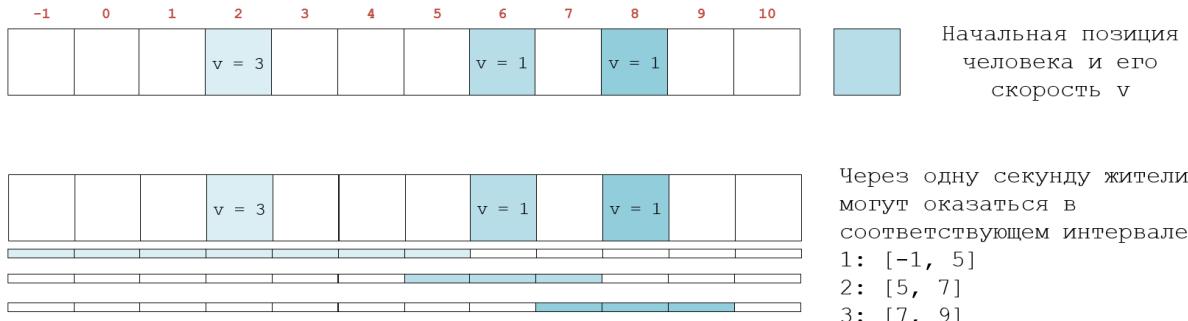


Рис. 11.6: Начальное расположение жителей и их возможное положение через секунду

```
#include <stdio.h>
#include <float.h>

double max(double a, double b) { return a > b ? a : b; }

double min(double a, double b) { return a < b ? a : b; }

int isPossible(int *speeds, int *positions, int nPeople, double time) {
    double left = -1e20;
    double right = 1e20;
    for (int peopleIndex = 0; peopleIndex < nPeople; peopleIndex++) {
        double possibleLeft = positions[peopleIndex]
            - speeds[peopleIndex]*time;
        double possibleRight = positions[peopleIndex]
            + speeds[peopleIndex]*time;
        left = max(left, possibleLeft);
        right = min(right, possibleRight);
    }
    return left <= right;
}
```

```

}

int main() {
    int n;
    scanf("%d", &n);

    int coordinates[n], speeds[n];
    for (int i = 0; i < n; i++)
        scanf("%d %d", &coordinates[i], &speeds[i]);

    double badTime = 0;
    double goodTime = 1e10;
    const double eps = 0.000001;
    while (goodTime - badTime > eps) {
        double middleTime = (badTime + goodTime) / 2;
        if (isPossible(speeds, coordinates, n, middleTime))
            goodTime = middleTime;
        else
            badTime = middleTime;
    }

    printf("%f", goodTime);

    return 0;
}

```

### 11.7.7 Задача о коровах и стойлах

#### Задача о коровах и стойлах

Имеется  $n$  стойл с координатами  $a_i$  и  $k$  коров ( $2 \leq k \leq n$ ). Необходимо расположить коров по одной в стойло таким образом, чтобы расстояние между ними было максимальным. В качестве ответа укажите максимальное возможное расстояние.

В качестве решения опишу функцию проверки, возможно ли получить расстояние `distance` при количестве коров `nCows` и `nCoordinates` стойлами с координатами `coordinates`.

```

int isPossible(const int *coordinates, int nCoordinates, int distance, int
→ nCows) {
    // загоняем корову в первое стойло
    int lastPosIndex = 0;
    int lastCoordinateWithCow = coordinates[lastPosIndex];
    nCows--;
    for (int coordinateIndex = 1;
         coordinateIndex < nCoordinates;
         coordinateIndex++) {
        int curCoordinate = coordinates[coordinateIndex];
        int currentDistance = curCoordinate - lastCoordinateWithCow;
        // если расстояние между текущим стойлом и стойлом с прошлой коровой

```

```
// стало не меньше проверяемого расстояния - загоняем корову в стойло
if (currentDistance >= distance) {
    lastPosIndex = coordinateIndex;
    lastCoordinateWithCow = coordinates[lastPosIndex];
    nCows--;
}
return nCows <= 0;
}
```

Возможно, что для проверяемого расстояния `distance` можно будет разместить большее количество коров, поэтому возвращаемое значение

```
return nCows <= 0;
```

### 11.7.8 Поиск максимального среднего арифметического на отрезке длины не менее $D$

**Поиск максимального среднего арифметического на отрезке длины не менее  $D$**

Имеется массив  $a$  размера  $n$ . Необходимо найти такой его отрезок из хотя бы  $D$  элементов, чтобы среднее арифметическое на отрезке было максимально.

Формально можно описать задачу так: пусть  $l$  и  $r - 1$  левая и правая граница искомого отрезка. Необходимо найти такие их значения, что:

$$\frac{\sum_{i=l}^{r-1} a_i}{r - l} \rightarrow \max$$

Пусть  $x'$  – предполагаемый ответ на задачу. Если бы мы могли узнать, является ли  $x'$  ответом на задачу, точное его значение можно найти через бинарный поиск.

Проверим, существует ли такие  $l$  и  $r - 1$  чтобы значение на подмассиве  $a[l..r - 1]$  было хотя бы  $x'$ :

$$\frac{\sum_{i=l}^{r-1} a_i}{r - l} \geq x'$$

Выполним ряд преобразований:

$$\frac{\sum_{i=l}^{r-1} a_i}{r - l} \geq x' \Leftrightarrow \sum_{i=l}^{r-1} a_i \geq x'(r - l) \Leftrightarrow \sum_{i=l}^{r-1} (a_i - x') \geq 0$$

Рассмотрим на реальных данных. Пусть  $D = 3$  и массив  $a$ :

0	1	2	3	4	5	6	7	8
a	6	7	5	3	2	6	4	1

Предположим, что проверяется гипотеза: существует такой отрезок, для которого  $x' = 5$ :

a	-	$x'$	0	1	2	3	4	5	6	7	8
			1	2	0	-2	-3	1	-1	-4	4

Чтобы быстро находить сумму на отрезке, воспользуемся префиксными суммами:

$$p_j = \sum_{i=0}^{j-1} (a_i - x')$$

p		0	1	2	3	4	5	6	7	8	9
		0	1	3	3	1	-2	-1	-2	-6	-2

С его помощью легко найти сумму на подмассиве:

$$\sum_{i=l}^{r-1} (a_i - x') = p[r] - p[l]$$

Таким образом, мы хотели бы понять, имеется ли такая пара  $l$  и  $r$ , что

$$\begin{cases} r - l \geq D \\ p[r] - p[l] \geq 0 \end{cases} \Leftrightarrow \begin{cases} l \leq r - D \\ p[l] \leq p[r] \end{cases} \quad (11.1)$$

Зафиксируем произвольное  $r$  и попытаемся определить, имеется ли хоть какое-нибудь  $l$  удовлетворяющее ограничениям выше. Пусть  $r = 9$ . Рассмотрим все  $a_i : l \leq r - D$ :

p		0	1	2	3	4	5	6	7	8	9
		0	1	3	3	1	-2	-1	-2	-6	-2

Элемент с индексом 5 соответствует заявленным ограничениям. Следовательно в исходном массиве имеется хотя бы одна подпоследовательность  $a[5..9-1]$  со средним арифметическим  $x' \geq 5$ :

a		0	1	2	3	4	5	6	7	8
		6	7	5	3	2	6	4	1	9

Единственная проблема – относительно высокие затраты по времени для работы с массивом  $p$  для поиска такой пары, которая удовлетворяла бы условиям системы 11.1. Перебор всех вариантов имеет сложность  $O(n^2)$ . Решить данную проблему можно с использованием массива префиксных минимумов:

$$m_i = \min(p[0..i])$$

Для нашего случая:

	0	1	2	3	4	5	6	7	8	9
p	0	1	3	3	1	-2	-1	-2	-6	-2
	0	1	2	3	4	5	6	7	8	9
m	0	0	0	0	0	-2	-2	-2	-6	-6

Имея такой массив, можно утверждать, что имеется пара  $(l, r)$ , удовлетворяющая ограничениям 11.1 если

$$m_{r-D} \leq p_r$$

---

// код представлен для случая, когда нумерация элементов массива ведётся с 1  
// в lBorder и rBorder записываются границы искомого интервала включительно.

```

double min(const double a, const double b) { return a < b ? a : b; }

int getBound(const int *a, const size_t n, double x, const int d,
             int *lBound, int *rBound) {
    double b[n];
    for (int i = 0; i < n; i++)
        b[i] = a[i] - x;

    double p[n + 1];
    p[0] = 0;
    for (int i = 1; i <= n; i++)
        p[i] = p[i - 1] + b[i - 1];

    double m[n + 1];
    m[0] = p[0];
    for (int i = 1; i <= n; i++)
        m[i] = min(m[i - 1], p[i]);

    for (int r = d; r <= n; r++)
        if (m[r - d] <= p[r]) {
            *rBound = r;
            // поиск левой границы
            int l = r - d;
            while (l >= 0 && m[l] <= m[r - d])
                l--;
            *lBound = l + 1;
            return 1;
        }
    return 0;
}

void getBorders(const int *a, const size_t n, const int d,
                int *lBorder, int *rBorder) {
    double minX = 0;
    double maxX = 1e9;
    while (maxX - minX > 0.000001) {
        double x = (minX + maxX) / 2;
        int r = -1;

```

```
int l = -1;
if (getBound(a, n, x, d, &l, &r))
    minX = x;
else
    maxX = x;
if (r != -1) {
    *rBorder = r;
    *lBorder = l;
}
}
```

---

## 11.8 Тернарный поиск

Пусть задана некоторая функция  $f(x)$ , которая вначале монотонно убывает, а потом монотонно возрастает на некотором интервале  $[l, r]$ . И нам нужно найти такой  $x$ , при котором  $f(x)$  минимальна. На каждой итерации будем делить отрезок от  $l$  до  $r$  на три равные части точками  $m_1$  и  $m_2$ :

$$m_1 = l + \frac{1}{3}(r - l) = \frac{2l - r}{3}$$

$$m_2 = l + \frac{2}{3}(r - l) = \frac{2r - l}{3}$$

В зависимости от значений  $m_1$  и  $m_2$  происходит изменение переменных  $l$  и  $r$ . Рассмотрим два случая. Первый вариант:

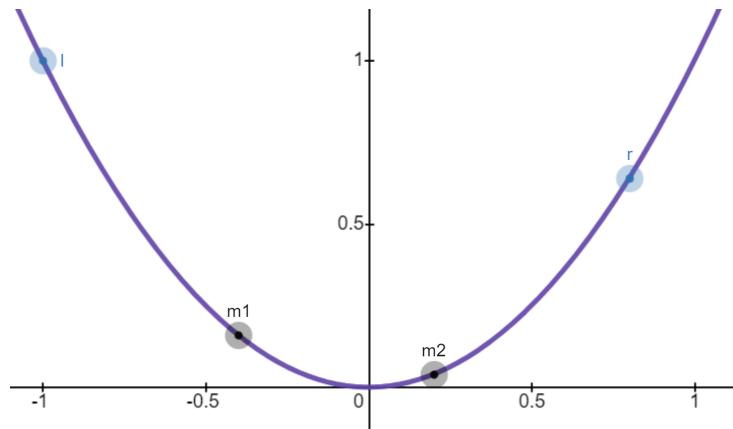


Рис. 11.7: Так как  $m_1 > m_2$ , искомая точка находится между  $m_1$  и  $r$ , поэтому на данной итерации  $l := m_1$

На следующей итерации для данного примера получим другую картину:

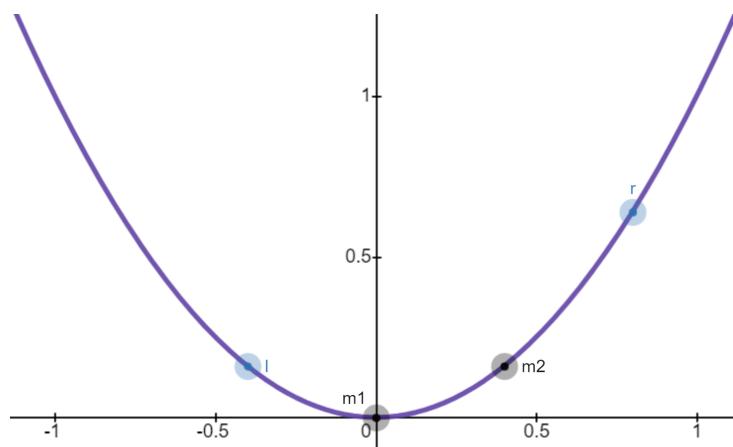


Рис. 11.8: Так как  $m_1 \leq m_2$ , искомая точка находится между  $l$  и  $m_2$ , поэтому на данной итерации  $r := m_2$

Опишем алгоритм тернарного поиска для нахождения минимума на отрезке:

---

#### Листинг 44 Тернарный поиск

---

```
double l = ...;           // левая граница интервала
double r = ...;           // правая граница интервала
double eps = 0.0000001    // погрешность измерения
while (r - l > eps) {
    double m1 = l + (r - l) / 3;
    double m2 = l + 2*(r - l) / 3;
    if (m1 > m2)
        l = m1;           // ситуация на рисунке \ref{fig:9}
    else
        r = m2;           // ситуация на рисунке \ref{fig:10}
}
```

---

Для ситуации, когда выполняется поиск максимума меняется лишь знак в 7 строке:

---

```
if (m1 < m2)
    l = m1;
else
    r = m2;
```

---

# Глава 12

## Сортировки

**Алгоритм сортировки** — это алгоритм для упорядочивания элементов в массиве. В случае, когда элемент в массиве имеет несколько полей, поле, служащее критерием порядка, называется **ключом** сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

### 12.1 Обезьяняя сортировка

Данная сортировка носит теоретический характер. *BogoSort*, также известный как сортировка перестановок, глупая сортировка, медленная сортировка, дробовик или обезьянья сортировка, является особенно неэффективным алгоритмом, основанным на парадигме генерации и тестирования. Алгоритм последовательно генерирует перестановки своих входных данных, пока не получит отсортированные.

Если *BogoSort* использовать для сортировки колоды карт, то сначала в алгоритме нужно проверить, лежат ли все карты по порядку, и если не лежат, то случайным образом перемешать её, проверить лежат ли теперь все карты по порядку, и повторять процесс, пока колода не будет отсортирована.

Алгоритм обезьяньей сортировки:

1. Генерация перестановки массива.
2. Если массив не отсортирован, вернуться к пункту 1.

Шаг 1	<table border="1"><tr><td>6</td><td>4</td><td>2</td><td>5</td><td>7</td><td>1</td><td>3</td></tr></table>	6	4	2	5	7	1	3
6	4	2	5	7	1	3		
Шаг 2	<table border="1"><tr><td>6</td><td>5</td><td>2</td><td>1</td><td>4</td><td>7</td><td>3</td></tr></table>	6	5	2	1	4	7	3
6	5	2	1	4	7	3		
Шаг 3	<table border="1"><tr><td>4</td><td>7</td><td>3</td><td>6</td><td>5</td><td>1</td><td>2</td></tr></table>	4	7	3	6	5	1	2
4	7	3	6	5	1	2		
Шаг 4	<table border="1"><tr><td>5</td><td>1</td><td>4</td><td>3</td><td>7</td><td>2</td><td>6</td></tr></table>	5	1	4	3	7	2	6
5	1	4	3	7	2	6		
	...							
Шаг N	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	1	2	3	4	5	6	7
1	2	3	4	5	6	7		

Рис. 12.1: Bogosort

Сложность по времени:  $O(n * n!)$ . Нижняя граница:  $\Omega(n)$ . Вам может повезти, и сразу элементы отсортируются так, как нужно. Но в среднем отсортированную

последовательность вы получите через  $n!$  перестановок. В прошлом примере такое можно получить за 5040 раз.

Запустив его на своей машине, я получил значение 5033.897984 генерации массивов в среднем.

## 12.2 Сортировка выбором

### Сортировка выбором

Дан массив  $a$  размера  $n$ . Необходимо упорядочить элементы массива по неубыванию сортировкой выбором.

Сложность по времени:  $O(n^2)$ .

Сортировка выбором является одним из простых алгоритмов сортировки. Её преимущество заключается небольшом количестве операций обмена. Массив разделяется на 2 части: отсортированную и неотсортированную. На  $i$ -ой итерации цикла происходит следующее:

- Выполняется поиск позиции  $\text{minPos}$  – позиции минимального элемента среди элементов в неотсортированной части (начиная с индекса  $i$ , заканчивая  $n - 1$ ).
- Элементы на позиции  $i$  и  $\text{minPos}$  обмениваются.

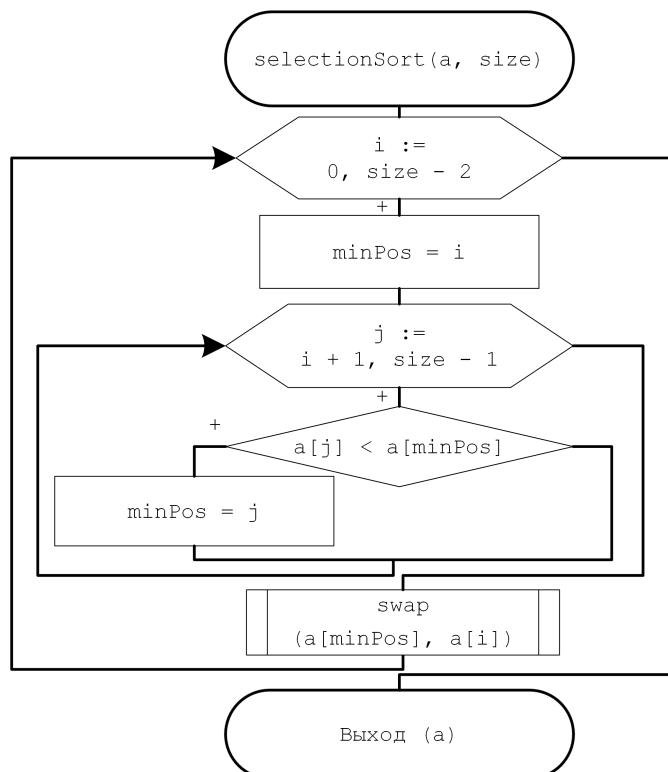


Рис. 12.2: Блок-схема сортировки выбором

---

**Листинг 45** Обезьянья сортировка
 

---

```

#include <stdio.h>
#include <stdlib.h>

// обмен элементов по адресу a и b
void swap(int *a, int *b) {
}

// проверка упорядоченности массива
int isSorted(const int *a, size_t size) {
    for (int i = 1; i < size; i++)
        if (a[i - 1] > a[i])
            return 0;
    return 1;
}

// перемешивание элементов массива
void shuffleArray(int *a, size_t size) {
    for (size_t i = 0; i < size; ++i)
        swap(&a[i], &a[(rand() % size)]);
}

// сортировка, возвращает количество перемешиваний
int bogoSort(int *a, size_t size) {
    int count = 0;
    while (!isSorted(a, size)) {
        shuffleArray(a, size);
        count += 1;
    }
    return count;
}

// эксперимент
int main() {
    int a[] = {6, 4, 2, 5, 7, 1, 3};
    int n = 7;

    double s = 0;
    double nExpr = 500000.0;
    for (int i = 0; i < nExpr; i++) {
        s += bogoSort(a, n);
        shuffleArray(a, n);
    }

    printf("%f", s / nExpr);

    return 0;
}
  
```

---

---

**Листинг 46** Сортировка выбором
 

---

```

void selectionSort(int *a, const int size) {
    for (int i = 0; i < size - 1; i++) {
        int minPos = i;
        for (int j = i + 1; j < size; j++)
            if (a[j] < a[minPos])
                minPos = j;
        swap(&a[i], &a[minPos]);
    }
}
  
```

---

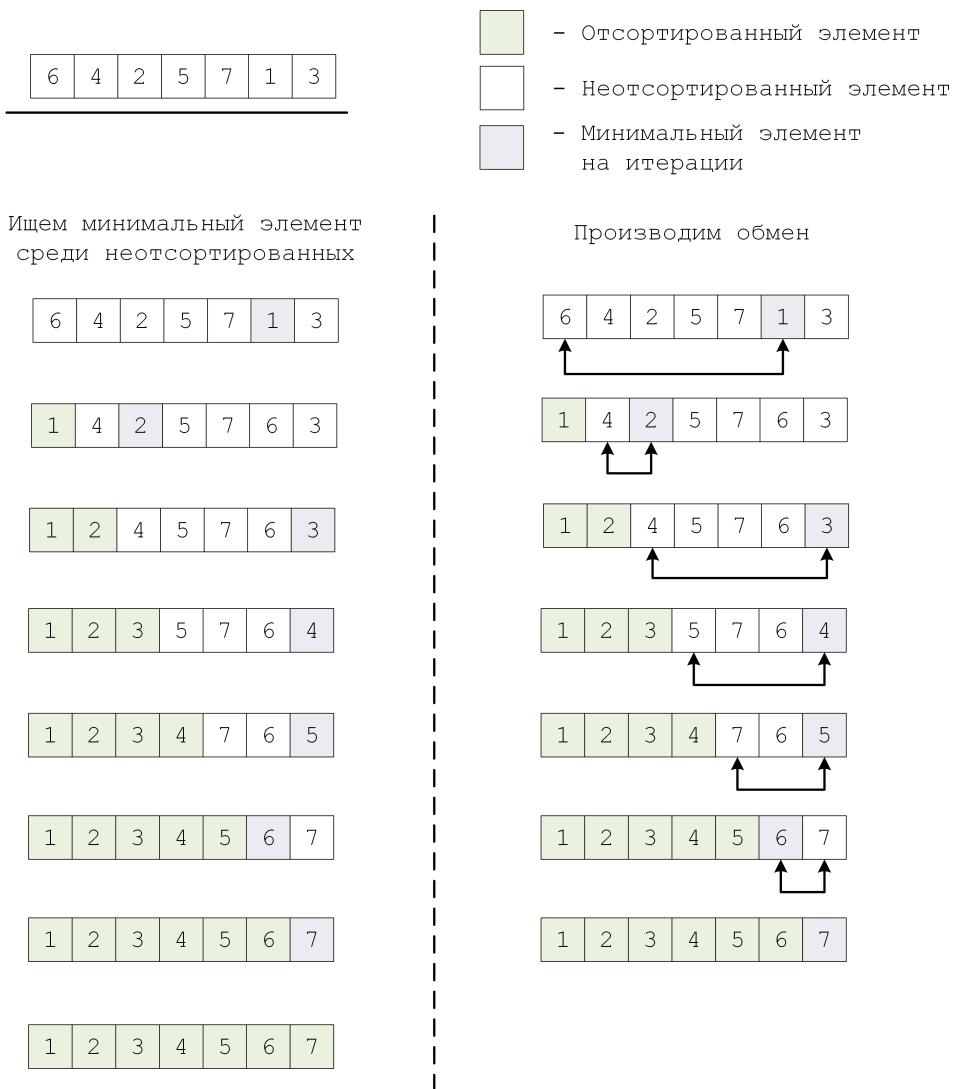


Рис. 12.3: Сортировка выбором

При сортировке выбором число сравнений элементов не зависит от их начального порядка. На первом шаге выполняется  $n - 1$  сравнение, на втором —  $n - 2$  и т. д. Следовательно, общее число сравнений равно

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) = \frac{(n - 1) * n}{2}$$

Порядок функции временной сложности составляет  $O(n^2)$ .

## 12.3 Сортировка вставками

### Сортировка вставками

Дан массив  $a$  размера  $n$ . Необходимо упорядочить элементы массива по неубыванию сортировкой вставками.

Сложность по времени:  $O(n^2)$ ,  $\Omega(n)$ .

Сортировку вставками часто применяют, если исходный массив практически отсортирован: то есть относительный порядок элементов более-менее напоминает верный. Опишем алгоритм в словесно-формульном виде:

1. Запоминаем элемент, подлежащий вставке.
2. Перебираем справа налево отсортированные элементы и сдвигаем каждый элемент вправо на одну позицию, пока не освободится место для вставляемого элемента.
3. Вставляем элемент на освободившееся место.

Пункты 1–3 выполняем для всех элементов массива, кроме первого. Процесс сортировки массива изображен на рисунке 12.5. Блок-схема на рисунке 12.4.

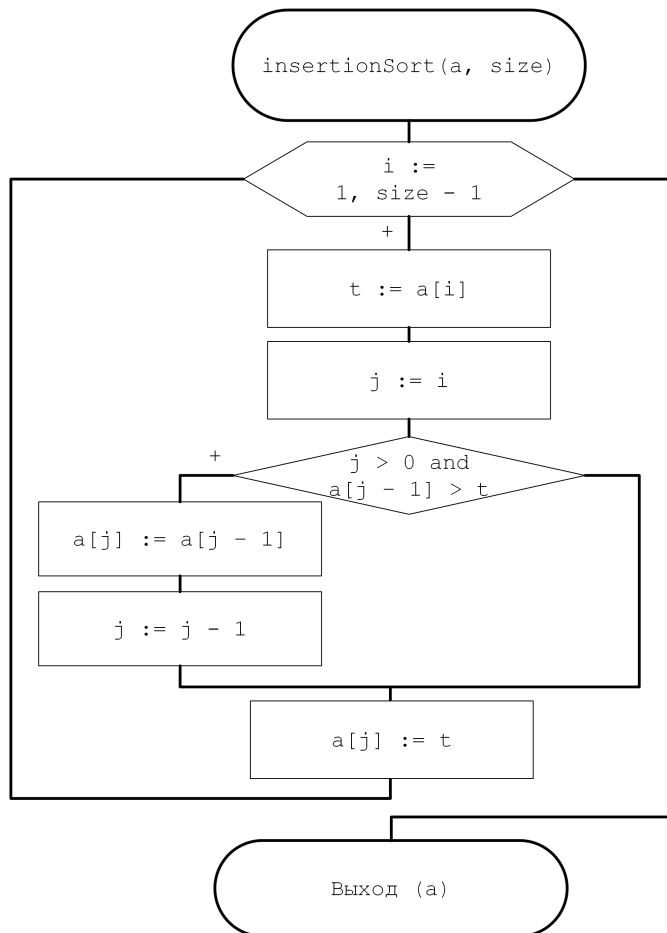


Рис. 12.4: Блок-схема сортировки вставками

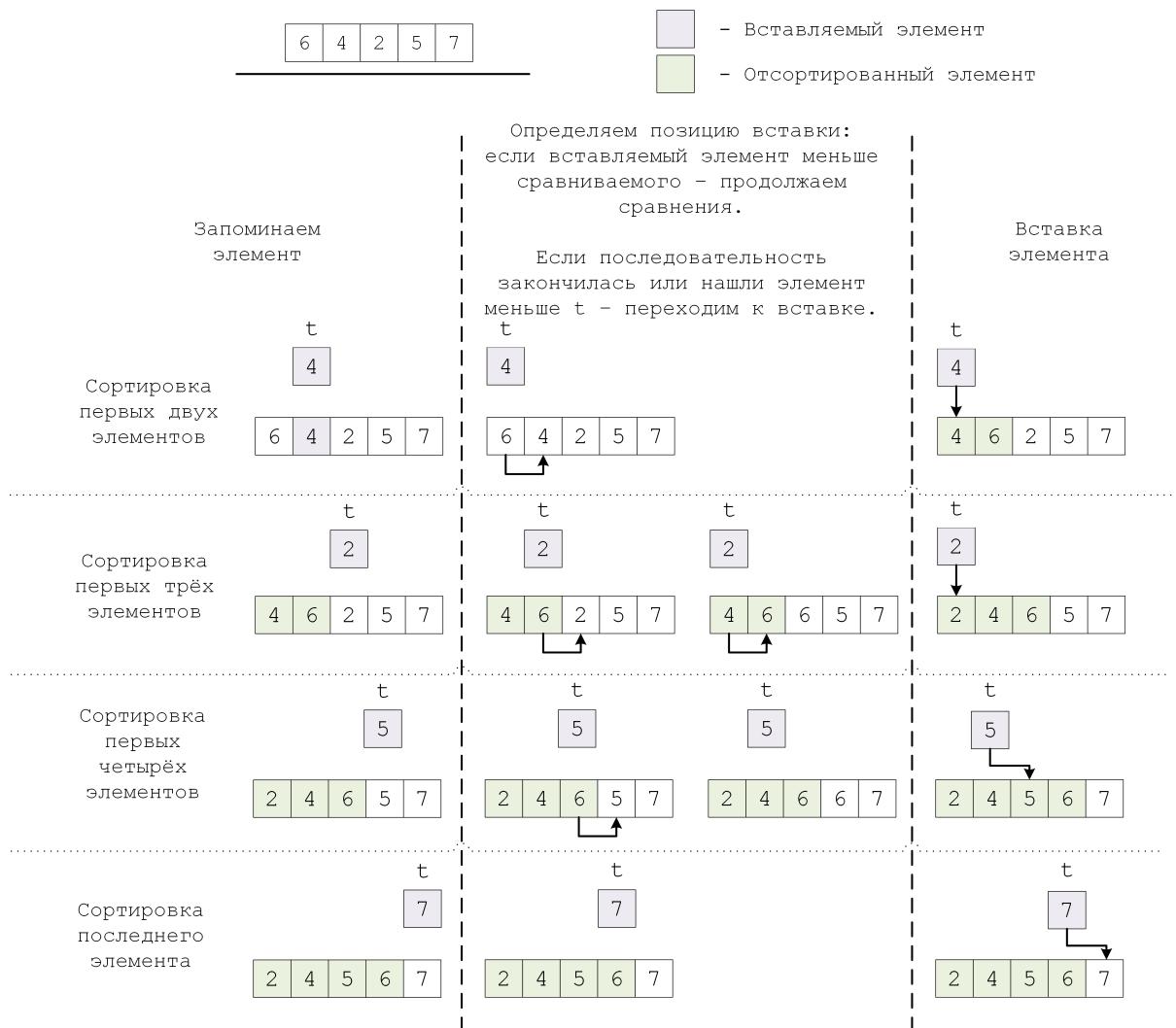


Рис. 12.5: Сортировка вставками

Проведём анализ наилучшего и наихудшего случая. Если массив уже отсортирован, то будет проведено  $n$  сравнений. Тогда порядок функции временной сложности –  $O(n)$ . В наихудшем случае –  $O(n^2)$  (массив упорядочен в обратном порядке).

#### Листинг 47 Сортировка вставками

```
void insertionSort(int *a, const size_t size) {
    for (size_t i = 1; i < size; i++) {
        int t = a[i];
        int j = i;
        while (j > 0 && a[j - 1] > t) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = t;
    }
}
```

## 12.4 Обменная сортировка

### Обменная сортировка

Дан массив  $a$  размера  $n$ . Необходимо упорядочить элементы массива по неубыванию обменной сортировкой.

Сложность по времени:  $O(n^2)$ .

Идея обменной сортировки заключается в том, что два элемента, нарушающие требуемый порядок, меняются местами. Как и в методе выбора, совершаются проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу массива.

Если рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в банке с водой, причем вес каждого соответствует его значению. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу. Такой метод известен под именем «пузырьковая сортировка».

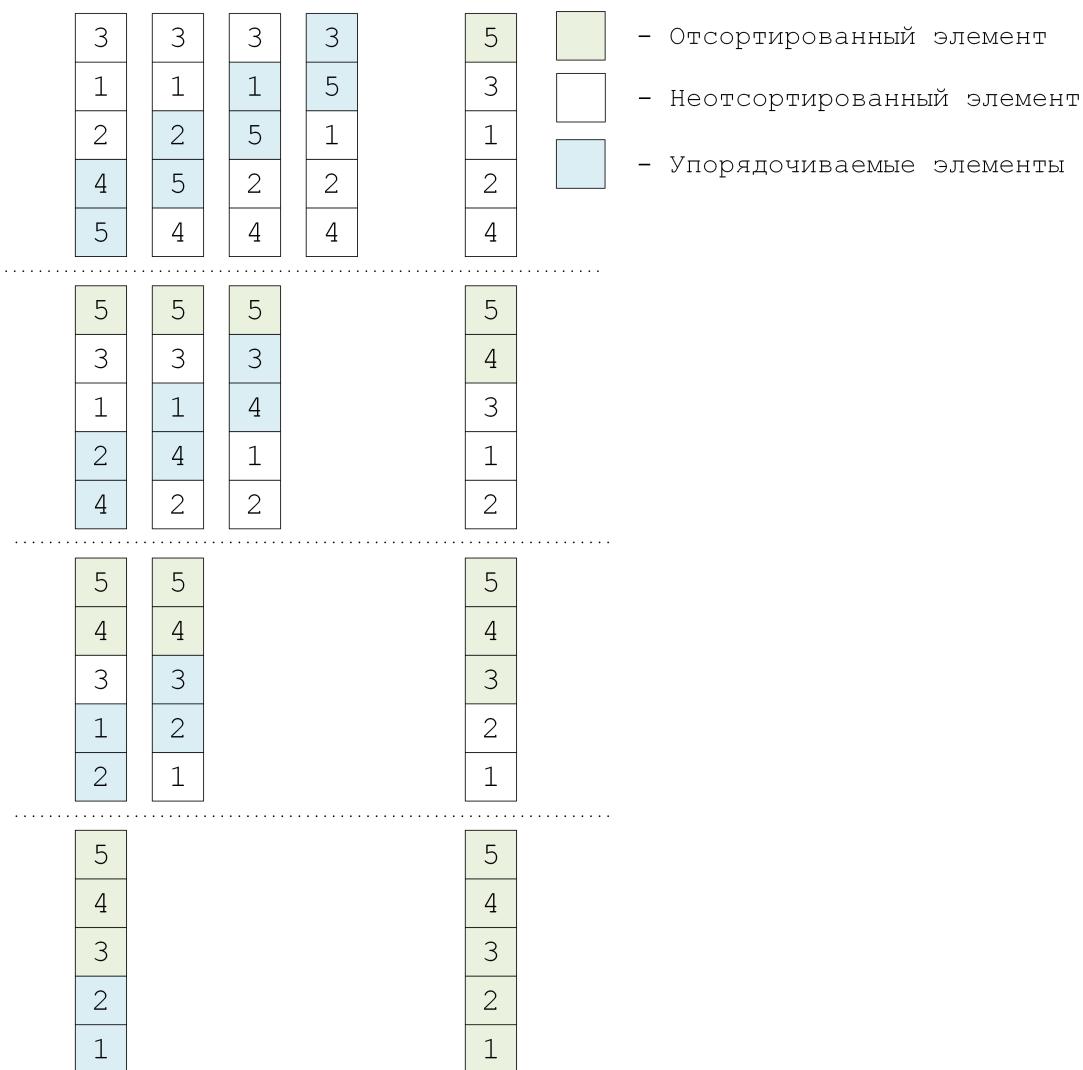


Рис. 12.6: Идея пузырьковой сортировки

Возможная реализация (сдвигает минимум вниз):

---

#### Листинг 48 Пузырьковая сортировка

---

```
void bubbleSort(int *a, size_t size) {
    for (size_t i = 0; i < size - 1; i++)
        for (size_t j = size - 1; j > i; j--)
            if (a[j - 1] > a[j])
                swap(&a[j - 1], &a[j]);
}
```

---

Существуют определенные модификации алгоритмов сортировки:

- После каждого шага может быть сделана проверка, были ли совершены перестановки в течение данного шага. Если перестановок не было, то массив упорядочен и дальнейших шагов не требуется;
- В течение шага фиксируется последний элемент, участвующий в обмене. В очередном проходе этот элемент и все предшествующие в сравнении не участвуют, т. к. все элементы до этой позиции уже отсортированы.

Модификации носят теоретический характер. Сложность сортировки всё-таки остаётся достаточно большой  $O(n^2)$ , чтобы быть использованной в реальных проектах.

## 12.5 Сортировка расческой

Основная идея 'расчёски' в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы 'причёсываем' массив, постепенно разглаживая на всё более аккуратные пряди. Первонаучальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения  $k$ , оптимальное значение которой равно примерно 1,247:

$$k = \frac{1}{1 - \frac{1}{e^{-\phi}}}$$

где  $e$  — основание натурального логарифма, а  $\phi$  — золотое сечение.

Сначала расстояние между элементами максимально, то есть равно  $n - 1$ . Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройтись вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы, как и в сортировке пузырьком, но такая итерация одна.

---

**Листинг 49** Сортировка расческой
 

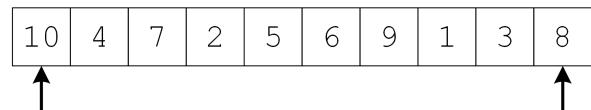
---

```

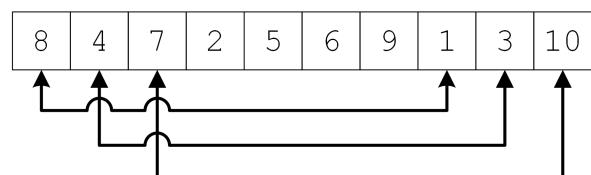
void combsort(int *a, const size_t size) {
    size_t step = size;
    int swapped = 1;
    while (step > 1 || swapped) {
        if (step > 1)
            step /= 1.24733;
        swapped = 0;
        for (size_t i = 0, j = i + step; j < size; ++i, ++j)
            if (a[i] > a[j])
                swap(&a[i], &a[j]);
            swapped = 1;
    }
}
  
```

---

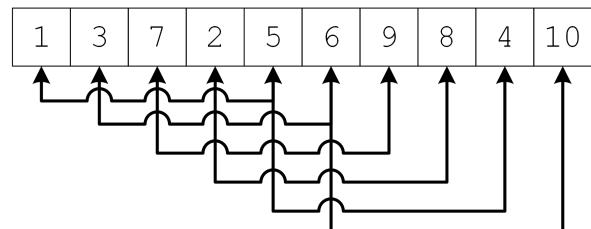
Шаг 1 ( $t = n - 1 = 9$ )



Шаг 2 ( $t := t / 1.2473 = 7$ )



Шаг 3 ( $t := t / 1.2473 = 5$ )



Шаг 4 ( $t := t / 1.2473 = 4$ )

Шаг 5 ( $t := t / 1.2473 = 3$ )

Шаг 6 ( $t := t / 1.2473 = 2$ )

Шаг 7 ( $t := t / 1.2473 = 1$ )

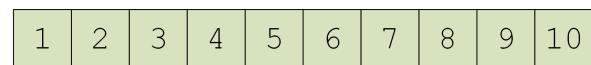


Рис. 12.7: Изменение шага при сортировке расческой

## 12.6 Сортировка подсчётом

### Сортировка подсчётом

Дан массив  $a$  размера  $n$ . Необходимо упорядочить элементы массива по неубыванию сортировкой подсчётом.

Сложность по времени:  $O(n + k)$ ,

Требуется дополнительная память:  $O(k)$ , где

$$k = \max(a) - \min(a) + 1$$

Сортировка подсчетом применяется в том случае, если  $\max(a) - \min(a)$  относительно невелико по отношению к количеству элементов массива.

Пусть дан массив  $a = \{1, 1, 3, 3, 2, 2, 2, 2, 1, 2, 3\}$ . Создадим вспомогательный массив, где  $i$ -ый элемент будет хранить количество значений, равных  $\min(a)$ ,  $i + 1$ -ый элемент ответит за количество встреченных  $\min(a) + 1, \dots$ , а  $k$ -ый - количество  $\max(a)$ .

Интервал  $[\min(a), \max(a)]$  представлен тремя значениями. То есть для сортировки последовательности нам пригодится вспомогательный массив  $b$  размера 3. Осталось подсчитать, сколько раз встречался каждый элемент массива  $a$  и получить мульти множества на массиве  $b$ :

$$a = \{1, 1, 3, 3, 2, 2, 2, 2, 1, 2, 3\}$$

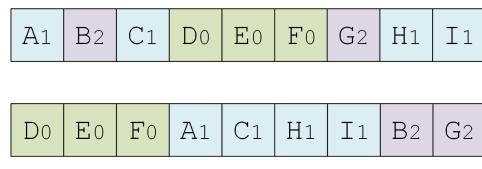
$$b = \{3, 5, 3\}$$

Таким образом, мы знаем, что  $\min(a) = 1$  встречался 3 раза,  $\min(a) + 1 = 2$  встретился 5 раз,  $\min(a) + 2 = 3$  встретился 3 раза.

Последним действием нам нужно перезаписать массив  $a$ :

$$a = \{1, 1, 1, 2, 2, 2, 2, 3, 3, 3\}$$

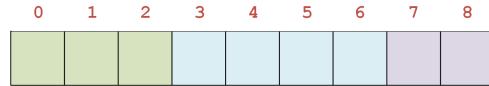
Рассмотрим такой случай. Пусть элементы массива – это не просто числа, а какие-то более сложные объекты, которые требуется отсортировать по ключу. От того, что мы отсортируем только одни ключи, не отсортируются объекты. Укажем объекты в виде букв, а в качестве индекса будет выступать значение ключа. Исходный массив и его отсортированная версия представлены на рисунке ниже:



Посчитаем, сколько раз встречался тот или иной ключ:

	0	1	2
cnt	3	4	2

Создадим массив для записи результата, места в котором зарезервируем под элементы с соответствующим ключом:



Если найти массив префиксных сумм  $p$  для массива с количеством ключей, можно найти индексы начала каждой из групп:

$$p[0] = 0 \quad p[i] = p[i - 1] + cnt[i - 1]$$

$$p[0] = 0 \quad p[1] = p[0] + cnt[0] = 3 \quad p[2] = p[1] + cnt[1] = 7$$

Таким образом, массив  $p$ :



Пройдёмся по исходному массиву и будем последовательно наносить элементы в результирующий массив. При занесении элемента определенного типа, будем увеличивать значение  $p[i]$ . Этап представлен на рисунке 12.8.

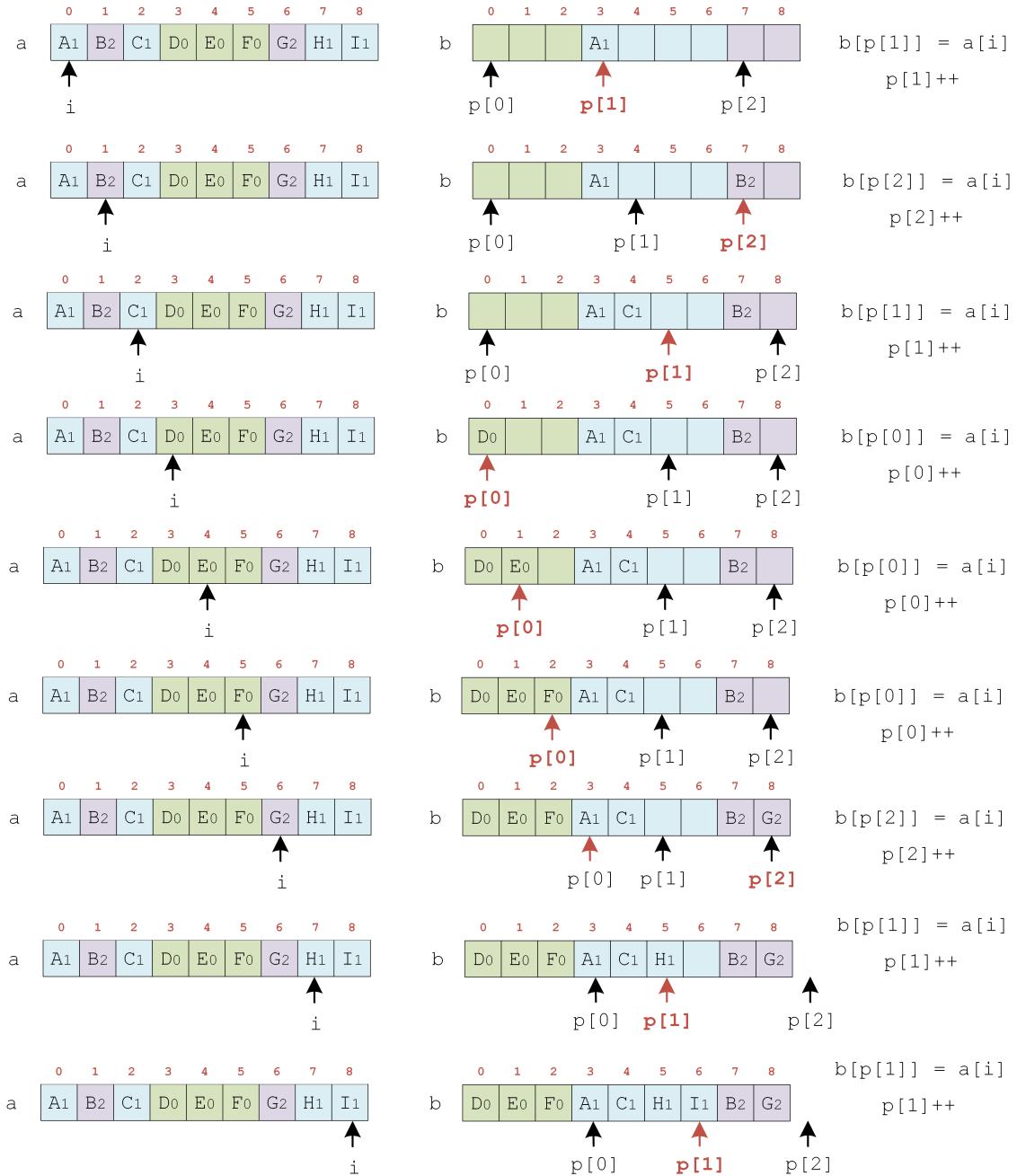


Рис. 12.8: Последний этап сортировки

Важной особенностью сортировки является её устойчивость. Данный аспект будет использован для реализации цифровой сортировки.

---

**Листинг 50** Сортировка подсчётом
 

---

```
#include <stdlib.h>

void getMinMax(const int *a, size_t size, int *min, int *max) {
    *min = a[0];
    *max = a[0];
    for (int i = 1; i < size; i++) {
        if (a[i] < *min)
            *min = a[i];
        else if(a[i] > *max)
            *max = a[i];
    }
}

void countSort(int *a, const size_t size) {
    int min, max;
    getMinMax(a, size, &min, &max);
    int k = max - min + 1;

    // выделение памяти под динамический массив из k элементов,
    // где каждый из элементов равен 0
    int *b = (int*)calloc(k, sizeof(int));
    for (int i = 0; i < size; i++)
        b[a[i] - min]++;

    int ind = 0;
    for (int i = 0; i < k; i++) {
        int x = b[i];
        for (int j = 0; j < x; j++)
            a[ind++] = min + i;
    }

    // освобождение памяти, выделенной под динамический массив
    free(b);
}
```

---

## 12.7 Цифровая сортировка

Рассмотрим случай, когда разброс между значениями стал ещё больше. Сортировка будет показана на небольших значениях, а потом обобщена для больших значений ключей.

0	1	2	3	4	5	6	7	8	9
01100011	11001010	01100110	11011001	11110000	10100101	00111100	00001010	01010110	10101111

Выполним сортировку подсчётом на основании двух последних битов:

0	1	2	3	4	5	6	7	8	9	
01100011	11001010	01100110	11011001	11110000	10100101	00111100	00001010	01010110	10101111	
00	01	10	11			00	01	10	11	
<b>cnt</b>	2	2	4	2		<b>p</b>	0	2	4	8
0	1	2	3	4	5	6	7	8	9	
11110000	00111100	11011001	10100101	11001010	01100110	00001010	01010110	01100011	10101111	

В силу стабильности сортировки подсчётом, сохраняется относительный порядок элементов. Повторим данную сортировку для последующих пар битов:

0	1	2	3	4	5	6	7	8	9	
11110000	00111100	11011001	10100101	11001010	01100110	00001010	01010110	01100011	10101111	
00	01	10	11			00	01	10	11	
<b>cnt</b>	2	3	3	2		<b>p</b>	0	2	5	8
0	1	2	3	4	5	6	7	8	9	
11110000	01100011	10100101	01100110	01010110	11011001	11001010	00001010	00111100	10101111	

После двух сортировок замечаем, что последние 4 бита будут отсортированы. Выполним сортировку ещё два раза и получим отсортированные значения:

0	1	2	3	4	5	6	7	8	9	
11110000	01100011	10100101	01100110	01010110	11011001	11001010	00001010	00111100	10101111	
00	01	10	11			00	01	10	11	
<b>cnt</b>	2	2	4	2		<b>p</b>	0	2	4	8
0	1	2	3	4	5	6	7	8	9	
11001010	00001010	01010110	11011001	01100011	10100101	01100110	10101111	11110000	00111100	
00	01	10	11			00	01	10	11	
<b>cnt</b>	2	3	2	3		<b>p</b>	0	2	5	7
0	1	2	3	4	5	6	7	8	9	
00001010	00111100	01010110	01100011	01100110	10100101	10101111	11001010	11011001	11110000	

На самом деле, вы можете выполнить реализацию сортировки самостоятельно. Но шанс того, что она будет работать быстрее библиотечной функции `qsort` довольно мал.

## 12.8 Сортировка слиянием

### Сортировка слиянием

Пусть имеются отсортированные по неубыванию последовательности  $a$  размера  $n$  и  $b$  размера  $m$ . Необходимо получить массив  $c$  размера  $n + m$ , в котором элементы так же отсортированы по неубыванию.

Пример:

$$a = \{1, 3, 3, 5, 10, 10, 35\}$$

$$b = \{1, 2, 2, 8\}$$

Тогда:

$$c = \{1, 1, 2, 2, 3, 3, 5, 8, 10, 10, 35\}$$

Сортировка должна проходить до тех пор, пока остались необработанные элементы. Мы записываем в результат элемент из первого массива в одном из двух случаев:

- Если закончились элементы второго массива;
- Если элементы есть в обоих массивах, и сравниваемое значение первого массива меньше, чем значение второго массива. В противном случае записываем элемент из второго массива.

Опишем функцию, которая выполняла заявленные действия:

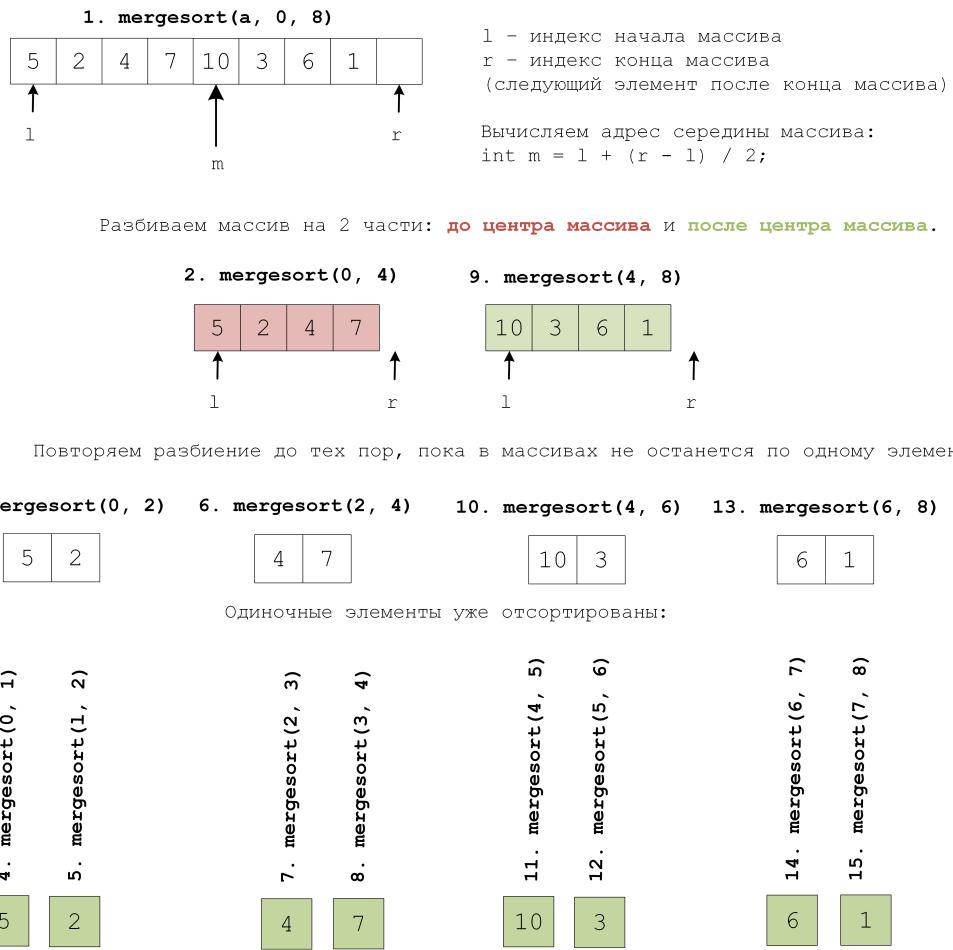
---

```
void merge(const int *a, const int n,
           const int *b, const int m, int *c) {
    int i = 0, j = 0;
    while (i < n || j < m) {
        if (j == m || i < n && a[i] < b[j]) {
            c[i + j] = a[i];
            i++;
        } else {
            c[i + j] = b[j];
            j++;
        }
    }
}
```

---

Давайте разовьём идею сортировки слиянием на произвольные массивы. Для этого потребуется знание рекурсивных функций:

## Рекурсивный спуск




---

Если функция выполнила разбиения и нам известно, что и первая и вторая часть массива отсортирована – можно выполнять слияние левой и правой части.

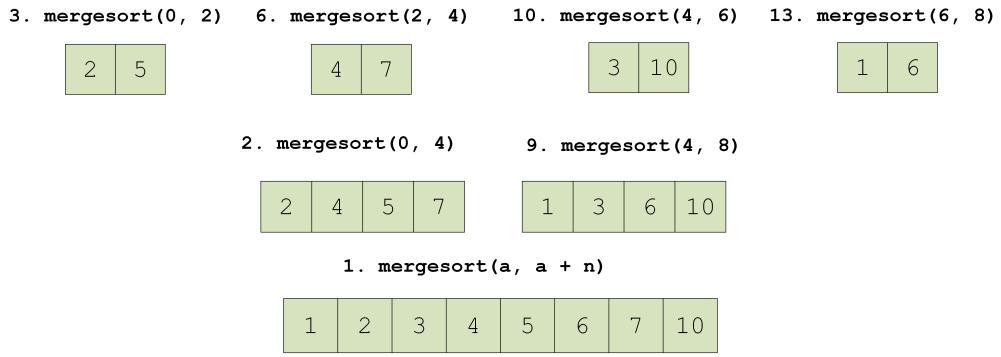


Рис. 12.9: Сортировка слиянием

При сортировке слиянием на рекурсивном спуске массив разбивается на подмассивы пополам, до тех пор, пока в подмассиве не останется один элемент. Один элемент является отсортированным сам по себе. Когда разбиение закончилось, постепенно объединяются подмассивы. В процессе объединения они сортируются. Сложность алгоритма сортировки по времени:  $O(n \log n)$ , по памяти  $O(n)$ .

---

**Листинг 51** Сортировка слиянием
 

---

```

#include <stdio.h>
#include <malloc.h>
#include <memory.h>

void merge(const int *a, const int n,
           const int *b, const int m, int *c) {
    int i = 0, j = 0;
    while (i < n || j < m) {
        if (j == m || i < n && a[i] < b[j]) {
            c[i + j] = a[i];
            i++;
        } else {
            c[i + j] = b[j];
            j++;
        }
    }
}

void mergeSort_(int *source, int l, int r, int *buffer) {
    int n = r - l;
    if (n <= 1)
        return;

    // определяем середину последовательности
    // и рекурсивно вызываем функцию сортировки для каждой половины
    int m = (l + r) / 2;
    mergeSort_(source, l, m, buffer);
    mergeSort_(source, m, r, buffer);

    // производим слияние элементов, результатом сохраняем в буфере
    merge(source + l, m - l, source + m, r - m, buffer);
    // переписываем сформированную последовательность с буфера
    // в исходный массив
    memcpy(source + l, buffer, sizeof(int) * n);
}

void mergeSort(int *a, int n) {
    // создаём буфер из которого будут браться элементы массива
    int *buffer = (int*)malloc(sizeof(int) * n);
    mergeSort_(a, 0, n, buffer);
    free(buffer);
}

int main() {
    int a[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
    size_t n = sizeof(a) / sizeof(int);

    mergeSort(a, n);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}

```

---

## 12.9 Куча и сортировка кучей

Структура данных, которая хранит некоторое множество элементов. Поддерживает операции:

- `insert(x)` – добавляет элемент в кучу.
- `getMin()` – возвращает минимальный элемент, лежащий в куче.
- `removeMin()` – удаляет минимальный элемент из кучи. Если элементов несколько, удаляется только один из них.

Рассмотрим несколько реализаций на массиве:

---

```

1 insert(x): // O(1)
2     a[n++] = x;
3
4 getMin(): // O(n)
5     res = ∞
6     for i = 0...n - 1:
7         res = min(res, a[i]);
8     return res;
9
10 removeMin(): // O(n)
11     minIndex = 0
12     for i = 1...n - 1:
13         if a[i] < a[minIndex]:
14             minIndex = i
15     swap(a[minIndex], a[--n])

```

---

Можно несколько модифицировать подход и хранить в `a[m]` значение минимума, тогда:

---

```

1 getMin(): // O(1)
2     return a[m];
3
4 insert(x): // O(1)
5     a[n++] = x;
6     if (x < a[m])
7         m = n - 1;
8
9 removeMin(): // O(n)
10    swap(a[m], a[--n])
11    m = 0;
12    for i = 1...n - 1:
13        if a[i] < a[m]
14            m = i

```

---

Предположим, что массив `a` является упорядоченным и минимальный элемент хранится в конце массива, тогда:

---

```

1 getMin(): // O(1)
2     return a[n - 1];
3
4 insert(x): // O(n)
5     insertionPos = n
6     while (x > a[insertionPos] && insertionPos > 0):
7         a[insertionPos + 1] = a[insertionPos]
8         insertionPos--
9     a[insertionPos] = x
10    n++

```

---

```

11
12 removeMin(): // O(1)
13     n--;

```

С другой стороны, можно разбить массив на  $\sqrt{n}$  блоков, где  $n$  – максимальный размер структуры данных. Вставка бы заключалась в размещении элемента в последнем блоке (если там есть место, а если места нет – размещение происходило бы в следующем блоке). Вставка работала бы за  $O(\sqrt{n})$ . Для поиска минимума в СД достаточно найти минимальное значение среди минимальных значений блоков  $O(\sqrt{n})$ . Для удаления – выполнить поиск минимума. И закинуть последнее значение из последней кучи на освободившееся место  $O(\sqrt{n})$ .

Рассмотрим следующий случай. Представим кучу в виде дерева на массиве. Пронумеруем элементы дерева и элементы массива. Соответствие индексу элемента массива и дереву представлено на рисунке 17.1. Дерево заполнено по уровням. Заполнены будут все уровни (кроме, возможно, последнего).

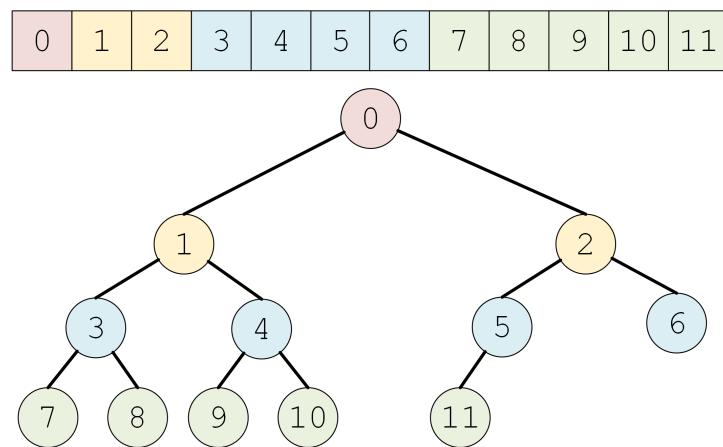


Рис. 12.10: Соответствие элемента массива узлу дерева

Показанное дерево на рисунке 17.1 является бинарным. У каждого родителя не более двух детей. Например, у родителя 0 детьми являются узлы 1 и 2, у родителя 2 – узлы 5, 6 и так далее.

Рассмотрим произвольный узел  $i$  и определим способ вычисления индексов потомков:

$$i_{left\_child} = 2 * i_{parent} + 1$$

$$i_{right\_child} = 2 * i_{parent} + 2$$

Если нам известен номер ребёнка, номер родителя вычисляется как:

$$i_{parent} = \left\lfloor \frac{i_{child} - 1}{2} \right\rfloor$$

В процессе создания кучи будем поддерживать следующую идею: необходимо, чтобы значение в родителе было меньше или равно, чем значение в любом из потомков (пример на рисунке 17.2).

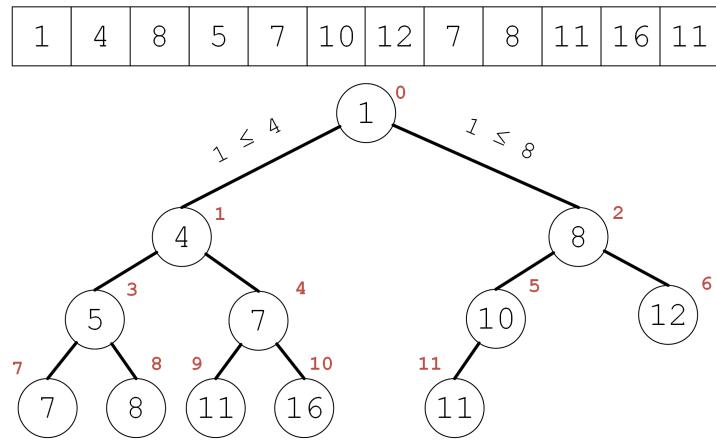


Рис. 12.11: Пример кучи

Куча отображается в виде дерева, чтобы лучше понимать, какие взаимосвязи есть между элементами массива. Опишем операции при взаимодействии с такой структурой. Поиск минимума осуществить просто - это значение  $a[0]$ :

```
getMin(): // O(1)
    return a[0];
```

Опишем вставку. Добавим элемент в конец массива:

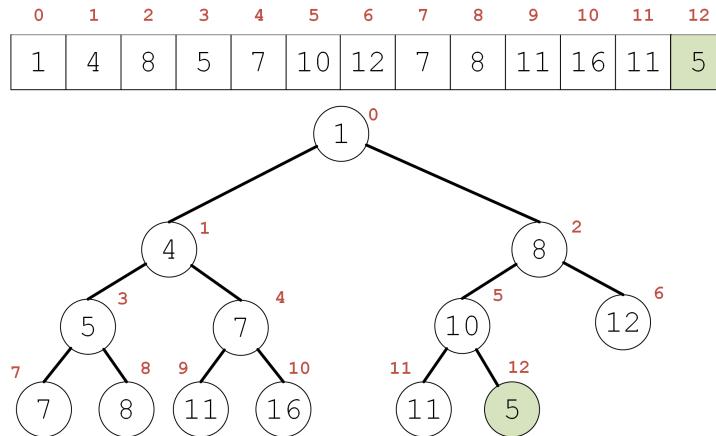


Рис. 12.12: Состояние кучи после вставки в конец

Но он нарушает требование кучи к упорядоченности элементов. Будем последовательно обменивать ребёнка и родителя до тех пор, пока вставляемый элемент не окажется на правильном месте:

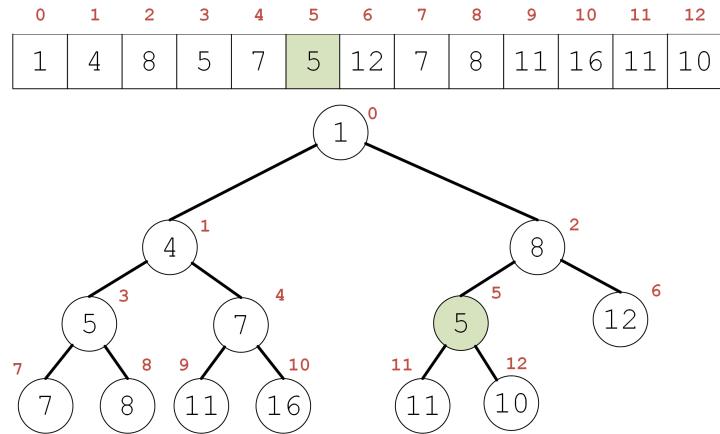


Рис. 12.13: Обмен элементов на 5 и 12 позициях

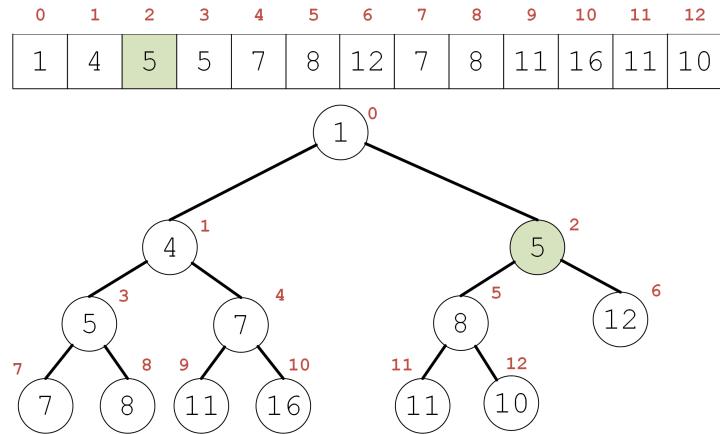


Рис. 12.14: Обмен элементов на 2 и 5 позициях

Обмены будут происходить до тех пор, пока мы не доберёмся до родителя, или родитель не станет меньше вставляемого элемента:

---

```

1 insert(x): // O(log n)
2     a[n++] = x
3     childPos = n - 1
4     parentPos = (childPos - 1) / 2
5     while a[childPos] < a[parentPos] and childPos *$\\ne\$* 0:
6         swap(a[childPos], a[parentPos])
7         childPos = parentPos;
8         parentPos = (childPos - 1) / 2

```

---

Удаление проходит в несколько этапов:

- Перекидываем последний элемент в корень:

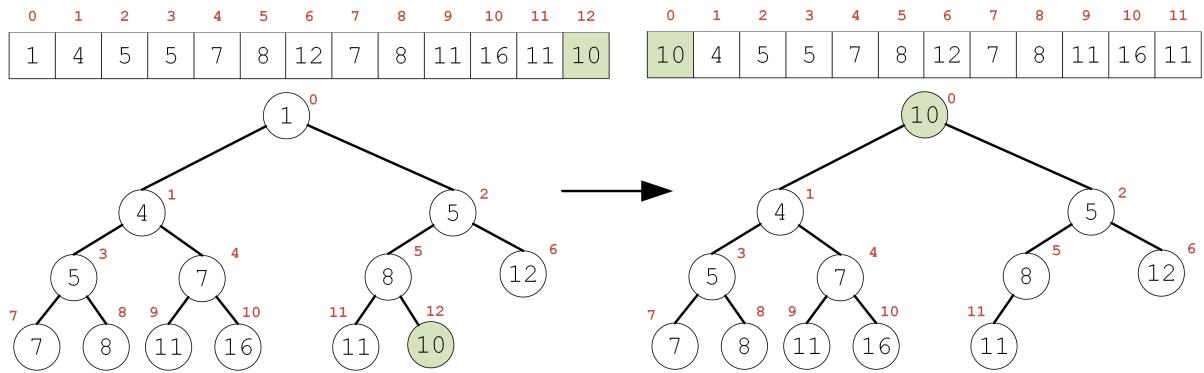


Рис. 12.15: Перенос последнего элемента в корень

- Сравниваем вставляемое значение с потомками. Если находится ПОТОМКИ с меньшим значением - производится обмен:

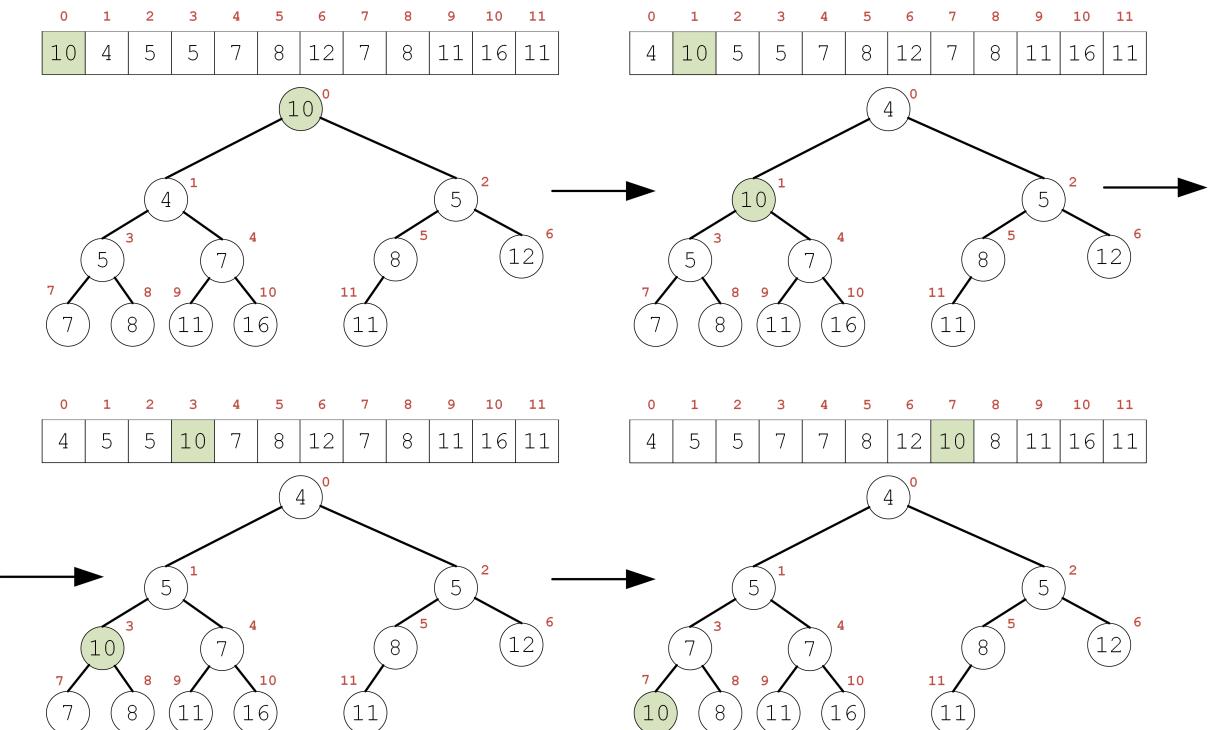


Рис. 12.16: Проталкивание элемента

```

1 getMinChildIndex(parentIndex):
2     leftChild = 2 * parentIndex + 1
3     rightChild = leftChild + 1
4     if (rightChild < size): // есть два ребёнка
5         return a[leftChild] < a[rightChild] ? leftChild : rightChild
6     else: // есть один ребёнок
7         return leftChild
8
9 haveChild(parentIndex):
10    return 2*parentIndex + 1 < size
11
12 removeMin(): // O(logN)
13    swap(a[0], a[--size]);

```

```

14     parentIndex = 0;
15     while haveChild(parentIndex, *size):
16         minChild = getMinChildIndex(a, parentIndex, *size);
17         if (a[minChild] < a[parentIndex])
18             swap(a[minChild], a[parentIndex]);
19             parentIndex = minChild;
20         else:
21             break;

```

Сортировку кучей можно представить из двух этапов:

1. Закинуть элементы в кучу.
2. На  $i$ -ом шаге доставать элемент из кучи и сохранить в отсортированный массив.

```

1 // n - размер сортируемого массива
2 heap h;
3 for i = 0...n - 1:
4     h.insert(a[i])
5 for i = 0...n - 1:
6     a[i] = h.getMin();
7     h.removeMin();

```

Единственный недостаток такого подхода: требуется отдельная память под кучу. Но можно организовать вычисления иначе.

Пусть требуется отсортировать массив  $a$ . Организуем кучу на памяти, отведенной под массив  $a$ . Пример:

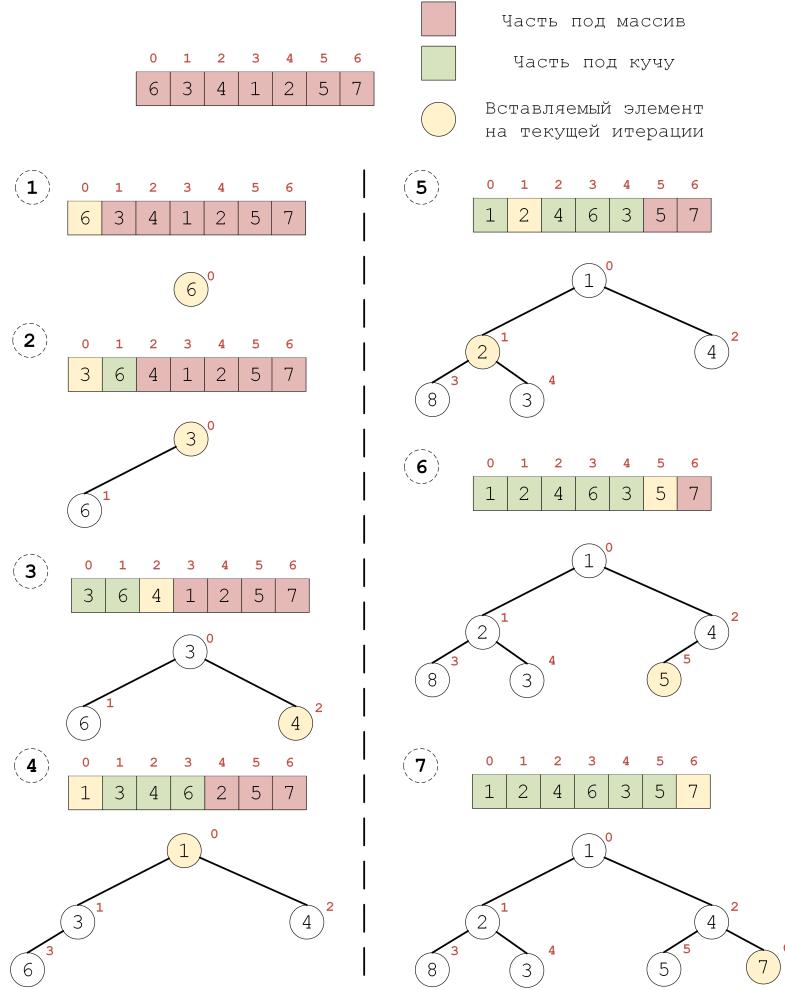


Рис. 12.17: Процесс построения кучи на массиве  $a$

После этого выполним удаление минимальных элементов из кучи, пока она не станет пуста. Иллюстрация этого представлена на рисунке ??.

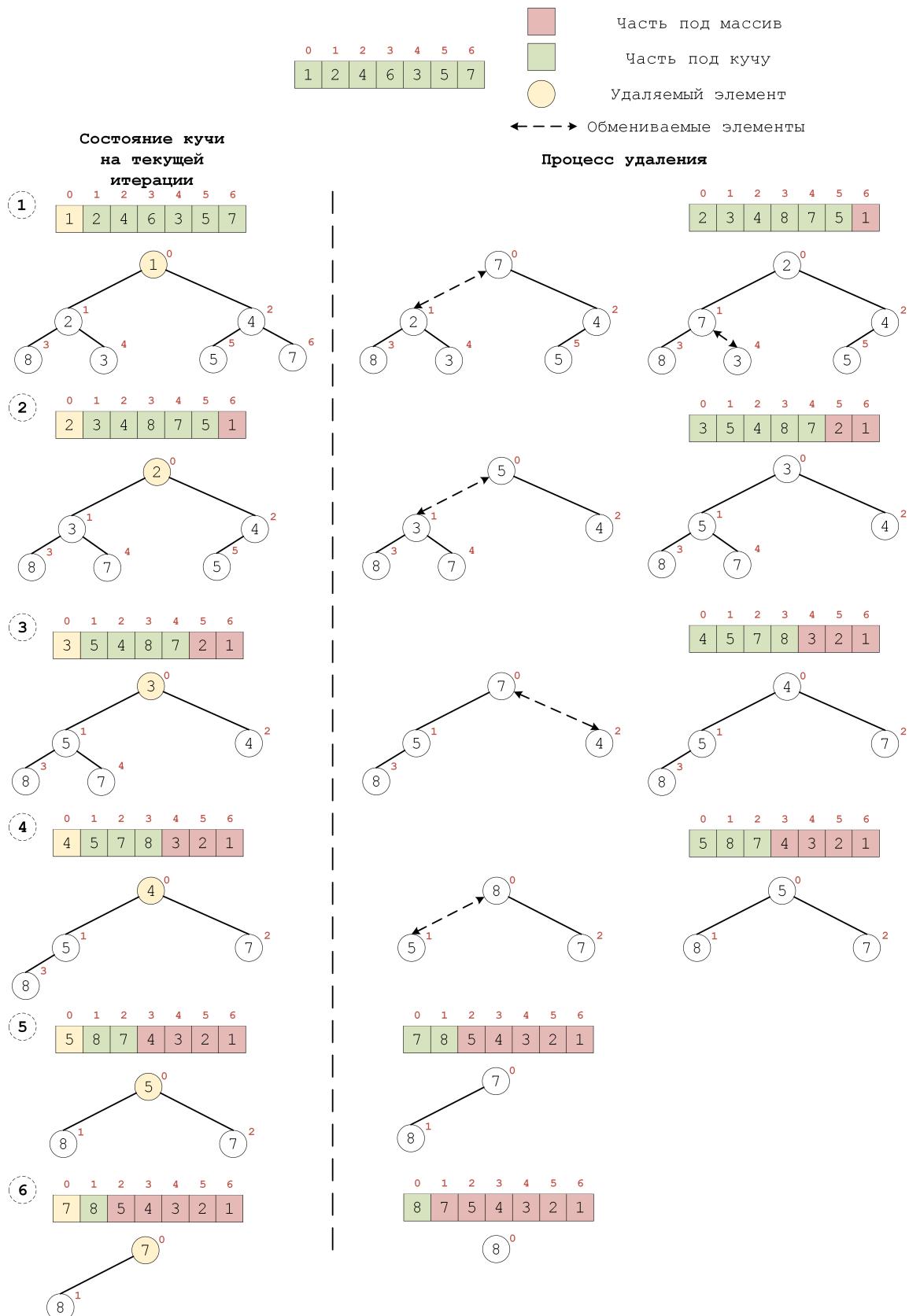


Рис. 12.18: Процесс удаления элементов из кучи. Можно заметить, что массив сортируется по невозрастанию

Реализуем функцию вставки элемента в кучу, удаление элемента из кучи и алгоритм сортировки:

---

### Листинг 52 Операции над кучей

---

```

void insertHeap(int *a, size_t *size, int x) {
    a[(*size)++] = x;
    size_t childIndex = *size - 1;
    size_t parentIndex = (childIndex - 1) / 2;
    while (a[childIndex] < a[parentIndex] && childIndex != 0) {
        swap(&a[childIndex], &a[parentIndex]);
        childIndex = parentIndex;
        parentIndex = (childIndex - 1) / 2;
    }
}

bool hasLeftChild(size_t parentIndex, size_t size) {
    return 2 * parentIndex + 1 < size;
}

bool hasRightChild(size_t parentIndex, size_t size) {
    return 2 * parentIndex + 2 < size;
}

size_t getLeftChildIndex(size_t parentIndex) {
    return 2 * parentIndex + 1;
}

size_t getMinChildIndex(const int *a, size_t size, size_t parentIndex) {
    size_t minChildIndex = getLeftChildIndex(parentIndex);
    size_t rightChildIndex = minChildIndex + 1;
    if (hasRightChild(parentIndex, size))
        if (a[rightChildIndex] < a[minChildIndex])
            minChildIndex = rightChildIndex;
    return minChildIndex;
}

void removeMinHeap(int *a, size_t *size) {
    *size -= 1;
    swap(&a[0], &a[*size]);
    size_t parentIndex = 0;
    while (hasLeftChild(parentIndex, *size)) {
        size_t minChildIndex = getMinChildIndex(a, *size, parentIndex);
        if (a[minChildIndex] < a[parentIndex]) {
            swap(&a[minChildIndex], &a[parentIndex]);
            parentIndex = minChildIndex;
        } else break;
    }
}

```

---

---

**Листинг 53** Сортировка кучей
 

---

```
void heapSort(int *a, size_t size) {
    size_t heapSize = 0;
    while (heapSize != size)
        insertHeap(a, &heapSize, a[heapSize]);
    while (heapSize)
        removeMinHeap(a, &heapSize);
}
```

---

## 12.10 Быстрая сортировка

Быстрая сортировка (англ. *quicksort*), часто называемая *qsort* (по имени в стандартной библиотеке языка C) — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром во время его работы в МГУ в 1960 году.

Алгоритм сортировки Хоара

1. Выбираем случайный элемент массива в качестве разделителя.
2. Располагаем элементы меньшие разделителя в первой части массива, а большие — во второй.
3. Если число элементов в первой части массива больше 1, то применяем к ней алгоритм в целом, иначе конец алгоритма.
4. Если число элементов во второй части массива больше 1, то применяем к ней алгоритм в целом, иначе конец алгоритма.

Опишем функцию `split`, которая разделяет элементы массива на две части: первая группа элементов меньше  $x$ , вторая — больше или равна  $x$ :

```
1 split(left, right, x):
2     iWrite = left
3     for iRead = left..right-1:
4         if a[iRead] < x:
5             swap(a[iRead], a[iWrite])
6             iWrite++
7     return iWrite
```

---

Тогда сортировка может быть описана так:

```
1 sort(left, right):
2     if right - left <= 1:
3         return
4     else:
5         x = a[rand(left, right - 1)]
6         middle = split(left, right, x)
7         sort(left, middle)
8         sort(middle, right)
```

---

Однако она не будет работать, например, для массива с равными элементами в силу специфики работы функции `split`<sup>1</sup>.

<sup>1</sup>Вы можете легко в этом убедиться, если выполните сортировку для двух равных элементов.

## Поиск $k$ -ой порядковой статистики. Алгоритм Хоара

Идея алгоритма Хоара может быть использована для поиска  $k$ -ой порядковой статистики<sup>2</sup>. Запишем алгоритм псевдокодом:

```

1 find(left, right):
2     if right - left <= 1: // a[k] == a[left]
3         return a[left]
4     else:
5         x = a[rand(left, right - 1)]
6         middle = split(left, right, x)
7         sort(left, middle)
8         if k < middle
9             return find(left, middle, k)
10        else
11            return find(middle, right, k);

```

Таким образом, можно найти  $k$ -ую порядковую статистику без сортировки массива.

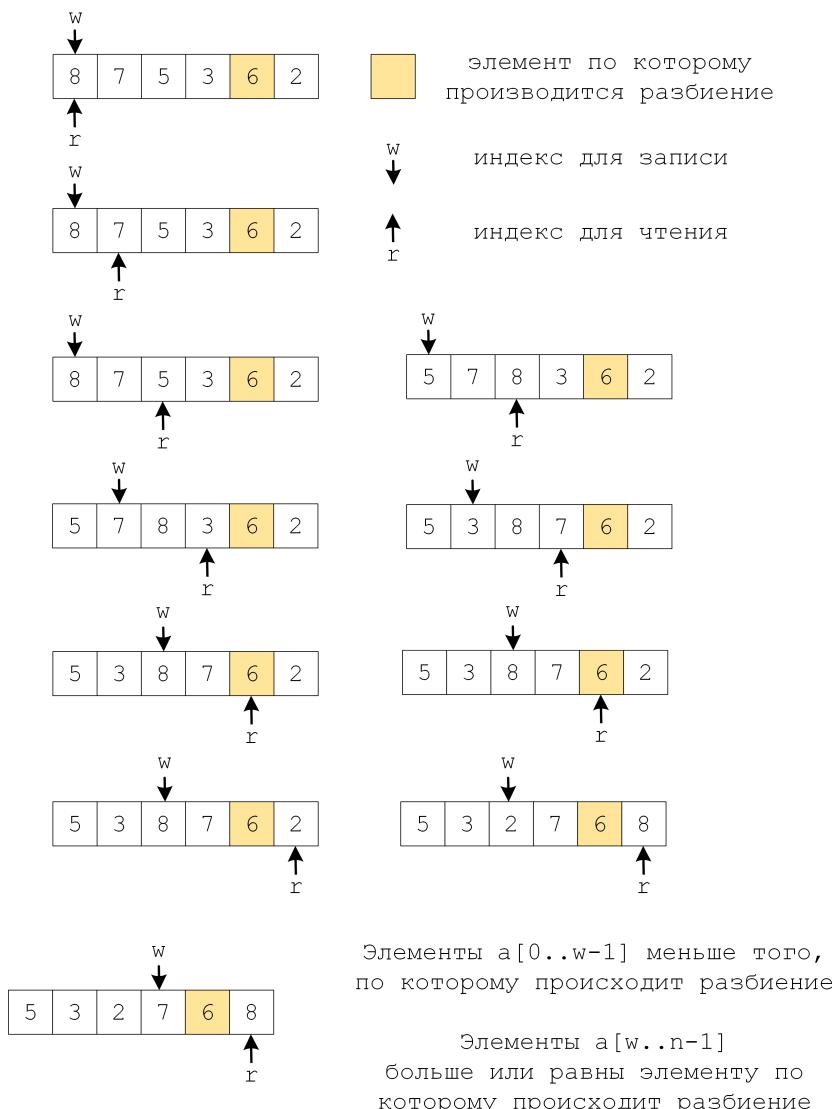


Рис. 12.19: Разбиение последовательности функцией *split*

<sup>2</sup> **$k$ -ой порядковой статистикой** называется элемент, который будет стоять на  $k$ -ом месте в отсортированном массиве.

# Глава 13

## Система непересекающихся множеств

Пусть задано  $N$  множеств. Изначально каждое множество состоит из одного элемента. Назовём **лидером множества** некоторый элемент, который принадлежит данному множеству<sup>1</sup>. Пусть  $N$  равно 6. Графически можно представить так:

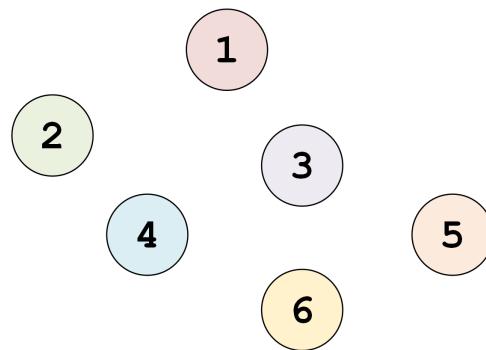


Рис. 13.1: Шесть множеств из одного элемента. Каждый элемент является лидером своего множества

Возникла необходимость в разработке такой структуры данных, которая позволяла бы выполнять следующие операции:

- `union(x, y)` – Объединять два множества, к которым принадлежат элементы  $x$  и  $y$ .
- `get(x)` – Определять множество, в котором лежит элемент  $x$ .

При этом:

1. Множества не имеют повторяющихся элементов.
2. После объединения множеств элементы не могут его покинуть.

Когда поступит запрос на определение множества, в которому принадлежит элемент  $x$ , мы будем возвращать лидера множества (см рисунок 13.2). Элементы 1, 2, 4 принадлежат множеству с лидером 1. Элемент 3 принадлежит к множеству с лидером 3 и состоит из одного элемента. Элементы 5, 6 принадлежат к множеству с лидером 6. Запрос `get(4)` должен возвращать значение 1, а `get(5)` - значение 6.

<sup>1</sup>Принадлежность элемента множеству будем обозначать цветом. Лидеры множества выделены **полужирным** начертанием

### 13.1 Наивный подход

Самая простая идея заключается в следующем: создадим массив  $p$ , где  $i$ -й элемент будет хранить значение лидера множества для элемента  $i$ . Например, для рисунка 13.2 можно было бы создать следующий массив:

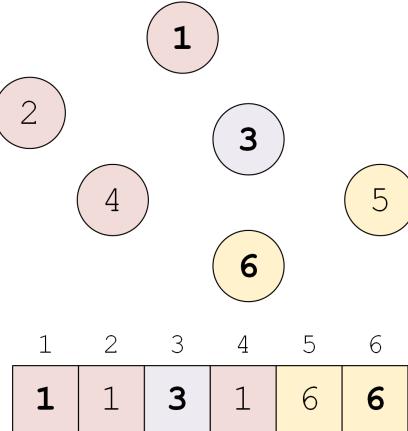


Рис. 13.2: Три множества и значения массива  $p$

Определить лидеров просто: если  $p[i] = i$ , значит  $i$  является лидером. Чтобы найти лидера множества для элемента, к которому принадлежит  $x$  достаточно узнать значение  $p[i]$ . Данный запрос работает за  $O(1)$ .

Очевидным является факт, что когда каждый элемент представляет собой множество из одного элемента, массив  $p$  выглядит так:

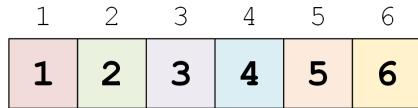


Рис. 13.3: Инициализация массива  $p$

Процедура объединения выглядит сложнее. Предположим, мы хотим объединить множества, к которым принадлежат элементы 2 и 6. Тогда необходимо найти лидера множества 2 (значение 1) и лидера множества 6 (значение 6) и заменить в массиве  $p[i]$  все единицы на шестёрки или шестёрки на единицы:

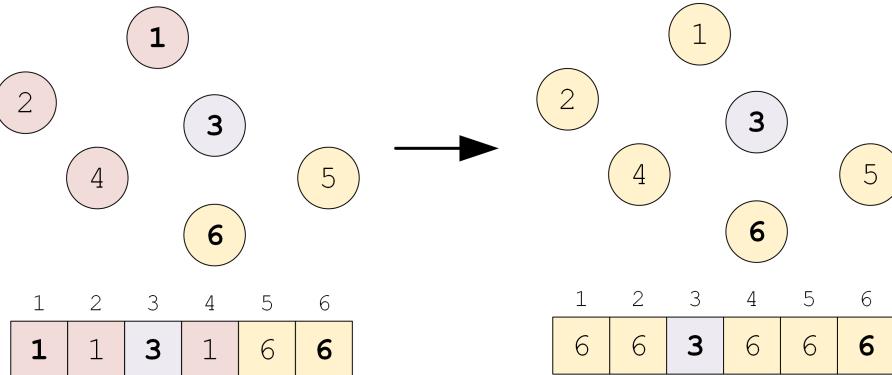


Рис. 13.4: Выполнение операции  $union(2, 6)$

Сложность такого алгоритма объединения  $O(N)$ . Рассмотрим ряд модификаций, которые могут ускорить решение.

## 13.2 Модификация №1 - Переход к спискам

Можно для каждого элемента-лидера хранить список принадлежащих ему элементов. Тогда при объединении двух множеств элементы одного списка будут добавлены к элементам другого списка. Рассмотрим на примере. Было 6 множеств, над которыми последовательно выполнялись запросы. Изменения списков:

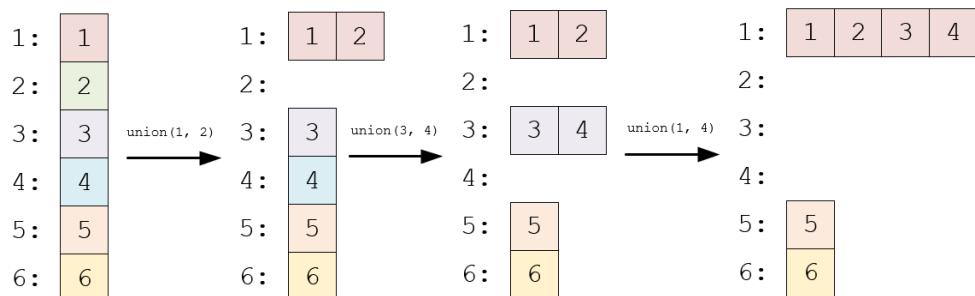


Рис. 13.5: Процесс объединения множеств

Идеально это выглядит так:

---

```

1 union(x, y):
2     x = get(x)
3     y = get(y)
4     for i : list[x]:
5         p[i] = y           // обновляем необходимые элементы массива p
6         list[y].add(i)    // перекидываем элементы из списка
7     list[x] = []          // опустошаем список

```

---

В общем случае хочется, чтобы объединение списков происходило быстро. И было бы хорошей идеей дописывать к списку большей длины список меньшей длины. В примере на рисунке 13.6 является более предпочтительным первый вариант, так как он потребует всего лишь одно обновление в массиве  $p$  вместо двух.

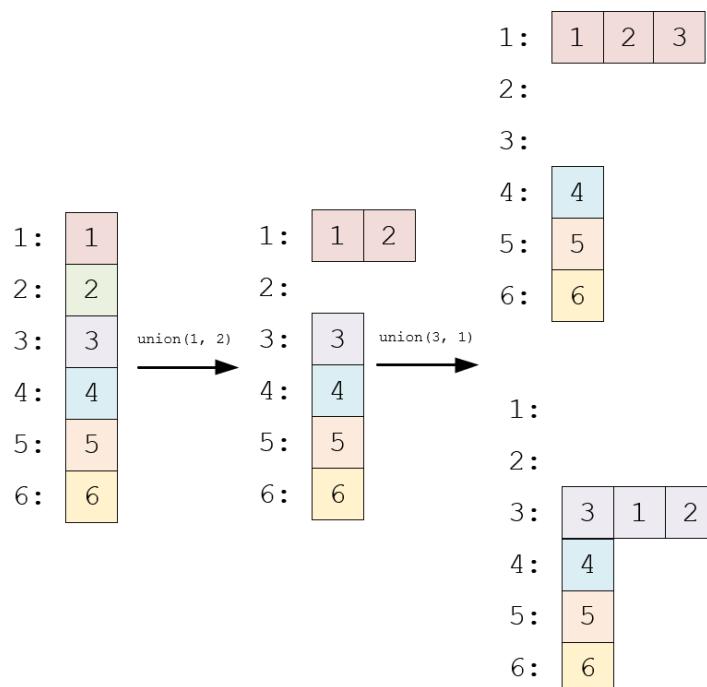


Рис. 13.6: Варианты объединения

В одном случае придётся обновить один элемент массива  $p$  (с индексом 3), в другом – два элемента (с индексами 1 и 2).

Если учесть размеры списка, получится:

---

```

1 union(x, y):
2     x = get(x)
3     y = get(y)
4     if list[x].size() > list[y].size():
5         swap(x, y)
6     for i : list[x]:
7         p[i] = y
8         list[y].add(i)
9     list[x] = []

```

---

Амортизированное время на операцию объединения в таком случае займёт  $O(\log N)$ .

### 13.3 Модификация №2 - Переход к деревьям

Представим множества в виде деревьев. В качестве корня будем хранить лидера множества:

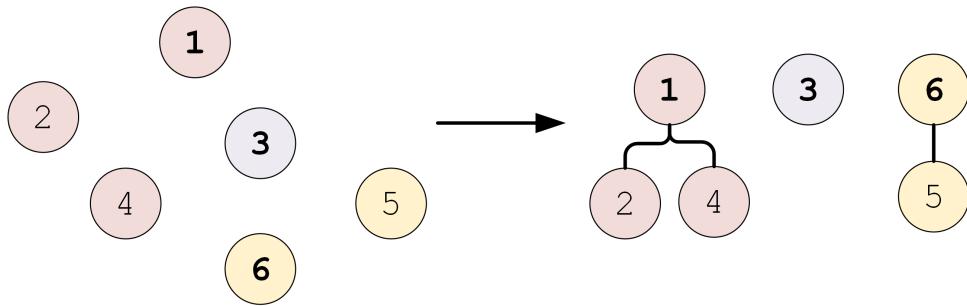


Рис. 13.7: Представление в виде дерева

Теперь в качестве элементов массива  $p$  будут выступать следующие:  $p[i]$  – родитель  $i$ -ого элемента дерева. Если  $p[i] = i$  значит  $i$  – лидер множества. Массив для прошлого рисунка:

1	2	3	4	5	6
<b>1</b>	1	<b>3</b>	1	6	<b>6</b>

Чтобы найти идентификатор множества для элемента  $x$  будем подниматься до корня:

---

```

1 get(x):
2     while (p[x] != x)
3         x = p[x]
4     return x;

```

---

Больший интерес представляет операция объединения. Выполнить это можно довольно просто: необходимо к корню одного дерева подвесить другой корень (очевидно, что для объединяемых множеств необходимо прежде всего найти их лидеров):

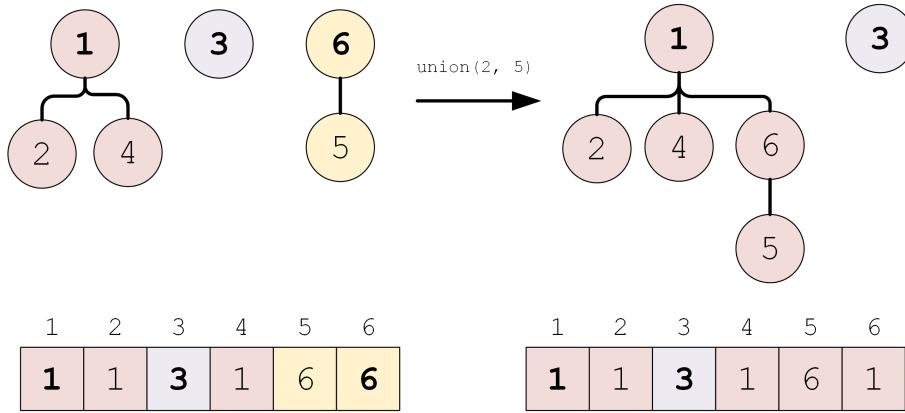


Рис. 13.8: Объединение множеств

Опишем решение псевдокодом:

---

```

1 get(x):
2     id1 = get(x)
3     id2 = get(y)
4     p[id1] = id2

```

---

Проблема такой реализации состоит в том, что, если элементы выстраиваются последовательно, и будут выполнены запросы *get*, время будет линейным. Для борьбы этим будем учитывать высоту дерева, и подвешивать к более высокому дереву более низкое дерево:

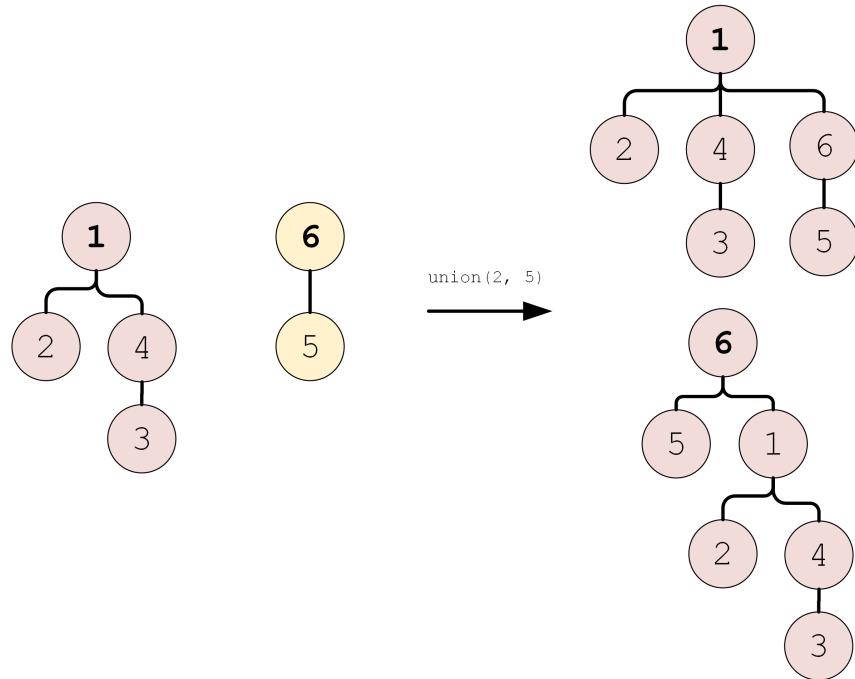


Рис. 13.9: Варианты объединения множеств.

Введём дополнительный массив *r*, который будет определять высоту дерева (ранг). При объединении множеств ищем лидеров, и сравниваем их ранги. Если ранг одного из деревьев больше, значит, оно является более высоким, и к нему нужно другое дерево. Если ранги равны, не так и важно, кого с кем объединять. После объединения необходимо обновить ранги. Процесс описан на рисунке 13.10.

Опишем объединение:

---

```

1 union(x, y):
2     id1 = get(x)
3     id2 = get(y)
4     // сделаем так, чтобы r[id1] стал меньше или равен r[id2]
5     if r[id1] > r[id2]:
6         swap(id1, id2)
7     // привязываем к низкому дереву высокое
8     p[id1] = id2
9     if (r[id1] == r[id2])
10        r[id2]++

```

---

и получение идентификатора:

---

```

1 get(x):
2     while (x != p[x]):
3         x = p[x]
4     return x

```

---

Можете ознакомиться с примером объединения множеств:

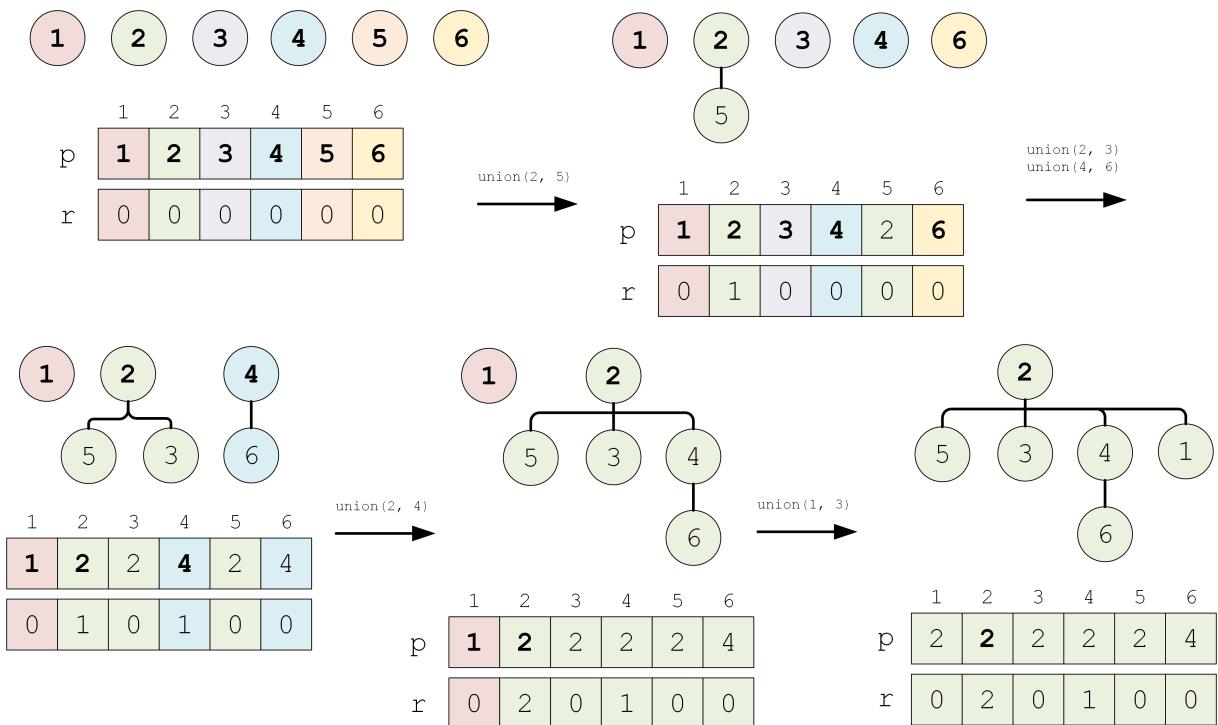


Рис. 13.10: Объединение множеств

Можно найти ещё один способ ускорения времени работы функции `get`. Будем вычислять лидера рекурсивно. В процессе рекурсивного спуска ищем лидера, в процессе подъёма (возвращаясь к исходному элементу) – обновим ссылки на родителя.

---

```

1 get(x):
2     if p[x] != x:
3         p[x] = get(p[x])
4     return p[x]

```

---

После выполнения функции, родителем будет не тот, кто непосредственно выше, а лидер.

В процессе данных вычислений будет уменьшаться высота дерева (рисунок 13.11). Такая эвристика называется эвристикой сжатия путей. Благодаря данным подходам

операции `get` и `union` работают за обратную функцию Аккермана  $O(\alpha(m, n))$ , где  $m$  – число выполненных операций `get` и  $n$  – число элементов.

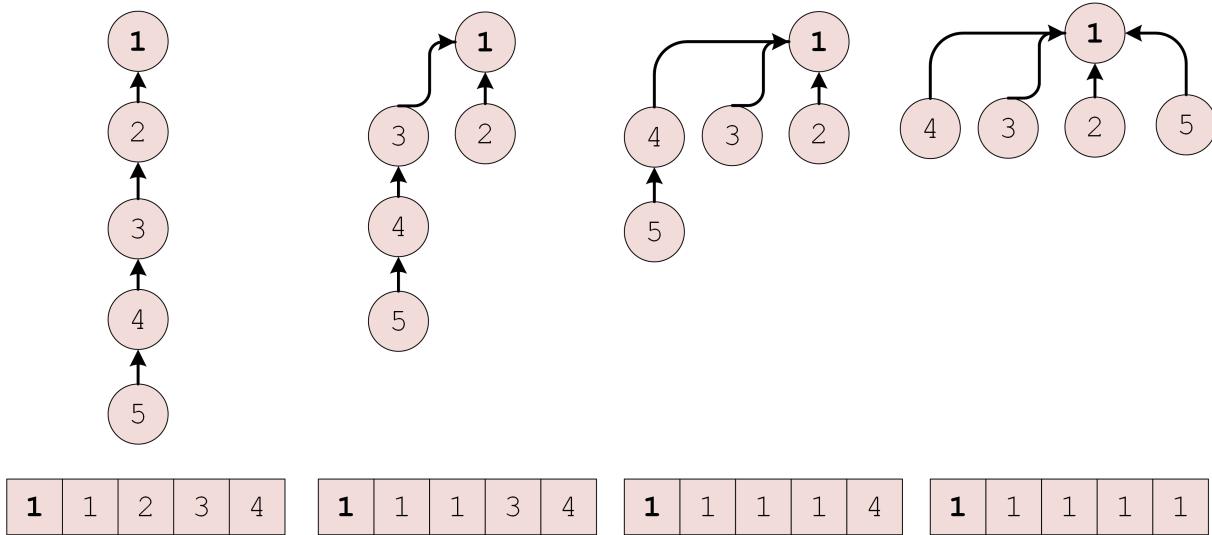


Рис. 13.11: Рекурсивный подъем функции `get`

Помимо просто поддержания множеств как таковых, можно поддерживать ассоциативные и коммутативные функции на них. Операция  $\otimes$  называется **ассоциативной**, если её результат не зависит от того, в каком порядке ее вычислять, то есть если  $(a \otimes b) \otimes c = a \otimes (b \otimes c)$ . Операция  $\otimes$  называется **коммутативной**, если её результат не зависит от перестановки оперируемых, то есть если  $a \otimes b = b \otimes a$ . Например, можно считать сумму или минимум на всех элементах множества. Тогда код `union` получается следующий. Естественно, массивы `sum` и `min` нужно правильно инициализировать:

---

```

1 union(x, y):
2     x = get(x)
3     y = get(y)
4     if r[x] > r[y]:
5         swap(x, y)
6     p[x] = y
7     if (r[x] == r[y]):
8         r[y]++
9         sum[y] += sum[x];
10        // min[y] = min(min[x], min[y])

```

---

## 13.4 Задача про людей

Допустим, имеется  $n$  человек, которые стоят в одной шеренге. Имеется два вида запросов:

- `delete(i)` – убрать  $i$ -го человека из шеренги.
- `get_first(i)` – возвращает человека с позицией не меньше  $i$  или -1 если за позицией  $i$  никого нет.

Оптимизировать процесс объединения множеств не получится: нам нужно удаляемое значение присоединить к последующему, ранговая эвристика не будет работать.

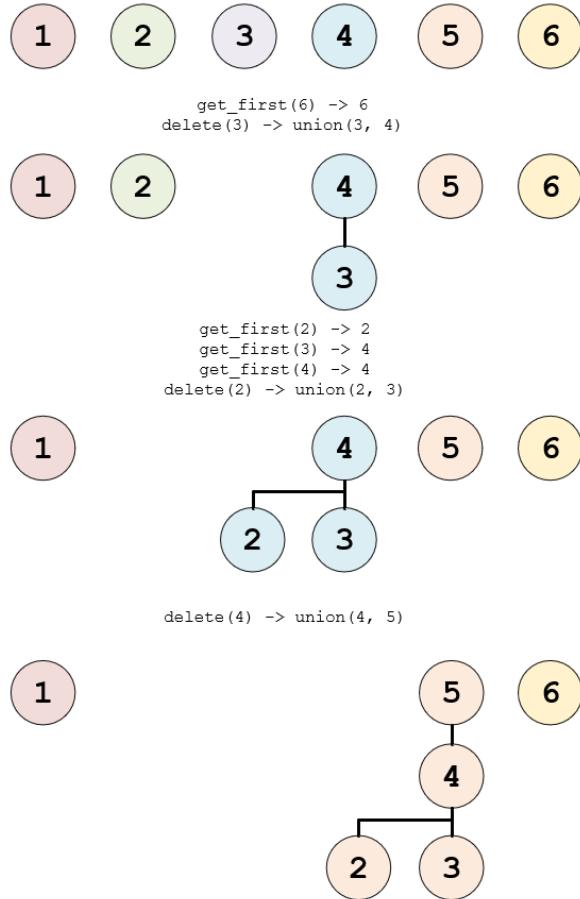


Рис. 13.12: Примеры запросов

Но за счёт эвристики сжатия пути можно добиться амортизированного  $O(\log N)$  на запрос<sup>2</sup>.

### 13.5 Алгоритм Краскала

**Алгоритм Краскала** — эффективный алгоритм построения минимального остовного дерева<sup>3</sup>. Пример минимального остовного дерева представлен на рисунке 13.13.

<sup>2</sup>Хорошей практикой было бы добавление фиктивного элемента. Например, при удалении элемента 6 (который являлся последним), присоединять всё дерево к элементу 7. Если при запросе `get_first` получалось значение 7 – выводим ответ -1.

<sup>3</sup>**Минимальное остовное дерево** (или минимальное покрывающее дерево) в связанным взвешенным неориентированным графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер. **Остовное дерево графа** — это дерево, подграф данного графа, с тем же числом вершин, что и у исходного графа. Неформально говоря, остовное дерево получается из исходного графа удалением максимального числа рёбер, входящих в циклы, но без нарушения связности графа. Остовное дерево включает в себя все  $n$  вершин исходного графа и содержит  $n - 1$  ребро взвешенного связного неориентированного графа.

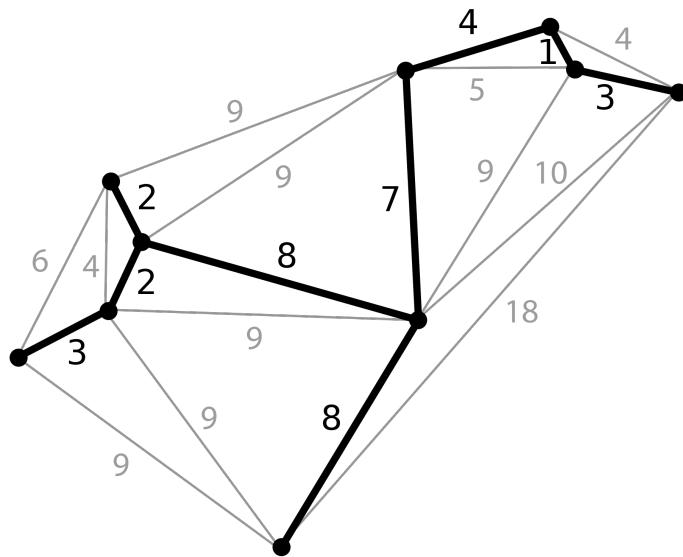
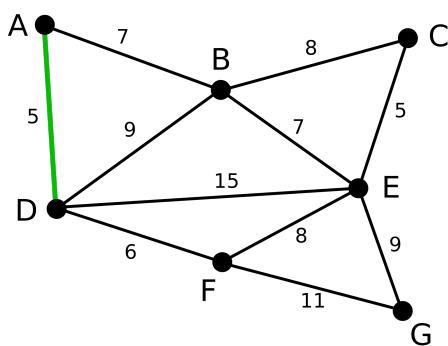


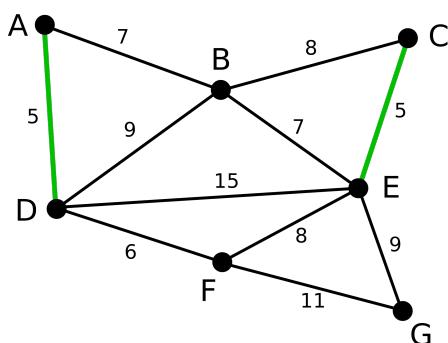
Рис. 13.13: Минимальное оствовное дерево.

Данный алгоритм является жадным<sup>4</sup>. Если отсортировать рёбра по неубыванию длин, последовательно добавлять их в граф при условии, что оно соединяет две вершины, между которыми ещё нет построенного пути, при обработке всех рёбер будет получено минимальное оствовное дерево. Например, решение такой задачи может помочь минимизировать общую протяженность прокладываемых дорог для связи населённых пунктов между собой.

Рассмотрите последовательность шагов на примере:

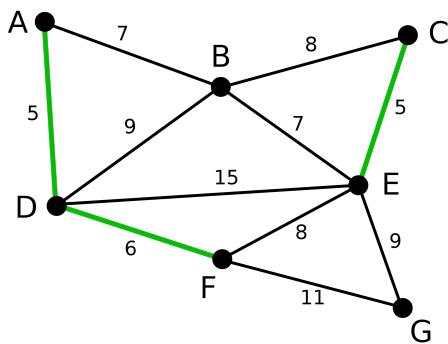


Имеем два ребра длины 5. Добавим ребро  $AD$ .

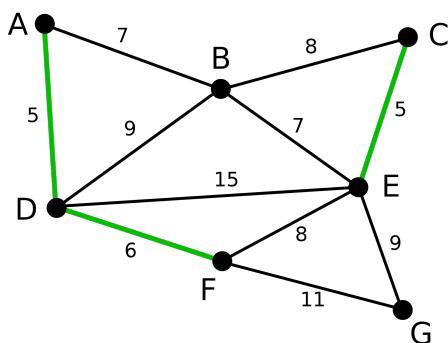


Следующее рассматриваемое ребро -  $CE$ . Так как оно соединяет вершины, между которыми ещё нет пути (путь пока что имеется только из вершины  $A$  в вершину  $D$ ), добавляем ребро.

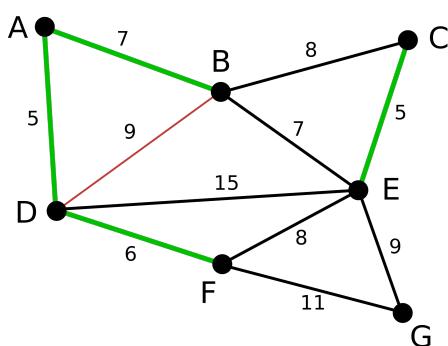
<sup>4</sup>Жадный алгоритм – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.



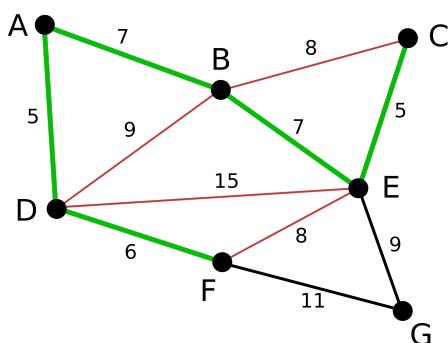
Рассмотрим ребро  $DF$ . Так как оно соединяет вершины, между которыми нет пути, добавляем ребро в дерево.



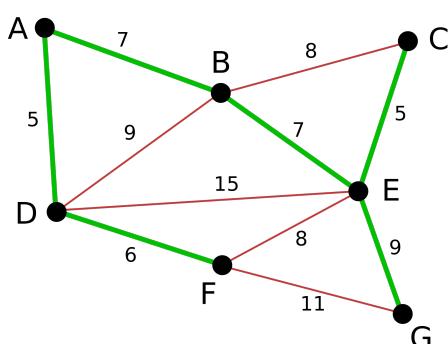
Рассмотрим ребро  $DF$ . Так как оно соединяет вершины, между которыми нет пути, добавляем ребро в дерево.



Рассмотрим ребро  $AB$ . Так как оно соединяет вершины, между которыми нет пути, добавляем ребро в дерево. Если когда-то мы будем рассматривать ребро  $BD$ , его не будет смысла добавлять в минимальное оставное дерево, так как между вершинами  $B$  и  $D$  имеется путь  $B - A - D$ .



Следующее добавляемое ребро –  $BE$ . После его добавления отпадает необходимость в рёбрах  $BC$ ,  $DE$ ,  $FE$ .



Алгоритм завершается добавлением ребра  $EG$  с весом 9.

Псевдокод решения задачи:

---

```
1 // сортировка ребёр по их длине
2 sort(edges)
3
4 // w - суммарный вес рёбер
5 w = 0
6 for e in edges:
7     if get(e.u) != get(e.v):
8         union(e.u, e.v)
9         w += e.w
```

---

Так как операции `get` и `union` работают крайне быстро, можем сказать, что основная нагрузка алгоритма выпадает на сортировку ребёр. Таким образом, время работы алгоритма составит  $O(E \log E)$ , где  $E$  – количество рёбер.

# Глава 14

## Множества. Представление числовых множеств в ЭВМ

Введём некоторые определения. **Структура данных** — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс.

Рассмотрим какую-нибудь известную вам СД, например массив. У него имеется две операции:

- `get(i)` - получить по индексу `i`
- `put(i, x)` - записать по индексу `i` значение `x`

Будем последовательно изучать структуры данных и организовывать написанные решения в библиотеку.

### 14.1 Организация библиотек в языке программирования С

Всё, что пишется, лучше собирать в библиотеки, чтобы можно было повторно использовать необходимые участки кода. Будем руководствоваться следующим: книга Никлауса Вирта называлась Алгоритмы + Структуры данных = Программы. Поэтому разделим всю нашу библиотеку на две составляющие. Первая – алгоритмы, вторая – структуры данных. Создадим папку `libs` с поддиректориями:

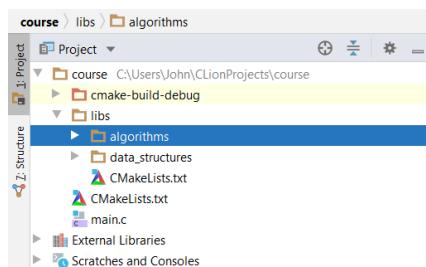
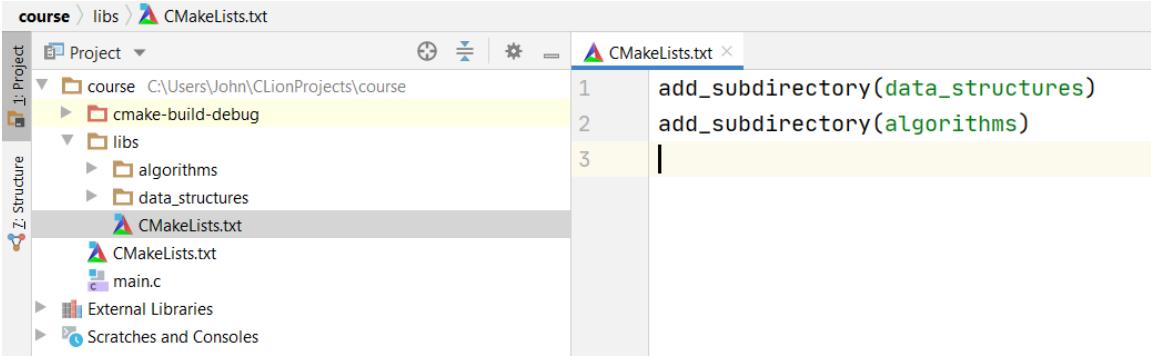


Рис. 14.1: Структура проекта

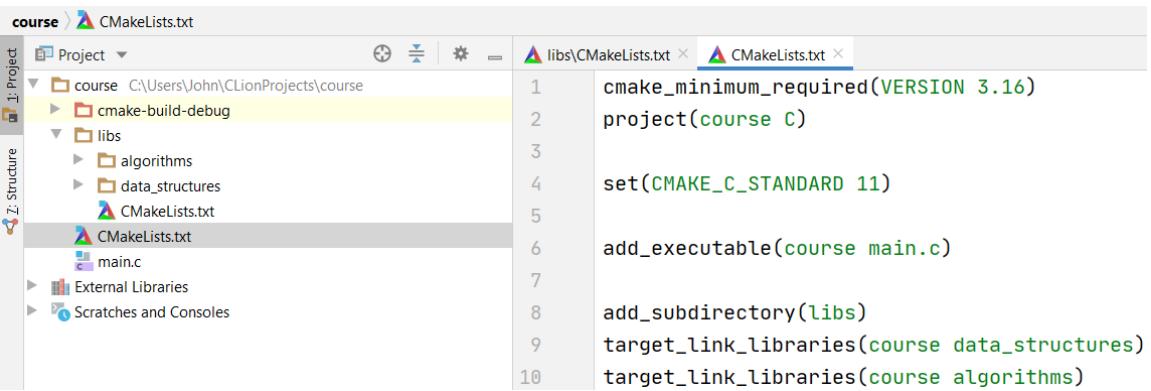
В *libs/CMakeLists* добавим наши поддиректории:



```
course > libs > CMakeLists.txt
Project 1: course
  course C:\Users\John\CLionProjects\course
    cmake-build-debug
    libs
      algorithms
      data_structures
      CMakeLists.txt
        CMakeLists.txt
        main.c
    External Libraries
    Scratches and Consoles
```

```
1 add_subdirectory(data_structures)
2 add_subdirectory(algorithms)
```

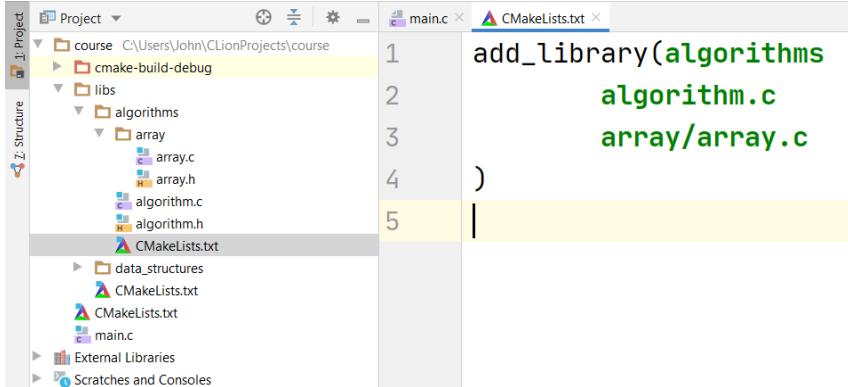
Обновим *CMakeLists* проекта:



```
course > CMakeLists.txt
Project 1: course
  course C:\Users\John\CLionProjects\course
    cmake-build-debug
    libs
      algorithms
      data_structures
      CMakeLists.txt
        CMakeLists.txt
        main.c
    External Libraries
    Scratches and Consoles
```

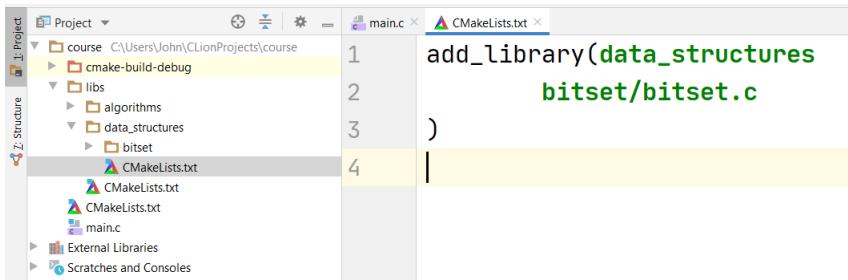
```
1 cmake_minimum_required(VERSION 3.16)
2 project(course C)
3
4 set(CMAKE_C_STANDARD 11)
5
6 add_executable(course main.c)
7
8 add_subdirectory(libs)
9 target_link_libraries(course data_structures)
10 target_link_libraries(course algorithms)
```

В директориях *algorithms* и *data\_structures* создадим свои *CMakeLists* файлы и выполним добавление .c файлов:



```
course > libs > algorithms > array > CMakeLists.txt
Project 1: course
  course C:\Users\John\CLionProjects\course
    cmake-build-debug
    libs
      algorithms
        array
          array.c
          array.h
          algorithm.c
          algorithm.h
        CMakeLists.txt
      data_structures
      CMakeLists.txt
      main.c
    External Libraries
    Scratches and Consoles
```

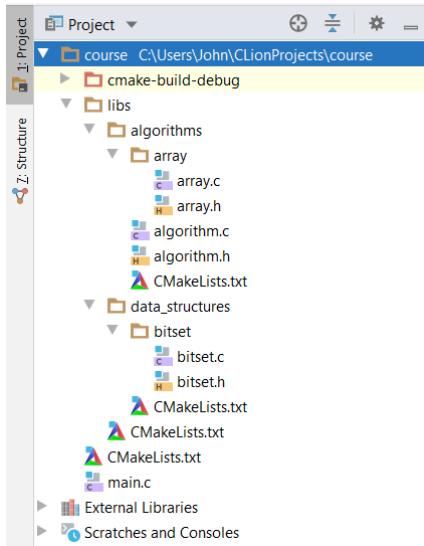
```
1 add_library(algorithms
2   algorithm.c
3   array/array.c
4 )
5 |
```



```
course > libs > data_structures > bitset > CMakeLists.txt
Project 1: course
  course C:\Users\John\CLionProjects\course
    cmake-build-debug
    libs
      algorithms
      data_structures
        bitset
        CMakeLists.txt
      CMakeLists.txt
      main.c
    External Libraries
    Scratches and Consoles
```

```
1 add_library(data_structures
2   bitset/bitset.c
3 )
4 |
```

Итоговая структура проекта:



представлена заголовочными файлами и файлами с реализацией. По мере изучения материалов, библиотека будет пополняться новыми функциями. Начнём с наполнения библиотеки для работы с массивами. *array.h*-файл содержит заголовки функций:

```

1 #ifndef INC_ARRAY_H
2 #define INC_ARRAY_H
3
4 #include <stddef.h>
5
6 // ввод массива data размера n
7 void inputArray_(int *a, size_t n);
8 // вывод массива data размера n
9 void outputArray_(const int *a, size_t n);
10
11 // возвращает значение первого вхождения элемента x
12 // в массиве a размера n при его наличии, иначе - n
13 size_t linearSearch_(const int *a, const size_t n, int x);
14
15 // возвращает позицию вхождения элемента x
16 // в отсортированном массиве a размера n при его наличии, иначе - SIZE_MAX
17 size_t binarySearch_(const int *a, const size_t n, int x);
18 // возвращает позицию первого элемента равного или большего x
19 // в отсортированном массиве a размера n
20 // при отсутствии такого элемента возвращает n
21 size_t binarySearchMoreOrEqual_(const int *a, const size_t n, int x);
22
23 // вставка элемента со значением value
24 // в массив data размера n на позицию pos
25 void insert_(int *a, size_t *n, size_t pos, int value);
26 // добавление элемента value в конец массива data размера n
27 void append_(int *a, size_t *n, int value);
28 // удаление из массива data размера n элемента на позиции pos
29 // с сохранением порядка оставшихся элементов
30 void deleteByPosSaveOrder_(int *a, size_t *n, size_t pos);
31 // удаление из массива data размера n элемента на позиции pos
32 // без сохранения порядка оставшихся элементов
33 // размер массива data уменьшается на единицу
34 void deleteByPosUnsaveOrder_(int *a, size_t *n, size_t pos);
35

```

```

36 // возвращает значение 'истина' если все элементы
37 // массива data размера n соответствует функции-предикату predicate
38 // иначе - 'ложь'
39 int all_(const int* a, size_t n, int (*predicate)(int));
40 // возвращает значение 'истина' если хотя бы один элемент
41 // массива data размера n соответствует функции-предикату predicate
42 // иначе - 'ложь'
43 int any_(const int *a, size_t n, int (*predicate)(int));
44 // применяет функцию predicate ко всем элементам массива source
45 // размера n и сохраняет результат в массиве dest размера n
46 void forEach_(const int *source, int *dest, size_t n, const int (*
    predicate)(int));
47 // возвращает количество элементов массива data размера n
48 // удовлетворяющих функции-предикату predicate
49 int countIf_(const int * a, size_t n, int (*predicate)(int));
50
51 // удаляет из массива data размера n все элементы, удовлетворяющие
52 // функции-предикату deletePredicate, записывает в n новый размер
53 // массива
54 void deleteIf_(int * a, size_t * n, int (*deletePredicate)(int));
55
56 #endif

```

---

Файл имеет защиту от двойного включения:

---

```

1 #ifndef INC_ARRAY_H
2 #define INC_ARRAY_H
3
4 // ...
5
6 #endif

```

---

Это позволит безопасно подключать библиотеку в другие файлы. Содержание *array.c* файла в текущей реализации:

---

```

1 #include <stdio.h>
2 #include <limits.h>
3 #include <assert.h>
4 #include "array.h"
5
6
7 void inputArray_(int * const a, const size_t n) {
8     for (size_t i = 0; i < n; i++)
9         scanf("%d", &a[i]);
10 }
11
12 void outputArray_(const int * const a, const size_t n) {
13     for (size_t i = 0; i < n; i++)
14         printf("%d ", a[i]);
15     printf("\n");
16 }
17
18 void append_(int * const a, size_t * const n, const int value) {
19     a[*n] = value;
20     (*n)++;
21 }
22
23 void insert_(int * const a, size_t * const n, const size_t pos,
24               const int value) {
25     assert(pos < *n);
26     if (*n != 0) {
27         size_t lowBound = (pos == 0) ? SIZE_MAX : pos;

```

```

28     (*n)++;
29     for (size_t i = *n; i != lowBound; i--)
30         a[i] = a[i - 1];
31     a[pos] = value;
32 } else {
33     (*n)++;
34     a[pos] = value;
35 }
36 }

37 void deleteByPosSaveOrder_(int *a, size_t *n, const size_t pos) {
38     for (size_t i = pos; i < *n - 1; i++)
39         a[i] = a[i + 1];
40     (*n)--;
41 }

42 void deleteByPosUnsaveOrder_(int *a, size_t *n, size_t pos) {
43     a[pos] = a[*n - 1];
44     (*n)--;
45 }

46 size_t linearSearch_(const int *a, const size_t n, int x) {
47     for (size_t i = 0; i < n; i++)
48         if (a[i] == x)
49             return i;
50     return n;
51 }

52 int any_(const int *a, size_t n, int (*predicate)(int)) {
53     for (size_t i = 0; i < n; i++)
54         if (predicate(a[i]))
55             return 1;
56     return 0;
57 }

58 int all_(const int *a, size_t n, int (*predicate)(int)) {
59     for (size_t i = 0; i < n; i++)
60         if (!predicate(a[i]))
61             return 0;
62     return 1;
63 }

64 int countIf_(const int * const a, const size_t n, int (*predicate)(int))
65 {
66     int count = 0;
67     for (size_t i = 0; i < n; i++)
68         count += predicate(a[i]);
69     return count;
70 }

71 void deleteIf_(int * const a, size_t * const n, int (*deletePredicate)(int)) {
72     size_t iRead = 0;
73     while (iRead < *n && !deletePredicate(a[iRead]))
74         iRead++;
75     size_t iWrite = iRead;
76     while (iRead < *n) {
77         if (!deletePredicate(a[iRead])) {
78             a[iWrite] = a[iRead];
79             iWrite++;
80         }
81     }
82 }
83 
```

```

86         }
87         iRead++;
88     }
89     *n = iWrite;
90 }
91
92 void forEach_(const int *source, int *dest, const size_t n, const int (*
93     predicate)(int)) {
94     for (size_t i = 0; i < n; i++)
95         dest[i] = predicate(source[i]);
96 }
97
98 size_t binarySearch_(const int *a, size_t n, int x) {
99     size_t left = 0;
100    size_t right = n - 1;
101    while (left <= right) {
102        size_t middle = left + (right - left) / 2;
103        if (a[middle] < x)
104            left = middle + 1;
105        else if (a[middle] > x)
106            right = middle - 1;
107        else
108            return middle;
109    }
110    return SIZE_MAX;
111 }
112
113 size_t binarySearchMoreOrEqual_(const int *a, size_t n, int x) {
114     if (a[0] >= x)
115         return 0;
116     size_t left = 0;
117     size_t right = n;
118     while (right - left > 1) {
119         size_t middle = left + (right - left) / 2;
120         if (a[middle] < x)
121             left = middle;
122         else
123             right = middle;
124     }
125     return right;
126 }
```

---

Важно отметить следующий момент: модификатор `const` к данным может быть указан в заголовочном файле. Однако `const` к указателям является избыточным. Но в файле с реализацией он может быть поставлен.

Вы можете дополнить файлы своими функциями. Чтобы использовать библиотеки в своём проекте, достаточно выполнить их подключение:

---

```

1 #include "libs/algorithms/array/array.h"
2
3 int main() {
4     int a[5];
5     inputArray_(a, 5);
6     outputArray_(a, 5);
7     return 0;
8 }
```

---

## 14.2 Представление числовых множеств и мульти-множеств

Рассмотрим способы представления множеств. **Множеством** называется произвольный набор (совокупность, класс, семейство) каких-либо объектов. Объекты, входящие во множество, называются его **элементами**. Если объект  $x$  является элементом множества  $A$ , то говорят, что  $x$  принадлежит  $A$  и пишут  $x \in A$ .

Основные операции над множествами:

- **Объединением** множеств  $A$  и  $B$  называется множество  $C$ , состоящее из элементов множеств  $A$  и  $B$ :

$$\begin{aligned} C &= A \cup B = \{x, y : x \in A, y \in B\} \\ A &= \{1, 2, 3\} \quad B = \{3, 4, 5\} \quad C = A \cup B = \{1, 2, 3, 4, 5\} \end{aligned}$$

- **Пересечением** множеств  $A$  и  $B$  называется множество  $C$ , состоящее из элементов, которые принадлежат и множеству  $A$  и множеству  $B$ :

$$\begin{aligned} C &= A \cap B = \{x : x \in A \text{ и } x \in B\} \\ A &= \{1, 2, 3\} \quad B = \{3, 4, 5\} \quad C = A \cap B = \{3\} \end{aligned}$$

- **Разностью** множеств  $A$  и  $B$  называется множество  $C$ , состоящее из элементов, принадлежащих множеству  $A$ , но не принадлежащих множеству  $B$ :

$$\begin{aligned} A \setminus B &= \{x : x \in A \text{ и } x \notin B\} \\ A &= \{1, 2, 3\} \quad B = \{3, 4, 5\} \quad A \setminus B = \{1, 2\} \end{aligned}$$

- **Симметрической разностью** множеств  $A$  и  $B$  называется множество  $C$ , состоящее из элементов множества  $A$ , которых нет в  $B$  и элементов множества  $B$ , которых нет в  $A$ :

$$\begin{aligned} A \Delta B &= \{x : x \in A \text{ и } x \notin B \text{ или } x \in B \text{ и } x \notin A\} \\ A &= \{1, 2, 3\} \quad B = \{3, 4, 5\} \quad A \Delta B = \{1, 2, 4, 5\} \end{aligned}$$

В любой конкретной задаче приходится иметь дело только с подмножествами некоторого, фиксированного для данной задачи, множества. Его принято называть универсальным (универсумом) и обозначать символом  $U$ . Например, при сборке некоторого изделия универсальным множеством естественно назвать множество всех деталей и сборочных элементов, из которых это изделие состоит.

- **Дополнением** множества  $A$  до универсума называется множество:

$$\begin{aligned} \overline{A} &= \{x : x \in U \text{ и } x \notin A\} = U \setminus A \\ A &= \{1, 2, 3\} \quad U = \{1, 2, \dots, 10\} \quad \overline{A} = \{4, 5, 6, 7, 8, 9, 10\} \end{aligned}$$

Рассмотрим некоторые представления числовых множеств в памяти ЭВМ:

- На логическом массиве или массиве битов.
- На неупорядоченном массиве.
- На упорядоченном массиве.

Каждый из способов представления обладает своими преимуществами и недостатками, о которых станет понятно в процессе их реализации.

### 14.2.1 Реализация множества на массиве битов

Существует простой способ реализовать множества и операции над ними: использовать побитовые операции. Рассмотрим множество, универсумом которого являются числа от 0 до 25. Для множества будем использовать лишь одну 32-битовую переменную. Если значение  $i$ -го бита равняется единице – элемент есть в множестве, иначе – отсутствует. Пример:



Рис. 14.2: Представление множества в памяти ЭВМ.

К сожалению, тип `int` не является фиксированным в вопросах размера, а для нашего приложения, это было бы важно. В случае смены платформы, размер `int` может измениться на 16 бит, и написанные программы с такими множествами станут некорректными. Описания типов данных с фиксированным размером содержатся в `stdint.h`.

Язык С позволяет определять имена новых типов данных с помощью ключевого слова `typedef`:

---

<sup>1</sup> // `typedef` тип имя;

---

где `тип` – это любой существующий тип данных, а `имя` – это новое имя для данного типа. На самом деле здесь не создается новый тип данных, а определяется новое имя существующему типу.

Фрагмент файла `stdint.h`:

---

```

1 typedef signed char int8_t;
2 typedef unsigned char uint8_t;
3 typedef short int16_t;
4 typedef unsigned short uint16_t;
5 typedef int int32_t;
6 typedef unsigned uint32_t;
7 _MINGW_EXTENSION typedef long long int64_t;
8 _MINGW_EXTENSION typedef unsigned long long uint64_t;

```

---

Мы будем использовать побитовые операции для реализации операций над множествами, и в условиях задачи тип `uint32_t` вполне подходит.

#### Добавление элемента в множество

Для добавления элемента в множество, необходимо выставить бит для соответствующего элемента в единицу:

$$A \cup \{x\} \Rightarrow A := A \mid (1 \ll x)$$

Неиспользуемые биты										Используемые биты для представления множества													
31	30	29	28	27	26	25	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
$A$	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	1	0	1	1	1	1	0	
$A \cup \{8\}$	0	0	0	0	0	0	0	...	0	0	0	1	0	1	0	1	0	1	1	1	1	0	

$$A \cup \{8\} = \{1, 2, 3, 4, 6, 10\} \cup \{8\} = \{1, 2, 3, 4, 6, 8, 10\}$$

### Удаление элемента из множества

Для удаления элемента из множества, необходимо выставить бит для соответствующего элемента в ноль:

$$A \setminus \{x\} \Rightarrow A := A \& \sim(1 \ll x)$$

Неиспользуемые биты										Используемые биты для представления множества													
31	30	29	28	27	26	25	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
$A$	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	1	0	1	1	1	1	0	
$A \setminus \{6\}$	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	0	0	0	1	1	1	1	0

$$A \setminus \{6\} = \{1, 2, 3, 4, 6, 10\} \setminus \{6\} = \{1, 2, 3, 4, 10\}$$

### Объединение

Объединение множеств реализуется за счёт применения побитового ИЛИ:

$$A \cup B \Rightarrow A | B$$

Неиспользуемые биты										Используемые биты для представления множества													
31	30	29	28	27	26	25	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
$A$	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	1	0	1	1	1	1	0	
$B$	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0
$A \cup B$	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	1	1	1	1	1	1	1	0

$$A \cup B = \{1, 2, 3, 4, 6, 10\} \cup \{1, 4, 5, 6\} = \{1, 2, 3, 4, 5, 6, 10\}$$

## Пересечение

Пересечение множеств реализуется через побитовое И:

$$A \cap B \Rightarrow A \& B$$

	Неиспользуемые биты										Используемые биты для представления множества														
	31	30	29	28	27	26	25	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
A	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0				
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0			
$A \cap B$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0			

$$A \cap B = \{1, 2, 3, 4, 6, 10\} \cap \{1, 4, 5, 6\} = \{1, 4, 6\}$$

## Разность

$$A \setminus B \Rightarrow A \& \sim B$$

	Неиспользуемые биты										Используемые биты для представления множества														
	31	30	29	28	27	26	25	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
A	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0				
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0			
$A \setminus B$	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0			

$$A \setminus B = \{1, 2, 3, 4, 6, 10\} \setminus \{1, 4, 5, 6\} = \{2, 3, 10\}$$

## Симметрическая разность

$$A \Delta B \equiv (A \setminus B) \cup (B \setminus A) \Rightarrow A \wedge B$$

	Неиспользуемые биты										Используемые биты для представления множества														
	31	30	29	28	27	26	25	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
A	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0				
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0			
$A \Delta B$	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0				

$$A \Delta B = \{1, 2, 3, 4, 6, 10\} \Delta \{1, 4, 5, 6\} = \{2, 3, 5, 10\}$$

## Дополнение

$$\overline{A} \Rightarrow \sim A$$



$$\overline{A} = U \setminus \{1, 2, 3, 4, 6, 10\} = \{0, 5, 7, 8, 9, 11, \dots, 25\}$$

Следует обратить внимание, что после применения операции инверсии, изменяются и неиспользуемые биты. Поэтому в данной реализации нельзя выполнять сравнение множеств  $A$  и  $B$ , как  $A == B$ . Придётся или модифицировать операцию дополнения сбрасыванием неиспользуемых битов, или усложнить операцию сравнения. Остановимся на первом варианте. Пусть  $n$  - количество неиспользуемых бит в представлении множества, тогда операция дополнения:

$$\overline{A} \Rightarrow (\sim A \ll n) \gg n$$

## Проверка множеств на равенство

$$A = B \Rightarrow A == B$$

## Проверка на то, что множество $B$ нестрого включает множество $A$

$$A \subseteq B \Rightarrow B \& A == A$$

## Проверка на то, что множество $B$ строго включает множество $A$

$$A \subset B \equiv A \subseteq B \text{ и } A \neq B \Rightarrow (B \& A == A) \&& (A != B)$$

Реализация может быть следующей. Оформление выполним в виде библиотеки.

Заголовочный файл:

---

```

1 #ifndef INC_BITSET_H
2 #define INC_BITSET_H
3
4 #include <stdint.h>
5 #include <stdbool.h>
6
7 typedef struct bitset {
8     uint32_t values;      // множество
9     uint32_t maxValue;   // максимальный элемент универсума
10 } bitset;
11
12 // возвращает пустое множество с универсумом 0, 1, ..., maxValue

```

```

13 bitset bitset_create(unsigned maxValue);
14
15 // возвращает значение 'истина', если значение value имеется в множестве set
16 // иначе - 'ложь'
17 bool bitset_in(bitset set, unsigned value);
18
19 // возвращает значение 'истина', если множества set1 и set2 равны
20 // иначе - 'ложь'
21 bool bitset_isEqual(bitset set1, bitset set2);
22
23 // возвращает значение 'истина' если множество subset
24 // является подмножеством множества set, иначе - 'ложь'.
25 bool bitset_isSubset(bitset subset, bitset set);
26
27 // добавляет элемент value в множество set
28 void bitset_insert(bitset *set, unsigned value);
29
30 // удаляет элемент value из множества set
31 void bitset_deleteElement(bitset *set, unsigned value);
32
33 // возвращает объединение множеств set1 и set2
34 bitset bitset_union(bitset set1, bitset set2);
35
36 // возвращает пересечение множеств set1 и set2
37 bitset bitset_intersection(bitset set1, bitset set2);
38
39 // возвращает разность множеств set1 и set2
40 bitset bitset_difference(bitset set1, bitset set2);
41
42 // возвращает симметрическую разность множеств set1 и set2
43 bitset bitset_symmetricDifference(bitset set1, bitset set2);
44
45 // возвращает дополнение до универсума множества set
46 bitset bitset_complement(bitset set);
47
48 // вывод множества set
49 void bitset_print(bitset set);
50
51 #endif

```

---

Файл реализации приведём частично, сфокусировавшись на пояснении некоторых функций:

```

1 #include <stdio.h>
2 #include <assert.h>
3
4 #include "bitset.h"
5
6 int bitset_checkValue(bitset *a, unsigned value) {
7     return value >= 0 && value <= a->maxValue;
8 }
9
10 bitset bitset_create(unsigned set.MaxValue) {
11     assert(set.MaxValue < 32);
12     return (bitset) {0, set.MaxValue};
13 }
14
15 // ...

```

---

В примере выше можем заметить использование макроса `assert`, определенного в `assert.h`. Если выражение внутри `assert` будет ложно, будет выдана ошибка.

Например, вы вызове

---

```
1 bitset a = bitset_create(50);
```

---

будет выдана ошибка:

```
C:\Users\John\CLionProjects\course\cmake-build-debug\course.exe
Assertion failed!

Program: C:\Users\John\CLionProjects\course\cmake-build-debug\course.exe
File: C:\Users\John\CLionProjects\course\libs\data_structures\bitset\bitset.c, Line 11

Expression: max < 32
```

Использование `assert` позволяет проверять некоторые предусловия перед выполнением тела функции. Какой-нибудь программист может надеяться на то, что наши множества обрабатывают значения до 50 включительно. И если не добавить `assert(max < 32)` он будет верить, что всё нормально: программа же не выкинула никаких ошибок. Это поможет избавиться от ошибок, вызванных неправильным использованием библиотеки.

Для функции вычисления объединения двух множеств потребуем, чтобы операция могла быть осуществлена только в случае, если универсумы совпадают:

---

```
1 bitset bitset_intersection(bitset set1, bitset set2) {
2     assert(set1 maxValue == set2 maxValue);
3     return (bitset){set1.values & set2.values, set1 maxValue};
4 }
```

---

Последний интересный момент можно заметить при выводе сообщения:

---

```
1 void bitset_print(bitset set) {
2     printf("{");
3     int isEmpty = 1;
4     for (int i = 0; i <= set maxValue; ++i) {
5         if (bitset_in(set, i)) {
6             printf("%d, ", i);
7             isEmpty = 0;
8         }
9     }
10    if (isEmpty)
11        printf("}\n");
12    else
13        printf("\b\b}\n");
14 }
```

---

Символ `\b` удаляет последний выводимый символ. Более того, функция вызывает внутри функцию проверки наличия элемента в множестве. Страйтесь использовать ранее написанные фрагменты и не допускать дублирования.

Особенности данного способа:

- Подходит только для малых по мощности числовых множеств с небольшим разбросом возможных значений.
- Операции над множествами работают крайне быстро, легко реализуются и имеют сложность  $O(1)$ .

## 14.2.2 Реализация множества и мульти множества на массиве типа `char`

Иногда возникают задачи, в которых требуется реализовать множество элементов на логическом массиве. В стандарте C90 не имеется логического типа. Для этих целей можно использовать тип `char`, который занимает 1 байт и для представления множества из  $N$  элементов потребуется  $N$  байт<sup>1</sup>. Данный способ часто используется для представления множеств в олимпиадном программировании.

### Задача о подсчёте букв

С клавиатуры вводятся строка из символов латинского алфавита (строчных). Необходимо определить, сколько различных символов содержала последовательность, которые встретились хотя бы раз.

Сложность по времени:  $O(n)$ .

Можно создать массив `letterset` из 26 элементов (латинский алфавит содержит 26 букв), каждый из которых проинициализируем нулём. Если мы встретили букву  $a$  – выставим `letterset[0]` в единицу, встретим  $b$  – выставим `letterset[1]` в единицу и т.д. По окончанию обработки символьной последовательности мы получим массив, где  $i$ -ый элемент отвечает: встречалась ли  $i$ -ая буква латинского алфавита в последовательности. Если встречалась, то значение элемента равняется 1, иначе – 0.

```

1 #include <stdio.h>
2
3 #define ABC_POWER 26
4
5 void getLetterset(const char *string, char *letterset) {
6     int i = 0;
7     while (string[i] != '\0') {
8         letterset[string[i] - 'a'] = 1;
9         i++;
10    }
11 }
12
13 int countNonZero(const char *a, int n) {
14     int res = 0;
15     for (int i = 0; i < n; i++)
16         res += a[i] != 0;
17     return res;
18 }
19
20 int main() {
21     char s[100];
22     gets(s);
23
24     char letterset[ABC_POWER] = {0};
25     getLetterset(s, letterset);
26
27     printf("%d", countNonZero(letterset, ABC_POWER));
28
29     return 0;
30 }
```

---

<sup>1</sup>Всегда считал излишним тратить один байт под элемент множества, Более продвинутые техники будут показаны позже.

Однако часто приходится работать с мульти множеством в таком представлении. Усложним немного прошлую задачу:

### Задача о подсчёте букв

С клавиатуры вводятся  $n$  строчных символов латинского алфавита. Необходимо определить, сколько различных символов содержала последовательность, которые встретились четное количество раз.

Сложность по времени:  **$O(n)$** .

В отличие от прошлой задачи, при встрече очередной буквы мы будем не выставлять `letterset[pos]` в единицу, а увеличивать его значение на 1. По окончанию обработки символьной последовательности мы получим массив, где  $i$ -ый элемент отвечает: сколько раз встречалась  $i$ -ая буква латинского алфавита в последовательности.

```

1 #include <stdio.h>
2
3 // покажем решение через указатель
4 void get_letterset(char *s, int *letterset) {
5     while (*s != '\0') {
6         letterset[*s - 'a']++;
7         s++;
8     }
9 }
10
11 int main() {
12     char s[100];
13     gets(s);
14
15     int letterset[26] = {0};
16     get_letterset(s, letterset);
17
18     for (int i = 0; i < 26; i++)
19         if (letterset[i] % 2 == 0 && letterset[i] != 0)
20             printf("%c ", i + 'a');
21
22     return 0;
23 }
```

### 14.2.3 Множество на неупорядоченном массиве

Рассмотрим случай с неупорядоченными массивами. Выполним предположение, что размеры в процессе оперирования ими не превышают некоторой константы. Организуем работу с массивом так: если элемент принадлежит множеству – его значение имеется в массиве, иначе – элемент отсутствует. Пусть

$$A = \{1, 5, 7, 10\} \quad B = \{5, 6, 7\}$$

Тогда массивы могли быть следующими:



Порядок элементов в массиве зависит от порядка их включения в множество. Если в множество  $A$  добавляется элемент, он появится в самом конце:



Если в прошлых вариантах представления можно было быстро проверить, имеется ли элемент в множестве (за  $O(1)$ ), здесь это сделать сложнее. Требуется применение линейного поиска ( $O(N)$ ). Перед тем как добавить или удалить элемент из множества, необходимо проверить, имеется ли он там. Если элемент есть – вставка не производится, если его нет – при удалении множество остаётся без изменения.

Так как у нас имеется библиотека с функциями для работы с одномерными массивами, по максимуму постараемся ей воспользоваться. Опишем часть реализации. Объявление типа множество:

---

```

1 typedef struct unordered_array_set {
2     int *data;           // элементы множества
3     size_t size;         // количество элементов в множестве
4     size_t capacity;    // максимальное количество элементов в множестве
5 } unordered_array_set;

```

---

Для создания множества будем пользоваться двумя функциями. Первая возвращает множество, заданного размера:

---

```

1 unordered_array_set unordered_array_set_create(size_t capacity) {
2     return (unordered_array_set) {
3         malloc(sizeof(int) * capacity),
4         0,
5         capacity
6     };
7 }

```

---

Вторая функция создаёт множество из элементов одномерного массива  $a$  размера  $size$ :

---

```

1 unordered_array_set unordered_array_set_create_from_array(const int *a,
2     size_t size) {
3     unordered_array_set set = unordered_array_set_create(size);
4     for (size_t i = 0; i < size; i++)
5         unordered_array_set_insert(&set, a[i]);
6     unordered_array_set_shrinkToFit(&set);
7     return set;
}

```

---

В прошлой реализации можно заметить функцию **shrinkToFit**, которая освобождает неиспользуемую оперативную память, отведённую под множество:

---

```

1 static void unordered_array_set_shrinkToFit(unordered_array_set *a) {
2     if (a->size != a->capacity) {
3         a->data = (int*)realloc(a->data, sizeof(int) * a->size);
4         a->capacity = a->size;
5     }
6 }

```

---

Чем может помочь вам второй способ создания множества? Проведению тестирования. Можно создать литералы массивов и из них создавать множества для теста:

---

```

1 void unionTest() {
2     unordered_array_set set1 =
3         unordered_array_set_create_from_array((int[]){1, 2}, 2);
4     unordered_array_set set2 =
5         unordered_array_set_create_from_array((int[]){2, 3}, 2);
6     unordered_array_set resSet =
7         unordered_array_set_union(set1, set2);
8     unordered_array_set expectedSet =
9         unordered_array_set_create_from_array((int[]){1, 2, 3}, 3);
10    assert(unordered_array_set_isEqual(resSet, expectedSet));
11
12    unordered_array_set_delete(set1);
13    unordered_array_set_delete(set2);
14    unordered_array_set_delete(resSet);
15    unordered_array_set_delete(expectedSet);
16 }
```

---

при наличии функции isEqual:

---

```

1 int unordered_array_set_isEqual(unordered_array_set set1,
2     unordered_array_set set2) {
3     if (set1.size != set2.size)
4         return 0;
5     qsort(set1.data, set1.size, sizeof(int), compare_ints);
6     qsort(set2.data, set2.size, sizeof(int), compare_ints);
7     return memcmp(set1.data, set2.data, sizeof(int)*set1.size) == 0;
8 }
```

---

Для реализации операции поиска потребуется функция линейного поиска:

---

```

1 int unordered_array_set_in(unordered_array_set *set, int value) {
2     return linearSearch_(set->data, set->size, value);
3 }
```

---

Вставка элемента:

---

```

1 void unordered_array_set_isAbleAppend(unordered_array_set *set) {
2     assert(set->size < set->capacity);
3 }
4
5 void unordered_array_set_insert(unordered_array_set *set, int value) {
6     if (unordered_array_set_in(set, value) == set->size) {
7         unordered_array_set_isAbleAppend(set);
8         append_(set->data, &set->size, value);
9     }
10 }
```

---

Написание всех остальных операций отведём для самостоятельного рассмотрения.

Особенности представления множества на неупорядоченном массиве:

- Подходит для разреженных множеств.
- Операции над множествами работают относительно медленно.

#### 14.2.4 Множество на упорядоченном массиве

Если же при оперировании множествами поддерживать упорядоченность, можно ускорить время работы программы. Однако потребуется знание более сложных техник. В связи с этим можно выделить следующие особенности:

- Подходит для разреженных множеств.
- Алгоритмы работают быстрее, чем алгоритмы на неупорядоченных массивах, но более сложные в реализации.

# Глава 15

## Одномерное динамическое программирование

Динамическое программирование<sup>1</sup> — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать (А. Кумок).

Метод динамического программирования, который иногда называют методом динамической оптимизации — это подход к решению сложных задач путем разбиения их на более мелкие части и запоминания результатов решения этих частей.

Давайте разберем пример:

$$1 + 2 = ?$$

$$1 + 2 + 3 = ?$$

$$1 + 2 + 3 + 4 = ?$$

$$1 + 2 + 3 + 4 + 5 = ?$$

$$1 + 2 + 3 + 4 + 5 + 6 = ?$$

Если бы перед вами поставили задачу посчитать значения данных сумм, вы, вероятно, использовали бы результаты вычислений на предыдущих этапах, чтобы найти значение последующего выражения. Динамическое программирование позволяет нам избежать повторных вычислений, путем запоминания промежуточных результатов. То есть путем запоминания результатов проблем, которые мы уже решили.

Чтобы успешно решить задачу динамикой, необходимо определиться со следующим:

1. Состояние динамики: параметр(ы), однозначно задающие подзадачу;
2. Значения начальных состояний, называемых базой;
3. Переходы между состояниями: формула пересчёта / перехода;
4. Порядок пересчёта / обхода;
5. Положение ответа на задачу: иногда это сумма или, например, максимум из значений нескольких состояний.

---

Глава написана совместно с Барышниковой Варварой.

<sup>1</sup> Я считаю, что алгоритмы динамического программирования часто легче разработать заново, чем искать их готовую реализацию в какой-либо книге. Однако пока вы не понимаете динамическое программирование, оно вам кажется каким-то шаманством. (Стивен Скиена)

В данной главе в качестве языка листингов выбран C++, он позволяет абстрагироваться от тонкостей C и больше сфокусироваться на алгоритмах. Пространство имён `std` подключено по этим же соображениям.

## Вычисление префиксных сумм

Для примера возьмем задачу нахождения префиксных сумм (посчитать сумму чисел в массиве, предшествующие  $i$ -му элементу).

1	2	3	4	4	5	6	1
1	3	6	10	14	19	25	26

Рис. 15.1: Массив и соответствующий ему префиксный массив

Обозначим за  $F(i)$  значение суммы на префикссе. Известно, что

$$F(0) = 0$$

$$F(1) = a[1]$$

Для вычисления последующих значений  $F(i)$  потребуется сумма  $F(i-1)$  на префикссе длины  $i - 1$ . То есть для данного примера:

$$F(i) = F(i-1) + a[i] \quad i \geq 2$$

Можно обратить внимание на сходство ДП и метода математической индукции. В последнем случае также для доказательства какого-то утверждения, необходимо проверить его на малых числах, а затем доказать, что если для малых чисел теорема верна, то она верна и для большего.

### Состояние

Состояние описывает, какую задачу решает ДП. Оно должно содержать в себе некоторые параметры задачи. Из состояния мы должны получить ответ. Чаще всего (но не всегда) состояние ДП можно извлечь из вопроса или условия самой задачи.

Например, для получения массива префиксных сумм, состояние будет:  $F(i)$  – сумма всех чисел от 1 до  $i$ . Здесь число  $i$  будет параметром задачи. А желаемый ответ – сумма соответствующих чисел. Но не всегда всё так просто, и состояние может быть описано двумя, тремя и т.д. числами и иногда даже целыми множествами.

### База

Мы не можем взять и из ничего начать решать задачу на основе более простых. Всегда должны быть такие задачи или условия, ответ на которые мы знаем сразу же при анализе исходной задачи (исходя из логики или разбора случаев, но не прибегая к динамике). Такие состояния называются **базой**. Другими словами, база – это набор самых простых состояний, значения для которых мы получаем не из динамики, а, в основном, из каких-либо математических соображений. Обычно размер базы зависит от формул перехода.

Хорошей базой в задаче о префиксных суммах является число  $F[0]$  – сумма чисел на префикссе размера 0.

## Формулы перехода

**Формулы перехода** задают правила, по которым происходит переход от простых задач к сложным. В самом простом случае каждое простое состояние зависит только от предыдущего. Правда, довольно часто это не так. Бывают переходы из двух, трех и большего числа состояний. В случае префиксных сумм, примером формулы является:

$$F(i) = F(i - 1) + a[i]$$

Из формулы видно, что для вычисления очередной суммы, достаточно знать предыдущую сумму. Поэтому, база размера 1 вполне нас устраивала. Возвращаясь к понятию базы, в общем случае, база подбирается так, чтобы при вычислении динамики по формулам, никогда не было необходимости смотреть в неподсчитанные значения. Стоит отметить, что обычно по формулам построить базу не составляет труда.

Формулы могут быть очень разнообразными, как и количество параметров, от которых зависит динамика. Если есть проблемы с формулой, то придется менять (модифицировать) состояние.

## Порядок обхода

**Порядок обхода** – последовательность вычисления состояний динамики. Мы говорим, что решаем задачу через более простые. Однако не всегда понятно, что такое более простые задачи. Поэтому определение порядка обхода важно.

В случае с префиксной суммой, легче перебирать по возрастанию параметра. Другие порядки встречаются в более сложных задачах. Если не удается подобрать порядок обхода, то может помочь специальная техника, называемая ленивой динамикой.

## Ответ

**Ответ** – значение, которое мы хотим получить от динамики. Очень часто в состоянии мы храним ответ на задачу, поэтому подчас достаточно найти ее самое последнее значение. Однако, иногда цель задачи в том, чтобы посчитать несколько значений.

Для начала разберем базовые задачи для описания состояния динамики которых хватит одного параметра, и соответственно требуют для своего решения одномерный массив. Такие задачи называются задачами на одномерное динамическое программирования.

## 15.1 Сумма чисел от 1 до $n$

### Сумма чисел от 1 до $n$

Найти сумму чисел:

$$F(n) = 1 + 2 + \dots + n$$

---

Совершенно нормально чувствовать себя в замешательстве, впервые столкнувшись с динамическим программированием. Совет: не тушуйтесь. Не упускайте возможность стать еще умнее.

Необходимо выстроить решение задачи таким образом, чтобы значение  $F(i)$  описалось на другие, рассчитанные ранее  $F$ .

Ответим на вопросы, для решения задачи:

- Состояние динамики: параметр(ы), однозначно задающие подзадачу. Состояние динамики можно определять значением параметра  $i$ . Пусть  $F(i)$  – сумма чисел от 1 до  $i$ . Функцию  $F$  часто называют **целевой**.
- Значения начальных состояний. В качестве начального состояния можно взять следующее:

$$F(1) = 1$$

- Переходы между состояниями: формула пересчёта. Необходимо понять, как  $F(i)$  связано с  $F(i - 1)$ . Для данной задачи:

$$F(i) = F(i - 1) + i$$

- Порядок пересчёта: последовательно считаем  $F(i)$  для  $i \in [2, n]$ .

- Положение ответа на задачу. Ответ будет записан в  $F(n)$ .

Применим подход, присущий динамическому программированию: вначале решим маленькую задачу и её решение будем использовать для решения большой задачи. Полученные сведения выше используем для вычисления  $F(4)$ :

$$F(1) = 1$$

$$F(2) = 1 + 2 = F(1) + 2 = 1 + 2 = 3$$

$$F(3) = 1 + 2 + 3 = F(2) + 3 = 3 + 3 = 6$$

$$F(4) = 1 + 2 + 3 + 4 = F(3) + 4 = 6 + 4 = 10$$

После произведения вычислений мы можем дать ответ для любого  $i$  от 1 до  $n$ , выдав значение  $F(i)$ .

## 15.2 Техники кэширования: мемоизация и табуляция

**Кэш** или **кеш** – промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью.

### Числа Фибоначчи

Найти  $n$ -ый член последовательности Фибоначчи.

Известно, что можно выразить  $n$ -ый член последовательности через два предыдущих.

Рекурсивная функция:

---

```
// возвращает n-ый член последовательности Фибоначчи
long long F(const int n) {
    if (n < 2)
        return n;
    else
        return F(n - 2) + F(n - 1);
}
```

---

Главный плюс – быстро и легко пишется. Из минусов – долго работает за счёт повторных рекурсивных вызовов.

Добавление **мемоизации**<sup>2</sup> решает проблему. При очередном вычислении значения мы сохраняем его в массив и только потом осуществляем возврат. Это позволяет избежать повторных расчётов: если значение было посчитано ранее, оно и будетозвращено:

---

#### Листинг 54 Вычисление чисел Фибоначчи с использованием мемоизации

---

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

using namespace std;

int F(const int n) {
    static vector<int> cache(41, 0);

    if (cache[n] != 0)
        return cache[n];

    if (n < 2)
        cache[n] = n;
    else
        cache[n] = F(n - 2) + F(n - 1);

    return cache[n];
}

int main() {
    cout << F(1) << '\n'; // 1
    cout << F(4) << '\n'; // 3
    cout << F(10) << '\n'; // 55

    return 0;
}
```

---

Основной недостаток такого подхода – наличие глубоких рекурсивных вызовов. Решить данный недостаток можно с использованием **табуляции** – техники, которая в первую очередь сфокусирована на заполнении кэша, а не на поиске решения

---

<sup>2</sup>Мемоизация – техника кэширования, которая использует заново ранее вычисленные решения подзадач.

подпроблемы. Вычисление значений, которые необходимо поместить в кэш легче всего в данном случае выполнять итеративно, а не рекурсивно. Возможное решение с использованием техники табуляции выглядело бы так:

---

**Листинг 55** Вычисление чисел Фибоначчи с использованием табуляции

---

```
#include <iostream>
#include <vector>
#include <cassert>

using namespace std;

long long F(const int n) {
    static vector<long long> cache;
    if (cache.empty()) {
        // будет найдено 92 числа Фибоначчи
        cache.push_back(0);
        cache.push_back(1);

        int i = 2;
        while (cache.back() > 0) {
            cache.push_back(cache[i - 1] + cache[i - 2]);
            i++;
        }
        cache.pop_back();
    }
    assert(n < cache.size());
    return cache[n];
}

int main() {
    cout << F(1) << '\n'; // 1
    cout << F(2) << '\n'; // 1
    cout << F(3) << '\n'; // 2
    cout << F(4) << '\n'; // 3

    return 0;
}
```

---

### 15.3 Задача о зайце

Задачи о чём-то прыгающем – классика динамического программирования. Нельзя пропустить это.

### Задача о зайце

В зоопарке появился заяц. Чтобы он не скучал, директор зоопарка распорядился поставить в его клетке лесенку. Теперь наш зайчик может прыгать по лесенке вверх, перепрыгивая через ступеньки.

Лестница имеет определенное количество ступенек  $N$ . Заяц может одним прыжком преодолеть не более  $K$  ступенек. Директору любопытно, сколько различных способов есть у зайца добраться до вершины лестницы при заданных значениях  $K$  и  $N$ .

Например, если  $K = 3$  и  $N = 4$ , то существуют следующие маршруты:  $1 + 1 + 1$ ,  $1 + 1 + 2$ ,  $1 + 2 + 1$ ,  $2 + 1 + 1$ ,  $2 + 2$ ,  $1 + 3$ ,  $3 + 1$ . Т.е. при данных значениях у зайца всего 7 различных маршрутов добраться до вершины лестницы.

Тестовые данные:

$N$	$K$	Результат
3	1	1
7	2	21
10	3	274

Ответим на вопросы, для решения задачи:

- Состояние динамики: параметр(ы), однозначно задающие подзадачу. Состояние динамики можно определять значением параметра  $i$ . Пусть  $F(i)$  – количество способов добраться со ступеньки  $i$  до последней ступеньки.
- Значения начальных состояний. В качестве начального состояния можно взять следующее:

$$F(n) = 1$$

Если вы находитесь на финише, у вас есть один способ оказаться там. Было бы некорректным говорить, что количество способов равняется нулю.

- Переходы между состояниями: формула пересчёта. Попробуем понять, каким образом связаны состояния между собой. Несколько тривиальных выкладок. Пусть  $K = 3$ , тогда заяц по условию задачи может прыгать хотя бы на одну ступеньку вверх, значит:

$$F(n - 1) = F(n) = 1$$

Мы решили задачу с одной ступенькой. Несложно убедиться в том, что нет другого способа добраться до вершины лестницы.

Предположим, что ступенек две.  $n$  – вершина,  $(n - 1)$  – ступенька перед ней,  $(n - 2)$  – ещё одна ступенька. Если бы он не мог прыгать на две ступеньки вверх, а только на одну:

$$F(n - 2) = F(n - 1) = 1$$

но если бы мог:

$$F(n - 2) = F(n - 1) + F(n) = 1 + 1 = 2$$

У зайца есть два способа: прыгнуть сразу на конечную или через промежуточную ступеньку.

Продолжим:

$$F(n-3) = F(n-2) + F(n-1) + F(n) = 4$$

$$F(n-4) = F(n-3) + F(n-2) + F(n-1) = 7$$

В общем случае:

$$F(i) = \sum_{j=1}^{j=k} \overbrace{F(i+j)}^{[i+j \leq n]} \quad i \in [1, n]$$

Оговоримся о нотации. Обозначение  $\overbrace{X}^{[LE]}$  говорит, что  $X$  участвует в вычислении выражения только в том случае, если истинно условие  $LE$ .

4. Порядок пересчёта: последовательно считаем  $F(i)$  в сторону уменьшения  $i$ .
5. Положение ответа на задачу. Судя из формулировки целевой функции, задача заключается в вычислении  $F(1)$  (количество способов добраться с 1 ступеньки до последней).

Можно решать данную задачу иначе, если задать функцию другим образом. Пусть  $F(i)$  – количество способов добраться со ступеньки 1 до ступеньки  $i$ . Ответ будет записан в  $F(n)$ .

Очевидно, что для начальной вершины:

$$F(1) = 1$$

Заяц по условию задачи может прыгать хотя бы на одну ступеньку вверх (и попасть на ступеньку 2 со ступеньки 1), значит:

$$F(2) = F(1) = 1$$

Из последнего выражения следует очевидная мысль: сколько было путей от первой ступеньки до первой, столько же маршрутов от первой ступеньки до второй.

Предположим, что ступенек три. Если бы он не мог прыгать на две ступеньки, то попасть на третью ступеньку он мог бы только со второй ступеньки:

$$F(3) = F(2) = 1$$

но если бы мог прыгать и на одну, и на две ступеньки:

$$F(3) = F(1) + F(2) = 1 + 1 = 2$$

В общем случае:

$$F(i) = \overbrace{F(i-1)}^{[i>1]} + \overbrace{F(i-2)}^{[i>2]} + \cdots + \overbrace{F(i-k)}^{[i>k]} = \sum_{j=1}^{j=k} \overbrace{F(i-j)}^{[i>j]} \quad i \in [2, n]$$

## 15.4 Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет

Вычисление количества способов выдать сдачу  $n$  рублей известными номиналами монет

Имеются монеты номиналом 1, 3, 5, 10 рублей. Сколько существует способов вернуть сдачу из  $n$  рублей номиналами данных монет (порядок выдачи монет важен).

Пусть  $n = 4$  тогда существуют следующие способы выдачи:

$$4 = 1 + 1 + 1 + 1 \quad 4 = 1 + 3 \quad 4 = 3 + 1$$

Для  $n = 4$  ответ равен 3.

Решить задачу можно рекурсивно без мемоизации:

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

using namespace std;

int getN Ways(int balance) {
    static vector<int> denominations = {1, 3, 5, 10};

    if (balance == 0)
        return 1;
    else {
        int res = 0;
        for (const auto denomination : denominations)
            if (balance - denomination >= 0)
                res += getN Ways(balance - denomination);

        return res;
    }
}

int main() {
    int balance;
    cin >> balance;

    cout << getN Ways(balance);

    return 0;
}
```

В значении 48 достигается 1640016810 вариантов и время вычисления составило около 10 секунд. Для 49 количество вариантов вызвало переполнение типа *int*, но расчёты заняли около 16 секунд.

Решим через динамическое программирование. Обозначим через  $F(i)$  количество способов вернуть сдачу величины  $i$ . Сдать сдачу 0 рублей можно единственным спо-

собом, поэтому:

$$F(0) = 1$$

Выдать сдачу один рубль можно весьма просто: выдать 1 рубль и останется сдать ещё как-то 0 рублей (нам даже неважно, как это будет происходить, это уже было посчитано ранее):

$$F(1) = \overbrace{F(0)}^{\begin{array}{c} \text{Выдав монету номиналом 1,} \\ \text{останется выдать суммарно} \\ \text{монет номиналом 0} \end{array}} = 1$$

Количество способов выдать один рубль совпадает с количеством способов выдать 0 рублей.

Если требуется выдать два рубля, можно отдать 1 рубль, и ещё как-то нужно будет выдать 1 рубль:

$$F(2) = F(1) = 1$$

При увеличении количества выдаваемых монет получим следующее:

$$F(3) = \overbrace{F(2)}^{\begin{array}{c} \text{Выдав монету номиналом 1,} \\ \text{останется выдать суммарно} \\ \text{монет номиналом 2} \end{array}} + \overbrace{F(0)}^{\begin{array}{c} \text{Выдав монету номиналом 3,} \\ \text{останется выдать} \\ \text{монет номиналом 0} \end{array}} = 1 + 1 = 2$$

$$F(4) = F(3) + F(1) = 2 + 1 = 3$$

$$F(5) = F(4) + F(2) + F(0) = 3 + 1 + 1 = 5$$

Самостоятельно подсчитайте значения для  $F(6)$ ,  $F(7)$ ,  $F(8)$ ,  $F(9)$ ,  $F(10)$ . В общем случае:

$$F(k) = \overbrace{F(k-10)}^{k \geq 10} + \overbrace{F(k-5)}^{k \geq 5} + \overbrace{F(k-3)}^{k \geq 3} + \overbrace{F(k-1)}^{k \geq 1}$$

Исходный код представлен на листинге 56.

**Листинг 56** Задача о количестве способов выдачи сдачи  $n$  рублей с определенными номиналами монет

---

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

using namespace std;

int main() {
    vector<long long> F(51, 0);

    F[0] = 1;
    for (int balance = 1; balance < F.size(); balance++) {
        for (const auto denomination : {1, 3, 5, 10})
            if (balance - denomination >= 0)
                F[balance] += F[balance - denomination];
        else
            break;
    }

    for (int balance = 1; balance < F.size(); balance++)
        cout << balance << " - " << F[balance] << '\n';

    return 0;
}
```

---

Формула может быть записана и в таком виде:

$$F(k) = \sum_{\substack{i \in \text{номиналы} \\ \text{монет}}}^{\overbrace{F(k-i)}^{k \geq i}}$$

## 15.5 Вычисление минимального количества монет, необходимого для выдачи сдачи $n$ рублей известными номиналами монет

**Вычисление минимального количества монет, необходимого для выдачи сдачи  $n$  рублей известными номиналами монет**

Имеются монеты номиналом 1, 3, 5, 10 рублей. Какое минимальное количество монет необходимо, чтобы выдать сдачу  $n$  рублей.

При решении задачи можно было бы следовать жадно: выдавать сдачу крупными монетами, пока это возможно. Потом остаток сдавать более мелкими. Всегда ли работает данный способ? Для данного набора – да, но в общем случае – нет. Это легко можно заметить на примере суммы в 18 монет и номиналах 1, 9, 10.

Решим задачу методом динамического программирования. Обозначим через  $F(n)$  – минимальное количество монет достоинством 1, 3, 5, 10 для возврата сдачи номиналом  $n$ . Будем решать простую задачу и её решение использовать для решения более сложных:

$$F(0) \rightarrow F(1) \rightarrow F(2) \rightarrow \dots \rightarrow F(n-1) \rightarrow F(n)$$

Осталось понять, как знание  $F(i)$  позволяет вычислить  $F(i + 1)$  (найти формулу пересчёта).

Посчитаем  $F(i)$  для  $i \geq 10$ :

$$F(i) = \min \left( \begin{array}{c} \text{выдадим одну монету номиналом} \\ \text{один, сумму номиналом } i - 1 \\ \text{выдадим оптимально} \\ 1 + F(i - 1) \\ + \cdots + \\ \text{выдадим одну монету номиналом} \\ \text{десять, сумму } i - 10 \text{ монет} \\ \text{выдадим оптимально} \\ 1 + F(i - 10) \end{array} \right)$$

Для  $i \in [1, n]$ :

$$F(i) = \min_{j \in \text{номиналы монет}} \left( 1 + \overbrace{F(i - j)}^{i \geq j} \right) = 1 + \min_{j \in \text{номиналы монет}} \left( \overbrace{F(i - j)}^{i \geq j} \right)$$

Приведем некоторые вычисления вручную:

$$F(0) = 0$$

$$F(1) = 1 + \min(F(0)) = 1 + 0 = 1$$

$$F(2) = 1 + \min(F(1)) = 1 + 1 = 2$$

$$F(3) = 1 + \min(F(2), F(0)) = 1 + \min(2, 0) = 1 + 0 = 1$$

$n$	0	1	2	3	4	5	6	7	8	9	10	11	12
$F_n$	0	1	2	1	2	1	2	3	2	3	1	2	3

Программное решение в листинге 57.

## 15.6 Получение произвольного минимального по количеству набора монет известного номинала, которые можно сдать сдачу $n$ рублей

Получение произвольного минимального по количеству набора монет известного номинала, которые можно сдать сдачу  $n$  рублей

Имеются монеты номиналом 1, 3, 5, 10 рублей. Какие именно монеты нужны для того, чтобы выдать сдачу  $n$  рублей минимальным количеством монет (устроит любой набор).

Воспользуемся решением прошлой задачи, только в этот раз будем запоминать ещё одно значение, которое укажем в скобках: монету, которая будет выдана при данном  $n$  при условии, что дальнейшая выдача монет будет оптимальна:

$n$	0	1	2	3	4	5	6	7	8
$F_n$	0	1 (1)	2 (1)	1 (3)	2 (1)	1 (5)	2 (1)	3 (1)	2 (3)

---

**Листинг 57** Вычисление минимального количества монет, необходимого для выдачи сдачи  $n$  рублей известными номиналами монет

---

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

int main() {
    int sum;
    cin >> sum;

    vector<int> F(sum + 1, INT_MAX);

    F[0] = 0;
    for (int balance = 1; balance < F.size(); balance++)
        for (int denomination : {1, 3, 5, 10})
            if (balance - denomination >= 0)
                F[balance] = min(F[balance],
                                  1 + F[balance - denomination]);

    cout << F[sum];

    return 0;
}
```

---

С этой таблицей можно работать так: пусть возникла необходимость выдать 7 монет (что можно сделать за 3 шага). На текущем этапе  $n = 7$  выдаётся монета номиналом 1 и выполняется переход в клетку с  $n = 6$ .

$n$	0	1	2	3	4	5	6	7	8
$F_n$	0	1 (1)	2 (1)	1 (3)	2 (1)	1 (5)	2 (1)	3 (1)	2 (3)

При  $n = 6$  снова выдаётся монета номиналом 1 и выполняется переход в  $n = 5$ .

$n$	0	1	2	3	4	5	6	7	8
$F_n$	0	1 (1)	2 (1)	1 (3)	2 (1)	1 (5)	2 (1)	3 (1)	2 (3)

При  $n = 5$  снова выдаётся монета номиналом 5 и выполняется переход в  $n = 0$ .

$n$	0	1	2	3	4	5	6	7	8
$F_n$	0	1 (1)	2 (1)	1 (3)	2 (1)	1 (5)	2 (1)	3 (1)	2 (3)

Так как нам остаётся выдать 0 рублей, выдача монет закончена.

Итого получаем:

$$7 = 1 + 1 + 5$$

---

**Листинг 58** Получение произвольного минимального по количеству набора монет известного номинала, которые можно сдать сдачу  $n$  рублей

---

```

#include <climits>
#include <iostream>
#include <vector>

using namespace std;

struct F {
    int n_steps; // количество ходов для лучшего решения
    // при текущем количестве монет
    int best_coin; // выдаваемая монета по оптимальной стратегии
    // при текущем количестве монет
};

std::vector<F> get_F(int maxSum) {
    std::vector<F> F(maxSum + 1);

    F[0].n_steps = 0;
    for (int balance = 1; balance < F.size(); balance++) {
        F[balance].n_steps = INT_MAX;
        for (int denomination : {1, 3, 5, 10}) {
            if (balance - denomination >= 0) {
                int nSteps = 1 + F[balance - denomination].n_steps;
                if (nSteps < F[balance].n_steps) {
                    F[balance].n_steps = nSteps;
                    F[balance].best_coin = denomination;
                }
            }
        }
    }

    return F;
}

vector<int> make_request(const vector<F> &F, int balance) {
    vector<int> coins;
    while (balance != 0) {
        coins.push_back(F[balance].best_coin);
        balance -= F[balance].best_coin;
    }

    return coins;
}

int main() {
    auto F = get_F(500);

    int sum;
    cin >> sum;

    for (auto &x : make_request(F, sum))
        cout << x << ' ';

    return 0;
}

```

---

## 15.7 Задача о игре

### Компьютерная игра

Во многих старых играх с двумерной графикой можно столкнуться с подобной ситуацией. Какой-нибудь герой прыгает по платформам, которые висят в воздухе. Он должен перебраться от одного края экрана к другому. При этом при прыжке с одной платформы на соседнюю, у героя уходит  $|y_2 - y_1|$  единиц энергии, где  $y_1$  и  $y_2$  – высоты, на которых расположены эти платформы. Кроме того, у героя есть суперприем, который позволяет перескочить через платформу, но на это затрачивается  $3 * |y_3 - y_1|$  единиц энергии.

Предположим, что вам известны высоты всех платформ в порядке от левого края до правого. Сможете ли вы найти, какое минимальное количество энергии потребуется герою, чтобы добраться с первой платформы до последней?

Тестовые данные:

$n$	$a$	Результат
3	1 5 10	9
3	1 5 2	3
5	3 10 1 5 2	9

Зададим целевую функцию. Пусть  $F(i)$  – количество энергии, которое будет затрачено на то, чтобы добраться с  $i$ -ой до последней платформы. Если герой уже находится на финише, затраты энергии не потребуются:

$$F(n) = 0$$

Добраться до последней платформы с предпоследней можно только простым прыжком:

$$F(n-1) = |a(n) - a(n-1)|$$

С предпоследней можно добраться как суперприёмом, так простым прыжком. Нас интересует вариант с минимальными затратами энергии:

$$F(n-2) = \min(\overbrace{|a(n-1) - a(n-2)| + F(n-1)}^{\text{обычный прыжок}}, \overbrace{3 * |a(n) - a(n-2)| + F(n-2)}^{\text{суперприём}})$$

В произвольной клетке нас ожидает выбор между двумя альтернативами:

$$F(i) = \min(|a(i+1) - a(i)| + F(i+1), 3 * |a(i+2) - a(i)| + F(i+2))$$

Итоговое значение может быть найдено в  $F(1)$ .

## 15.8 Задача о зайце с запрещенными ступеньками

### Задача о зайце с запрещенными ступеньками

Условие задачи совпадает с задачей на странице 368. Требуется найти решение со сложностью  $O(n)$ , где  $n$  – количество ступенек в лестнице. Дополнительно добавим ступеньки, на которых заяц не может появляться. Гарантируется, что начальная клетка свободна.

Необходимо найти количество способов добраться с первой ступеньки до последней.

Пусть  $F(i)$  – количество различных путей, которыми заяц может добраться до  $i$ -й клетки, не заходя в занятые.

Скажем, что  $F(1)$  – изначальная позиция зайца, то есть в начале своего движения он стоял на этом самом месте и пришел в него (до начала событий задачи) одним каким-то способом, тогда можно написать

$$F(1) = 1$$

Очевидно, что прийти в состояние  $F(2)$  можно только одним способом: из позиции, где заяц находился изначально, причем внимательно смотрим на клетку, и если на нее заходить нельзя, то обходим стороной, оставляя ее равной 0 (т.к. в нее попасть невозможно никаким способом). Несложно показать, что для произвольной клетки  $i \in [2, n]$  количество способов будет находиться так:

$$F(i) = \begin{cases} 0, & \text{если клетка недоступна} \\ \sum_{j=1}^k \overbrace{F(i-j)}^{i>j}, & \text{если клетка доступна} \end{cases}$$

Можно на каждой итерации  $i$  поддерживать сумму последних не более  $k$  элементов:

$$s = \sum_{j=1}^k \overbrace{F(i-j)}^{i>j},$$

и использовать её для получения алгоритма сложности  $O(n)$ . Каждое из решений представлено на листингах ниже.

---

**Листинг 59** Количество способов добраться до последней ступеньки с ограничением на занятые ступеньки  $O(n * k)$

---

```
#include <iostream>
#include <vector>
#include <set>
#include <cassert>

using namespace std;

vector<long long> getF(const set<int> &blockedCellsNumbers,
                       size_t n, size_t k) {
    // проверяем, что начальная ступенька доступна, и что
    // количество ступенек не равно нулю.
    assert(blockedCellsNumbers.find(1) == blockedCellsNumbers.end() && n != 0);

    vector<long long> F(n + 1, 0);
    vector<bool> isBlocked(n + 1);
    for (auto &i : blockedCellsNumbers)
        if (i <= n)
            isBlocked[i] = true;
        else
            break;

    F[1] = 1;
    for (size_t i = 2, iBound = min(k, n); i <= iBound; i++) {
        if (!isBlocked[i])
            for (size_t j = 1; j < i; j++)
                F[i] += F[j];
    }

    for (size_t i = k + 1; i <= n; i++) {
        if (!isBlocked[i])
            for (size_t j = i - k; j < i; j++)
                F[i] += F[j];
    }

    return F;
}

int main() {
    auto F = getF({2, 4, 5}, 8, 3);

    // количество способов попасть в 8 ступеньку
    // при условии, что заблокированы ступеньки 2, 4, 5,
    // и максимальная длина прыжков равна 3
    cout << F[8];

    return 0;
}
```

---

---

**Листинг 60** Количество способов добраться до последней ступеньки с ограничением на занятые ступеньки  $O(n)$

---

```
#include <iostream>
#include <vector>
#include <set>
#include <cassert>

using namespace std;

vector<long long> getF(const set<int> &blockedCellsNumbers,
                       size_t n, size_t k) {
    // проверка, что начальная ячейка не запрещена и что
    // количество ступенек хотя бы одна
    assert(blockedCellsNumbers.find(1) == blockedCellsNumbers.end() && n >= 1);

    vector<long long> F(n + 1, 0);
    vector<bool> isBlocked(n + 1);
    for (auto &i : blockedCellsNumbers)
        if (i <= n)
            isBlocked[i] = true;
        else
            break;

    F[1] = 1;
    long long s = 0;
    for (size_t i = 2, iBound = min(k, n); i <= iBound; i++) {
        s += F[i - 1];
        if (!isBlocked[i])
            F[i] = s;
    }

    for (size_t i = k + 1; i <= n; i++) {
        s -= F[i - k - 1];
        s += F[i - 1];
        if (!isBlocked[i])
            F[i] = s;
    }

    return F;
}

int main() {
    auto F = getF({2, 4, 5}, 8, 3);

    // Количество способов попасть в 8 ступеньку
    // При условии, что заблокированы ступеньки 2, 4, 5
    // и максимальная длина прыжков равна 3
    cout << F[8];

    return 0;
}
```

---

## 15.9 Поиск возрастающей подпоследовательности наибольшей длины

### Поиск возрастающей подпоследовательности наибольшей длины

Дана числовая последовательность  $a_1, a_2, \dots, a_n$ . Требуется найти длину наибольшей возрастающей подпоследовательности и вывести её (подпоследовательность может быть получена из последовательности путем вычеркивания некоторых элементов).

Пример:

$$a = \{-1, 2, 3, 0, 1, 2\}$$

Наибольшая по длине возрастающая последовательность:

$$a' = \{-1, 0, 1, 2\}$$

Пусть  $F(i)$  – максимальная длина возможной получаемой последовательности, при условии, что  $a[i]$  включается в последовательность. При решении данной задачи на каждом шаге нам придётся выбирать включаем ли мы текущий элемент в подпоследовательность или нет. Если включаем, то пытаемся добавленный элемент присоединить к какому-то следующему, чтобы суммарная длина цепочки была максимальной.

Начнём решать задачу с конца. Если элемент один, он будет составлять пока что последовательность максимальной длины:

$$F(n) = 1$$

Если мы рассматриваем пару элементов, возможны две ситуации:

- $a[n - 1] < a[n] \Rightarrow F(n - 1) = 2$
- $a[n - 1] \geq a[n] \Rightarrow F(n - 1) = 1$

Если обобщить два случая, получим:

$$F(n - 1) = 1 + [a[n - 1] < a[n]]$$

Рассмотрим историю с тремя элементами на примере:

$$a = \{\dots, 1, 2, 3\}$$

Единичку можно привязать хоть к двойке, хоть к тройке. Максимальная длина последовательности в первом случае:

$$F(n - 2) = 1 + F(n - 1) = 3, \text{ так как } a[n - 2] < a[n - 1]$$

Во втором случае:

$$F(n - 2) = 1 + F(n) = 2, \text{ так как } a[n - 2] < a[n]$$

Очевидно, что нам нужно остановиться на более длинном варианте. Таким образом, можно записать формулу так:

$$F(n-2) = 1 + \max \left( \overbrace{F(n-1)}^{[a[n-2] < a[n-1]]}, \overbrace{F(n)}^{[a(n-2) < a(n)]} \right)$$

Тогда для произвольного  $i$  получим:

$$F(i) = 1 + \max_{j=i+1}^n \left( \overbrace{F(j)}^{[a(i) < a(j)]} \right)$$

Ответ на задачу будет определяться по формуле:

$$x = \max_{i=1}^n (F(i))$$

Благодаря такому подходу, был найден алгоритм со сложностью  $O(n^2)$  по времени (хотя возможно построить алгоритмы, обладающие меньшей сложностью по времени ( $O(n \log n)$ )).

Для восстановления последовательности потребуется использование дополнительной памяти. Будем во вспомогательном массиве запоминать индекс элемента, который должен быть выбран следующим для получения последовательности максимальной длины:

Индекс	1	2	3	4	5	6	7	8
Последовательность	4	2	6	5	7	12	3	8
$F(i)$								1
Индекс оптимального следующего элемента							-1	

Таблица 15.1: База

Индекс	1	2	3	4	5	6	7	8
Последовательность	4	2	6	5	7	12	3	8
$F(i)$	4	4	3	3	2	1	2	1
Индекс оптимального следующего элемента	4	3	5	5	8	-1	8	-1

Таблица 15.2: Вычисленные состояния и индексы оптимальных следующих элементов

Из таблицы можно получить, что максимальная длина возможной возрастающей подпоследовательности равняется четырём. Для восстановления последовательности достаточно найти любое максимальное  $F(i)$ , вывести этот элемент по индексу  $i$  и проследовать по индексу оптимального следующего элемента до тех пор, пока не будет найдено -1. Самостоятельно восстановите 2 последовательности длины 4, пользуясь значениями из таблицы выше ( $a'_1 = \{4, 5, 7, 8\}$ ,  $a'_2 = \{2, 6, 7, 8\}$ ).

## 15.10 Поиск оптимального города, для доставки мусора

### Поиск оптимального города, для доставки мусора

Дано  $n$  ( $n \geq 1$ ,  $n$  – чётно) городов. Города соединены кольцевой двусторонней автодорогой. В каждом городе генерируется  $a_i$  единиц мусора (схема на рис. 15.2).

Стоимость провоза единицы груза между соседними городами равняется 1. Требуется выбрать один город и построить в нём мусороперерабатывающий завод таким образом, чтобы минимизировать стоимость доставки мусора.

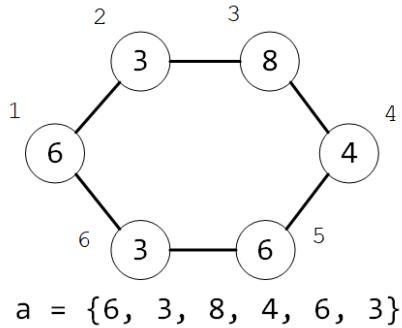


Рис. 15.2: Расположение шести городов с указанием количества генерируемого мусора. Если выбрать в качестве завода первый город, общая стоимость доставки составит:

$$3 * 1 + 8 * 2 + 4 * 3 + 6 * 2 + 3 * 1 = 46$$

Пусть  $F(i)$  – общие затраты на доставку мусора, при условии, что завод поставлен в городе  $i$ . Попробуем вычислить базу и формулы перехода. Чтобы решение прошло проще, представим пока что, что  $n = 6$ . Стоимость доставки мусора в первый и второй город:

$$F(0) = a_0 * 0 + a_1 * 1 + a_2 * 2 + a_3 * 3 + a_4 * 2 + a_5 * 1$$

$$F(1) = a_0 * 1 + a_1 * 0 + a_2 * 1 + a_3 * 2 + a_4 * 3 + a_5 * 2$$

Посмотрим, каким образом связаны  $F(1)$  и  $F(2)$ :

$$F(1) - F(0) = a_0 - a_1 - a_2 - a_3 + a_4 + a_5$$

Если бы  $F(0)$  было подсчитано, то для поиска  $F(1)$  стало достаточно вычесть из  $F(0)$  сумму  $a_1 + a_2 + a_3$  и прибавить  $a_4 + a_5 + a_0$ . Если писать более формально:

$$F(i) = F(i-1) - \sum_{j=0}^{\frac{n}{2}} a[(i+j) \bmod n] + \sum_{j=\frac{n}{2}}^{j < n} a[(i+j) \bmod n] \quad i \neq 0$$

При попытке закодировать алгоритм и не применять никаких модификаций будет получено решение за  $O(n^2)$ . Однако, хотелось бы как-то снизить сложность вычислений. Пусть

$$F(i) = F(i-1) + G(i) \quad i \geq 0$$

где  $G(i)$  – требуемая к добавлению  $F(i - 1)$  для получения  $F(i)$ :

$$G(i) = - \sum_{j=0}^{j < \frac{n}{2}} a[(i+j) \bmod n] + \sum_{j=\frac{n}{2}}^{j < n} a[(i+j) \bmod n]$$

$$G(0) = -a_0 - a_1 - a_2 + a_3 + a_4 + a_5$$

$$G(1) = a_0 - a_1 - a_2 - a_3 + a_4 + a_5$$

$$G(2) = a_0 + a_1 - a_2 - a_3 - a_4 + a_5$$

$$G(i) = G(i - 1) + 2 * a_{i-1} - 2 * a_{(i-1+\frac{n}{2}) \bmod n} \quad i \geq 0$$

Приведём решение в листинге 61.

---

### Листинг 61 Поиск оптимального города, для доставки мусора

---

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n;
    cin >> n;

    vector<int> a(n);
    for (int i = 0; i < n; i++)
        cin >> a[i];

    vector<int> F(n);
    for (int i = 0; i < n; i++)
        F[0] += min(i - 0, n - i) * a[i];

    vector<int> G(n);
    for (int i = 0; i < n; i++)
        G[0] += (i < n / 2) ? -a[i] : +a[i];

    for (int i = 1; i < n; i++) {
        G[i] = G[i - 1] + 2 * (a[i - 1] - a[(i + n / 2 - 1) % n]);
        F[i] = F[i - 1] + G[i];
    }

    cout << min_element(begin(F), end(F)) - begin(F) + 1;

    return 0;
}
```

---

# Глава 16

## Двумерное динамическое программирование

В случаях, когда целевая функция зависит не от одного, а от двух параметров, мы имеем дело с двумерным динамическим программированием.

### 16.1 Получение количества двоичных последовательностей длины $n$ , не содержащих две единицы подряд

Получение количества двоичных последовательностей длины  $n$ , не содержащих две единицы подряд

Требуется вычислить количество  $n$ -значных чисел в системе счисления с основанием 2, таких что их запись не содержит двух подряд идущих единиц.

В качестве целевой функции возьмём  $F(i, j)$  – количество чисел, которые можно получить без двух единиц подряд, формируя  $i$ -ый разряд, при условии, что на его место ставится значение  $j$ .

Проведем расчеты для целевой функции:

$$F(n, 0) = 1 \quad F(n, 1) = 1$$

Для произвольного  $i \neq n$  получим:

$$F(i, 0) = F(i + 1, 0) + F(i + 1, 1) \quad F(i, 1) = F(i + 1, 0)$$

Ответом будет выступать сумма  $F(1, 0) + F(1, 1)$ .

Если попробовать провести расчет для  $n = 5$  можно получить следующую картину:

$i$	1	2	3	4	5
$F(i, 0)$	8	5	3	2	1
$F(i, 1)$	5	3	2	1	1

Можно заметить, что для произвольного  $n$  ответ является числом Фибоначчи  $F(n + 2)$ , где

$$\begin{aligned} F(1) &= 1, & F(2) &= 1, \\ F(i) &= F(i - 1) + F(i - 2) \text{ для } i > 2 \end{aligned}$$

Программное решение представлено на листинге 62.

---

**Листинг 62** Получение количества двоичных последовательностей длины  $n$ , не содержащих две единицы подряд

---

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

using namespace std;

long long F(size_t n) {
    vector<vector<long long>> F(n + 1, vector<long long>(2));

    F[0][0] = 1;
    F[0][1] = 1;
    for (size_t i = 1; i < n; i++) {
        F[i][0] = F[i - 1][0] + F[i - 1][1];
        F[i][1] = F[i - 1][0];
    }

    return F[n][0] + F[n][1];
}

int main() {
    cout << F(5);

    return 0;
}
```

---

## 16.2 Количество путей из верхнего левого угла в правый нижний на поле размера $n$ на $m$

Количество путей из верхнего левого угла в правый нижний на поле размера  $n$  на  $m$

В верхнем-левом углу таблицы сидит черепашка. Она умеет ходить только вниз, вправо и вниз-вправо. Черепаха хочет попасть в нижнюю правую клетку. Требуется найти количество возможных путей.

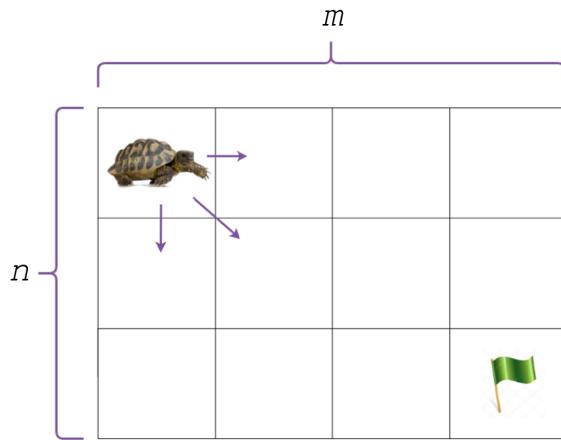
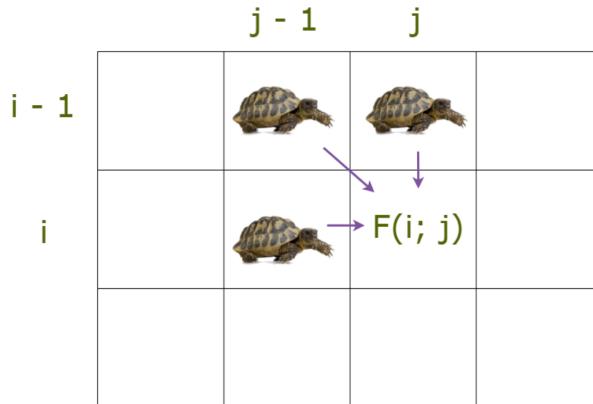


Рис. 16.1: Возможные движения черепашки

Обозначим на  $F(i, j)$  – количество путей в точку  $(i, j)$  из точки  $(1, 1)$ . Тогда задача сведётся к вычислению  $F(n, m)$ .

Попробуем вычислить формулу перехода. Для этого прибегнем к вопросу «откуда мы можем попасть в клетку с координатами  $(i, j)$ ?». Поскольку черепашка может ходить только вправо, вниз или вправо-вниз, то в любую клетку мы можем прийти либо сверху, либо слева, либо сверху-слева:



Отойдём немного от задачи. Пусть есть город  $A$ , в который ведёт  $x$  путей, и город  $B$ , в который ведёт  $y$  путей, и имеется некоторый город  $C$ , в который можно попасть только из  $A$ , или только из  $B$  ( $A$  и  $B$  не связаны). Общее количество путей в город  $C$  будет находиться как сумма  $x + y$ .

Тогда для нашей задачи  $F(i, j)$  будет определяться как сумма количества путей в те вершины, откуда мы можем попасть в  $(i, j)$ :

$$F(i, j) = F(i - 1, j) + F(i, j - 1) + F(i - 1, j - 1)$$

И вот тут мы дошли до того момента, когда найти порядок обхода становится отдельной подзадачей. Необходимо чтобы во время вычисления значения динамики клетки  $(i, j)$ , были уже посчитаны значения для всех клеток, от которых зависит это состояние:  $F(i - 1, j)$ ,  $F(i, j - 1)$ ,  $F(i - 1, j - 1)$ .

Очень часто используется подход, при котором просчитывается первый столбец и первая строка. И только потом последовательно находятся значения в оставшихся клетках. Рассмотрим на примере:

$F(i, j)$	$F(i, j)$	$F(i, j)$	$F(i, j)$
1	1 1 1 1	1 1 1 1	1 1 1 1
		1 ↘ ↗ ↘	1 ↘ ↗ ↘
		1 ↗ ↘ ↗	1 ↗ ↘ ↗
		1	1
		1	1

$F(i, j)$	$F(i, j)$	$F(i, j)$	$F(i, j)$
1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
1 3 5 ↘ ↗ ↘	1 3 5 ↗ ↘ ↗	1 3 5 ↘ ↗ ↘	1 3 5 ↗ ↘ ↗
1 ↗ ↘ ↗	1 ↗ ↘ ↗	1 ↗ ↘ ↗	1 ↗ ↘ ↗
1	1	1	1

Рис. 16.2: Процесс заполнения матрицы  $F$ 

Последовательность вычислений:

$$F(1, 1) = 1$$

$$F(1, j) = 1 \quad j \in [1, m]$$

$$F(i, 1) = 1 \quad i \in [1, n]$$

$$F(i, j) = F(i - 1, j) + F(i, j - 1) + F(i - 1, j - 1)$$

Ответ будет получен в  $F(n, m)$ .

---

**Листинг 63** Количество путей из верхнего левого угла в правый нижний на поле размера  $O(n * m)$

---

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

using namespace std;

long long F(size_t n, size_t m) {
    vector<vector<long long>> F(n + 1, vector<long long>(m + 1));

    F[1][1] = 1;
    for (int i = 1; i <= n; i++) F[i][1] = 1;
    for (int j = 1; j <= m; j++) F[1][j] = 1;

    for (int i = 2; i <= n; i++)
        for (int j = 2; j <= m; j++)
            F[i][j] = F[i - 1][j] + F[i][j - 1] + F[i - 1][j - 1];

    return F[n][m];
}

int main() {
    cout << F(3, 5);

    return 0;
}
```

---

Можно решать задачу с конца: обозначим на  $F(i, j)$  – количество путей из точки  $(i, j)$  в точку  $(n, m)$ . Тогда задача сводится к вычислению  $F(1, 1)$ .

$$F(n, m) = 1$$

$$F(i, m) = 1, \quad i \in [1, n]$$

$$F(n, j) = 1, \quad j \in [1, m]$$

Остальные ячейки будут заполнены по формуле:

$$F(i, j) = F(i + 1, j) + F(i, j + 1) \quad i \in [1, n], j \in [1, m]$$

при движении по матрице снизу-вверх и справа-налево.

### 16.3 Минимальная сумма при движении по полю размера $n$ на $m$

#### Минимальная сумма при движении по полю размера $n$ на $m$

В условиях прошлой задачи добавилось условие: в каждой из ячеек поля записано число. При попадании в ячейку некоторый баланс (изначально равный нулю) изменяется на значение, записанное в ячейке поля. Начальная ячейка тоже содержит число.

Требуется вычислить минимальное возможное значение, которое можно получить после прохождения лабиринта.

Обозначим на  $F(x, y)$  – максимальное количество монет, которое можно собрать по пути, двигаясь из точки  $(1, 1)$  в точку  $(x, y)$ . На самом деле, глобально ничего не меняется при решении этой и прошлой задач.

Пусть есть город  $A$ , при оптимальном движении в котором будет собрано  $F(A)$  монет, и город  $B$ , при оптимальном движении в котором будет собрано  $F(B)$  монет и имеется некоторый город  $C$  (в котором хранится  $x$  монет), и в него можно попасть только из  $A$  или из  $B$  ( $A$  и  $B$  не связаны). Если бы стояла задача минимизировать прибыль, очевидно, что лучше приходить из того города, где накопленное количество монет меньше, тогда

$$F(C) = \min(F(A), F(B)) + z$$

Пользуясь опытом прошлой задачи, выходим на следующую базу, последовательность вычислений и формул пересчёта:

$$F(1, 1) = a[1, 1]$$

$$F(1, j) = F(1, j - 1) + a[1, j] \quad j \in [2, m]$$

$$F(i, 1) = F(i - 1, 1) + a[i, 1] \quad i \in [2, n]$$

$$F(i, j) = \min(F(i - 1, j), F(i, j - 1), F(i - 1, j - 1)) + a(i, j) \quad i \in [2, n], j \in [2, m]$$

Ответ будет найден в  $F(n, m)$ .

## 16.4 Поиск пути с минимальными суммарными перепадами высот

### Поиск пути с минимальными суммарными перепадами высот

Отряду нужно пересечь прямоугольное поле размера  $m \times n$  квадратов, двигаясь из левого верхнего угла в правый нижний и перемещаясь между соседними квадратами только в двух направлениях – вправо и вниз.

Поле не очень ровное, но у отряда есть карта, на которой отмечена высота каждого квадрата. Опасность перехода с квадрата высоты  $h_1$  на соседний квадрат высоты  $h_2$  оценивается числом  $|h_2 - h_1|$ . Опасность всех переходов в пути суммируется.

Выясните, какова минимальная опасность пути из квадрата  $(1, 1)$  в квадрат  $(m, n)$ .

Обозначим на  $F(x, y)$  минимальную опасность перехода, которая будет получена в процессе движения из точки  $(1, 1)$  в точку  $(x, y)$ . В качестве примера поля и оптимального пути рассмотрите следующий:

$1 \rightarrow 2 \rightarrow 3$	5		0	1	2	4
3	8	4	2	7	3	6
2	6	1	3	7	6	8
1	7	2	4	8	7	8

$h$                                      $F$

Итак, мы знаем, что отряд может перемещаться между соседними квадратами только в двух направлениях – вправо и вниз. Значит, в любую клетку мы можем прийти либо сверху, либо слева. Мы помним, что опасность перехода с квадрата высоты  $h_1$  на соседний квадрат высоты  $h_2$  оценивается числом  $|h_2 - h_1|$ . То есть мы должны определить числовую характеристику опасности попадания из клетки  $(i - 1, j)$  и клетки  $(i, j - 1)$  в текущую клетку  $(i, j)$ . Опасность перехода слева:

$$\text{penaltyLeft} = |h[i][j] - h[i][j - 1]|$$

Опасность перехода сверху:

$$\text{penaltyUp} = |h[i][j] - h[i - 1][j]|$$

И затем найдем минимум из двух таких опасностей. И не забудем, что в итоге у нас должна получиться сумма всех опасностей, через которые мы прошли (то есть дополнительно добавим значение динамики на предыдущем шаге, при этом учитывая, из какой клетки мы приедем):

$$F(i, j) = \min(F(i, j - 1) + \text{penaltyLeft}, F(i - 1, j) + \text{penaltyUp})$$

Таким образом, последовательность вычисления состояний:

$$F(1, 1) = 0$$

$$F(1, j) = F(1, j - 1) + |h[1][j] - h[1][j - 1]| \quad j \in [2, m]$$

$$F(i, 1) = F(i - 1, 1) + |h[i][1] - h[i - 1][1]| \quad i \in [2, n]$$

$$F(i, j) = \min(F(i - 1, j) + |h[i][j] - h[i - 1][j]|, F(i, j - 1) + |h[i][j] - h[i][j - 1]|)$$

## 16.5 Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет с ограничением на количество монет определенного номинала

**Вычисление количества способов выдать сдачу  $n$  рублей известными номиналами монет с ограничением на количество монет определенного номинала**

Имеются монеты номиналом 1, 2, 5 рублей. Сколько существует способов сдать сдачу  $n$  рублей при условии, что в кассе имеется не более двух монет номиналом 1.

Целевая функция может быть определена как:  $F(n, k)$  – количество способов выдать сдачу  $n$  при условии, что можно использовать не более  $k \in [0, 2]$  монет достоинством рубль.

Рассуждения: если вам нужно выдать сдачу 0 рублей, это всегда можно сделать, поэтому:

$$F(0, k) = 1 \quad k \in [0, 2]$$

Последовательно будем увеличивать  $n$  и вычислять значения для различных  $k$ .

Пусть  $n = 1$ . Для  $F(1, 0)$  решений нет, поэтому  $F(1, 0) = 0$ . Нельзя сдать сдачу в один рубль, не имея в кассе рубль. При вычислении  $F(1, 1)$  задумаемся, в какое состояние мы перейдем, если выдадим рубль. Ответ в формуле:

$$F(1, 1) = F(0, 0)$$

так как осталось выдать 0 рублей, но монетой в рубль мы воспользовались. Если осталось две однорублёвые монеты, то  $F(1, 2) = F(0, 1) = 1$ .

Пусть  $n = 2$ . Вычислим  $F(2, k)$  для  $k \in [0, 2]$ :

$$\begin{aligned} F(2, 0) &= \overbrace{F(0, 0)}^{\text{выдано 2 рубля}} = 1 \\ F(2, 1) &= \overbrace{F(0, 1)}^{\text{выдано 2 рубля}} + \overbrace{F(1, 0)}^{\text{выдан рубль}} = 1 + 0 = 1 \\ F(2, 2) &= \overbrace{F(0, 2)}^{\text{выдано 2 рубля}} + \overbrace{F(1, 1)}^{\text{выдан рубль}} = 1 + 1 = 2 \end{aligned}$$

Для произвольного  $n \neq 0$ :

$$F(n, 0) = \overbrace{F(n - 2, 0)}^{[n \geq 2]} + \overbrace{F(n - 5, 0)}^{[n \geq 5]}$$

$$F(n, k) = \overbrace{F(n - 1, k - 1)}^{[n \geq 1]} + \overbrace{F(n - 2, k)}^{[n \geq 2]} + \overbrace{F(n - 5, k)}^{[n \geq 5]} \quad k \neq 0$$

## 16.6 Вычисление минимального количества монет, необходимое для выдачи сдачи $n$ рублей известными номиналами монет с ограничением на количество монет определенного номинала

**Вычисление минимального количества монет, необходимое для выдачи сдачи  $n$  рублей известными номиналами монет с ограничением на количество монет определенного номинала**

Имеются монеты номиналом 1, 2, 5 рублей. Определите минимальное количество монет, которое необходимо, чтобы сдать сдачу  $n$  рублей при условии, что в кассе имеется не более двух монет номиналом 1 рубль.

Пусть  $F(n, k)$  – минимальное количество монет, необходимое для возврата сдачи размера  $n$  при условии, что осталось не более  $k$  монет достоинством один рубль ( $k \in [0, 2]$ ). Из опыта решения прошлых задач получим:

$$F(0, k) = 0 \quad k \in [0, 2]$$

$$F(n, 0) = 1 + \min \left( \overbrace{F(n - 2, 0)}^{[n \geq 2]}, \overbrace{F(n - 5, 0)}^{[n \geq 5]}, \infty \right) \quad n \neq 0$$

$$F(n, k) = 1 + \min \left( \overbrace{F(n - 1, k)}^{[n \geq 1]}, \overbrace{F(n - 2, k - 1)}^{[n \geq 2]}, \overbrace{F(n - 5, k)}^{[n \geq 5]}, \infty \right) \quad n \neq 0, k \neq 0$$

В формулах выше появилась бесконечность. Если все состояния динамики, в которые мы можем попасть из текущего не имеют решений, то и текущее состояние не имеет решений.

Например, нам нужно выдать сдачу 1 рубль, не имея монет данного номинала. Тогда:

$$F(1, 0) = 1 + \min(\infty) = \infty$$

Нет таких состояний, в которые мы можем попасть из  $F(1, 0)$ . Бесконечность позволяет пометить их как не имеющие решений.

---

**Листинг 64** Вычисление количества способов выдать сдачу  $n$  рублей известными номиналами монет с ограничением на количество монет определенного номинала

---

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<long long>> getF(size_t s, int maxNOnes) {
    vector<vector<long long>> F(
        s + 1,
        vector<long long>(maxNOnes + 1, 0)
    );

    for (int j = 0; j <= maxNOnes; j++) F[0][j] = 1;

    vector<int> denominations = {1, 2, 5};
    for (int sum = 1; sum <= s; sum++)
        for (int remainOnes = 0; remainOnes <= maxNOnes; remainOnes++)
            if (remainOnes)
                for (auto denomination : denominations) {
                    if (sum >= denomination) {
                        bool isOne = denomination == 1;
                        F[sum][remainOnes] +=
                            F[sum - denomination][remainOnes - isOne];
                    }
                }
            else
                for (auto denomination : denominations)
                    if (sum >= denomination && denomination != 1)
                        F[sum][remainOnes] +=
                            F[sum - denomination][remainOnes];
    return F;
}

int main() {
    auto F = getF(10, 2);

    cout << F[4][2]; // 4 способа:
                    // 4 = 2 + 2
                    // 4 = 1 + 1 + 2
                    // 4 = 1 + 2 + 1
                    // 4 = 2 + 1 + 1

    return 0;
}
```

---

## 16.7 Вычисление количества способов выдать сдачу $n$ рублей известными номиналами монет с ограничением на количество монет

**Вычисление количества способов выдать сдачу  $n$  рублей известными номиналами монет с ограничением на количество монет**

Имеются монеты номиналом 1, 2, 5 рублей. Сколько существует способов вернуть сдачу из  $n$  рублей номиналами данных монет при условии, что можно использовать не более  $k = 5$  монет.

Зададим целевую функцию:  $F(n, k)$  - количество способов вернуть сдачу  $n$  рублей, используя ровно  $k$  монет. Параметры целевой функции:

- $n$  – сумма, которую необходимо выдать;
- $k$  – количество оставшихся монет.

Решение:

$$\begin{aligned} F(0, k) &= 1 \quad k \in [0, 5] \\ F(1, 0) &= 0 \quad F(1, k) = F(0, k - 1) = 1 \\ F(2, 0) &= 0 \quad F(2, k) = \overbrace{F(1, k - 1)}^{\substack{\text{выдадим монету} \\ \text{номиналом 1}}} + \overbrace{F(0, k - 1)}^{\substack{\text{выдадим монету} \\ \text{номиналом 2}}} \end{aligned}$$

Итого для произвольных  $n \neq 0$  и  $k$  решение может быть найдено следующим образом:

$$F(n, 0) = 0$$

$$F(n, k) = \sum_{j \in \text{номиналы}} \overbrace{F(n - j, k - 1)}^{[n \geq j, i > 0]}, \quad k \neq 0$$

## 16.8 Задача об оптимальном спуске с горы

**Задача об оптимальном спуске с горы**

В одном из горнолыжных курортов Италии проводятся соревнования по горнолыжному спуску. Каждому спортсмену предстоит скатиться с горы на лыжах.

На любом этапе спуска участник получает определенное число очков. После прохождения трассы очки суммируются. Участник, набирающий наибольшее количество очков, выигрывает.

Гора представляет собой треугольник, в качестве элементов которого выступают целые числа — очки за прохождение этапа. На каждом уровне спортсмену предоставляется выбор — двигаться вниз влево или вниз вправо. Начало спуска — в самой высокой точке горы, конец в одной из самых низких.

Требуется найти максимальное количество очков, которое может набрать спортсмен.

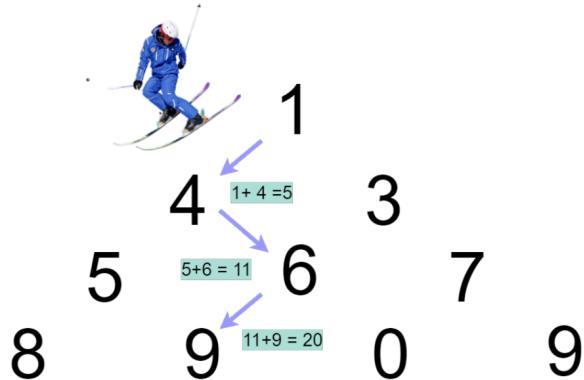
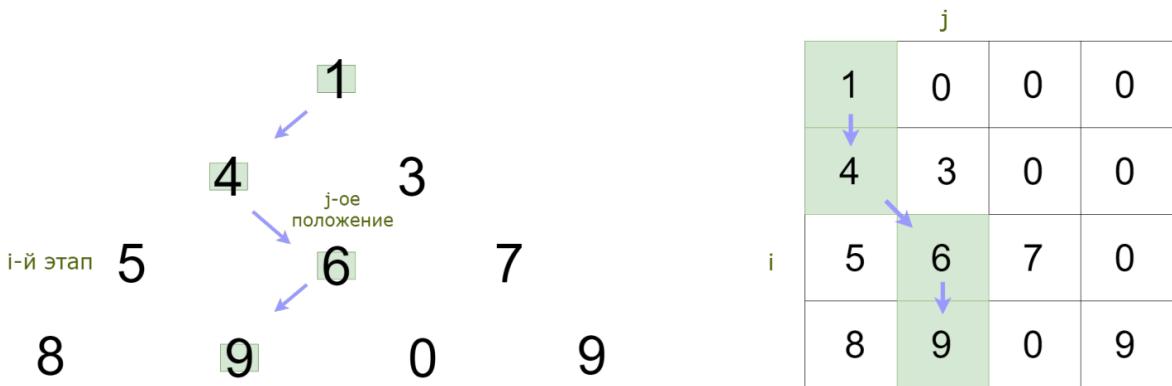


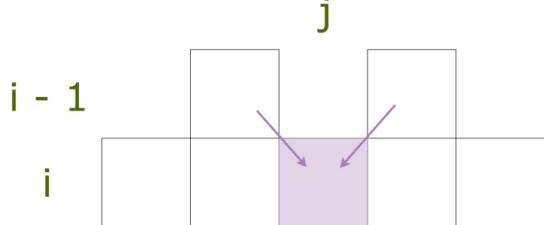
Рис. 16.3: Один из оптимальных маршрутов для лыжника в условиях задачи

Пусть  $F(i, j)$  – максимальная сумма очков, которую можно получить при достижении  $i$  этапа спуска в положении  $j$ . Тогда графическое представление задачи и матрица условия будет выглядеть так, как представлено на рисунке 16.4.

Рис. 16.4: Формирование матрицы  $a$  из известных значений условия задачи

Определимся с формулой перехода. Мы знаем, что лыжник может двигаться вниз-влево или вниз-вправо из определенного положения на горе. Это означает, что в некоторую клетку  $(i, j)$  мы можем попасть либо сверху-слева, либо сверху-справа. Среди всех таких путей нужно найти минимальный.

Наглядное представление:



Представление на матрице:

	$j - 1$	$j$	
$i - 1$	0	0	0
$i$		0	0
		0	0
			0

Рис. 16.5: Множество вариантов попасть в положение  $(i, j)$

---

**Листинг 65** Задача об оптимальном спуске с горы
 

---

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n;
    cin >> n;

    vector<vector<int>> a;
    vector<vector<int>> F;
    for (int i = 0; i < n; i++) {
        a.emplace_back(i + 1);
        F.emplace_back(i + 1);
        for (int j = 0; j <= i; j++)
            cin >> a[i][j];
    }

    F[0][0] = a[0][0];
    for (int i = 1; i < n; i++) {
        F[i][0] = F[i - 1][0] + a[i][0];
        for (int j = 1; j < i; j++)
            F[i][j] = max(F[i - 1][j - 1], F[i - 1][j]) + a[i][j];
        F[i][i] = F[i - 1][i - 1] + a[i][i];
    }

    cout << *max_element(begin(F[n - 1]), end(F[n - 1]));
}

return 0;
}
  
```

---

Из рисунка выше можно получить формулу:

$$F(i, j) = \max(F(i - 1, j - 1), F(i - 1, j)) + a[i][j]$$

Решение в листинге 65.

## 16.9 Задача о счастливых билетах

**Задача о счастливых билетах**

Подсчитайте количество счастливых билетов среди всевозможных строк длины  $n$ , состоящих из цифр ( $n$  чётно). Билет называется счастливым, если сумма цифр первой половины билета совпадает с суммой цифр второй половины билета.

Без каких-нибудь оптимизаций сложность по времени составит  $O(10^n)$ . Однако, мы ставим цель уменьшить затраты по времени для данной задачи.

Попробуем найти  $F(s, k)$  – количество способов представить сумму  $s$ , если у нас осталось  $k$  цифр. Рассмотрим случай, когда билеты четырёхзначные. Получается, что билет разделяется на две группы по 2 цифры. Минимальная сумма, которую можно получить при помощи двух цифр равняется 0, максимальная – 18.

Вычислим значения  $F(s, 2)$  для  $s \in [0, 18]$ . Тогда ответ к задаче мог быть легко найден по формуле:

$$n_{tickets} = \sum_{s=0}^{s \leq 18} F(s, 2)^2$$

Решим задачу при помощи ленивой динамики. Воспользуемся мемоизацией. Решение представлено в листинге 66.

Если кратко описать порядок и формулы перехода, получим, что

$$F(0, k) = 1 \quad F(s, 0) = 0 \quad s \neq 0$$

$$F(s, k) = \sum_{i=0}^{\min(9, s)} F(s - i, k - 1) \quad k \neq 0$$

## 16.10 Задача о покупке билетов

### Покупка билетов

За билетами выстроилась очередь из  $n$  человек, каждый из которых хочет купить 1 билет. На всю очередь работала только одна касса. Самые сообразительные быстро заметили, что, как правило, несколько билетов в одни руки кассир продаёт быстрее, чем когда эти же билеты продаются по одному. Поэтому они предложили нескольким подряд стоящим людям отдавать деньги первому из них, чтобы он купил билеты на всех.

Однако для борьбы со спекулянтами кассир продавала не более 3-х билетов в одни руки, поэтому договориться таким образом между собой могли лишь 2 или 3 подряд стоящих человека. Известно, что на продажу  $i$ -му человеку из очереди одного билета кассир тратит  $a_i$  секунд, на продажу двух билетов —  $b_i$  секунд, трех —  $c_i$  секунд.

Требуется найти минимальное время, за которое могли быть обслужены все покупатели.

Начнём решать задачу с конца. Зададим целевую функцию. Пусть  $F(i, j)$  – минимальное количество денежных средств, затраченных хвостом очереди начиная с  $i$  до  $n$ , если  $i$ -ый участник купит  $j$  билетов.

Предположим, что в очереди стоит  $n$  участников. Последний может купить билеты только для себя:

$$F(n, 1) = a[n, 1] \quad F(n, 2) = F(n, 3) = \infty$$

Предпоследний может купить билет только для себя:

$$F(n - 1, 1) = a[n - 1, 1] + F(n, 1)$$

---

**Листинг 66** Задача о счастливых билетах
 

---

```

#include <iostream>
#include <map>
#include <cassert>
#include <cmath>

using namespace std;

long long int get_F(int s, int k) {
    // словарь будет содержать вычисленные значения  $F(s, k)$ 
    static map<pair<int, int>, unsigned long long int> F;
    // если необходимо выдать сумму 0, то неважно, сколько цифр
    // у нас осталось, так как всегда можно использовать нули
    if (s == 0)
        return 1;

    // если всё-таки сумма ненулевая, а цифр больше нет,
    // тогда и решений нет
    if (k == 0)
        return 0;

    // если ответ при таких  $s$  и  $k$  считался ранее, вернём его
    auto it = F.find({s, k});
    if (it != F.end())
        return it ->second;

    // иначе высчитаем количество способов получить сумму  $s$ 
    // при помощи  $k$  цифр
    long long int value = 0;
    for (int digit = 0, maxPossibleDigit = min(9, s); digit <=
         maxPossibleDigit; digit++)
        value += get_F(s - digit, k - 1);

    // сохраняем ответ (чтобы не считать его вновь)
    F[{s, k}] = value;

    return value;
}

int main() {
    int n;
    cin >> n;

    assert(n % 2 == 0);

    int halfLen = n / 2;
    int maxSum = 9 * halfLen;
    unsigned long long int nTickets = 0;
    for (int i = 0; i <= maxSum; i++)
        nTickets += pow(get_F(i, halfLen), 2);

    cout << nTickets << endl;

    return 0;
}
  
```

---

или на двоих:

$$F(n - 1, 2) = a[n - 1, 2]$$

На троих билеты купить до сих пор нельзя ( $F(n - 1, 3) = \infty$ ).

Продолжим вычисления:

$$F(n - 2, 1) = a[n - 2, 1] + \min(F(n - 1, 1), F(n - 1, 2))$$

$$F(n - 2, 2) = a[n - 2, 2] + F(n, 1)$$

$$F(n - 2, 3) = a[n - 2, 3]$$

В общем случае получим:

$$F(i, j) = \begin{cases} \infty, & \text{если } i + j > n \\ a(i, j), & \text{если } i = n \text{ и } j = 1 \\ a(i, j) + \min_{k=1}^3 (F(i + j, k)), & \text{если } i + j \leq n \end{cases}$$

Оптимальное решение будет заключаться в поиске значения выражения:

$$\min_i^3 (F(1, i))$$

Можно свести задачу к одномерному динамическому программированию. Пусть  $F(i)$  – минимальная стоимость обслуживания, суммарно требуемая на всех клиентов от 1 до  $i$ . Первый покупатель тогда может купить билет только для себя:

$$F(1) = a[1]$$

При учете следующих двух покупателей:

$$F(2) = \min(b[1], F(1) + a[2])$$

$$F(3) = \min(c[1], F[1] + b[2], F[2] + a[3])$$

Для  $i \geq 4$ :

$$F(i) = \min(c[i - 2], F[i - 2] + b[i - 1], F[i - 1] + a[i])$$

Ответ будет сохранён в  $F(n)$ .

## 16.11 О играх

### О играх

Дано  $2n$  спортсменов, которые выстроились в два ряда друг напротив друга.

Каждый спортсмен характеризуется своим ростом. Рост первого ряда записан в массиве  $a$ , второго ряда – в  $b$  (нумерация спортсменов с единицы). Требуется набрать команду для игры. Игровые будут выбираться слева-направо, можно взять только одного игрока из пары (или из первого, или из второго ряда, или ни из одного). А для того, чтобы не отдавать предпочтения одному из рядов, каждый следующий **выбранный** школьник должен стоять не в том же ряду, что предыдущий.

В новой игре считается, что чем больше суммарный рост участников, тем более хороший результат покажет команда при игре (количество игроков неважно). Определите максимальную сумму ростов.

Тест	Ожидаемый результат
<b>9 3 5 7 3</b>	29
<b>5 8 1 4 5</b>	
<b>1 2 9</b>	19
<b>10 1 1</b>	

Таблица 16.1: Тестовые данные. Оптимальный выбор в каждом тесте выделен

Попробуем решить данную задачу с начала. Пусть  $F(i, j)$  – максимальный рост первых  $i$  участников, при условии, что последним был выбран спортсмен из ряда  $j$ .

База:

$$F(0, 1) = 0 \quad F(0, 2) = 0$$

Выведем формулу перехода на примере первого ряда. Предположим, рассматривается спортсмен с номером  $i$ . Он может как быть взят в команду, так и пропущен. Если его взять в команду, то он продолжит цепочку, закончившуюся на спортсменах второго ряда. Но, может быть, лучше не брать его, и передать накопившееся значение по первому ряду дальше. Так как нужно максимизировать рост:

$$F(i, 1) = \max(F(i - 1, 2) + a[i], F(i - 1, 1)) \quad i \in [1, n]$$

Аналогично для второго ряда:

$$F(i, 2) = \max(F(i - 1, 1) + b[i], F(i - 1, 2)) \quad i \in [1, n]$$

Ответ:

$$x = \max(F(n, 1), F(n, 2))$$

---

**Листинг 67** Задача об игроках

---

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

using namespace std;

int main() {
    size_t n;
    cin >> n;

    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> a[i];

    vector<int> b(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> b[i];

    vector<vector<long long>> F(n + 1, vector<long long>(2, 0));
    for (int i = 1; i <= n; i++) {
        F[i][0] = max(F[i - 1][1] + a[i], F[i - 1][0]);
        F[i][1] = max(F[i - 1][0] + b[i], F[i - 1][1]);
    }

    cout << max(F[n][0], F[n][1]);

    return 0;
}
```

---

## 16.12 О художниках

### О художниках

Знаменитое объединение художников «Калевич жив!» поставило на поток производство шедевров. Объединение состоит из  $n$  художников, которые решили следующим образом организовать свою работу.

Каждый из художников будет использовать только один закрепленный за ним цвет. Цвета для всех художников различны, будем считать, что первый художник использует первый цвет, второй художник – второй цвет, и так далее. Каждый шедевр будет содержать все эти цвета. Чтобы наложить  $j$ -й цвет на  $i$ -й шедевр,  $j$ -му художнику потребуется  $t_{ij}$  единиц времени.

Во всем нужен порядок, поэтому работа художников упорядочена по следующим правилам:

- каждый шедевр сначала обрабатывается первым художником, потом вторым и т.д., то есть после окончания обработки  $j$ -м художником шедевр должен быть обработан  $(j+1)$ -ым (если  $j < n$ )
- **каждый из художников работает над шедеврами по порядку:** сначала над первым шедевром, потом над вторым и так далее;
- каждый художник может одновременно работать не более чем над одним шедевром, но времени на отдых им не требуется;
- как только  $j$ -й художник заканчивает работу над шедевром, тот мгновенно передается следующему художнику.

Считая, что художники приступят к работе в момент времени 0, найдите для каждого шедевра момент времени, когда он будет готов к продаже.

Тест	Ожидаемый результат
$t = \begin{pmatrix} 2 & 5 \\ 3 & 1 \\ 5 & 3 \\ 10 & 1 \end{pmatrix}$	$x = \{7, 8, 13, 21\}$

Таблица 16.2: Тестовые данные. Оптимальный выбор в каждом тесте выделен. Работа над 4-мя шедеврами двумя художниками

Для большей наглядности проиллюстрируем первый тест:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	2																			
	1				2				3											4

Пусть  $F(i, j)$  – минимальное время, которое потребуется  $j$ -ому художнику для работы над  $i$ -ым шедевром.

Попробуем найти формулу перехода, через последовательное вычисление  $F(i, j)$ . Первый художник сразу начнёт работать и закончит свою часть работы за  $t[1][1]$ :

$$F(1, 1) = t[1][1]$$

Второй художник может закончить свою работу только после того, как закончит работать первый, и его суммарное время составит:

$$F(1, 2) = F(1, 1) + t[1][2]$$

Рассмотрим работу над вторым шедевром. Первый художник, как только закончит работу над первым шедевром, так сразу же переключится на второй:

$$F(2, 1) = F(1, 1) + t[2][1]$$

Второй художник начнёт работу либо после того, как закончит первый шедевр, или только после того, как первый художник закончит второй шедевр. Поэтому:

$$F(2, 2) = \max(F(2, 1), F(1, 2)) + t[2][2]$$

Тогда для произвольного  $i > 1$ :

$$F(i, 1) = F(i - 1, 1) + t[i][1]$$

$$F(i, 2) = \max(F(i, 1), F(i - 1, 2)) + t[i][2])$$

Теперь предположим, что художников больше, чем двое. И если распространить рассуждения на них, получим:

$$F(1, 1) = t[1][1]$$

$$F(i, 1) = F(i - 1, j) + t[i][1] \quad i \in [2, n]$$

$$F(1, j) = F(1, j - 1) + t[1][j] \quad j \in [2, m]$$

$$F(i, j) = \max(F(i - 1, j), F(i, j - 1)) + t[i][j] \quad i \in [2, n], j \in [2, m]$$

Решении на листинге 68.

---

**Листинг 68** О художниках

---

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

using namespace std;

int main() {
    size_t n, m;
    cin >> n >> m;

    vector<vector<long long>> a(n + 1, vector<long long>(m + 1, 0));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> a[i][j];

    for (int i = 2; i <= n; i++)
        a[i][1] += a[i - 1][1];

    for (int j = 2; j <= m; j++)
        a[1][j] += a[1][j - 1];

    for (int i = 2; i <= n; i++)
        for (int j = 2; j <= m; j++)
            a[i][j] += max(a[i - 1][j], a[i][j - 1]);

    for (int i = 1; i <= n; i++)
        cout << a[i][m] << ' ';

    return 0;
}
```

---

### 16.13 Поиск подпоследовательности наибольшей длины с неотрицательными префиксными суммами

Поиск подпоследовательности наибольшей длины с неотрицательными префиксными суммами

Дана числовая последовательность, требуется найти длину наибольшей подпоследовательности (подпоследовательность может быть получена из последовательности путем вычеркивания некоторых элементов), префиксный массив сумм которой не содержит отрицательных значений.

Например, из последовательности 4 -4 1 -3 1 -3 можно взять элементы равные 4 1 -3 1 -3. Массив префиксных сумм: 0 4 5 2 3 0 не содержит отрицательных значений.

Пусть  $F(i, j)$  определяет максимальное значение префиксной суммы при условии, что мы использовали  $j$  элементов последовательности из  $i$ .

Для первого элемента:

$$F(1, 0) = 0 \quad F(1, 1) = a[1] = 4$$

Для второго элемента:

$$F(2, 0) = 0 \quad F(2, 1) = \max(a[2] + F(1, 0), F(1, 1)) = 4 \quad F(2, 2) = F(1, 1) + a[2] = 0$$

Для третьего элемента:

$$F(3, 0) = 0$$

$$F(3, 1) = \max(a[3], F(2, 1), F(1, 1)) = 4$$

$$F(3, 2) = \max(a[3] + \max(F(2, 1), F(1, 1)), F(2, 2)) = 5$$

$$F(3, 3) = F(2, 2) + a[3] = 1$$

Для четвертого элемента:

$$F(4, 0) = 0$$

$$F(4, 1) = \max(a[4], F(3, 1), F(2, 1), F(1, 1)) = 4$$

$$F(4, 2) = \max \left( \underbrace{a[4] + \max(F(3, 1), F(2, 1), F(1, 1))}_{\text{Если берём текущий элемент}}, \underbrace{F(3, 2), F(2, 2)}_{\text{Если игнорируем текущий}} \right) = 5$$

$$F(4, 3) = \max \left( \underbrace{a[4] + \max(F(3, 2), F(2, 2), F(1, 2))}_{\text{Если берём текущий элемент}}, \underbrace{F(3, 3)}_{\text{Если игнорируем текущий}} \right) = 2$$

$$F(4, 4) = F(3, 3) + a[3] = -2$$

При вычислении  $F(4, 4)$  получили отрицательное значение, что не удовлетворяет условию задачи. Выполним замену:

$$F(4, 4) = -\infty$$

Для произвольных  $i$  и  $j$  получим<sup>1</sup>:

$$F(i, 0) = 0$$

$$F(i, j) = \max(a[i] + \max_{k=j-1}^{i-1} F(k, j-1), \max_{k=j}^{i-1} F(k, j))$$

При получении отрицательных  $F(i, j)$  необходимо выполнить замену на  $-\infty$ .

## 16.14 Задача о рюкзаке

### О рюкзаке

Дано  $N$  предметов,  $a_i$  предмет имеет массу  $w_i > 0$  и стоимость  $p_i > 0$ . Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины  $W$  (вместимость рюкзака), а суммарная стоимость была максимальна.

Пусть  $b_i = 1$ , если мы берём  $i$ -ый предмет, иначе -  $b_i = 0$ . Тогда по условию задачи требуется:

$$\sum_{i=1}^N p_i b_i \rightarrow \max \quad \sum_{i=1}^N w_i b_i \leq W$$

При помощи наивного алгоритма можно добиться сложности  $O(2^N)$  за счёт перебора всех подмножеств множества предметов. Однако метод динамического программирования позволяет уменьшить сложность до  $O(NW)$ .

### Задача о суммах подмножеств

Рассмотрим более простой вариант задачи:

### Задача о суммах подмножеств

Имеется  $n$  слитков золота, каждый имеет свой вес. И рюкзак вместимости  $W$ .

Необходимо вычислить максимальный вес, который можно унести в рюкзаке. Вы не можете дробить слитки.

На ум приходят такие решения:

- Отсортировать слитки по возрастанию веса и будем брать от меньшего к большему, пока влезает в рюкзак.
- Брать слитки по убыванию веса и брать первый попавшийся, который помещается.

Это классические примеры жадных алгоритмов<sup>2</sup>. К сожалению, в задаче о рюкзаке подобные алгоритмы не работают. Легко увидеть на примере: вместимость рюкзака 10, веса слитков  $w = \{2, 3, 4, 4, 6, 9\}$ . В первом случае получим  $w' = \{2, 3, 4\}$ ,

<sup>1</sup>Ожидаемо, что это не единственный способ решить задачу. При использовании приоритетной очереди можно достичь меньшей сложности и по памяти, и по времени.

<sup>2</sup>Optimal solution can't be guaranteed by greedy algorithms, why? Because greedy is not the solution for everything. Перевод: "Почему жадные алгоритмы не гарантируют оптимальное решение? Потому что жадность не всегда хорошее решение."

во втором –  $w' = \{9\}$ . Ни один из вариантов не нашел оптимальное решение с суммарным весом 10.

Пусть  $F(i, w) = 1$ , если можно набрать суммарный вес  $w$ , используя только какие-то из первых  $i$  камней, иначе  $F(i, w) = 0$ .

Заметим, что для того, чтобы вычислить  $F(i, w)$  нам достаточно рассмотреть судьбу  $i$ -го камня: мы можем его либо взять положить в рюкзак или нет. Если мы его не берем, то нам нужно при помощи камней с номерами до  $i - 1$  набрать вес  $w$ . Если же мы его берем, то при помощи  $i - 1$  камня нам нужно набрать вес  $w - w_i$ . То есть, если хотя бы одно из этих состояний динамики равно 1, то и  $F(i, w)$  тоже равно 1. Иначе – не зависимо от того, берем ли мы камень или нет, набрать нужный вес нельзя.

Отдельно обратим внимание, что иногда мы не можем взять  $i$ -й камень, даже если захотим. Это происходит, когда вес, который мы хотим набрать, меньше веса  $i$ -го камня. В таком случае у нас нет выбора и  $F(i, w) = F(i - 1, w)$ .

Итоговая формула:

$$F(i, w) = \begin{cases} [F(i - 1, w - w_i) \text{ or } F(i - 1, w)], & \text{если } w_i \leq w \\ F(i - 1, w), & \text{в противном случае.} \end{cases}$$

Определимся с порядком обхода. Каждый раз мы смотрим только на две клетки из предыдущей строки, поэтому достаточно перебрать динамику в порядке возрастания  $i$  и затем набранный вес – в любом порядке, пусть тоже по возрастанию.

Возьмем в качестве базы ситуацию, когда мы использовали 0 слитков. Какой вес можно набрать, набрав 0 слитков? Верно, ноль. Поэтому  $F(0, 0) = 1$ . Остальные состояния вида  $F(0, i)$  будут равны 0. База примет вид

$$F(0, 0) = 1 \quad F(0, i) = 0 \quad i \neq 0$$

Ответ в задаче не так просто найти, как в прошлых. Потому что в  $F(n, W)$  будет находиться информация о том, можем ли мы заполнить рюкзак полностью. Нас же интересует максимальный вес, который мы можем набрать. Ответом будет наибольшее значение  $w$ , такое что  $F(n, w) = 1$ .

Программное решение на листинге 69. Если запустить программу при  $W = 10$  и  $w = \{2, 3, 4, 4, 6, 9\}$  будет получена таблица  $F$  со следующим содержимым:

Предмет (Вес)	0	1	2	3	4	5	6	7	8	9	10
0 (0)	+	-	-	-	-	-	-	-	-	-	-
1 (2)	+	-	+	-	-	-	-	-	-	-	-
2 (3)	+	-	+	+	-	+	-	-	-	-	-
3 (4)	+	-	+	+	+	+	+	+	-	+	-
4 (4)	+	-	+	+	+	+	+	+	+	+	+
5 (6)	+	-	+	+	+	+	+	+	+	+	+
6 (9)	+	-	+	+	+	+	+	+	+	+	+

Используя таблицу, можно узнать, какие предметы стоит взять, чтобы получить суммарный вес равный  $w$ . Достаточно продвигаться с ячейки  $F(6, w)$  вверх до тех пор, пока там есть плюс. Если мы двигаемся вверх, то предмет не берём. Если плюса сверху нет, мы вынуждены взять  $i$ -ый предмет, и тогда нам останется собрать не исходные  $w$  кг., а  $w - w_i$  кг.

---

**Листинг 69** Задача о суммах подмножеств

---

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    int W, n;
    cin >> W >> n;

    vector<int> w(n + 1);
    for (int i = 1; i <= n; ++i)
        cin >> w[i];

    vector<vector<bool>> F(n + 1, vector<bool>(W + 1, false));
    F[0][0] = true;
    for (int i = 1; i <= n; ++i)
        for (int j = 0; j <= W; ++j)
            F[i][j] = F[i - 1][j] || w[i] <= j && F[i - 1][j - w[i]];

    // получение ответа
    int maxWeight = W;
    while (maxWeight >= 0 && !F[n][maxWeight])
        --maxWeight;

    cout << maxWeight;

    return 0;
}
```

---

# Глава 17

## Продвинутые техники в динамическом программировании при решении олимпиадных задач

### 17.1 Matrix Exponentiation

Перед тем, как перейти к прочтению данного раздела убедитесь, что знаете:

- Матрицы и операции над ними.
- Алгоритм быстрого возведения в степень.

Динамическое программирование – довольно излюбленная тема олимпиадного программирования. Она оперирует состояниями и формулами перехода между ними. При небольшом количестве переходов проблем не наблюдается. Однако при больших  $n$  требуется использование более совершенных техник, об одной из которых пойдёт речь.

#### 17.1.1 Числа Фибоначчи

##### Числа Фибоначчи

Найти  $n$ -е ( $n \geq 10^{10}$ ) число Фибоначчи по модулю  $10^9 + 7$ .

Ещё раз обозначу: идея динамического программирования здесь работает, но при больших  $n$  потребуется много времени на вычисление. Рассмотрим уже известное:

$$F(0) = 0 \quad F(1) = 1 \quad F(i) = F(i - 1) + F(i - 2)$$

Пусть нам даны состояния  $F(i - 2)$  и  $F(i - 1)$ . Создадим из них матрицу-столбец  $A$ :

$$A = \begin{pmatrix} F(i - 1) \\ F(i - 2) \end{pmatrix}$$

Предположим, что наши состояния могут быть связаны некоторым таким образом, что последующее может быть получено умножением известных предыдущих состояний на матрицу  $M$ :

$$M * \begin{pmatrix} F(i - 1) \\ F(i - 2) \end{pmatrix} = \begin{pmatrix} F(i) \\ F(i - 1) \end{pmatrix}$$

Вычислим матрицу  $M$ :

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} F(i-1) \\ F(i-2) \end{pmatrix} = \begin{pmatrix} F(i-1) + F(i-2) \\ F(i-1) \end{pmatrix} = \begin{pmatrix} F(i) \\ F(i-1) \end{pmatrix}$$

Матрица  $M$  связывает состояния между собой:

$$\{F(0), F(1)\} \xrightarrow{*M} \{F(1), F(2)\} \xrightarrow{*M} \{F(2), F(3)\} \xrightarrow{*M} \dots \xrightarrow{*M} \{F(i-1), F(i)\} \xrightarrow{*M} \dots$$

Если выполнить умножение один раз, получим следующее состояние, если две – состояние через ещё одно. Несложно показать, что при необходимости найти  $i$ -ый член последовательности  $i \geq 1$  Фибоначчи достаточно найти:

$$\begin{pmatrix} F(i) \\ F(i-1) \end{pmatrix} = M^{i-1} * \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Осталось лишь применить схему быстрого возведения в степень для получения  $M^{i-1}$ . Таким образом, времененная сложность алгоритма  $O(\log N)$ .

### 17.1.2 Задача о треугольниках

#### Задача о треугольниках

Изначально дан треугольник. Над ним проводятся разбиения, показанные на рисунке 17.1.

Требуется найти количество треугольников, направленных острым углом вверх.

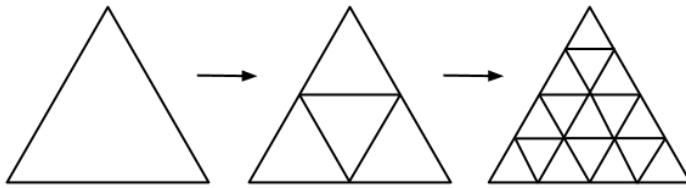


Рис. 17.1: Рисунок к задаче о треугольниках

Пусть  $i$ -е состояние динамики описывается двумя числами  $a_i$  и  $b_i$  – количество треугольников, которые смотрят вверх и вниз соответственно при условии, что проведено  $i$  разбиений.

$$F(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad F(1) = \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad F(2) = \begin{pmatrix} 10 \\ 6 \end{pmatrix}$$

Подберём такую матрицу  $M$ , которая связывала бы состояния. Треугольник, смотрящий острым углом вверх порождает 3 таких же треугольника, и ещё один треугольник, который "смотрит" вниз. И с точностью дооборот ситуация для треугольника, направленного острым углом вниз:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \xrightarrow{\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}^*} \begin{pmatrix} 3 \\ 1 \end{pmatrix} \xrightarrow{\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}^*} \begin{pmatrix} 10 \\ 6 \end{pmatrix} \xrightarrow{\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}^*} \dots$$

Таким образом,

$$F(i) = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}^i * \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

### 17.1.3 Расшифровка генома

#### Расшифровка генома

Дана строка  $s$ , представляющая некоторую последовательность нуклеотидов. Однако некоторые подпоследовательности длины 2 не могут входить в цепочку. Рассмотрим на примере. Пусть ДНК представляется 4 нуклеотидами:  $a, b, c, d$ , и запрещены цепочки:  $ab, ac, bd, cc, ca$ .

Требуется найти количество допустимых цепочек длины  $n$  в соответствии с ограничениями на запрещенные пары нуклеотидов.

Пусть  $i$ -е состояние описано так:

$$F(i) = \begin{pmatrix} a_i \\ b_i \\ c_i \\ d_i \end{pmatrix}$$

где  $a_i$  – количество корректных цепочек длины  $i$ , заканчивающихся на  $a$ ,  $b_i$  – количество корректных цепочек длины  $i$ , заканчивающихся на  $b$  и так далее.

Цепочка из одного символа является корректной, поэтому:

$$F(1) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Подумаем, какая бы могла быть матрица  $M$ , связывающая состояния. Допустим, на каком-то этапе  $i$  получилась ситуация, что цепочка заканчивалась на  $a$ . В силу наличия запрещенных пар, она может быть продолжена только одним из символов:  $a, d$ . Следовательно  $a_{i+1}$  будет вычисляться

$$a_{i+1} = a_i + d_i$$

Для других значений получим:

$$b_{i+1} = a_i + b_i + c_i$$

$$c_{i+1} = b_i + d_i$$

$$d_{i+1} = a_i + b_i + c_i + d_i$$

Матрица, позволяющая производить преобразование из  $F(i)$  в  $F(i+1)$  выглядит так:

$$M = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Тогда количество корректных цепочек длины  $n$  будет находиться как сумма элементов матрицы  $F$ :

$$F(i) = M^{i-1} * \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

### 17.1.4 Тетраэдр

#### Тетраэдр

Задан тетраэдр. Обозначим его вершины буквами  $A, B, C$  и  $D$  соответственно (рисунок 17.2).

В вершине тетраэдра  $D$  находится муравей. Муравей очень подвижный и не любит стоять на месте. В каждый момент времени он совершает один шаг от одной вершины к другой по некоторому ребру тетраэдра, оставаться на месте он не может.

Нужно посчитать каким количеством способов муравей может прийти из исходной вершины  $D$  в себя ровно за  $n$  шагов.

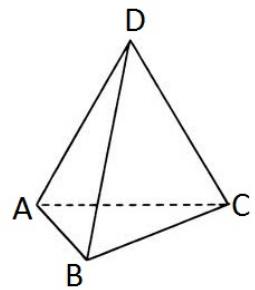


Рис. 17.2: Тетраэдр

Попробуйте выполнить решение самостоятельно.

## Глава 18

### Лабораторные работы

## 18.1 Лабораторная работа «Стандартный ввод и вывод»

**Цель работы:** получение навыков использования функций ввода и вывода стандартной библиотеки *stdio*.

**Содержание отчета:**

- Тема лабораторной работы
- Цель лабораторной работы
- Составить программу для
  - ввода и вывода символов и строк.
  - ввода и вывода значений каждого из базовых типов с использованием функций форматного ввода и вывода с соответствующими допустимыми для данного типа символами преобразований.
  - ввода и вывода значений модифицированных базовых типов.
  - ввода и вывода значений с использованием флагов, точности и ширины.
- Ответы на контрольные вопросы (дайте их без предварительной проверки на компьютере)
- Текст программы, посредством которой могут быть получены ответы
- Анализ допущенных ошибок (какие ответы не сошлись, объяснение увиденному поведению ПО).

**Контрольные вопросы:**

1. Что будет выведено при выполнении следующих операторов:

- (a) \_\_\_\_\_  
`printf("%hu\n", 2-3);`
- (b) \_\_\_\_\_  
`printf("%-4cd%3i\n", 65, 'A');`
- (c) \_\_\_\_\_  
`printf("%-7i %c\n", 12389, 'a');`
- (d) \_\_\_\_\_  
`printf("%4.2f\n", 345.789);`
- (e) \_\_\_\_\_  
`printf("%#o, %#X\n", 345, 345);`
- (f) \_\_\_\_\_  
`printf("%f\n", .019278912);`
- (g) \_\_\_\_\_  
`printf("%e\n", .0019278912e-1);`
- (h) \_\_\_\_\_  
`printf("%g\n", .019278912);`
- (i) \_\_\_\_\_  
`printf("%8.2f\n", 19.915);`
- (j) \_\_\_\_\_  
`printf("%8.2e\n", 19.915);`
- (k) \_\_\_\_\_  
`printf("%8g\n", 19.915);`

2. Какое значение вернет функция `printf("%4.2f", 345.789)` при успешном выполнении?

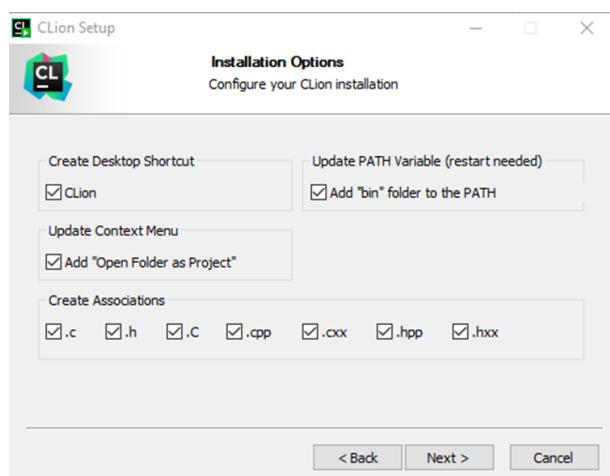
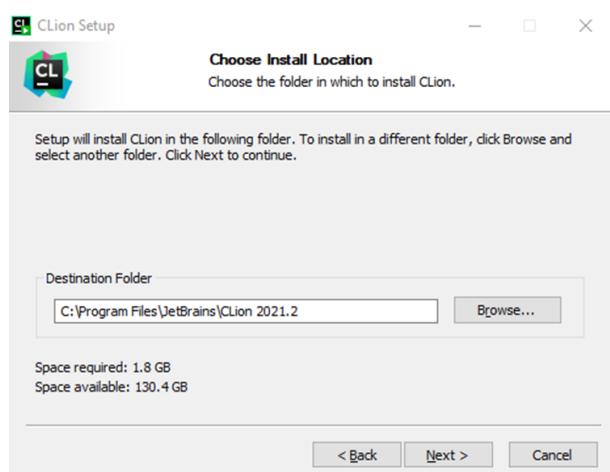
3. Какое значение вернет функция `scanf("%3f %4s %c", &f, s, &c)` при успешном выполнении?
4. Какие значения будут присвоены переменным соответствующих типов `f`, `s`, `c` после вызова функции `scanf("%3f %8s %c", &f, s, &c)`, если на клавиатуре набрано: 1234.56 Belgorod 308012?

## Пояснения к лабораторной работе

Для выполнения последующих работ вам потребуется компилятор и среда разработки<sup>1</sup>.

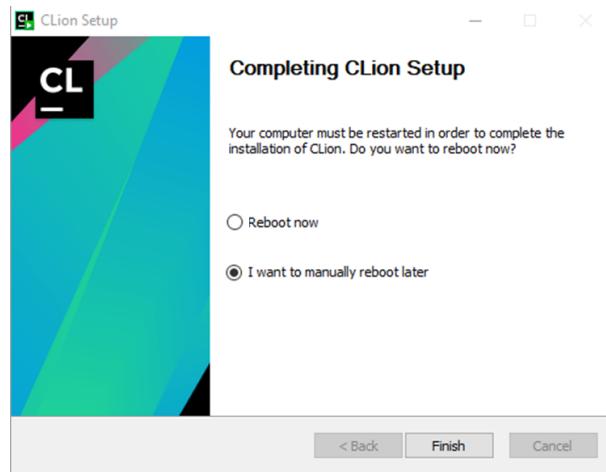
### Установка CLion:

1. Выбираем свою систему и скачиваем установщик CLion с официального сайта: <https://www.jetbrains.com/clion/download/> (Далее будут представлены инструкции для системы Windows)
2. Проходим процедуру установки:



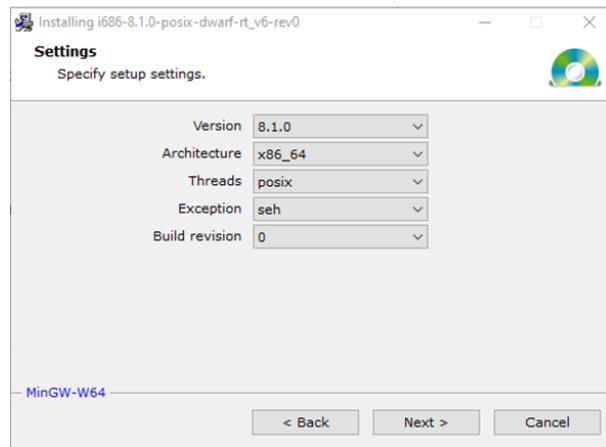
3. Закрываем установщик. Перезагрузку мы выполним после установки компилятора.

<sup>1</sup>Инструкция любезно предоставлена Николотовым Александром



### Установка компилятора:

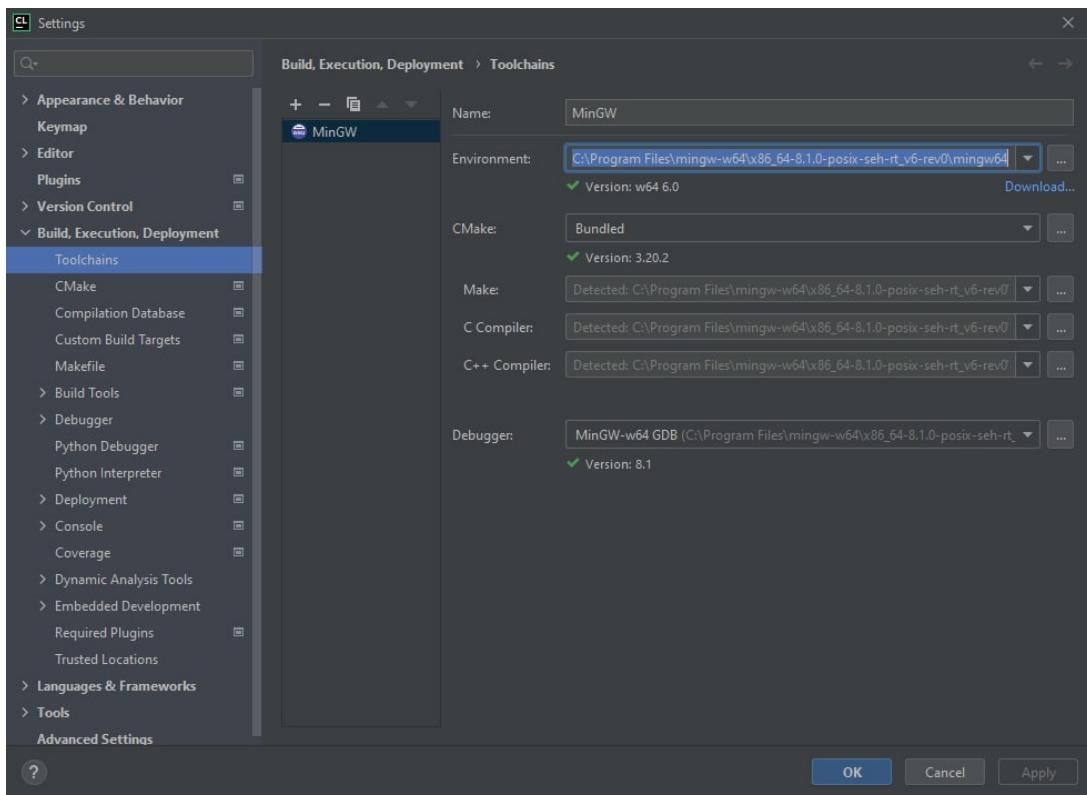
- Скачиваем установщик MinGw для своей системы: <https://www.mingw-w64.org/downloads/> (Для windows выбираем “MingW-W64-builds”).
- При установке выбираем “Architecture” - “x86\_64”, в случае если у вас установлена 64-битная версия ОС, или “I686” - в противном случае.



- После установки, перезагружаем ПК.

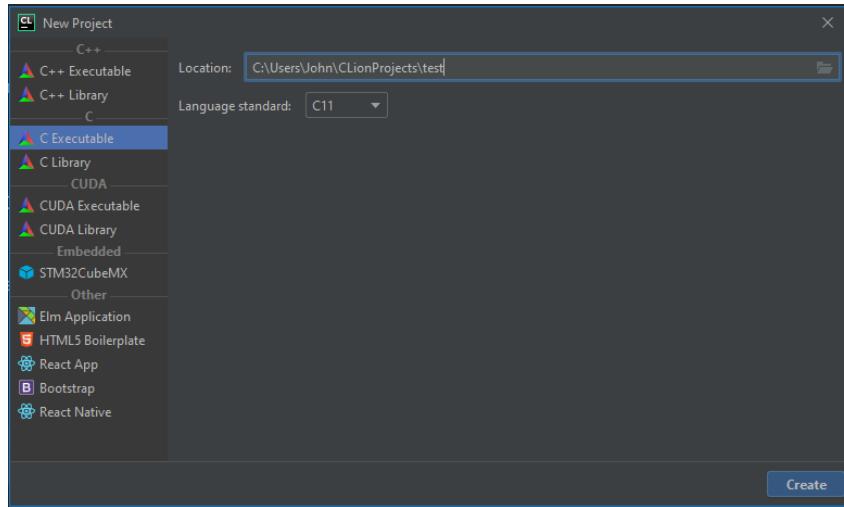
### Старт работы в CLion:

- Запускаем CLion, через меню переходим в: File / Settings / Build, Execution, Deployment / Toolchains.
- Выбираем “MinGW” и указываем путь к папке “mingw64”, например: “C:\Program Files\mingw-w64\x86\_64-8.1.0-posix-seh-rt\_v6-rev0\mingw64”.



3. Сохраняем настройки.

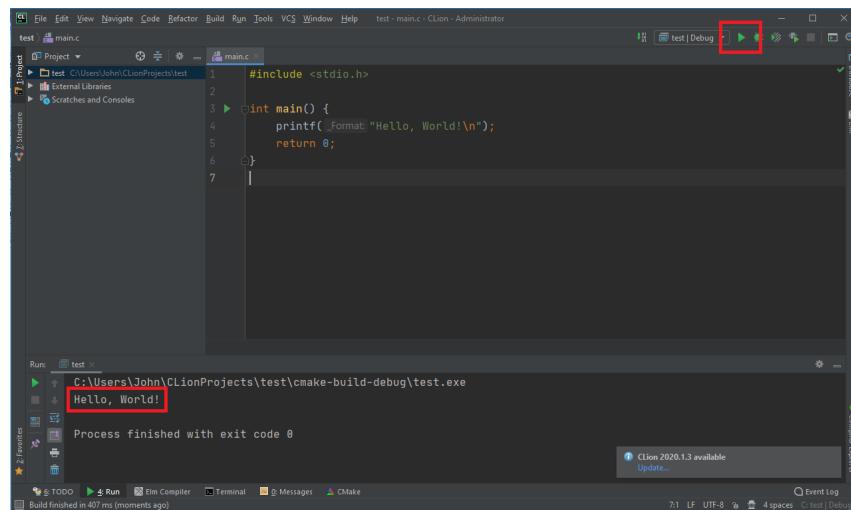
4. Создаем пустой C-проект (File / New project):



5. Запускаем наш Hello World

#### Пара советов:

- При возникновении любой внештатной ситуации во время установки или запуска проекта, гуглите. Важно понимать, что вы вряд ли первый, у кого возникли трудности на начальном этапе настройки среды программирования.
- Изучите среду. Она – ваш главный инструмент (после поисковика *google*). Настройте под себя тему, размер шрифта. Почитайте про комбинации клавиш и снippets. Важно чтобы **вам нравился ваш редактор и вы умели в нем ориентироваться**.



- Как можно скорее научитесь пользоваться отладчиком.

## 18.2 Лабораторная работа «Алгоритмы разветвляющиеся структуры»

**Цель работы:** получение навыков написания линейных и разветвляющихся алгоритмов.

### Содержание отчета:

- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
  - Название задачи.
  - Для задач со звездочкой приложить блок-схему.
  - Исходный код.
  - Скриншот с *codeforces* с указанием вердикта тестирующей системы.
- Вывод.

### Перечень задач:<sup>2</sup>

1. Минуты до Нового года (1283A)<sup>3</sup>
2. Опять двадцать пять! (630A)<sup>4</sup>
3. Игра (513A)<sup>5</sup>
4. Слоник (617A).
5. Задача про арбуз (4A)
6. \* Хипстер Вася (581A)
7. Солдат и бананы (546A)
8. Подсчёт функции (486A)
9. Освещение парка (1358A)
10. \*Ручки и карандаши (1244A)<sup>6</sup>
11. \*Покупка воды (1118A)
12. Блэкджек (104A)

---

<sup>2</sup>Щелчок по названию задачи переведёт на условие.

<sup>3</sup>Разбор задачи имеется в примере оформления на странице 422.

<sup>4</sup>Решения некоторых задач проще, чем кажется на первый взгляд. Если у вас возникает желание использовать циклы – это плохая идея.

<sup>5</sup>Для решения некоторых задач вам могут потребоваться не все исходные данные. Иногда достаточно воспользоваться частью.

<sup>6</sup>Если вы начинаете сгонять муху с монитора при помощи курсора мыши, пора выключать компьютер.

13. \*Сделай треугольник! (1064A)

14. Уравнение (1269A)

15. Компот (746A)

16. Кнопочные гонки (835A)

## Пояснения к лабораторной работе

Среди списка задач будут такие, в которых программа должна обработать несколько наборов тестовых данных. Это можно сделать при помощи циклов. Несмотря на то, что циклы ещё не были изучены, я оставлю шаблон, посредством которого вы можете описывать свои решения в таких случаях:

---

```
#include <stdio.h>

int main() {
    int n_sets;
    scanf("%d", &n_sets);

    for (int set_number = 1; set_number <= n_sets; set_number++) {
        // код для одного случая
    }

    return 0;
}
```

---

Только в этом случае допускается использовать циклы. Во всех других ситуациях использование циклов **категорически запрещено**.

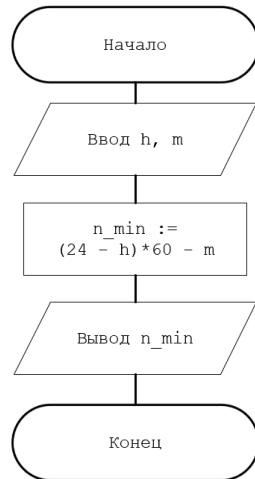
Рассмотрим в качестве примера первую задачу (Минуты до Нового года). В описании входных данных указано следующее:

*Первая строка входных данных содержит одно целое число  $t$  ( $1 \leq t \leq 1439$ ) – количество наборов входных данных.*

*Следующие  $t$  строки содержат наборы входных данных:  $i$ -я строка содержит время в виде двух целых чисел  $h$  и  $m$  ( $0 \leq h \leq 24$ ,  $0 \leq m < 60$ ). Гарантируется, что это время не равно полуночи, то есть одновременно не могут выполняться два условия  $h = 0$  и  $m = 0$ . Гарантируется, что  $h$  и  $m$  заданы без лидирующих нулей.*

входные данные	Скопировать
5 23 55 23 0 0 1 4 20 23 59	

Выполним решение задачи. Опишем алгоритм решения задачи при помощи блок-схемы.



Закодировав данный фрагмент получим:

---

```

int h, m;
scanf("%d %d", &h, &m);

int n_min = (24 - h)*60 - m;

printf("%d\n", n_min);
  
```

---

Вставим полученный фрагмент в функцию *main*:

---

```

#include <stdio.h>

int main() {
    int n_sets;
    scanf("%d", &n_sets);

    for (int set_number = 1; set_number <= n_sets; set_number++) {
        int h, m;
        scanf("%d %d", &h, &m);

        int n_min = (24 - h)*60 - m;

        printf("%d\n", n_min);
    }

    return 0;
}
  
```

---

На странице с задачей выбираем пункт "Отослать":

 Sponsored by Telegram

ГЛАВНАЯ ТОП СОРЕВНОВАНИЯ ТРЕНИРОВКИ АРХИВ ГРУППЫ РЕЙТИНГ EDU API К

ЗАДАЧИ ОТОСЛАТЬ МОИ ПОСЫЛКИ СТАТУС ВЗЛОМЫ ПОЛОЖЕНИЕ ЗАПУСК

А. Минуты до Нового года

ограничение по времени на тест: 1 секунда  
ограничение по памяти на тест: 256 мегабайт  
ввод: стандартный ввод  
выход: стандартный вывод

В следующем окне указываем нашу задачу, используемый язык и вставляем код:

Отослать решение  
Codeforces Round #611 (Div. 3)

Задача:	A - Минуты до Нового года
	стандартный ввод/вывод
	1 с, 256 МБ
Язык:	GNU GCC C11 5.1.0
<pre> 1 #include &lt;stdio.h&gt; 2 3 int main() { 4     int n_sets; 5     scanf("%d", &amp;n_sets); 6 7     for (int set_number = 1; set_number &lt;= n_sets; set_number++) { 8         int h, m; 9         scanf("%d %d", &amp;h, &amp;m); 10        int n_min = (24 - h)*60 - m; 11        printf("%d\n", n_min); 12    } 13 14    return 0; 15 }</pre>	
Исходный код:	

Внизу страницы нажимаем на кнопку "Отослать". Если всё пройдёт успешно, вы увидите в столбце "Вердикт" строку "Полное решение":

Мои посылки							
№	Когда	Кто	Задача	Язык	Вердикт	Время	Память
<a href="#">126443394</a>	19.08.2021 15:26	ispritchin	<a href="#">A - Минуты до Нового года</a>	GNU C11	<b>Полное решение</b>	0 мс	3600 КБ

## Ошибки, допускаемые при решении и рекомендации:

- **Игнорирование требований.** Если сказано, что использование циклов вне описанного назначения запрещено, это означает, что использование циклов вне описанного назначения запрещено. Совсем запрещено. Без исключений. Это может показаться странным, но каждое слово в лабораторной означает ровно то, что означает (понимаю, что в это сложно поверить).
- Отсутствие форматирования кода. В среде разработки *CLion* это можно сделать посредством комбинации клавиш *Ctrl+Alt+L*. Вы пишете код не для себя, **вы пишете код для других программистов**, так что будьте добры постараться.
- Использование конструкции:

---

```
// if (<логическое выражение1>
//      <оператор1>
// else {
//     if (<логическое выражение2>
//          <оператор2>
//     else
//          <оператор3>
// }
```

---

вместо:

---

```
// if (<логическое выражение1>
//     <оператор1>
// else if (<логическое выражение2>
//     <оператор2>
// else
//     <оператор3>
```

---

- Предположим, имеется несколько логических выражений, которые не могут быть попарно истинны (т. е. если выполняется логическое выражение1, то все остальные логические выражения будут давать ложь (и так для всех пар)). Правильный способ закодировать выглядит следующим образом:

---

```
// if (<логическое выражение1>
//     <оператор1>
// else if (<логическое выражение2>
//     <оператор2>
// else
//     <оператор3>
```

---

но не так:

---

```
// if (<логическое выражение1>
//     <оператор1>
// if (<логическое выражение2>
//     <оператор2>
// if (<логическое выражение3>
//     <оператор3>
```

---

Если в первом случае будет вычислено одно или два логических выражения, то во втором варианте посчитываются все три логических выражения.

- Стремитесь к коротким, лаконичным решениям без лишних конструкций. Уже закончились те времена, когда эффективность программиста измерялась количеством строк кода.
- Переменные обязаны быть объявлены максимально близко к месту первого их использования.
- Используйте корректирующее присваивание везде, где это необходимо:

---

```
x += a
```

---

вместо

---

```
x = x + a
```

---

- Не стоит отделять объявление и инициализацию переменных:

---

```
int a = 10; // правильно

int b;      // неправильно
b = 10
```

---

- Не объявляйте с инициализацией две и более переменных:

---

```
int a = 10;           // правильно
int b = 20;

int a = 10, b = 20; // неправильно
```

---

- Имеются несколько способов округлить результат деления вверх (рекомендую остановиться на втором или третьем варианте):

1. При помощи функции *ceil*, которая находится в стандартной библиотеке *math.h*<sup>7</sup>:

---

```
int x = ceil((double)a / b);
```

---

2. При помощи условного оператора *if – else*, тернарного оператора:

---

```
// через if-else
int x;
if (a % b)
    x = a / b + 1;
else
    x = a / b;

// через тернарный оператор ?: 
x = a % b ? a / b + 1 : a / b;
```

---

3. Если знать тот факт, что результатом вычисления логического выражения является значением 'истина' (1) или 'ложь' (0) можно решить задачу так:

---

```
x = a / b + (a % b != 0)
```

---

4. Пользуясь свойством<sup>8</sup>

$$\left\lceil \frac{a}{b} \right\rceil \equiv (a + b - 1) \operatorname{div} b$$

- После импортирования библиотек оставляйте пустую строку. Дополнительно рекомендуется вставлять пустые строки, если это улучшает читаемость кода, например, разделить ввод, обработку, и вывод:

<sup>7</sup> Вариант может создавать погрешности, так как не работает с большими числами в силу приведения типа из *long long* в *double*. Не рекомендуется к использованию.

<sup>8</sup>  $\lceil a \rceil$  – обозначение округления в большую сторону для выражения *a*

---

```
#include <stdio.h>

int main() {
    int n_sets;
    scanf("%d", &n_sets);
    // пустая строка, так как закончен ввод
    for (int set_number = 1; set_number <= n_sets; set_number++) {
        int h, m;
        scanf("%d %d", &h, &m);
        // пустая строка, так как закончен ввод
        int n_min = (24 - h)*60 - m;
        // пустая строка, так как закончилась обработка
        printf("%d\n", n_min);
    }
    // пустая строка перед return 0
    return 0;
}
```

---

- По возможности не осуществляйте вывод в ветках. Делайте так:

---

```
int max;
if (a > b)
    max = a;
else
    max = b;

printf("%d", max);
```

---

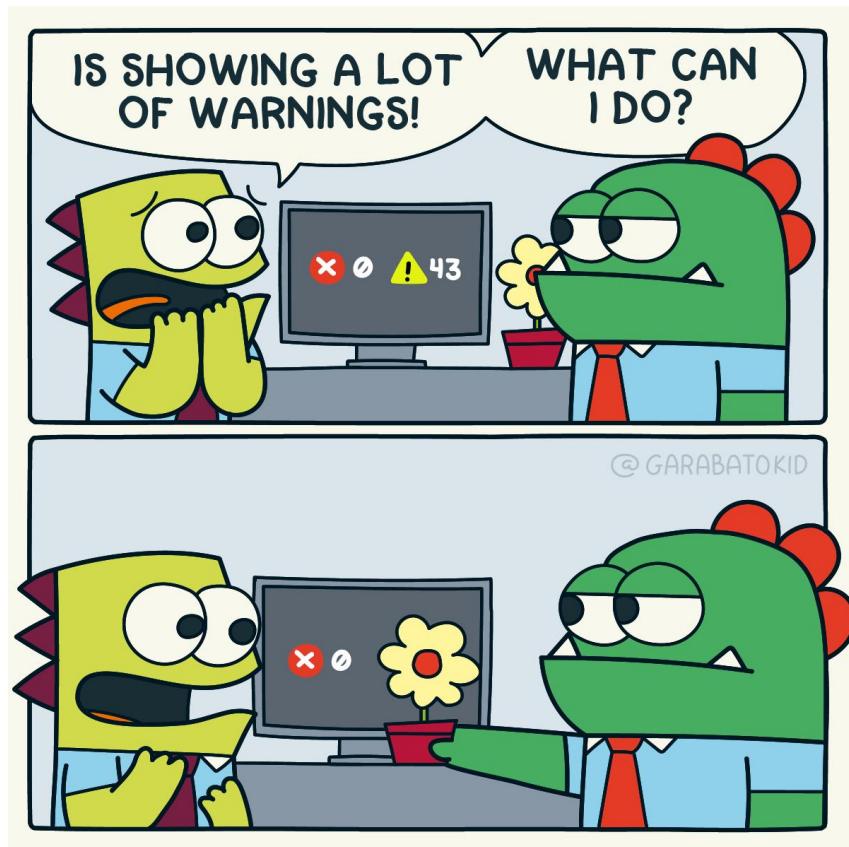
вместо

---

```
if (a > b)
    printf("%d", a);
else
    printf("%d", b);
```

---

- Игнорирование предупреждений компилятора.



- Плохое форматирование отчёта. Вы должны помнить: отчёт - лицо вашей работы. К качеству его оформления нужно отнестись так же ответственно, как и к решению задач. Перечислю ряд рекомендаций:
  - Формат отчёта – *pdf*.
  - Рекомендации к оформлению блок-схем:
    1. Для создания блок-схем рекомендуются следующие инструменты: *Microsoft Visio* или *draw.io*
    2. Стиль оформления блоков должен совпадать со стилем оформления блоков в пособии.
    3. Операция присваивания на блок-схеме: `:=`.
    4. Целочисленное деление на блок-схеме – `div`, вещественное – `/`.
    5. Блок-схема не должна содержать операции, присущие языкам программирования. Например: `x += a`. Последнее правильнее записать как `x := x + a`.
    6. Для выхода из блока 'решение' отсутствует обозначение '+'. Рекомендуется размещать данную ветку слева.
    7. Развилка обязана иметь два плеча, даже если по '-' ничего не происходит:

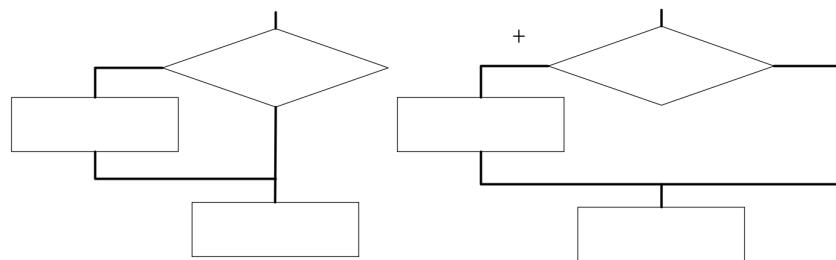
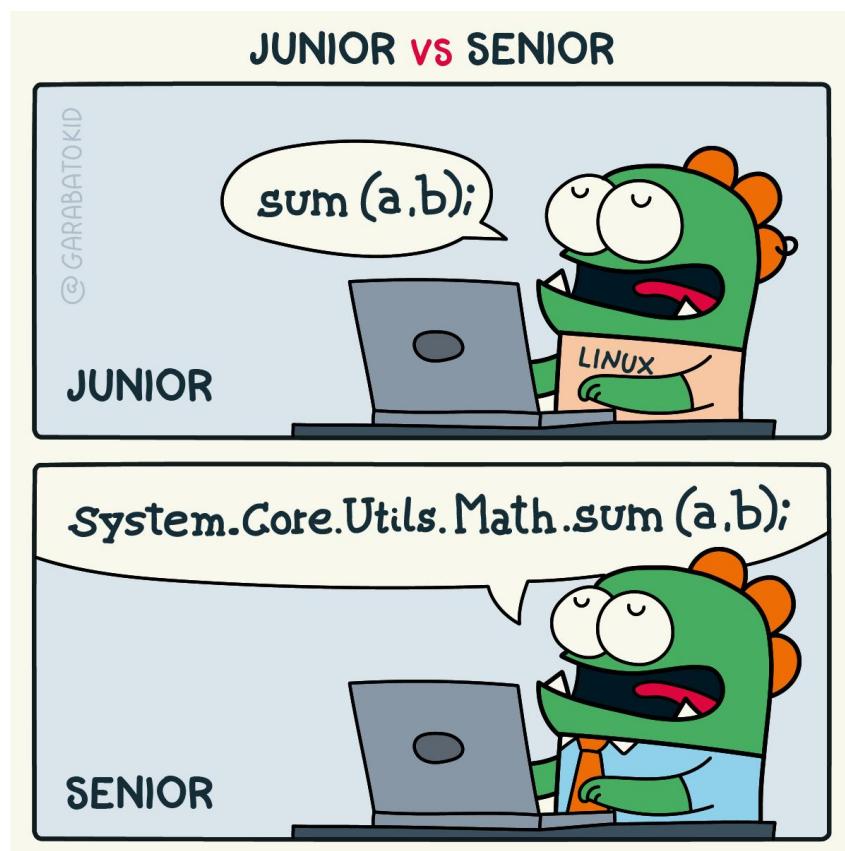


Рис. 18.1: Вариант оформления справа является правильным

8. Кегль текста в блоках блок-схем должен визуально совпадать с кеглем основного текста.

Код должен быть максимально чистым, идеи прозрачными. Не усложняйте пожалуйста:



Пример оформления отчёта представлен на следующей странице.

## Пример оформления отчета

### Лабораторная работа «Алгоритмы разветвляющейся структуры»

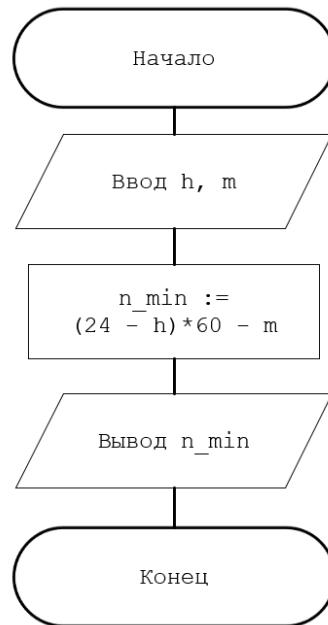
**Цель работы:** получение навыков написания линейных и разветвляющихся алгоритмов.

#### Содержание отчета:

- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
  - Название задачи.
  - Для задач с звездочкой приложить блок-схему.
  - Исходный код.
  - Скриншот с *codeforces* с указанием вердикта тестирующей системы.
- Вывод.

#### Задача №1. Минуты до Нового года (1283A).

Блок-схема алгоритма:



Код программы:

```
#include <stdio.h>

int main() {
    int nSets;
    scanf("%d", &nSets);

    for (int setNumber = 1; setNumber <= nSets; setNumber++) {
        int h, m;
        scanf("%d %d", &h, &m);

        int nMin = (24 - h) * 60 - m;

        printf("%d\n", nMin);
    }

    return 0;
}
```

Вердикт тестирующей системы:

Мои посылки							
№	Когда	Кто	Задача	Язык	Вердикт	Время	Память
126443394	19.08.2021 15:26	ispritchin	A - Минуты до Нового года	GNU C11	Полное решение	0 мс	3600 КБ

**Вывод:** в ходе выполнения лабораторной работы получены навыки написания линейных и разветвляющихся алгоритмов.

Рано или поздно вам придётся столкнуться с критикой ваших работ. Будьте готовы:



## Дополнительные материалы

Некоторые полезные рекомендации к оформлению могут быть найдены в [статье](#) (слово 'статье' кликабильно).

### 18.3 Лабораторная работа «Алгоритмы разветвляющиеся структуры»

**Цель работы:** закрепление навыков написания линейных и разветвляющихся алгоритмов.

#### Содержание отчета:

- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
  - Название задачи.
  - Для задач с звездочкой приложить блок-схему.
  - Задачи с двумя звездочками допускается пропустить.
  - Исходный код.
  - Скриншот с *codeforces* с указанием вердикта тестирующей системы.
- Вывод.

#### Перечень задач:<sup>9</sup>

1. Спасти Люка (624A)<sup>10</sup>
2. Поликарп и монеты (1551A)
3. Номер этажа (1426A)
4. Два кролика (1304A)<sup>11</sup>
5. \*\*Странная таблица (1506A)
6. Разделение последовательности (1102A)<sup>12</sup>

<sup>9</sup>Рекомендации:

- Уделите ОГРОМНОЕ внимание именованию переменных. Транслитерация запрещена. Представьте, что ваш код попадёт в международную команду, и транслитерация не поможет.
- В оформлении блок-схем необходимо придерживаться **исключительно** стиля пособия.
- Если что-то можно решить при помощи функции, надо использовать функции.
- Использование циклов и массивов в ходе лабораторной работы запрещено.
- Если используется тернарный оператор и вычисляемые выражения являются большими – предпочтите *if-else*. Не злоупотребляйте тернарными операторами.
- Для вывода сообщений "YES", "NO" используйте *printf* без спецификатора %s.

<sup>10</sup>В нескольких решениях задач, вместо явного приведения к *double* использовалось неявное: 1.0 \* .... Не делайте так.

<sup>11</sup>Избегайте дублирования вычислений. Создавайте вспомогательные переменные. Если видите, что какой-то фрагмент вычислений повторяется - создайте переменную.

<sup>12</sup>Попробуйте решить без условного оператора *if*. Подсказка: выводите значение логического выражения в *printf*.

7. Торт - это ложь (1519B)
8. \*Кто напротив? (1560B)
9. На лифте или по лестнице? (1054A)<sup>13</sup>
10. Паша и палка (610A)
11. Не NP (805A)
12. ExAb И нОд (1325A)
13. Три кучки с конфетами (1196A)
14. Театральная площадь (1A)<sup>14</sup>
15. Найти Амира (804A)
16. \* Пицца, пицца, пицца!!! (979A)<sup>15</sup>
17. Минимальное число (1101A)
18. \* Оплата без сдачи (1256A)
19. Магазины пончиков (1373A)
20. Сумма нечетных чисел (1327A)
21. \* Медведь Василий и треугольник (336A)
22. Водяная лилия (1199B)
23. \*\* Старт олимпиады (1539A)
24. Даша и лестница (761A)
25. Середина контеста (1133A)<sup>16</sup>
26. Чунга-Чанга (1181A)
27. Отопление (1260A)<sup>17</sup>
28. Комментаторские кабинки (990A)
29. Пересдача (991A)
30. Высота функции (1036A)
31. Пара игрушек (1023B)
32. Посмотрим футбол (195A)
33. \*\*Настя играет в компьютер (1136B)<sup>18</sup>

<sup>13</sup>В данной задаче сравниваются два способа добраться с одного этажа на другой. Было бы хорошей практикой результат вычисления одного способа сохранить в переменной *stairsTime*, а второй способ - в *liftTime*. Если не добавить слово *Time* вы будете сравнивать лестницу с лифтом, что звучит довольно странно.

<sup>14</sup>При необходимости округлить значения, следует использовать функции.

<sup>15</sup>Хорошее именование переменных может привести к исключению дублирования вычислений.

<sup>16</sup>Для вывода результата используйте `printf("%02d:%02d", hours, minutes);`

<sup>17</sup>Особое внимание именованию переменных в данной задаче.

<sup>18</sup>Допускается использование не более одного *if* или *if-else*.

## Пояснения к лабораторной работе

- Оформление должно быть аналогичным прошлой лабораторной работе.
- Требования к решениям остаются неизменными.
- Допускается использование следующих функций:

---

```
// возвращает минимальное значение переменных a и b
long long min2(long long a, long long b) {
    return a < b ? a : b;
}

// возвращает максимальное значение переменных a и b
long long max2(long long a, long long b) {
    return a > b ? a : b;
}

// возвращает максимальное значение среди переменных a, b, c
long long max3(long long a, long long b, long long c) {
    long long max;
    if (a > b && a > c)
        max = a;
    else
        max = b > c ? b : c;
    return max;
}

// возвращает значение дроби a/b, округлённое вверх
// примеры: 5 / 3 -> 2, 8 / 2 -> 4, 7 / 4 -> 2;
// пример вызова функции: ceil\frac(5, 3);
long long ceilFrac(long long a, long long b) {
    return a % b ? a / b + 1 : a / b;
}

// возвращает модуль числа x
long long abs(long long x) {
    return x >= 0 ? x : -x;
}
```

---

- Допустимо использование функций из библиотеки `math.h`.
- При оформлении блок-схемы, в которой используются вызовы функции, необходимо использовать блок 'предопределенный процесс'. Предположим, что требуется найти максимум и минимум двух переменных, тогда оформление следует выполнить следующим образом:

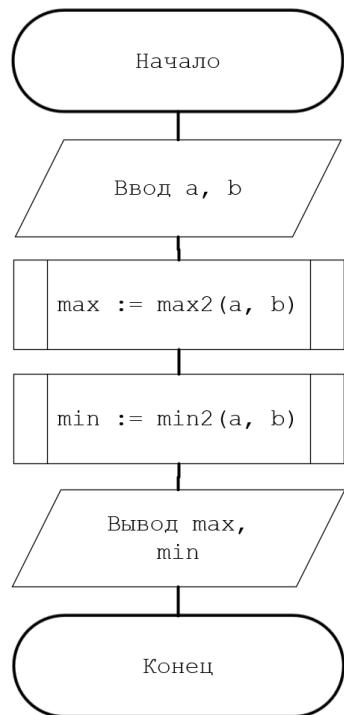


Рис. 18.2: Пример оформления блок-схемы с вызовами функций

## Дополнительные материалы

Рекомендуется прочесть главу 15 'Условные операторы' из книги Макконнелла 'Совершенный код'.

## 18.4 Лабораторная работа «Циклы. Введение в тестирование»

**Цель работы:** получение навыков написания циклических алгоритмов и проведения ручного тестирования.

### Содержание отчета:

- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
  - Условие задачи.
  - Для задач с звездочкой приложите блок-схему.
  - Задачи с двумя звездочками допускается пропустить.
  - Тестовые данные, на которых проверялось приложение.
  - Исходный код.
- Вывод по работе.

### Перечень задач<sup>19</sup>:

1. \* С клавиатуры вводятся  $n$  ( $n > 0$ ) чисел. Найти максимальное значение.
2. С клавиатуры вводится последовательность чисел. Признак конца ввода - 0. Найдите максимальное значение среди введенных. Если последовательность была пуста - выведите сообщение 'Последовательность пуста'. Для того чтобы использовать русский язык для вывода подключите *windows.h* в и функции main добавьте строку 5:

---

```
#include <stdio.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(CP_UTF8);

}
```

---

3. С клавиатуры вводятся  $n$  ( $n > 0$ ) чисел. Найти индекс первого минимального значения. Нумерация элементов - с нуля<sup>20</sup>.
4. С клавиатуры вводятся  $n$  ( $n > 0$ ) чисел. Найти индекс последнего максимального значения. Нумерация элементов - с нуля<sup>21</sup>.

<sup>19</sup>Проведите качественное тестирование приложений. Это указано в целях работы.

<sup>20</sup>Вариант для именования результирующей переменной *firstMinIndex*. В ходе проверки работ было замечено, что в процессе решения выделяются лишние переменные.

<sup>21</sup>Переменную для индекса можно назвать *lastMaxIndex*.

5. С клавиатуры вводятся  $n$  ( $n > 0$ ) чисел. Найти количество минимальных значений<sup>22</sup>.
6. \* С клавиатуры вводятся  $n$  ( $n > 0$ ) чисел. Найти разность между максимальным и минимальным значением.<sup>23</sup>
7. С клавиатуры вводится последовательность. Признак конца ввода - 0. Найти сумму четных чисел<sup>24</sup>.
8. Дано целое число  $n$  ( $n > 0$ ). Найти максимальную цифру в записи этого числа.
9. Вводится последовательность из натуральных чисел. Признак конца ввода 0. Вывести количество четных и нечетных чисел.
10. Дано целое число  $n$  ( $n > 0$ ). Найти произведение отличных от нуля цифр данного числа<sup>25</sup>.
11. Дано целое число  $n$  ( $n > 0$ ). Проверить, входит ли в запись числа  $n$  данная цифра  $digit$   $k$  раз<sup>26</sup>.
12. \* С клавиатуры вводятся целые числа. Признак конца ввода – ноль. Определить число, следующее за последним из введенных минимальных значений.
13. \* С клавиатуры вводятся целые числа. Признак конца ввода – ноль. Определить число, предшествующее первому из введенных максимальных значений.
14. С клавиатуры вводится символы. Признак конца ввода - символ перехода на новую строку '\n'<sup>27</sup>. Определить количество букв<sup>28</sup>.
15. С клавиатуры вводится символы. Признак конца ввода - символ перехода на новую строку '\n'. Определить количество согласных букв<sup>29</sup>.
16. С клавиатуры вводятся целые числа. Признак конца ввода – ноль. Определить, является ли вводимая последовательность упорядоченной по невозрастанию или по неубыванию или все элементы равны или последовательность не принадлежит ни к какой из групп<sup>30</sup>.

<sup>22</sup>Рекомендуемое имя для переменной-результата `minCount`. `Count` в конце намекает, что будет подсчитываться количество `min`.

<sup>23</sup>Проектируя решение, подумайте над вопросом: может ли вводимое значение быть больше максимума и меньше минимума одновременно. Подсказка: на каждой итерации цикла должно проверяться или одно или два условия в зависимости от входных данных.

При сравнении текущего значения с максимумом предпочтите вариант `x > max` вместо `max < x`. Первый случай более явно подчёркивает, что нашлось что-то, что больше максимума. А второй говорит, что максимум стал меньше `x`. Аналогично для минимума.

<sup>24</sup>Вы хотите больше хороших имен? Возьмите `evenSum`. Больше о хорошем стиле именования переменных на странице 440.

<sup>25</sup>Не используйте магические константы в духе 48 ('0') и 32 (' ', = 'а' - 'А'). **Магическими константами** называют плохую практику программирования, когда в исходном тексте встречается числовое значение и неочевиден его смысл.

<sup>26</sup>Рекомендация по тестированию: проверяйте крайние случаи, когда количество цифр в числе равно  $k$ , меньше на 1, больше на 1.

<sup>27</sup>Клавиша *Enter*.

<sup>28</sup>В задачах, в которых идёт оперирование символами, используйте функцию `getchar`. Буквы вводятся в одном регистре. Рекомендуется (не требуется) выделение функции под эту задачу.

<sup>29</sup>Предполагается, что буквы принадлежат одному регистру и одному языку. Мне кажется крайне удачной идеей использовать флагги `isLetter` и `isVowel` или `isConsonant`.

<sup>30</sup>Подсказка: в данной задаче очень легко допустить ошибку. Рекомендуется использовать три флага для поиска решения.

Таблица 18.1: Тестовые данные для задачи

Входные данные	Ожидаемый результат	Пояснение
0	'Последовательность пуста'	Последовательность может быть пустой.
1 0	'Последний элемент - минимальный'	Последний элемент минимальный.
1 2 0	2	Простая короткая последовательность, которая уже содержит какое-то значение после последнего минимального.
2 1 3 0	3	Перед минимальным элементом было ещё какое-то значение. Важно убедиться в том, что программа не выдаст 1.
3 1 3 1 2 0	2	Несколько минимальных. По условию задачи требовалось, чтобы было выведено число после последнего минимального.
5 5 0	'Последний элемент - минимальный'	Проверка последовательности из равных значений.

17. С клавиатуры вводятся символы. Признак конца ввода – точка. Определить сумму введенных цифр.
18. \*\* С клавиатуры вводятся символы (пробелы и цифры). Признак конца ввода – точка. Определить сумму введенных чисел.
19. \*\* С клавиатуры вводятся вещественные числа<sup>31</sup>. Признак конца ввода – ноль. Определить, является ли вводимая последовательность арифметической прогрессией.

## Пояснения к лабораторной работе

Основное отличие этой лабораторной от прошлых состоит в том, что вам самостоятельно придётся провести тестирование своих приложений. Вы должны сами определить набор тестовых данных. Постарайтесь разрабатывать наборы действуя по принципу 'от простого к сложному'. Рассмотрим на конкретной задаче.

*С клавиатуры вводятся целые числа. Признак конца ввода – ноль. Определить число, следующее за последним из введенных минимальных значений.* Какие же можно выделить случаи? Они представлены в таблице 18.1.

Некоторые задачи могут иметь несколько компонентов вводимых данных. Каждую компоненту описывайте с новой строки<sup>32</sup> (пример для задачи 11):

<sup>31</sup>Вещественные числа нельзя сравнивать на равенство. Правильный механизм сравнения находится в пояснениях к лабораторной работе.

<sup>32</sup>Если смысл компоненты неочевиден, укажите её имя из условия задачи.

Входные данные	Ожидаемый результат	Пояснение
$n = 14445$ $digit = 4$ $k = 3$	'YES'	Цифра встречается ровно требуемое количество раз.

Для задач похожих на №1 значение  $n$  можно не указывать (если будете указывать, ему следует выделить отдельную строку). Оно очевидно из количества элементов последовательности.

В процессе подбора тестовых данных проявите фантазию. Тесты должны отличаться. Чем лучше вы будете чувствовать возможные случаи, тем лучше сможете замечать потенциальные проблемы в своих решениях и решениях коллег.

Вы извлечете максимум пользы из этой лабораторной если выполнение будете производить в следующем порядке:

1. Подберёте и опишете тестовые данные.
2. Выполните построение блок-схемы.
3. Выполните прогон тестовых данных по блок-схеме.
4. Произведёте набор кода.
5. Произведёте тестирование приложения.

Для сравнения двух чисел на равенство используйте функцию  $fcompare^{33}$ :

```
#include <stdio.h>
#include <math.h>

#define EPS 1e-12

int fcompare(double a, double b) {
    return fabs(a - b) < EPS;
}

int main() {
    double a, b;
    scanf("%lf %lf", &a, &b);

    // не равно - !fcompare(a, b)
    if (!fcompare(a, b))
        printf("Equal");
    else
        printf("Not equal");

    return 0;
}
```

В задачах, где используется символьный ввод до какого-то признака конца ввода (например, `\n`), можно использовать следующий прием:

<sup>33</sup>Специально использован префикс  $f$  намекающий на сравнение вещественных чисел.

---

```
char symbol;
while ((symbol = getchar()) != '\n') {
    // ...
}
```

---

вместо

---

```
char symbol = getchar();
while (symbol != '\n') {
    // ...
    symbol = getchar();
}
```

---

### Рекомендации по именованию:

- Общие:
  - Придерживайтесь единообразия.
  - Не используйте транслитерацию.
  - Не допускайте орфографических ошибок (будет сложно найти имя, которое написано неправильно).
  - За чтением кода программисты проводят гораздо больше времени, чем за его написанием. Выбирайте имена так, чтобы они облегчали чтение кода, пусть даже за счет удобства его написания.
- Переменных:
  - Имя переменной должно являться существительным.
  - Имя переменной должно быть написано в `camelCase`<sup>34</sup>, даже если первое слово является именем собственным. Примеры: `lastMinIndex`, `moscowPosition`.
  - Избегайте общих имён `result`, `index`, `variable`, `data`, `temp`, и т. п. если в этом нет прямой необходимости.
  - Не называйте переменные `1` и `0`, так как они могут быть спутаны с единицей (`1`) и нулюм (`0`) соответственно.
  - Многие программы включают переменные, содержащие вычисляемые значения: суммы, средние величины, максимумы и т. д. Дополняя такое имя спецификатором вроде `Total`, `Sum`, `Average`, `Max`, `Min`, укажите его в конце имени.
  - Спецификатор `Num/Number` не рекомендуется к использованию, потому что `numberOfCustomers` - количество клиентов, а `customerNumber` - переменная-индекс для вычисления конкретного покупателя в массиве покупателей. Чтобы не создавать путаницу, лучше использовать варианты `customerCount` или `customerTotal` и `customerIndex`.

<sup>34</sup>На самом деле, для языка С должен применяться `snake_case`, но в силу того, что дальше ожидается переход на C++, принято решение использовать `camelCase`-стиль сразу.

- Используйте следующие пары антонимов, если это уместно: `begin/end`, `first/last`, `locked/unlocked`, `min/max`, `next/previous`, `old/new`, `opened/closed`, `visible/invisible`, `source/target`, `source/destination`, `up/down`.
- Для именования переменных цикла используются `i`, `j`, `k`. Не используйте данные переменные для других целей. Если счётчик цикла используется вне цикла, используйте более выразительное имя. Если цикл длиннее нескольких строк, смысл переменной `i` легко забыть, поэтому в подобной ситуации лучше присвоить индексу цикла более выразительное имя.  
На самом деле, многие опытные программисты не используют имена `i`, `j`, `k`, а предпочитают что-то более выразительное:

---

```
int setsTotal;
scanf("%d", &setsTotal);

for (int setIndex = 1; setIndex <= setsTotal; setIndex++) {

}
```

---

Особенно стоит придерживаться данного правила для вложенных циклов.

- Переменные статуса не должны содержать в себе слово `flag`. Имя флага не должно включать фрагмент `flag`, потому что он ничего не говорит о его сути. Рекомендуется использовать префиксы `is` для данных переменных: `isError`, `isFound`, `isProcessingComplete`, что превращается в вопрос. Переменные-флаги не должны в своём имени содержать отрицание `not`.

- Функций:

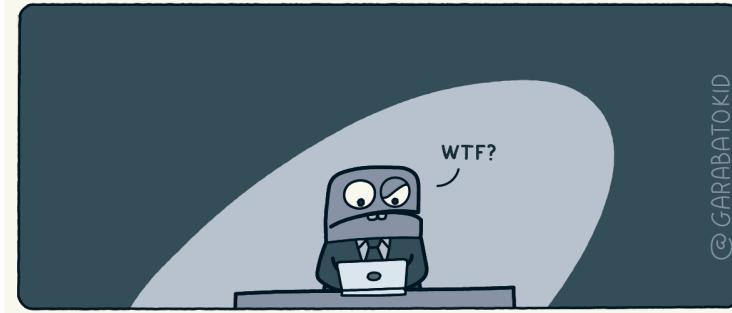
- Имя функции не должно являться существительным. Что делает функция `water()`?
- Разумные имена функций не должны содержать слов `be`, `do` и `perform` (быть, делать, выполнять). Естественно, они что-то делают и выполняют.

- Макросов:

- Имена макросов, объявленных через `#define` должны быть написаны `UPPER_SNAKE_CASE`.



MEANWHILE, IN THE CLIENT'S OFFICE



@GARABATOKID

## Пример оформления отчета

### Лабораторная работа «Циклы. Введение в тестирование»

**Цель работы:** получение навыков написания циклических алгоритмов и проведения ручного тестирования.

#### Содержание отчета:

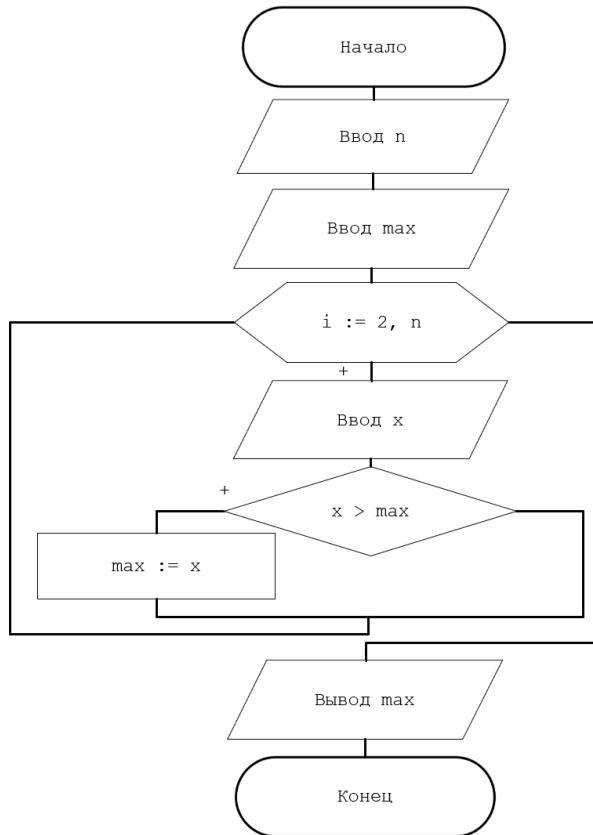
- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
  - Условие задачи.
  - Для задач с звездочкой приложить блок-схему.
  - Задачи с двумя звездочками допускается пропустить.
  - Тестовые данные, на которых проверялось приложение.
  - Исходный код.
- Вывод по работе.

**Задача №1.** \* С клавиатуры вводятся  $n$  ( $n > 0$ ) чисел. Найти максимальное значение.

Тестовые данные:

Входные данные	Ожидаемый результат	Пояснение
1	1	Последовательность из одного элемента, который сам по себе является максимумом.
1 2	2	Максимум обновляется в процессе его поиска
3 2 4 3 5 4	5	Максимум обновляется несколько раз

Блок-схема алгоритма:



Код программы:

---

```

#include <stdio.h>

int main () {
    int n;
    scanf("%d", &n);

    int max;
    scanf("%d", &max);
    for (int i = 2; i <= n; i++) {
        int x;
        scanf("%d", &x);

        if (x > max)
            max = x;
    }

    printf("%d", max);

    return 0;
}
  
```

---

**Вывод:** в ходе работы получены навыки написания циклических алгоритмов, получены навыки проведения ручного тестирования.

## 18.5 Лабораторная работа «Циклы»

**Цель работы:** закрепление навыков написания циклических алгоритмов.

**Содержание отчета:**

- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
  - Название задачи.
  - Для задач с звездочкой приложить блок-схему.
  - Исходный код.
  - Скриншот с *codeforces* с указанием вердикта тестирующей системы.
- Вывод по работе.

**Перечень задач<sup>35</sup>:**

1. Команда (231A)
2. \* Неправильное вычитание (977A)
3. \* Трамвай (116A)
4. Ваня и забор (677A)
5. Юра и заселение (467A)
6. Выбор команд (432A)
7. \* I\_love\_%username%(155A)
8. Нечётное множество (1542A)
9. \* Полицейские-рекруты (427A)
10. Задача Бахгольда (749A)
11. Мишка и старший брат (791A)
12. Открытки для друзей (1472A)
13. Ваня и кубики (492A)
14. Сайт отзывов (1511A)
15. \* Системный администратор (245A)
16. Покупка еды (1296B)
17. Проблемные обеды (276A)

---

<sup>35</sup>Во всех задачах использование массивов запрещено.

18. Хитрая сумма (598A)
19. Арья и Бран (839A)
20. Денежная система Геральдиона (560A)
21. Медведь и малина (385A)
22. Возрастающая последовательность (11A)
23. Расписание Алёны (586A)
24. \*\* Нечетная сумма (797B)

Оформление работы аналогично лабораторным 2a и 2b.

## 18.6 Лабораторная работа «Введение в функции»

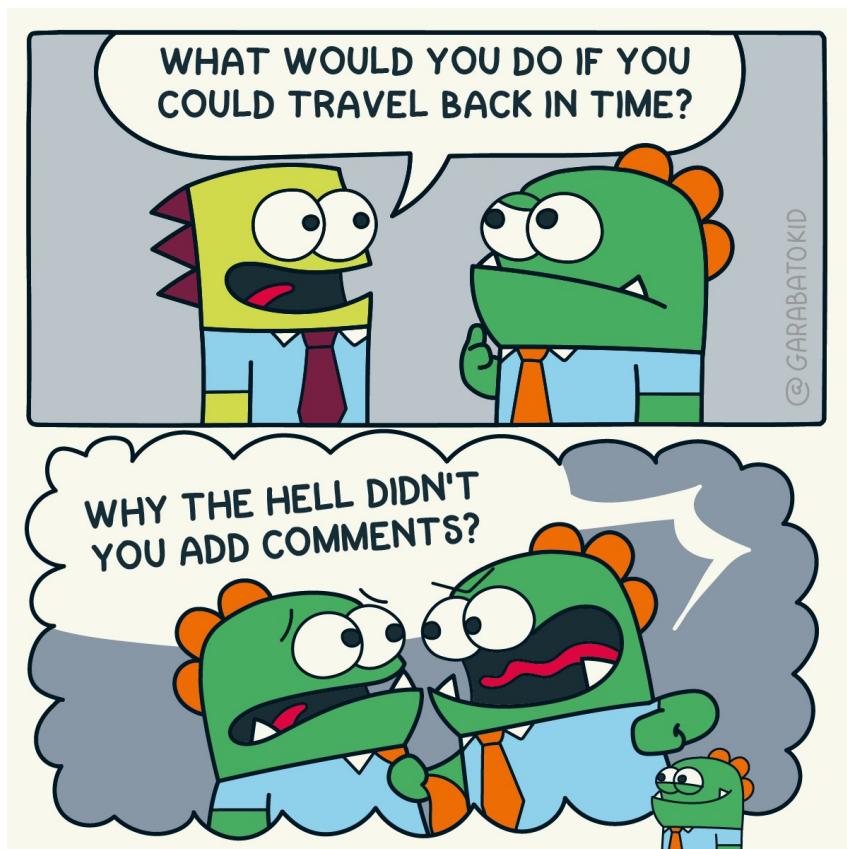
**Цель работы:** получение навыков написания функций при решении простых задач. Закрепление навыков разработки алгоритмов разветвляющейся и циклической структуры. Получение навыков формулирования спецификаций к разрабатываемым функциям.

### Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Решения задач. Для каждой задачи указать:
  - Условие задачи.
  - Тестовые данные<sup>36</sup>.
  - Исходный код функции и её спецификацию.
- Задачи с двумя звездочками допускается не решать.

### Требования к лабораторной работе:

- Требования к оформлению спецификаций:



<sup>36</sup> Дополняйте тестовые данные, в задачах на побитовые операции представлением числа в определенной системе счисления (по задаче).

- Фрагменты кода (заголовок и имена переменных в назначении) должны быть выделены моноширинным шрифтом (например, Courier New)<sup>37</sup>.
- **Каждый** из формальных параметров функции должен прозвучать в назначении.
- Если функция возвращает какое-то значение, и это считается основной задачей функции, её назначение должно начинаться со слова "возвращает".
- В спецификациях функций "истина" и "ложь" должны быть заключены в кавычки.
- Допускается не описывать спецификацию функции, которая была описана в отчете ранее.
- Используйте модификатор `const` везде, где это возможно.
- **Не используйте** структуры вида:

```
if (<логическое выражение>)
    return 1;
return 0;
```

```
if (<логическое выражение>)
    return 1;
else
    return 0;
```

вместо:

```
return <логическое выражение>
```

- При возврате выражения

```
return <выражение>
```

**не** ставьте скобки:

```
return (<выражение>)
```

- Функции не должны выводить никаких значений (если это не сказано в задании). Любые выводы внутри функции являются побочным эффектом.
- Функцией возведения в степень в задачах на побитовые операции пользоваться запрещено. Умножение на  $2^n$  и деление на  $2^n$  в задачах на побитовые операции замените на побитовые сдвиги.

### Условия задач:

1. Напишите функцию *abs* для вычисления модуля вещественного числа *x*.

---

<sup>37</sup>Отличный шрифт для параметров необходим для того, чтобы они несливались с остальным текстом назначения.

2. Напишите функцию *max2*, которая возвращает максимальное значение из двух целочисленных переменных типа *int*<sup>38</sup>.
3. Напишите функцию *max3*, которая возвращает максимальное значение из трёх целочисленных переменных типа *int*. Используйте при решении функцию *max2*.
4. Напишите функцию *getDistance*, которая вычисляет расстояние между двумя точками, заданными целочисленными координатами  $(x_1, y_1), (x_2, y_2)$ <sup>39</sup>
5. Напишите функцию *solveX2*, которая выводит корни квадратного уравнения:

$$ax^2 + bx + c = 0 \quad (a \neq 0)$$

Найденные корни должны быть выведены в теле функции. Если действительных корней нет - вывести соответствующее сообщение<sup>40</sup>.

6. Написать функцию *isDigit*, которая возвращает значение 'истина', если символ *x* является цифрой, 'ложь' - в противном случае.
7. Напишите функцию *swap*, которая принимает две переменные типа *float* и обменивает их значения<sup>41</sup>.
8. Напишите функцию *sort2*, которая упорядочивает значения *a* и *b* типа *float*. Т.е. если  $a > b$  то после выполнения функции значение переменной *a* должно быть меньше значения переменной *b* (при решении используйте функцию *swap* из прошлой задачи).
9. Напишите функцию *sort3*, которая упорядочивает значения переменных *a*, *b*, *c* типа *float* таким образом, чтобы:

$$a \leq b \leq c$$

(при решении используйте функцию *sort2* из прошлой задачи).

10. Написать функцию, которая возвращает значение 'истина', если можно составить треугольник с целочисленными сторонами *a*, *b*, *c* ( $a, b, c \in N$ ), 'ложь' - в противном случае (при решении вам потребуется *sort3* для целочисленных переменных)<sup>42</sup>.

---

<sup>38</sup>Тело функций для задач 1, 2, 3, 6 должны содержать одну строку и не использовать вспомогательные переменные.

<sup>39</sup>Ещё раз о принципе *DRY*: не дублируйте вычисления! Не считайте дважды  $x_2 - x_1$  и  $y_2 - y_1$ .

Для извлечения квадратного корня не рекомендуется писать `pow(x, 0.5)`. Предпочтите запись `sqrt(x)`.

Внутри функций не создавайте переменную *distance*. Из названия *getDistance* очевидно, что функция возвращает расстояние (*distance*).

<sup>40</sup>Обратите внимание, что в задании имеется слово 'выводит', следовательно, вывод внутри функции допустим.

Не дублируйте вычисление квадратного корня. Не извлекайте корень из отрицательных чисел.

<sup>41</sup>Грамотное назначение для данной функции: обменивает значения переменных по адресам *a* и *b*.

<sup>42</sup>Не передавайте адреса переменных в функции, если не планируете производить их изменение (переменных по тем адресам). После вызова функции длины переданных сторон не должны измениться.

Увы, но для каждого типа в языке С придётся писать свою функцию *sort3*. А по пути и оставшиеся.

11. Напишите функцию *getTriangleTypeLength*, которая возвращает значение 0, если треугольник со сторонами  $a, b, c$  является остроугольным, 1 – если прямоугольным, 2 – тупоугольным, -1 – если треугольник с такими сторонами не существует.
12. Напишите функцию *isPrime*, которая возвращает значение 'истина', если число является простым, иначе – 'ложь'. Приложите 3 вариации<sup>43</sup>:
- Без оптимизаций
  - С оптимизацией перебора до  $\sqrt{N}$ .
  - С оптимизацией перебора до  $\sqrt{N}$  и шагом 2.
13. Напишите функцию *deleteOctNumber*, которая удаляет цифру *digit* в записи данного восьмеричного числа  $x$ <sup>44</sup>:

Входные данные	Выходные данные
$3179_{10} = 110'001'101'011_2 = 6153_8$ <i>digit</i> = 1	$653_8 = 110'101'011_2 = 427_{10}$
$9_{10} = \underline{1}'001_2 = \underline{11}_8$ <i>digit</i> = 1	$0_{10}$
$37_{10} = 100'101_2 = 45_8$ <i>digit</i> = 1	$45_8 = 100'101_2 = 37_{10}$

14. Напишите функцию *swapPairBites*, которая меняет местами соседние цифры пар в двоичной записи данного натурального числа. Обмен начинается с младших разрядов. Непарная старшая цифра остается без изменения<sup>45</sup>.

Входные данные	Выходные данные
$77_{10} = 1001101_2$	$1001110_2 = 78_{10}$
$165_{10} = 10100101_2$	$1011010_2 = 90_{10}$

15. Напишите функцию *invertHex*, которая преобразует число  $x$ , переставляя в обратном порядке цифры в шестнадцатеричном представлении данного натурального числа.

<sup>43</sup>Извлечение квадратного корня в оптимизированных версиях должен осуществляться единожды. При проверке логических выражений в цикле не используйте сложные вычисления, если это возможно.

Назначения для всех трёх случаев одинаковы, так как функции делают одно и то же. Как правило, текст назначения не должен содержать информацию о действиях, которые производит функция (например, имеет ли она какие-то оптимизации и т.п.).

Написанные алгоритмы проверьте на значениях от 1 до 5. Как правило, ошибка обнаруживается на данных числах.

В каждом из случаев алгоритм должен закончить работу, если был найден хотя бы один нетриальный делитель.

<sup>44</sup>Исходное число вводится, и результат вычислений выводится в десятичной системе счисления. В таблице указаны представления чисел в нескольких системах счисления, чтобы лучше отразить преобразования, которые осуществляются по условиям задачи.

Использовать не более одного цикла. Чтобы получить последнюю цифру числа  $x$  в восьмеричном представлении достаточно выполнить `digit = x & 7`, а чтобы получить следующую – используйте побитовые сдвиги на 3 вправо.

<sup>45</sup>Решение должно быть выполнено без подсчёта количества битов.

Входные данные	Выходные данные
$77_{10} = 100'1101_2 = 4D_{16}$	$D4_{16} = 1101'0100_2 = 212_{10}$
$2732_{10} = 1010'1010'1100_2 = AAC_{16}$	$CAA_{16} = 1100'1010'1010_2 = 3242_{10}$

16. Напишите функцию *isBinPoly*, которая возвращает значение 'истина', если число  $x$  является палиндромом в двоичном представлении, иначе - 'ложь'<sup>46</sup>.

Входные данные	Выходные данные
$27_{10} = 11011_2$	<i>YES</i>
$454_{10} = 111000110_2$	<i>NO</i>

17. Даны два двухбайтовых целых  $sh_1$  и  $sh_2$ . Получить целое число, последовательность четных битов которого представляет собой значение  $sh_1$ , а последовательность нечетных – значение  $sh_2$ .

Входные данные	Выходные данные
$sh_1 = 0000000000001100_2 = 12_{10}$	$000000000000000000000000110100100_2 = 420_{10}$
$sh_2 = 000000000010010_2 = 18_{10}$	
$sh_1 = 011111100000000_2 = 32512_{10}$	$00101010101010000000000000000000_2 = 715784192_{10}$
$sh_2 = 000000000000000_2 = 0_{10}$	

18. Вывести восьмеричное представление записи числа  $x$ <sup>47</sup>.
19. Определить максимальную длину последовательности подряд идущих битов, равных единице в двоичном представлении данного целого числа.

Входные данные	Выходные данные
$x = 61454_{10} = 1111000000001110_2$	4
$x = 11_{10} = 1011_2$	2

20. \*\* Выполнить циклический сдвиг в двоичном представлении данного натурального числа  $x$  на  $k$  битов влево. Обратите внимание на механизм сдвига.

Входные данные	Выходные данные
$x = 27_{10} = 11011_2$	$10111_2 = 23_{10}$
$k = 1$	
$x = 27_{10} = 11011_2$	$1111_2 = 15_{10}$
$k = 2$	
$x = 42_{10} = 101010_2$	$10101_2 = 21_{10}$
$k = 1$	

<sup>46</sup>Решение должно быть выполнено без подсчёта количества битов.

<sup>47</sup>Вариантом `printf("%o", x)` пользоваться запрещено. Решение должно достигаться за один цикл. Будет ошибкой получить десятичное представление числа и выводить его.

21. \*\* Дано длинное целое неотрицательное число. Получить число, удалив каждую вторую цифру в двоичной записи данного числа, начиная со старших цифр.

Входные данные	Выходные данные
$x = 1_{10} = 1_2$	$1_2 = 1_{10}$
$x = 2_{10} = 10_2$	$1_2 = 1_{10}$
$x = 3_{10} = 11_2$	$1_2 = 1_{10}$
$x = 4_{10} = 100_2$	$10_2 = 2_{10}$
$x = 10_{10} = 1010_2$	$11_2 = 3_{10}$
$x = 40_{10} = 101000_2$	$110_2 = 6_{10}$

22. \*\* Дано целое неотрицательное число. Получить число перестановкой битов каждого байта данного числа в обратном порядке.

Входные данные	Выходные данные
$x = 61455_{10} = 1111000000001111_2$	$11111110000_2 = 4080_{10}$
$x = 43605_{10} = 1010101001010101_2$	$0101010110101010_2 = 21930_{10}$

Перечень задач<sup>48</sup>:

1. \*\*Новогодняя гирлянда (1279A)
2. \*\*Квадрат? (1351B)
3. \*\*Поход за едой (876A)
4. \*\*Пакеты с монетами (1037A)

---

<sup>48</sup>Просто приятный бонус. Если будут выделены функции, не нужно писать к ним спецификации. Оформление - аналогично задачам с *codeforces*.

**Пример оформления задачи:**

**Задача №1:** Опишите функцию для вычисления среднего арифметического модулей двух целочисленных переменных.

Пример тестовых данных:

Входные данные	Выходные данные
1 1 -3	2.0
2 1 2	1.5
3 -1 -3	2.0

Спецификация функции *absAverage*:

1. Заголовок: `double absAverage(int a, int b).`
2. Назначение: возвращает среднее суммы модулей значений *a* и *b*.

---

```
// возвращает среднее суммы модулей значений a и b.
double absAverage(int a, int b) {
    if (a < 0)
        a = -a;
    if (b < 0)
        b = -b;
    return (a + b) / 2.0;
}
```

---

## 18.7 Лабораторная работа «Обработка одномерных массивов с использованием функций»

**Цель работы:** получение навыков написания функций при решении задач на одномерные массивы.

### Содержание отчета:

- Тема лабораторной работы.
- Номер варианта.
- Цель лабораторной работы.
- Решения задач:
  - Для первого блока необходимо решить все задачи, кроме задач с двумя звездочками. Для задач с двумя звездочками достаточно приложить код без спецификации.
  - Для второго блока необходимо решить все задачи, кроме задач с двумя звездочками. Использовать вариант оформления №1 для задач с номерами  $a$  и  $b$ , где:
 
$$a = n_{\text{номер варианта по журналу}} \bmod 9 + 1$$

$$b = (n_{\text{номер варианта по журналу}} + 5) \bmod 9 + 1$$
  - Необязательные для решения задачи (с двумя звёздочками) допускается оформлять произвольно.

### Требования к лабораторной работе:

- Каждая задача должна быть протестирована на всех наборах входных данных.<sup>49</sup> После внесения любых исправлений должно выполняться повторное тестирование приложения.
- Любой вывод, который осуществляет ваше приложение, должно осуществляться из функции `main`. Функция `printf` может находиться только в функции вывода массива (или других специализированных функциях вывода) или в функции `main`.
- Каждое решение должно содержать весь исходный код.
- Каждая функция в качестве комментария, должна содержать спецификацию. Помните: "Худший комментарий - тот, который врёт". Текст спецификации должен сходиться с тем, что делает функция.
- Спецификация должна содержать все аспекты требований к данным. Например, если написанная функция ожидает отсортированный массив, в её назначении должно быть указание на этот факт: "...в отсортированном массиве  $a$  размера  $n$ ". Спецификации в коде должны быть описаны аналогично требованиям лабораторной 4а.

---

<sup>49</sup>Если при проверке на каком-то из существующих в пособии тестовых данных будет найден баг, выполнение работы дополнится скриншотами результатов запуска всех тестов из перечисленных.

- Указание спецификаций отдельно от кода не требуется в обоих вариантах оформления.

**Задания к лабораторной работе:**

1. Реализовать следующие функции<sup>50</sup>:

- Ввод массива  $a$  размера  $n$ .
- Вывод массива  $a$  размера  $n$ .
- Поиск позиции элемента со значением  $x$  с начала массива.
- Поиск позиции первого отрицательного элемента.
- \*\*Поиск позиции элемента с начала массива (по функции-предикату).
- Поиск позиции последнего четного элемента.
- \*\*Поиск позиции с конца массива (по функции-предикату).
- Подсчёт количества отрицательных элементов.
- \*\* Подсчёт элементов массива, удовлетворяющих функции-предикату.
- Изменение порядка элементов массива на обратный.
- Проверка является ли последовательность палиндромом.
- Сортировка массива выбором.
- Алгоритм удаления из массива всех нечетных элементов.
- Вставка элемента в массив с сохранением относительного порядка других элементов.
- Добавление элемента в конец массива.
- Удаление элемента с сохранением относительного порядка других элементов.
- Удаление элемента без сохранения относительного порядка других элементов.
- \*\* Реализуйте циклический сдвиг массива влево на  $k$  позиций<sup>51</sup>.
- \*\* Реализуйте функцию *forEach*, которая применяет функцию  $f$  к элементам массива  $a$  размера *size*.
- \*\* Реализуйте функцию *any*, которая возвращает значение 'истина', если хотя бы один элемент массива  $a$  размера *size* удовлетворяют функции-предикату  $f$ , иначе – 'ложь'.
- \*\* Реализуйте функцию *all*, которая возвращает значение 'истина', если все элементы массива  $a$  размера *size* удовлетворяют функции-предикату  $f$ , иначе – 'ложь'.

---

<sup>50</sup> Для задач поиска, возвращать значение  $-1$ , если элемент не найден.

Если реализуется вариант с функцией-предикатом, решение простого варианта необязательно.

<sup>51</sup> Допускается использование дополнительной памяти.

- \*\* Реализуйте функцию *arraySplit*, которая разделяет элементы массива *a* размера *size* на элементы, удовлетворяющие функции-предикату *f*, сохраняя в массиве *b*, иначе – в массиве *c*.

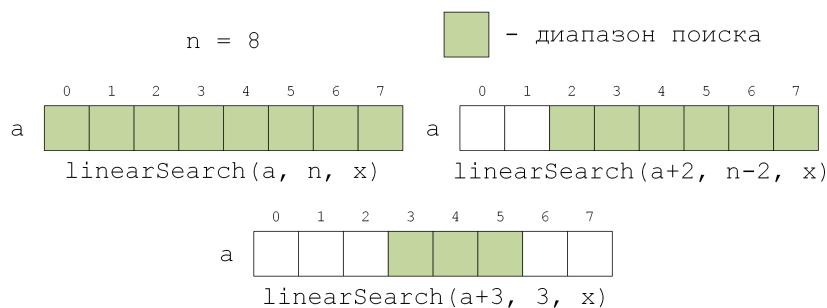
**Для задач без \*\* выполните самостоятельную проверку реализованных функций (сверившись с пособием).**

2. Перечень задач:

- Если возможно, то упорядочить данный массив размера *n* по убыванию, иначе массив **оставить без изменения**.<sup>52</sup> Примечание: вы можете (но не обязаны) в функции проверки на уникальность элементов использовать функцию линейного поиска. Предположим, у вас имеется массив *a* размера *n*. И необходимо проверить, встречается ли элемент *a[i]* в подмассиве *a[i+1..n-1]*. На этот вопрос можно ответить вызовом:

```
linearSearch(a + i, a + i + 1, n - i - 1)
```

Почему это работает? Применяя сложение / вычитание указателя с константой, мы производим смещение указателя к последующим / предыдущим элементам:



Входные данные	Выходные данные
1 2 4	4 2 1
4 2 4	4 2 4
1 3 1 4	1 3 1 4
4 2 3 1	4 3 2 1

- Дана целочисленная последовательность. Упорядочить по неубыванию часть последовательности<sup>53</sup>, заключенную между **первым вхождением максимального** значения и **последним вхождением минимального**.

Входные данные	Выходные данные
10 3 2 1 0	10 1 2 3 0
0 3 2 1 10	0 1 2 3 10
10 5 4 4 7 8 10 10	10 4 5 4 7 8 10 10

<sup>52</sup>Массив не может быть упорядочен по убыванию, если содержит одинаковые элементы. Вспомогательным массивом не пользоваться

<sup>53</sup>В решении задачи использовать любую сортировку, описанную ранее. Не писать специализированную версию сортировки. Используйте данный подход для решения последующих задач.

3. Если данная последовательность не упорядочена ни по неубыванию, ни по невозрастанию, найти среднее геометрическое<sup>54</sup> положительных членов.<sup>55</sup>

Входные данные	Выходные данные
4 1 2	2
9 1 3 3 0	3
2 -1 -1 0	2
-1 -2 -1	0
2 4 3	2.884499
-1 -1 -1	"Последовательность упорядочена"
1 2 4	"Последовательность упорядочена"
4 2 2	"Последовательность упорядочена"

Примечание: среднее геометрическое  $G$  от  $n$  чисел определяется как:

$$G(x_1, x_2, \dots, x_n) = \sqrt[n]{x_1 \cdot x_2 \cdot \dots \cdot x_n}$$

4. Если число  $x$  встречается в данной целочисленной последовательности, то упорядочить по неубыванию часть последовательности после первого вхождения  $x$ .

Входные данные	Выходные данные
16 8 4 2 1	16 8 4 1 2
$x = 4$	
16 8 4 2 1	16 1 2 4 8
$x = 16$	
16 8 4 2 1	16 8 4 2 1
$x = 1$	
16 8 4 2 1	16 8 4 2 1
$x = 3$	

5. Даны две последовательности. Получить упорядоченную по невозрастанию последовательность, состоящую из тех членов первой последовательности, которых нет во второй<sup>56</sup>.

<sup>54</sup>Средним геометрическим нескольких положительных вещественных чисел называется такое число, которым можно заменить каждое из этих чисел так, чтобы их произведение не изменилось.

<sup>55</sup>Функции проверки на упорядоченность не должны содержать лишние проверки. Если на каком-то этапе можно сказать, что массив неупорядочен - необходимо прервать работу функции.

<sup>56</sup>Используйте сортировку обоих массивов перед получением последовательности. Вы обязаны воспользоваться отсортированностью массивов и должны получить алгоритм сложностью  $O(N + M)$ , где  $N$  и  $M$  - размеры отсортированных массивов. Использование функций `deleteByPosSaveOrder` `linearSearch` запрещено.

Входные данные	Выходные данные
$a = \{1, 2, 4\}$	$c = \{2, 1\}$
$b = \{4\}$	
$a = \{1, 2, 2, 4\}$	$c = \{2, 2\}$
$b = \{1, 4\}$	
$a = \{1, 2, 2, 4\}$	$c = \{4, 2, 2, 1\}$
$b = \{3\}$	
$a = \{1, 3, 3, 4, 6, 8, 8, 9, 10, 12, 12, 13, 15, 15\}$	$c = \{1, 4, 6, 10, 12, 12, 13, 15, 15\}$
$b = \{0, 3, 5, 5, 8, 9, 9, 11, 14, 14\}$	

6. Данна целочисленная последовательность, содержащая как положительные, так и отрицательные числа. Упорядочить последовательность следующим образом: сначала идут отрицательные числа, упорядоченные по невозрастанию, потом положительные, упорядоченные по неубыванию<sup>57</sup>.

Входные данные	Выходные данные
3 2 1 1 -4 -5 -6	-4 -5 -6 1 1 2 3
-3 -2 -1 0 1 2 3 4	-1 -2 -3 0 1 2 3 4
1 2 3 4 5	1 2 3 4 5

7. Данна целочисленная последовательность (по определению содержащая как положительные, так и отрицательные элементы) и целое число  $x$ . Определить, есть ли  $x$  среди членов последовательности, и если нет, то найти члены последовательности, ближайшие к  $x$  снизу и сверху<sup>58</sup>.

Входные данные	Выходные данные
1 3 6 2 5 $x = 6$	" $x$ - элемент последовательности"
1 3 6 2 5 $x = 4$	3 5
1 3 6 2 5 $x = 0$	$-\infty, 1$
1 3 6 2 5 $x = 7$	6 $\infty$
1 1 5 5 5 $x = 3$	1 5

8. Данна целочисленная последовательность. Получить массив из уникальных элементов последовательности<sup>59</sup>.

<sup>57</sup> Для решения используйте произвольную функцию сортировки и функцию *reverse*. Использование двух сортировок будет считаться ошибкой.

<sup>58</sup> Не использовать функции сортировки. Выполнить поиск за один проход с использованием лишь одной функции.

<sup>59</sup> Перед получением массива из уникальных элементов выполните его сортировку. Не использовать функцию `deleteByPosSaveOrder`, и функции для подсчёта количества элементов в массиве `a` со значением `x`.

Входные данные	Выходные данные
1 2 4 1 2	4
1 2 3 4 5	1 2 3 4 5
1 1 1 1 1	"Последовательность пуста"

9. Определить, можно ли, переставив члены данной целочисленной последовательности длины  $n$  ( $n > 1$ ), получить геометрическую прогрессию с знаменателем  $q$  ( $|q| \neq 1$ ). Разрешимое допущение: знаменатель прогрессии – целое число.

Входные данные	Выходные данные
1 2 4	"Yes"
1 2 5	"No"
1 1	"No"
0 1	"No"
1 3 0	"No"
-1 -4 -16 2 8	"Yes"
1 2 -4 -8 -16	"No"
1 1 1 1 -1	"No"
0 0 0	"No"
1 2 4 4 4 4 8	"No"
1 -1 1	"No"

10. \*\*Найти сумму четных цифр элементов массива из положительных чисел<sup>60</sup>.

### Пример оформления задачи из первого блока

**Задача №1. Ввод массива  $a$  размера  $n$ .**

Код функции:

```
// ввод массива a размера n
void inputArray(int * const a, const size_t n) {
    for (size_t i = 0; i < n; i++)
        scanf("%d", &a[i]);
}
```

Спецификация функции должна находиться только в комментарии функций. Написание остальных спецификаций опционально.

### Пример оформления задачи из второго блока

Вариант оформления №1 может быть найден на странице 266. Он содержит:

1. Тестовые данные.

<sup>60</sup>Попробуйте использовать функцию `forEach`.

2. Выделение подзадач. Подзадачи требует выделения специализированных функций. Например, ввод числа не является подзадачей, а ввод массива – является. Подзадачи не должны содержать слово "если". Как они будут связаны будет отражено на блок-схеме в укрупнённых блоках. Первое слово подзадачи – существительное.
3. Алгоритм в укрупненных блоках. Блок-схема в укрупнённых блоках не может содержать блоки предопределенный процесс. Обратите внимание, что ввод и вывод массива обозначаются в блоке 'процесс'. Блок 'данные' используется исключительно для ввода и вывода элементарных типов данных.
4. Исходный код с комментариями к функциям (спецификациями).

Вариант оформления №2 аналогичен варианту №1 но содержит только:

1. Тестовые данные.
2. Исходный код с комментариями к функциям (спецификациями).

## 18.8 Лабораторная работа «Бинарный поиск»

**Цель работы:** получение навыков использования алгоритмов бинарного поиска для решения задач оптимизации.

### Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Решения задач:
  - Название задачи.
  - Исходный код.
  - Вердикт тестирующей системы.
  - Задачи со звёздочкой являются обязательными для получения наивысшей оценки по работе.
  - Задачи с двумя звёздочками не являются обязательными.

### Задания к лабораторной работе:

Для получения доступа к некоторым задачам потребуется регистрация на курсе 'ITMO Academy: пилотный курс' в вкладке EDU [по ссылке](#). Дополнительно можете изучить материалы, касаемые бинарного поиска там. После регистрации на курсе все задачи должны открываться корректно.

Для некоторых задач вам потребуется сортировка исходных данных. Так как эффективные алгоритмы не изучены, используйте сортировку *qsort* из стандартной библиотеки С. Функция

---

```
void qsort(void *ptr, size_t count, size_t size,
           int (*comp)(const void *, const void *));
```

---

имеет 4 параметра:

- `ptr` - указатель на начало массива, который требуется сортировать
- `count` - количество элементов в массиве
- `size` - размер одного элемента массива в байтах
- `comp` - указатель на функцию-компаратор, которая возвращает отрицательное целое значение, если первый аргумент меньше второго аргумента, положительное значение, если первый аргумент больше второго и ноль, если аргументы равны.

---

```

#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int compare_ints(const void* a, const void* b) {
    int arg1 = *(const int*)a;
    int arg2 = *(const int*)b;

    if (arg1 < arg2) return -1;
    if (arg1 > arg2) return 1;
    return 0;
}

int main() {
    int a[SIZE] = {5, 2, 4, 3, 1};

    qsort(a, SIZE, sizeof(int), compare_ints);

    for (size_t i = 0; i < SIZE; i++)
        printf("%d ", a[i]);

    return 0;
}

```

---

Перечень задач:

1. Двоичный поиск.
2. Ближайшее слева.
3. Ближайшее справа.
4. Быстрый поиск в массиве.
5. Веревочки.
6. Очень Легкая Задача.
7. Ночная работа.
8. Компьютерная игра.
9. Книги.
10. Евгений и плейлист.
11. Алена и узкий холодильник.
12. Модные числа.
13. Пара тем.
14. \* Чемпионат мира.
15. \* Максимальная медиана.

16. \*Разделение массива.

17. \*\* Гамбургеры.

Для чтения исходных данных воспользуйтесь следующим кодом:

---

```
#include <stdio.h>
#include <string.h>

#define MAX_RECIPE_LENGTH 100

void getRecipe(char *recipe, int *nBread, int *nSausage, int *nCheese) {
    // strlen - функция, определенная в string.h
    // для вычисления длины строки
    int nIngredients = strlen(recipe);
    *nBread = 0;
    *nSausage = 0;
    *nCheese = 0;
    for (int ingredientIndex = 0; ingredientIndex < nIngredients;
         ingredientIndex++) {
        switch (recipe[ingredientIndex]) {
            case 'B':
                (*nBread)++;
                break;
            case 'S':
                (*nSausage)++;
                break;
            case 'C':
                (*nCheese)++;
                break;
        }
    }
}

int main() {
    char recipe[MAX_RECIPE_LENGTH + 1]; // +1 - под ноль-символ

    int nBread, nSausage, nCheese;
    gets(recipe);
    getRecipe(recipe, &nBread, &nSausage, &nCheese);

    //...
    return 0;
}
```

---

18. \*\* Slay the Dragon.
19. \*\*Детский праздник.<sup>61</sup>
20. \*\*Удаление двух элементов<sup>62</sup>

---

<sup>61</sup>Очень хорошая задача особенно для случая, когда вы не разобрались с функциями. Когда решал самостоятельно, потратил около 1.5 часов времени, чтобы осознать, что не так с Неправильным ответом на 6 тесте. Вызов для вашей нервной системы.

<sup>62</sup>Эту задачу можно решить без бинарного поиска. Вспоминается высказывание: я предполагаю, что если единственный инструмент, который вы имеете – молоток, то заманчиво рассматривать всё как гвозди.)

## 18.9 Лабораторная работа «Рекурсивные функции»

**Цель работы:** получение навыков написания рекурсивных функций.

**Содержание отчета:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Решения задач. Для каждой задачи указать:
  - Условие задачи.
  - Исходный код рекурсивных функций без спецификаций.
- Задачи с одной звездочкой **обязаны** содержать функцию-обёртку.

**Требования к лабораторной работе:**

- При реализации функций не использовать циклы.
- Если рекурсивная функция требует обёртки, требуется её указывать.
- Так как рекурсия представляет собой на верхнем уровне команду ветвления, убедитесь, что возврат значения имеется по всем веткам выполнения функции.

Для решения задач, связанных с обработкой символов вы можете использовать стандартную библиотеку `<ctype.h>` для проверки на то, является ли символ цифрой, буквой и т п:

---

```
#include <ctype.h>

//...
isdigit('8'); // 'истина'
isspace(' '); // 'истина'
isspace('\t'); // 'истина'
isspace('1'); // 'ложь'
isupper('A'); // 'истина'
```

---

**Условия задач:**

1. Определить количество цифр в тексте, вводимом с клавиатуры. Текст заканчивается символом перехода на новую строку `\n`.<sup>63</sup>
2. Вывести данное натуральное число в восьмеричной системе счисления.
3. Дан знаменатель и первый член геометрической прогрессии. Вычислить  $n$ -й член прогрессии.

---

<sup>63</sup>Текст вводится посимвольно. Строки для хранения текста не использовать. Функция не должна содержать тернарных операторов, содержать вспомогательные переменные, отличные от считанного символа и быть записана с использованием не более чем одного `if-else`.

4. Данна **упорядоченная по убыванию**<sup>64</sup> последовательность целых чисел. Определить, есть ли среди членов данной последовательности число  $x$ , и если есть, найти номер этого члена. Бинарным поиском не пользоваться.
5. Дан массив  $a$  размера  $n$  ( $n \geq 2$ ). Необходимо проверить, является ли он упорядоченным по неубыванию<sup>65</sup>.
6. \*Найти номер первого вхождения минимального значения в последовательность длины  $n$  (линейный поиск)<sup>66</sup>.
7. Даны натуральные числа  $a$  и  $b$ . Определить, могут ли эти числа быть **соседними** членами последовательности Фибоначчи. Последовательность Фибоначчи задается следующим образом<sup>67</sup>:

$$f_1 = f_2 = 1 \quad f_i = f_{i-1} + f_{i-2} \text{ для } i > 2$$

8. Вывести в обратном порядке символы данного текста, вводимого с клавиатуры, которые не являются цифрами. Текст заканчивается символом перехода на новую строку `\n`<sup>68</sup>.
9. Дан  $n$ -й член арифметической прогрессии, ее разность и значение  $n$ . Вычислить первый член прогрессии<sup>69</sup>.
10. \* С клавиатуры вводятся положительные вещественные числа  $a_1, a_2, \dots, a_n$ . Признак конца ввода – отрицательное число<sup>70</sup>. Вывести следующие значения:

$$\frac{a_n + a_{n-1}}{2}, \quad \frac{a_{n-1} + a_{n-2}}{2}, \quad \dots, \quad \frac{a_2 + a_1}{2}$$

11. Реализовать функцию `any`<sup>71</sup>, которая возвращает значение 'истина', если хотя бы один элемент удовлетворяет функции-предикату `f`, в противном случае – ложь.
12. Реализовать функцию `all`, которая возвращает значение 'истина', если все элементы удовлетворяют функции-предикату `f`, в противном случае – ложь.
13. Реализовать алгоритм бинарного поиска.
14. Реализовать сортировку выбором.

---

<sup>64</sup>Решение каким-то образом должно использовать упорядоченность элементов в массиве. Ответьте себе на вопрос: "Каким образом я использовал упорядоченность элементов массива?" Обратитесь к следующему примеру:  $a = \{10, 9, 8, 2, 0\}$ ,  $x = 3$ .

<sup>65</sup>Функцию-обёртку не использовать. Решение может быть записано с использованием не более чем одного `if-else`. Функция не должна содержать больше двух формальных параметров.

<sup>66</sup>Рекурсивная функция не должна использовать более четырёх параметров.

<sup>67</sup>Саму последовательность чисел Фибоначчи не хранить.

<sup>68</sup>Функция должна содержать ровно один рекурсивный вызов.

<sup>69</sup>Рекурсивная функция содержит не более четырёх строк и не создаёт внутри никаких вспомогательных переменных.

<sup>70</sup>Вводимую последовательность в массиве не хранить.

<sup>71</sup>И для задания 11, и для задания 12: функция должна содержать один `if-else`. Пользуйтесь ленивой схемой вычислений. Использовать ровно одну логическую операцию. Функция должна работать для массива размером 0. Для массива размера 0 функция `any` должна возвращать значение 0, а функция `all` – значение 1. Обе рекурсивные функции должны содержать три параметра и не иметь обёрток.

## 18.10 Лабораторная работа «Структуры. Функции для работы со структурами»

**Цель работы:** получение навыков написания функций для решения задач со структурами.

### Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Решения задач:
  - Текст задания.
  - Исходный код.
  - Решение задач со звёздочкой требуется для получения максимального балла.

### Требования к лабораторной работе:

- Каждая функция в качестве комментария, должна содержать спецификацию.
- Указание спецификаций отдельно от кода не требуется.

### Задания к лабораторной работе:

1. Опишем структуру point:

---

```
struct point {
    double x;
    double y;
};

typedef struct point point;
```

---

- (a) Объявите структуру point с инициализацией.
- (b) Реализуйте функцию ввода структуры point.  
Заголовок: `void inputPoint(point *p)`.
- (c) Реализуйте функцию вывода структуры point. Выводите данные в следующем формате (1.450; 1.850) с тремя знаками после запятой.  
Заголовок: `void outputPoint(point p)`.
- (d) Создайте две точки p1 и p2. Проведите их инициализацию в коде. Выполните присваивание точки p2 точке p1.
- (e) Создайте массив структур размера N=3. Реализуйте функции для его ввода `inputPoints` и вывода `outputPoints`<sup>72</sup>.  
Заголовок функции ввода: `void inputPoints(point *p, int n)`.  
Заголовок функции вывода: `void outputPoints(point *p, int n)`.

<sup>72</sup>Функция ввода должна содержать вызов `inputPoint`, функция вывода - `outputPoint`

- (f) Реализуйте функцию, которая принимает на вход две структуры типа `point` и возвращает точку, находящуюся посередине между точками `p1` и `p2`.

Заголовок: `point getMiddlePoint(point p1, point p2)`.

- (g) Реализуйте функцию `isEqualPoint`, которая возвращает значение 'истина', если точки совпадают (с погрешностью не более `DBL_EPSILON`, определённой в `<float.h>`)

Заголовок: `int isEqualPoint(point p1, point p2)`.

- (h) Реализуйте функцию, которая возвращает значение 'истина', если точка `p3` лежит ровно посередине между точками `p1` и `p2`.<sup>73</sup>

Заголовок: `int isPointBetween(point p1, point p2, point p3)`.

- (i) Реализуйте функцию `swapCoordinates` которая меняет значения координат `x` и `y` структуры типа `point`.

Заголовок: `void swapCoordinates(point *p)`.

- (j) Реализуйте функцию `swapPoints` которая обменивает две точки.

Заголовок: `void swapPoints(point *p1, point *p2)`.

- (k) Напишите фрагмент кода, в котором выделяется память под массив структур размера `N = 3`, после чего укажите инструкцию освобождения памяти.

- (l) Реализуйте функцию, которая находит расстояние между двумя точками.

Заголовок: `double getDistance(point p1, point p2)`.

- (m) Опишите функцию-компоратор для `qsort`, которая сортирует массив точек размера `N = 3` по увеличению координаты `x`, а при их равенстве – по координате `y`.

- (n) Опишите функцию-компоратор для `qsort`, которая сортирует массив точек размера `N = 3` по увеличению расстояния до начала координат.

2. Опишем структуру `line`, которая задаёт линию на плоскости уравнением

$$ax + by + c = 0$$

---

```
struct line {
    double a;
    double b;
    double c;
};
```

---

- (a) Реализуйте функцию `inputLine` ввода структуры `line`.

Заголовок: `void inputLine(line *line)`.

- (b) Инициализируйте структуру типа `line` при объявлении.

- (c) Реализуйте функцию `getLine` которая возвращает прямую по координатам точек.

Заголовок: `line getLineByPoints(point p1, point p2)`.

<sup>73</sup>Используйте функции `isEqualPoint` и `getMiddlePoint`.

Рекомендуемая реализация:

---

```
void outputLineEquation(line line) {
    printf("%+.2lf x%+.2lf y%+.2lf = 0", line.a, line.b, line.c);
}
```

---

- (d) Напишите код для создания линии из точек, без явного создания структур `p1` и `p2`.
- (e) Реализуйте функцию `outputLineEquation` вывода уравнения прямой `line`.  
Заголовок: `void outputLineEquation(line line);`.
- (f) Реализуйте функцию `isParallel`, которая возвращает значение 'истина', если прямые `line1` и `line2` параллельны, 'ложь' – в противном случае.  
Заголовок: `int isParallel(line l1, line l2);`.
- (g) Реализуйте функцию `isPerpendicular`, которая возвращает значение 'истина' если прямые `line1` и `line2` перпендикулярны, 'ложь' – в противном случае.  
Заголовок: `int isPerpendicular(line l1, line l2);`.
- (h) Определите, есть ли среди данных `n` прямых на плоскости (`n - const`) параллельные.  
Заголовок: `int hasParallelLines(line *lines, size_t n);`.
- (i) Реализуйте функцию `printIntersectionPoint`, которая выводит точку пересечения прямых `line1` и `line2`. Если точек пересечения нет – проинформируйте пользователя.  
Заголовок: `void printIntersectionPoint(line l1, line l2);`.

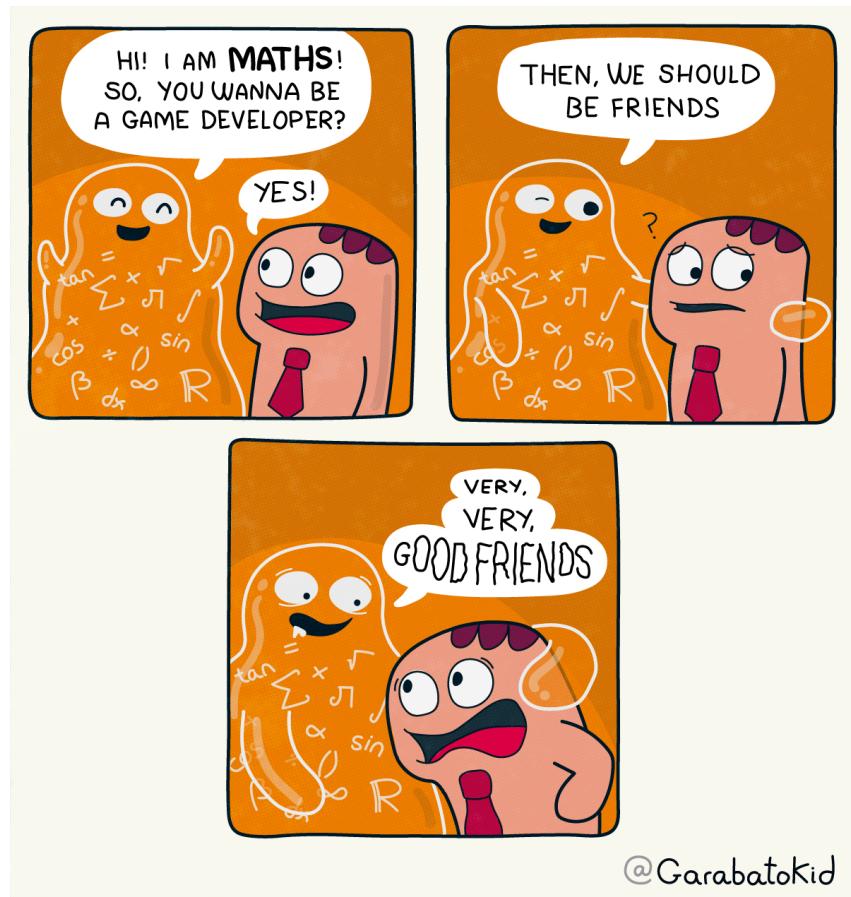
3. Опишите структуру `circle`, которая задаёт окружность посредством центра окружности `center(x0, y0)`, и радиуса `r`.

---

```
struct circle {
    point center; // центр окружности
    double r;      // радиус
};
```

---

- (a) Объявите с инициализацией структуру типа `circle`.
- (b) Объявите с инициализацией массив из двух структур типа `circle`.
- (c) Реализуйте функцию `inputCircle` ввода структуры `circle`.  
Заголовок: `void inputCircle(circle *a);`.
- (d) Реализуйте функцию `inputCircles` ввода массива структур `circle`.  
Заголовок: `void inputCircles(circle *a, size_t n);`.
- (e) Реализуйте функцию `outputCircle` вывода структуры `circle`.  
Заголовок: `void outputCircle(circle a);`.
- (f) Реализуйте функцию `outputCircles` вывода массива структур `circle`.  
Заголовок: `void outputCircles(circle *a, size_t n);`.



- (g) Реализуйте функцию `hasOneOuterIntersection`, которая возвращает значение 'истина', если окружность `c1` касается внешним образом окружности `c2`.

Заголовок: `int hasOneOuterIntersection(circle c1, circle c2)`.

- (h) Вводится массив из  $n$  окружностей ( $n$  вводится с клавиатуры). Реализуйте функцию, которая возвращает окружность, в которой лежит наибольшее количество окружностей. Если таких несколько – вернуть окружность с наименьшим радиусом.

- (i) \* Вводится массив из  $n$  окружностей ( $n$  вводится с клавиатуры). Реализуйте функцию сортировки окружностей, по неубыванию количества лежащих в ней окружностей. При равенстве количества последнего показателя, отсортировать по неубыванию радиуса.

4. Опишем структуру `fraction`:

---

```
struct fraction {
    int numerator; // числитель
    int denominator; // знаменатель
};
```

---

- (a) Реализуйте функцию `inputFraction` ввода структуры `fraction`. Пример ввода, который должен обрабатываться программой: '5/7', '2 / 17'.  
Заголовок: `void inputFraction(fraction *f)`.
- (b) Реализуйте функцию `inputFractions` ввода массива структур `fraction`.

Заголовок: `void inputFractions(fraction *f, size_t n).`

- (c) Реализуйте функцию `outputFraction` вывода структуры `fraction` в формате ' $5/7$ '.

Заголовок: `void outputFraction(fraction f).`

- (d) Реализуйте функцию `outputFractions` вывода массива структур `fraction`.

Заголовок: `void outputFractions(fraction *f, size_t n).`

- (e) Реализуйте функцию `gcd` возвращающую наибольший общий делитель.

Заголовок: `int gcd(int a, int b).`

- (f) Реализуйте функцию `lcm` возвращающую наименьшее общее кратное<sup>74</sup>.

Заголовок: `int lcm(int a, int b).`

- (g) Реализуйте функцию `simpleFraction` для сокращения дроби `a`.

Заголовок: `void simpleFraction(fraction *f).`

- (h) Реализуйте функцию `mulFractions` умножения двух дробей `a` и `b`.

Заголовок: `fraction mulFractions(fraction f1, fraction f2).`

- (i) Реализуйте функцию `divFractions` деления двух дробей `a` и `b`<sup>75</sup>.

Заголовок: `fraction divFractions(fraction f1, fraction f2).`

- (j) Реализуйте функцию `addFractions` сложения двух дробей `a` и `b`.

Заголовок: `fraction addFractions(fraction f1, fraction f2).`

- (k) Реализуйте функцию `subFractions` вычитания двух дробей `a` и `b`<sup>76</sup>.

Заголовок: `fraction subFractions(fraction f1, fraction f2).`

- (l) Реализуйте функцию для поиска суммы `n` дробей.

Заголовок: `fraction sumFractions(fraction *f, size_t n).`

5. \* Дан массив записей. Каждая запись содержит сведения о студенте группы: фамилию и оценки по 5 предметам. Удалить записи о студентах, имеющих более одной неудовлетворительной оценки<sup>77</sup>. Вывести фамилии оставшихся студентов. Указание: используйте структуру

---

```
#define N_MARKS 5

struct student {
    char surname[20];
    int marks[N_MARKS];
};
```

---

6. \* Дан массив, каждый элемент которого представляет собой временную отметку в рамках одного дня (запись из трех полей: часы, минуты и секунды). Упорядочить отметки в хронологическом порядке. Сравнение времени `t1` с `t2` оформить подпрограммой.

7. \* Определить время, прошедшее от `t1` до `t2`. Время предоставлено записью из трех полей: часы, минуты, секунды.

<sup>74</sup>Функция должна использовать в себе вызов `gcd`.

<sup>75</sup>Функция должна использовать в себе вызов `mulFractions`.

<sup>76</sup>Функция должна использовать в себе вызов `addFractions`.

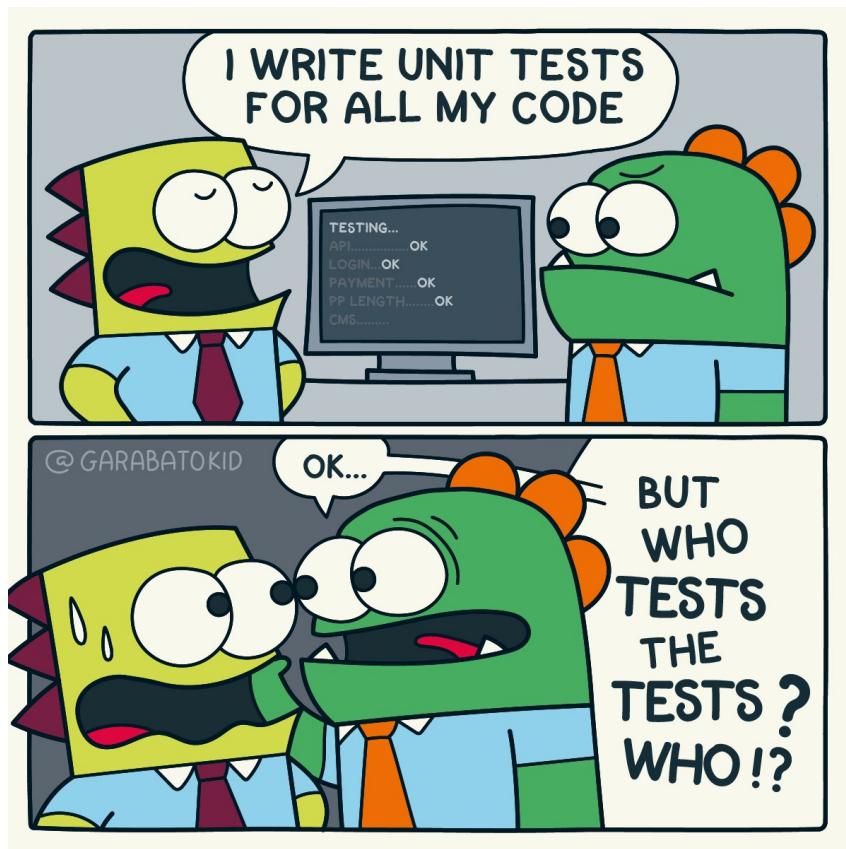
<sup>77</sup>Используйте алгоритм однопроходного удаления. Выделите функцию `isGoodStudent`, которая возвращает значение 'истина', если студент 'хороший', иначе – 'ложь'.

## 18.11 Лабораторная работа «Множества»

**Цель работы:** закрепление навыков работы со структурами, изучение простых способов представления множеств в памяти ЭВМ.

**Содержание отчета:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Решения задач:
  - Текст задания.
  - Исходный код библиотек (исходный и заголовочный файлы).
  - Исходный код тестов для любого (одного) произвольного типа представления множеств.



- Решение задач с *codeforces* требуется для получения максимального балла по лабораторной.
- Представление множеств на упорядоченных массивах требуется для получения максимального балла по лабораторной.

**Рекомендации по выполнению лабораторной работы:**

- Для реализации множеств вы должны **максимально** использовать описанные функции в `array.h`.
- Если какие-то функции можно выражать через другие – сделайте именно так.

- Если в процессе вычисления результата операций над множествами вы создаёте промежуточные множества, не забудьте освободить занимаемые ресурсы функцией `unordered_array_set_delete`.

### Задания к лабораторной работе:

1. Выполнить реализацию множества на типе `uint32_t`. Содержимое файла `bitset.h`:

---

```

1 #ifndef INC_BITSET_H
2 #define INC_BITSET_H
3
4 #include <stdint.h>
5 #include <stdbool.h>
6
7 typedef struct bitset {
8     uint32_t values;    // множество
9     uint32_t maxValue; // максимальный элемент универсума
10 } bitset;
11
12 // возвращает пустое множество с универсумом 0, 1, ..., maxValue
13 bitset bitset_create(unsigned maxValue);
14
15 // возвращает значение 'истина', если значение value имеется в множестве set
16 // иначе - 'ложь'
17 bool bitset_in(bitset set, unsigned int value);
18
19 // возвращает значение 'истина', если множества set1 и set2 равны
20 // иначе - 'ложь'
21 bool bitset_isEqual(bitset set1, bitset set2);
22
23 // возвращает значение 'истина' если множество subset
24 // является подмножеством множества set, иначе - 'ложь'.
25 bool bitset_isSubset(bitset subset, bitset set);
26
27 // добавляет элемент value в множество set
28 void bitset_insert(bitset *set, unsigned int value);
29
30 // удаляет элемент value из множества set
31 void bitset_deleteElement(bitset *set, unsigned int value);
32
33 // возвращает объединение множеств set1 и set2
34 bitset bitset_union(bitset set1, bitset set2);
35
36 // возвращает пересечение множеств set1 и set2
37 bitset bitset_intersection(bitset set1, bitset set2);
38
39 // возвращает разность множеств set1 и set2
40 bitset bitset_difference(bitset set1, bitset set2);
41
42 // возвращает симметрическую разность множеств set1 и set2
43 bitset bitset_symmetricDifference(bitset set1, bitset set2);
44
45 // возвращает дополнение до универсума множества set
46 bitset bitset_complement(bitset set);
47
48 // вывод множества set
49 void bitset_print(bitset set);
50
51 #endif

```

---

В качестве решения данного пункта приложите файлы *bitset.h* и *bitset.c*.

2. На неупорядоченном массиве:

---

```

1 #ifndef INC_UNORDERED_ARRAY_SET_H
2 #define INC_UNORDERED_ARRAY_SET_H
3
4 #include <stdint.h>
5 #include <assert.h>
6 #include <memory.h>
7 #include <stdio.h>
8 #include <stdbool.h>
9 #include "../algorithms/array/array.h"
10
11 typedef struct unordered_array_set {
12     int *data;           // элементы множества
13     size_t size;         // количество элементов в множестве
14     size_t capacity;    // максимальное количество элементов в множестве
15 } unordered_array_set;
16
17 // возвращает пустое множество для capacity элементов
18 unordered_array_set unordered_array_set_create(size_t capacity);
19
20 // возвращает множество, состоящее из элементов массива а размера size78.
21 unordered_array_set unordered_array_set_create_from_array(
22     const int *a, size_t size
23 );
24
25 // возвращает позицию элемента в множестве,
26 // если значение value имеется в множестве set, иначе - n
27 size_t unordered_array_set_in(unordered_array_set set, int value);
28
29 // возвращает позицию элемента в множестве,
30 // если значение value имеется в множестве set, иначе - n
31 size_t unordered_array_set_isSubset(unordered_array_set subset,
32                                     unordered_array_set set);
33
34
35 // возвращает значение 'истина', если элементы множеств set1 и set2 равны
36 // иначе - 'ложь'79
37 bool unordered_array_set_isEqual(
38     unordered_array_set set1, unordered_array_set set2
39 );
40
41 // возбуждает исключение, если в множество по адресу set
42 // нельзя вставить элемент
43 void unordered_array_set_isAbleAppend(unordered_array_set *set);
44
45 // добавляет элемент value в множество set
46 void unordered_array_set_insert(
47     unordered_array_set *set, int value
48 );
49
50 // удаляет элемент value из множества set
51 void unordered_array_set_deleteElement(
52     unordered_array_set *set, int value

```

---

<sup>78</sup> В реализации используйте функцию `unordered_array_set_insert`. Тело функции не должно содержать более 4 строк.

<sup>79</sup> Тело функции состоит из одной строки. Используйте другие (возможно ниже описанные) функции.

```

53 );
54
55 // возвращает объединение множеств set1 и set280.
56 unordered_array_set unordered_array_set_union(
57     unordered_array_set set1, unordered_array_set set2
58 );
59
60 // возвращает пересечение множеств set1 и set2
61 unordered_array_set unordered_array_set_intersection(
62     unordered_array_set set1, unordered_array_set set2
63 );
64
65 // возвращает разность множеств set1 и set281
66 unordered_array_set unordered_array_set_difference(
67     unordered_array_set set1, unordered_array_set set2
68 );
69
70
71 // возвращает дополнение до универсума множества set82
72 unordered_array_set unordered_array_set_complement(
73     unordered_array_set set, unordered_array_set universumSet
74 );
75
76 // возвращает симметрическую разность множеств set1 и set283
77 unordered_array_set unordered_array_set_symmetricDifference(
78     unordered_array_set set1, unordered_array_set set2
79 );
80
81 // вывод множества set
82 void unordered_array_set_print(unordered_array_set set);
83
84 // освобождает память, занимаемую множеством set
85 void unordered_array_set_delete(unordered_array_set set);
86
87 #endif

```

---

### 3. \* На упорядоченном массиве.

```

1 #ifndef INC_ORDERED_ARRAY_SET_H
2 #define INC_ORDERED_ARRAY_SET_H
3
4 #include <stdint.h>
5 #include <assert.h>
6 #include <memory.h>
7 #include <stdio.h>
8 #include <stdbool.h>
9 #include "../algorithms/array/array.h"
10
11 typedef struct ordered_array_set {
12     int *data;           // элементы множества

```

<sup>80</sup>Элементы первого множества уникальны, когда будете создавать множество-объединение, не используйте функцию `unordered_array_set_insert`, так как она имеет дополнительные проверки. А вот элементы второго множества можно добавлять данной функцией

<sup>81</sup>Запрещено использовать функцию `unordered_array_set_deleteElement`.

<sup>82</sup>Перед получением ответа проверьте, что множество `set` является подмножеством `universumSet`.

<sup>83</sup>Существует одна классическая ошибка: для вычисления симметрической разности НЕ создаются переменные для первой и второй разности множеств перед их объединением. Из-за этой причины ресурсы, выделенные промежуточным множествам, размещенных в динамической памяти, не освобождаются, что приводит к утечкам. Проверить правильно ли вы решили просто: убедитесь, что дважды вызывали функцию `unordered_array_set_delete`.

```

13     size_t size;           // количество элементов в множестве
14     size_t capacity;      // максимальное количество элементов в множестве
15 } ordered_array_set;
16
17 // возвращает пустое множество, в которое можно вставить capacity элементов
18 ordered_array_set ordered_array_set_create(size_t capacity);
19
20 // возвращает множество, состоящее из элементов массива a размера size
21 ordered_array_set ordered_array_set_create_from_array(const int *a,
22               size_t size);
23
24 // возвращает значение позицию элемента в множестве,
25 // если значение value имеется в множестве set,
26 // иначе - n
27 size_t ordered_array_set_in(ordered_array_set *set, int value);
28
29 // возвращает значение 'истина', если элементы множеств set1 и set2 равны
30 // иначе - 'ложь'84
31 bool ordered_array_set_isEqual(ordered_array_set set1,
32                               ordered_array_set set2);
33
34 // возвращает значение 'истина', если subset является подмножеством set
35 // иначе - 'ложь'85
36 bool ordered_array_set_isSubset(ordered_array_set subset,
37                                ordered_array_set set);
38
39 // возбуждает исключение, если в множество по адресу set
40 // нельзя вставить элемент
41 void ordered_array_set_isAbleAppend(ordered_array_set *set);
42
43 // добавляет элемент value в множество set
44 void ordered_array_set_insert(ordered_array_set *set, int value);
45
46 // удаляет элемент value из множества set
47 void ordered_array_set_deleteElement(ordered_array_set *set, int
48                                     value);
49
50 // возвращает объединение множеств set1 и set2
51 ordered_array_set ordered_array_set_union(ordered_array_set set1,
52                                         ordered_array_set set2);
53
54 // возвращает пересечение множеств set1 и set2
55 ordered_array_set ordered_array_set_intersection(ordered_array_set
56                                                 set1, ordered_array_set set2);
57
58 // возвращает разность множеств set1 и set2
59 ordered_array_set ordered_array_set_difference(ordered_array_set
60                                                 set1, ordered_array_set set2);
61
62 // возвращает симметрическую разность множеств set1 и set2
63 ordered_array_set ordered_array_set_symmetricDifference(
64                     ordered_array_set set1, ordered_array_set set2);
65
66 // возвращает дополнение до универсума universumSet множества set
67 ordered_array_set ordered_array_set_complement(ordered_array_set
68                                                 set, ordered_array_set universumSet);
69
70

```

<sup>84</sup>Решение должно использовать функцию `memcmp`.

<sup>85</sup>Особое внимание к корректности данной функции. Её реализации являются хорошим источником ошибок.

```

61 // вывод множества set
62 void ordered_array_set_print(ordered_array_set set);
63
64 // освобождает память, занимаемую множеством set
65 void ordered_array_set_delete(ordered_array_set set);
66
67 #endif

```

---

\* Задачи с *codeforces*:

1. Определи маршрут (1056A)
2. Пропущенная серия (440A)
3. Перестановка букв (1093B)
4. Тихий класс (1166A)
5. Щедрый Кефа (841A)
6. Перекраска собачек (1025A)
7. Ступени (1011A)
8. Башни (37A)
9. Бейджик (1020B)
10. Разнообразие - это хорошо (672B)
11. Игра: Банковские карты (777B)
12. Противоположности притягиваются (131B)

### Требования к лабораторной работе:

Вы **обязаны** выполнить автоматизированное тестирование разработанных алгоритмов. Опишем тестирование операции объединения. В функции `main` опишите тесты для проверки работы операции:

```

1 void test_unordered_array_set_union1() {
2     unordered_array_set set1 =
3         unordered_array_set_create_from_array((int[]){1, 2}, 2);
4     unordered_array_set set2 =
5         unordered_array_set_create_from_array((int[]){2, 3}, 2);
6     unordered_array_set resSet =
7         unordered_array_set_union(set1, set2);
8     unordered_array_set expectedSet =
9         unordered_array_set_create_from_array((int[]){1, 2, 3}, 3);
10    assert(unordered_array_set_isEqual(resSet, expectedSet));
11
12    unordered_array_set_delete(set1);
13    unordered_array_set_delete(set2);
14    unordered_array_set_delete(resSet);
15    unordered_array_set_delete(expectedSet);
16 }
17
18 void test_unordered_array_set_union2() {
19     unordered_array_set set1 =

```

```

20     unordered_array_set_create_from_array((int[]){1, 2, 3}, 3);
21     unordered_array_set set2 =
22         unordered_array_set_create_from_array((int[]){}, 0);
23     unordered_array_set resSet =
24         unordered_array_set_union(set1, set2);
25     unordered_array_set expectedSet =
26         unordered_array_set_create_from_array((int[]){1, 2, 3}, 3);
27     assert(unordered_array_set_isEqual(resSet, expectedSet));
28
29     unordered_array_set_delete(set1);
30     unordered_array_set_delete(set2);
31     unordered_array_set_delete(resSet);
32     unordered_array_set_delete(expectedSet);
33 }
```

---

Полученные тесты, объедините в функцию, которая вызывает все тесты по операции объединения:

```

1 void test_unordered_array_set_union() {
2     test_unordered_array_set_union1();
3     test_unordered_array_set_union2();
4 }
```

---

Полученные тесты по перечисленным ниже функциям объедините в функцию `test`:

```

1 void test() {
2     test_unordered_array_set_in();
3     test_unordered_array_set_isSubset();
4     test_unordered_array_set_insert();
5     test_unordered_array_set_deleteElement();
6     test_unordered_array_set_union();
7     test_unordered_array_set_intersection();
8     test_unordered_array_set_difference();
9     test_unordered_array_set_symmetricDifference();
10    test_unordered_array_set_complement();
11 }
```

---

В функции `main` проведите запуск всех тестов:

```

1 int main() {
2     test();
3
4     return 0;
5 }
```

---

Когда появилось свободное время, мне удалось посетить занятие коллеги, в котором обсуждалась тема множеств. Из разговора преподавателя со студентом, сдававшим лабораторную:

Преподаватель: натуральные числа начинаются с нуля или с единицы?

Студент: ... .

Преподаватель: по крайней мере в нашей стране.

Студент: ... .

Преподаватель: (пытается намекнуть, что натуральные числа используются при счёте) подсчитайте, сколько людей в аудитории.

Студент: один, два...

Преподаватель: с чего начали?

Студент: с Ивана Сергеевича!

## 18.12 Лабораторная работа «Реализация структуры данных «Вектор»

**Цель работы:** усовершенствование навыков в создании библиотек, получение навыков работы с системой контроля версий *git*.<sup>86</sup>



Содержание отчета:<sup>87</sup>

- Тема лабораторной работы.
- Цель лабораторной работы.
- Ссылка на открытый репозиторий с решением.
- Исходный код файлов<sup>88</sup>:
  - `vector.h` / `vector.c`
  - `*vectorVoid.h` / `vectorVoid.c`
  - `main.c`
- Результат выполнения команд<sup>89</sup>

<sup>86</sup>Было время, когда я трижды переделывал большую работу с базами данных, будучи студентом, так как ломал проект. Если бы я тогда пользовался системами контроля версий....

<sup>87</sup>Фрагменты отчета, помеченные звёздочкой требуются для получения максимального балла.

<sup>88</sup>Используйте такой шрифт, чтобы строки кода умещались по ширине.

<sup>89</sup>Оформление выполняется исключительно скриншотом.

```
git log --stat -- libs/data_structures/vector/ main.c
*git log --stat -- libs/data_structures/vectorVoid/
```

- Выводы по работе.

#### Требования:

- Обратите особое внимание на задания к лабораторной работе. В частности, на требование к коммитам в процессе выполнения работы. Если работа будет выполнена без них или будут отсутствовать промежуточные коммиты, она не будет засчитана. С целью улучшения умения чтения текста лабораторной, будет выдано дополнительное задание.

Когда-нибудь вы научитесь и этому... Я буду за вас рад ♡:



### Установка *Git* и настройка *GitHub*:

- Перейдите на официальный сайт <https://git-scm.com/downloads> для установки *Git*;



Рис. 18.3: Внешний вид страницы сайта

- Выберите свою операционную систему (далее все пояснения будут для обладателей лучшей операционной системы *Windows*);



Рис. 18.4: Внешний вид страницы сайта

3. Выберите установщик в соответствии с разрядностью вашей системы;

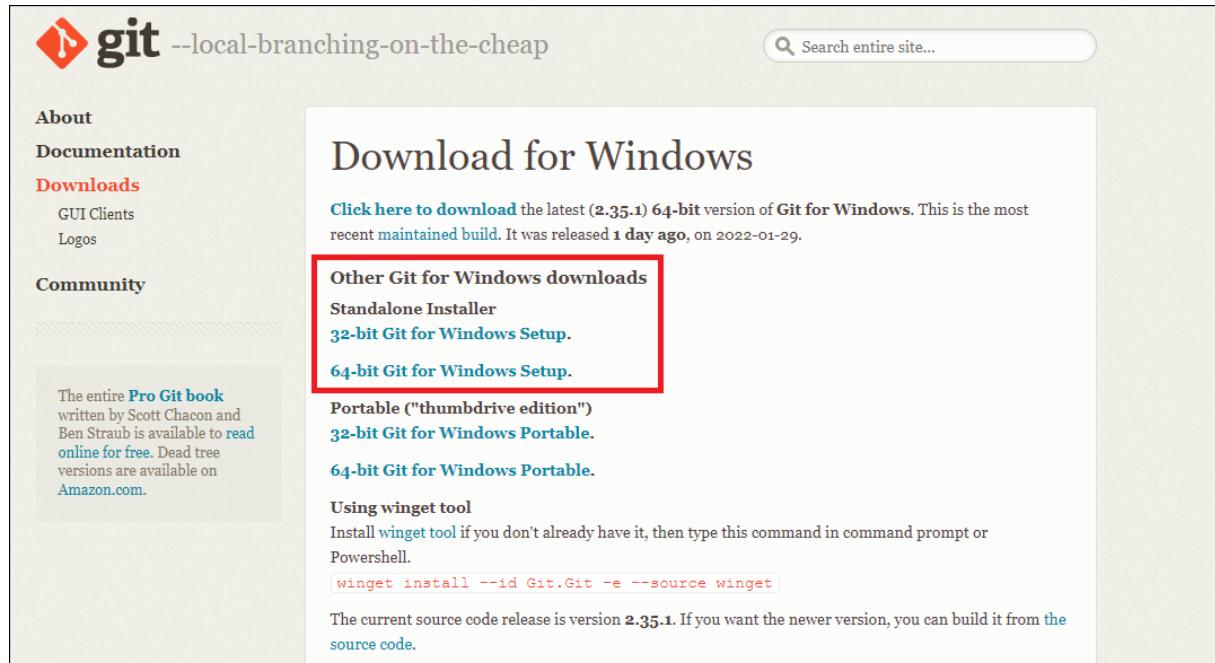


Рис. 18.5: Выбор установщика *Git*

4. Проходим процедуру установки. **Не меняйте** никакие параметры при установке. При желании, можно изменить **только** расположение файлов установщика;

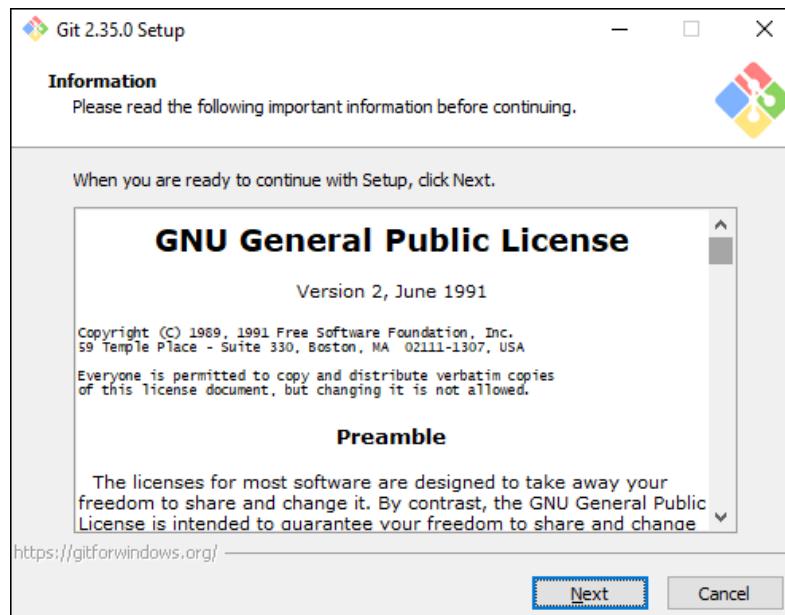


Рис. 18.6: Начальное окно установки

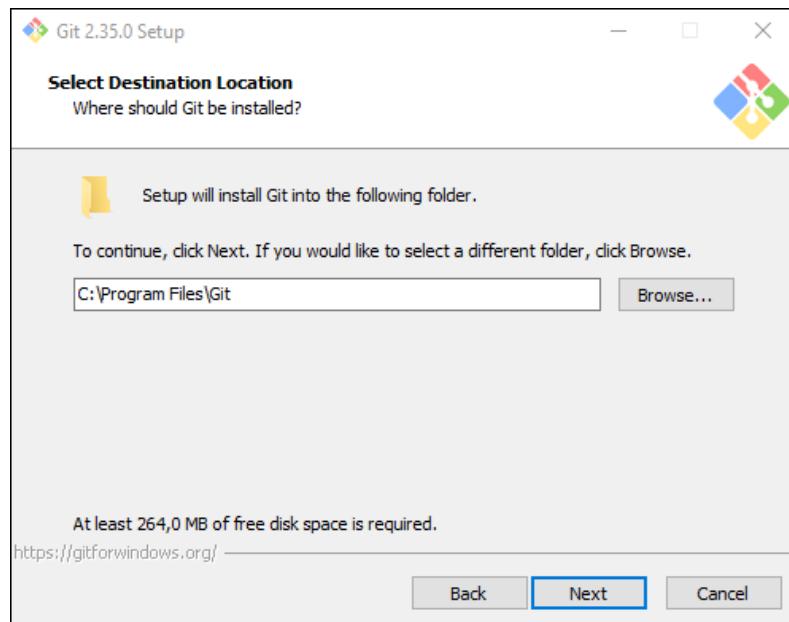


Рис. 18.7: Окно выбора расположения файлов установщика

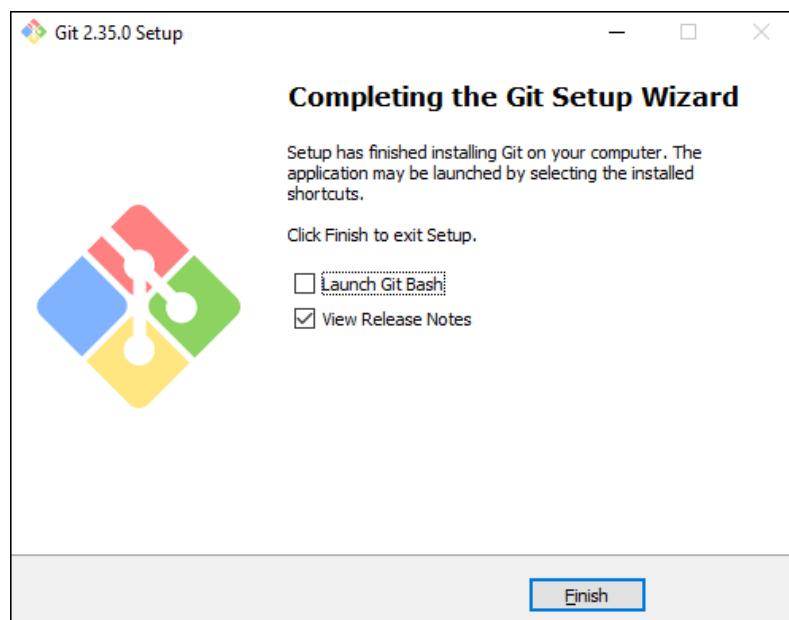
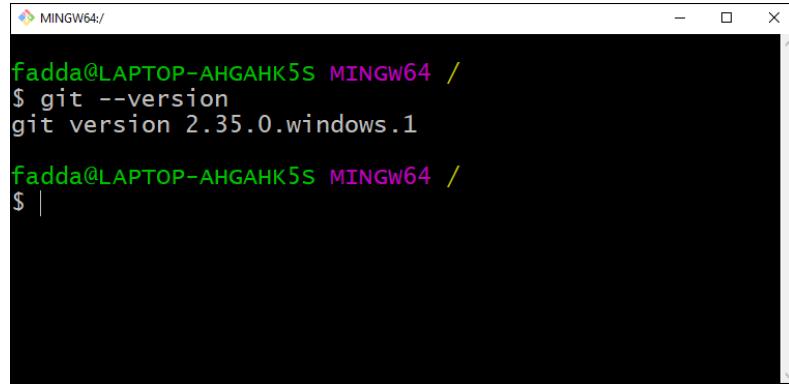


Рис. 18.8: Окно завершения установки *Git*

5. Закройте окно завершения установки и перезагрузите устройство;
6. Откройте *Git Bash*. Это можно сделать через поиск приложений Windows;
7. Введите команду "`git --version`". Если у вас была выведена версия *Git*, то поздравляю, у вас установлен *Git*! Однако, если версия не была выведена, попробуйте установить ещё раз;



```
Fadda@LAPTOP-AHGAHK5S MINGW64 /
$ git --version
git version 2.35.0.windows.1
```

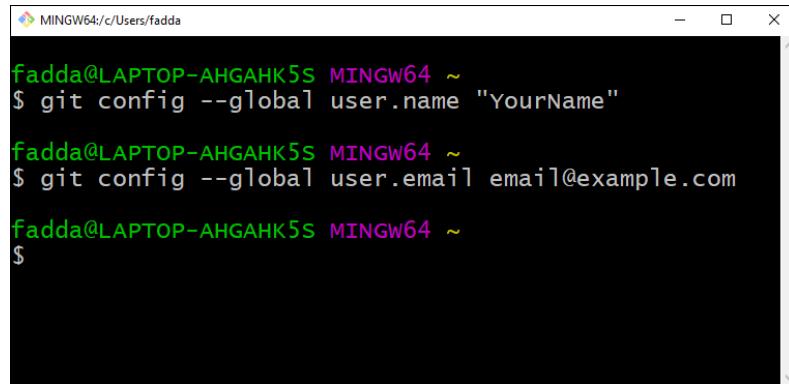
Рис. 18.9: Окно *Git Bash*

8. Введите данные команды, чтобы указать ваше имя и адрес электронной почты:

---

```
git config --global user.name "<Ваш псевдоним>"
git config --global user.email <Ваша электронная почта>
```

---



```
Fadda@LAPTOP-AHGAHK5S MINGW64 ~
$ git config --global user.name "YourName"

Fadda@LAPTOP-AHGAHK5S MINGW64 ~
$ git config --global user.email email@example.com

Fadda@LAPTOP-AHGAHK5S MINGW64 ~
$
```

Рис. 18.10: Окно *Git Bash*

9. Существует 2 способа перехода к определённой директории в *Git Bash*:

- 1) Использовать команду в *Git Bash* для перехода в определённую директорию:

---

```
cd "<расположение директории>"
```

---

Для того, чтобы вставить скопированный текст, используйте комбинацию клавиш (*Shift + Ins*), или через ПКМ по окну консоли.

```

MINGW64:/d/Test/Git project
$ cd "D:\\Test\\Git project"

```

Рис. 18.11: Окно *Git Bash*

- 2) Перейти в папку с нужной директорией, нажать на неё ПКМ и выбрать "*Git Bash Here*".

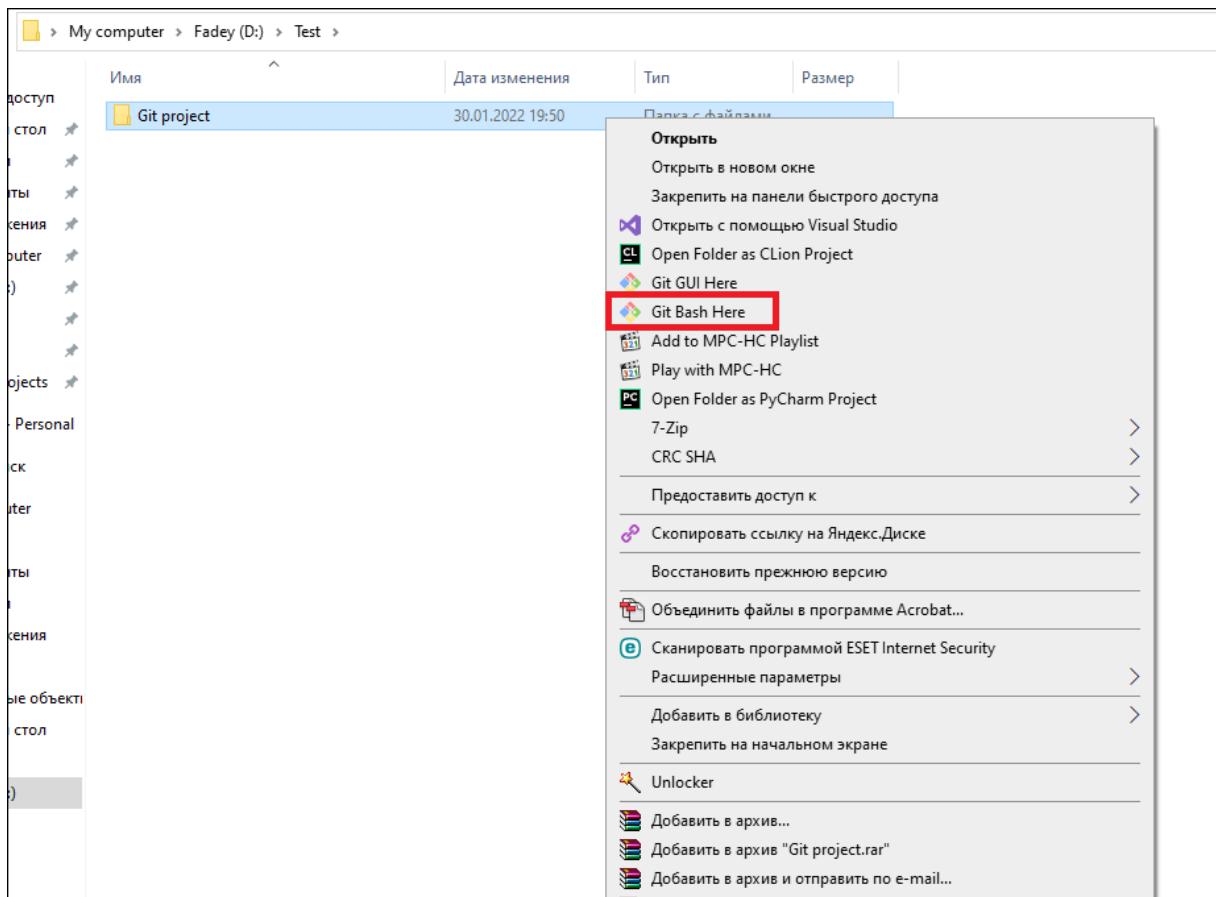
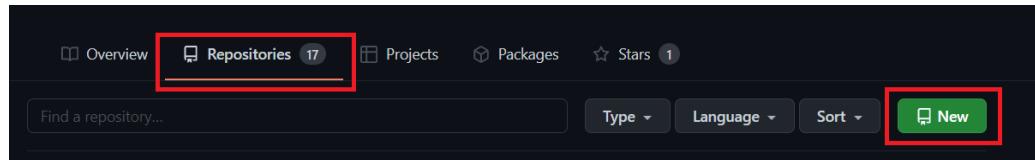


Рис. 18.12: Окно проводника

### Задания к лабораторной работе:

1. Перейдите на [github](#). Создайте **публичный** репозиторий для последующих лабораторных работ с произвольным названием:

- Перейдите во вкладку '*Repositories*' и нажмите на '*New*':



- Укажите имя репозитория и сделайте его публичным:

**Create a new repository**

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \* Repository name \*

ISPritchin / BasicsOfProgrammingCourse ✓

Great repository names are short and memorable. Need inspiration? How about [reimagined-octo-dollop](#)?

Description (optional)

**Public**  
Anyone on the internet can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

**Add a README file**  
This is where you can write a long description for your project. [Learn more](#).

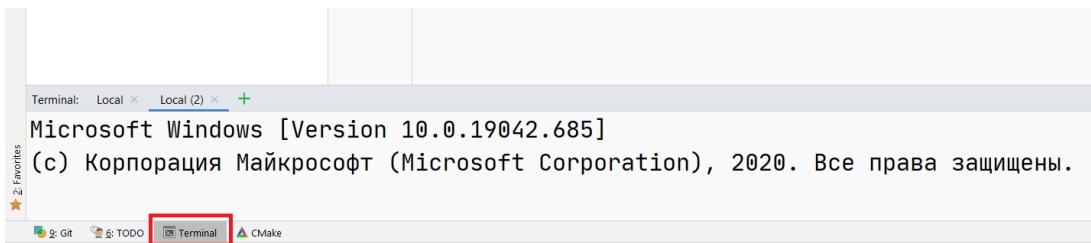
**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more](#).

**Choose a license**  
A license tells others what they can and can't do with your code. [Learn more](#).

**Create repository**

Рис. 18.13: Создание репозитория

- Нажмите на '*Create repository*'.
- Откройте папку проекта, например, в терминале *CLion* (просто перейдите во вкладку '*Terminal*') или в *Git Bash*:



и введите следующую последовательность команд только уже для своего репозитория (большая часть из них будет отображена после создания репозитория на *github*):

- `git init` – создает новый репозиторий
- `git add .` – команда сообщает *git*, что вы хотите включить изменения в конкретном файле в следующий коммит.
- `git commit -m "first commit"` – в кавычках указывается текст коммита – сопроводительное сообщение к изменяемым файлам.
- `git remote add origin https://github.com/ISPritchin/BasicsOfProgrammingCourse.git` – связывает локальный и удалённый репозиторий.
- `git push origin master` – публикация локальных изменений в удалённом репозитории.

Если всё пройдёт успешно, после последней команды вы увидите:

```
C:\Users\John\CLionProjects\course>git push origin master
Enumerating objects: 280, done.
Counting objects: 100% (280/280), done.
Delta compression using up to 6 threads
Compressing objects: 100% (252/252), done.
Writing objects: 100% (280/280), 207.58 KiB | 5.46 MiB/s, done.
Total 280 (delta 106), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (106/106), done.
To https://github.com/ISPritchin/BasicsOfProgrammingCourse.git
 * [new branch]      master -> master
```

Рис. 18.14: Отправка данных в удалённый репозиторий. Фрагмент выводимого сообщения

а после обновления страницы с репозиторием:

Commit Details	Date	
.idea	first commit	1 hour ago
cmake-build-debug	first commit	1 hour ago
libs	first commit	1 hour ago
CMakeLists.txt	first commit	1 hour ago
main.c	first commit	1 hour ago

2. Создайте заголовочный файл `libs\data_structures\vector\vector.h` и файл реализации `libs\data_structures\vector\vector.c`. Дополните `libs\data_structures\CMakeLists.txt`:

```
add_library(data_structures
    // прочие файлы библиотеки
    vector/vector.c
)
```

3. В заголовочном файле `vector.h` объявите структуру вектор:

```
typedef struct vector {
    int *data;           // указатель на элементы вектора
    size_t size;         // размер вектора
    size_t capacity;    // вместимость вектора
} vector;
```

4. Вектор представляет собой размещенный в динамической памяти массив, который автоматически расширяется по мере необходимости. Может быть представлен так:

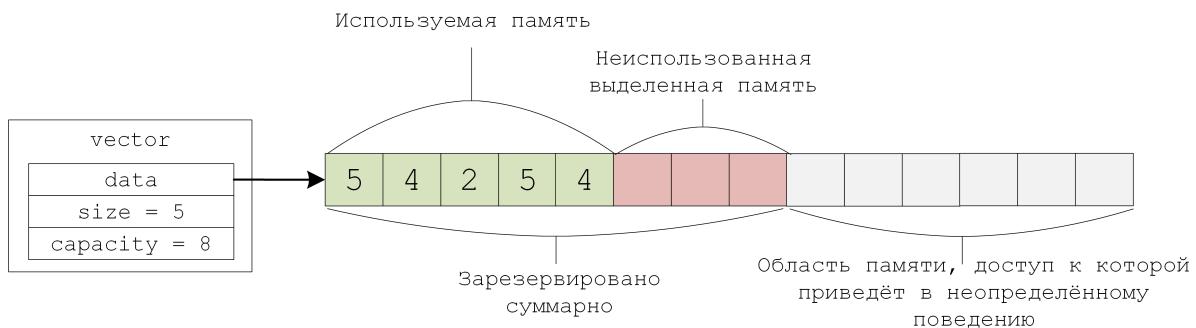
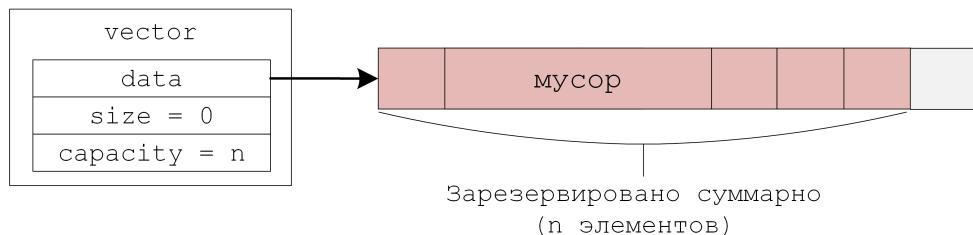


Рис. 18.15: Вектор из 5 элементов и вместимостью в 8 элементов

Реализуйте следующие функции для создания вектора и управления памятью:

- `vector createVector(size_t n)` – возвращает структуру-дескриптор вектор из `n` значений.



Требования к реализации:

- Пользователь должен иметь возможность создавать вектор размера 0.

- Если операционная система не может предоставить нужный объем памяти под размещение динамического массива, программа должна выводить сообщение в поток ошибок и заканчиваться с кодом 1:

---

```
fprintf(stderr, "bad alloc");
exit(1);
```

---

По окончанию реализации функции выполните следующий код:

---

```
#include <stdio.h>
#include <stdint.h>
#include "libs/data_structures/vector/vector.h"

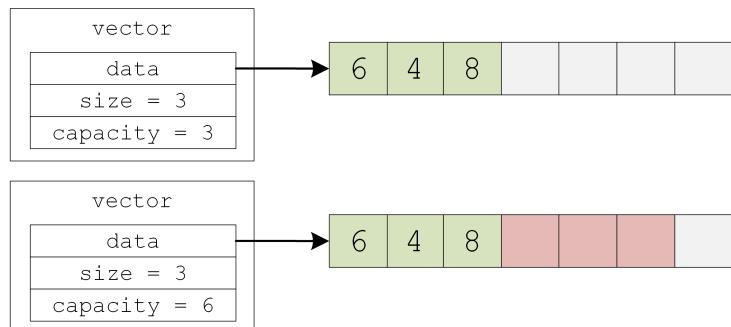
int main() {
    vector v = createVector(SIZE_MAX);

    return 0;
}
```

---

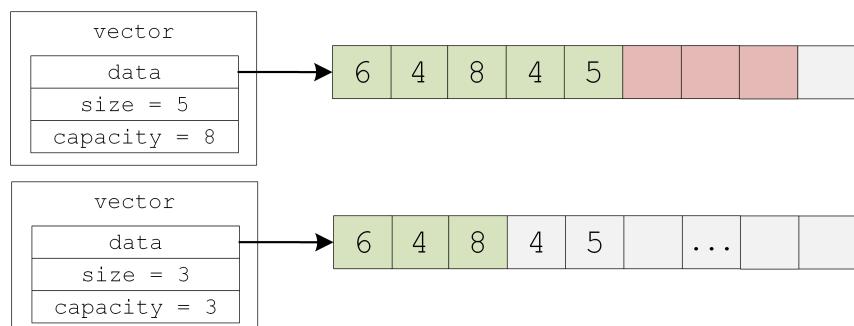
Результаты работы функции должны будут отобразиться в консоли. Сделайте соответствующие выводы.

- `void reserve(vector *v, size_t newCapacity)` – изменяет количество памяти, выделенное под хранение элементов вектора. Пример, когда новая вместимость больше:

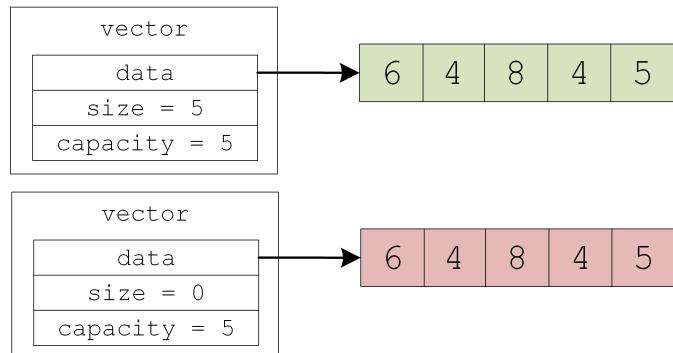


Должны соблюдаться следующие правила:

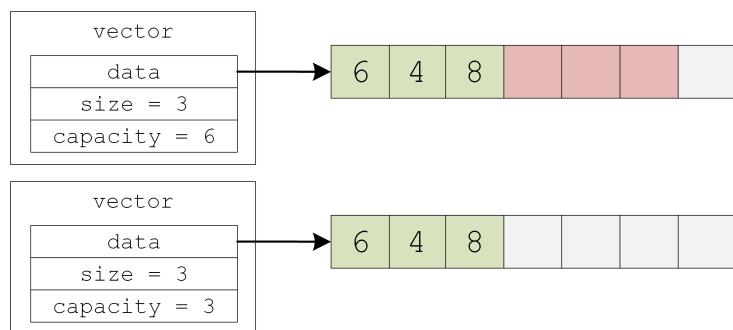
- Если в качестве `newCapacity` указано значение 0, в `data` должен быть записан `NULL`.
- Если значение `newCapacity` стало меньше `size`, тогда значение `size` должно равняться `newCapacity`:



- Если ОС не смогла выделить необходимый фрагмент памяти, вывести сообщение в поток ошибок и прервать выполнение программы.
- `void clear(vector *v)` – удаляет элементы из контейнера, но не освобождает выделенную память. Тело функции должно содержать ровно одну строку:



- `void shrinkToFit(vector *v)` – освобождает память, выделенную под неиспользуемые элементы. Тело функции должно содержать ровно одну строку.



- `void deleteVector(vector *v)` – освобождает память, выделенную векторы.

5. Сделайте коммит в репозитории:

---

```
git add .
git commit -m "memory usage of vector"
git push origin master
```

---

6. Реализуйте функции проверки на то, является ли вектор пустым (`bool isEmpty(vector *v)`) и полным (`bool isFull(vector *v)`). Тело каждой функции – одна строка кода<sup>90</sup>.
7. Реализуйте функцию `int getVectorValue(vector *v, size_t i)`, которая возвращает `i`-ый элемент вектора `v`. Тело функции – одна строка кода.
8. Реализуйте функции добавления и удаления элемента:

---

<sup>90</sup>Вектор пуст, если в нём нет 'полезных' элементов, хотя вместимость не обязана равняться нулю.  
Вектор полон, когда используется вся доступная вектору вместимость.

- `void pushBack(vector *v, int x)` – добавляет элемент `x` в конец вектора `v`. Если вектор заполнен, увеличьте количество выделенной ему памяти в 2 раза, используя `reserve`. Пример заполнения вектора представлен на рисунке 18.16.

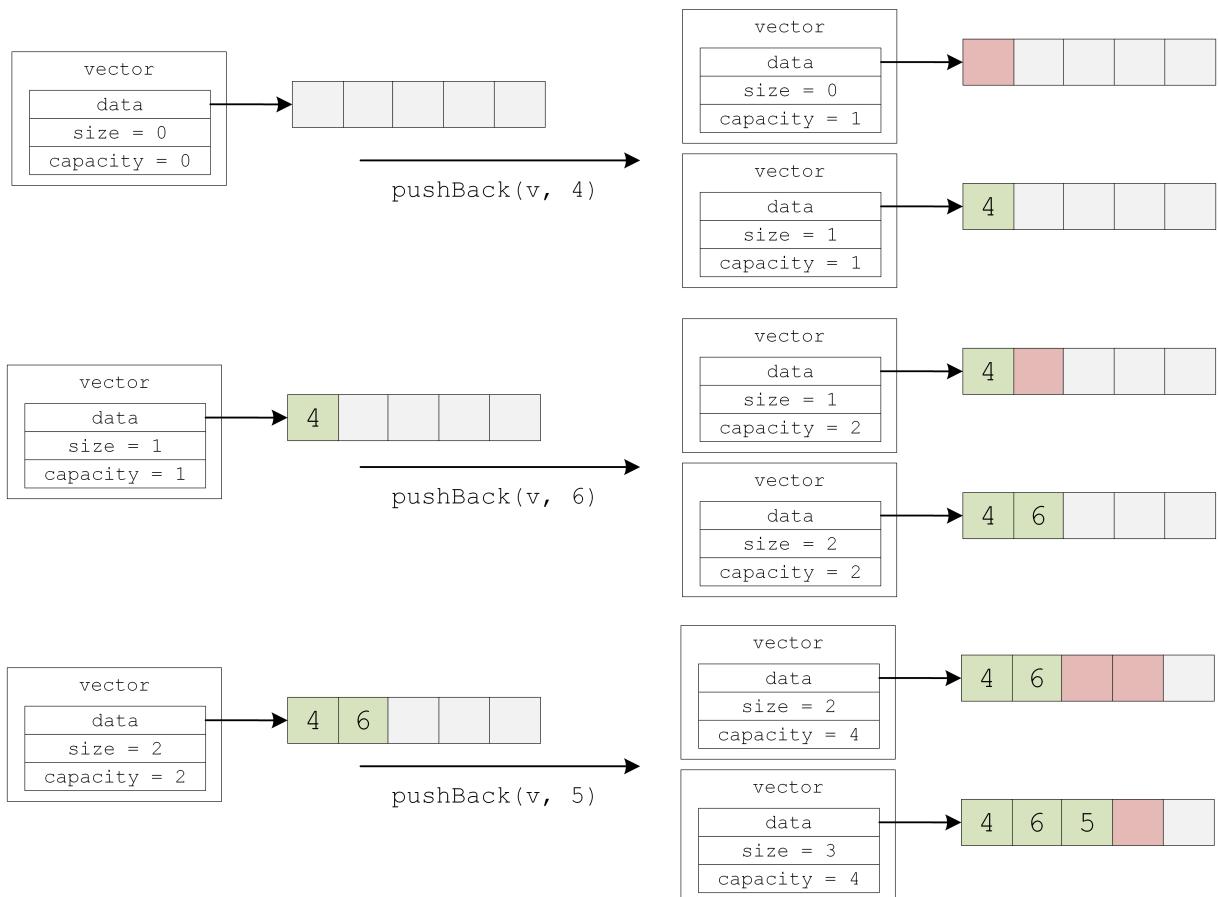


Рис. 18.16: Процесс заполнения элементов вектора

Протестируйте следующие случаи:

- Вектор пустой `void test_pushBack_emptyVector()`.
- Вектор заполнен `void test_pushBack_fullVector()`.

Обратите внимание на правила именования тестов. Последовательность составляющих следующая:

- (a) Слово `test`;
- (b) Разделяющее нижнее подчёркивание `_`;
- (c) Имя тестируемой функции, например, `pushBack`;
- (d) Разделяющее нижнее подчёркивание `_`;
- (e) Краткое описание случая `emptyVector`.

В файле `main.c` создайте функцию `test` для тестов. Все добавляемые тесты должны вызываться из неё.

```
void test() {
    test_pushBack_emptyVector();
    test_pushBack_fullVector();
    // последующие тесты
}
```

- `void popBack(vector *v)` – удаляет последний элемент из вектора. Функция должна 'выкидывать' в поток ошибок сообщение, если вектор пуст и закончить выполнение с кодом 1. Проверьте функциональность тестом, например, таким:

---

```
void test_popBack_notEmptyVector() {
    vector v = createVector(0);
    pushBack(&v, 10);

    assert(v.size == 1);
    popBack(&v);
    assert(v.size == 0);
    assert(v.capacity == 1);
}
```

---

9. Выполните коммит "append push / pop functions"
10. Реализуйте следующие дополнительные функции получения доступа к элементам:

- `int* atVector(vector *v, size_t index)` – возвращает указатель на `index`-ый элемент вектора. Если осуществляется попытка получить доступ вне пределов используемых элементов вектора, в поток ошибок должна выводиться ошибка: "`IndexError: a[index] is not exists`", где в качестве `index` указывается позиция элемента, к которому пытались осуществить доступ.
- `int* back(vector *v)` – возвращает указатель на последний элемент вектора.
- `int* front(vector *v)` – возвращает указатель на нулевой элемент вектора.

Выполните проверку реализованных функций. К сожалению, проверить ошибочные пути выполнения у вас не получится. Убедитесь, что обращение к существующим элементам корректно. Дополните тесты:

- `void test_atVector_notEmptyVector();`
- `void test_atVector_requestToLastElement()`<sup>91</sup>;
- `void test_back_oneElementInVector();`
- `void test_front_oneElementInVector();`

11. Выполните коммит "append access functions"
12. Выполните рефакторинг разработанного решения. Посмотрите, можете ли что-то улучшить? Сделайте код лучше настолько, насколько считаете возможным. Не забудьте перенести реализацию из `.h`-файла в соответствующий `.c`-файл.
13. Выполните коммит "refactoring".

---

<sup>91</sup>Часто ошибки заседают на пограничных случаях. Проверяйте их.

14. Выполните команду `git log --stat -- libs/data_structures/vector/ main.c` и приложите в отчёт результат выполнения. Пример на рисунке 18.17.

```
John@HOME-PC MINGW64 ~/CLionProjects/course (master)
$ git log --stat -- libs/data_structures/vector/ main.c
commit 08166a2175956a2b3bec01c52213c34bf261d206 (HEAD -> master, origin/master)
Author: ISPritchin <ISPritchin@gmail.com>
Date:   Sat Jan 29 20:09:39 2022 +0300

    refactoring

libs/data_structures/vector/vector.c | 78 ++++++-----+
libs/data_structures/vector/vector.h | 74 +----+-----+
2 files changed, 85 insertions(+), 67 deletions(-)

commit 9800800a1391fcc2b728d10699bcb7e76cf134a2
Author: ISPritchin <ISPritchin@gmail.com>
Date:   Sat Jan 29 20:03:43 2022 +0300

    append access functions

libs/data_structures/vector/vector.h | 16 ++++++-----+
main.c                            | 27 ++++++-----+
2 files changed, 42 insertions(+), 1 deletion(-)

commit f3f83d1142dc86abc168d2e2d0f1d88328358e09
Author: ISPritchin <ISPritchin@gmail.com>
Date:   Sat Jan 29 18:48:54 2022 +0300

    append push / pop functions

libs/data_structures/vector/vector.h | 32 ++++++-----+
main.c                            | 39 ++++++-----+
2 files changed, 69 insertions(+), 2 deletions(-)

commit cf03696a38e62fb8c01d78f6d220a09097fc9fc6
Author: ISPritchin <ISPritchin@gmail.com>
Date:   Sat Jan 29 17:07:44 2022 +0300

    memory usage of vector

libs/data_structures/vector/vector.h | 37 ++++++-----+
main.c                            | 17 +----+-----+
2 files changed, 40 insertions(+), 14 deletions(-)

commit b7d2963fd17620b037c284055fec548f43387845
Author: ISPritchin <ISPritchin@gmail.com>
Date:   Sat Jan 29 13:54:33 2022 +0300

    first commit

libs/data_structures/vector/vector.c |  5 +----+
libs/data_structures/vector/vector.h |  4 +----+
main.c                            | 20 ++++++-----+
3 files changed, 29 insertions(+)
```

Рис. 18.17: Результат выполнения запроса

Анализируя полученные результаты, мне показалось, что какой-то 'изюминки' в лабораторной не хватает. Так появилось продолжение. Оно необходимо только для получения максимального балла по работе:

1. Создадим тип `voidVector`, при помощи которого можно оперировать вектором произвольного типа. Для этого объявим структуру

---

```

#ifndef INC_VECTORVOID_H
#define INC_VECTORVOID_H

#include <limits.h>

typedef struct vectorVoid {
    void *data;           // указатель на нулевой элемент вектора
    size_t size;          // размер вектора
    size_t capacity;      // вместимость вектора
    size_t baseTypeSize;  // размер базового типа:
                          // например, если вектор хранит int -
                          // то поле baseTypeSize = sizeof(int)
                          // если вектор хранит float -
                          // то поле baseTypeSize = sizeof(float)
} vectorVoid;

#endif

```

---

в `libs\data_structures\vector\vectorVoid.h`

2. Подключите `vectorVoid.c` в `libs\data_structures\CMakeLists.txt`.

Попробуйте создать структуру типа `vectorVoid` в `main` и после сделайте коммит `"init vectorVoid"`.

3. Добавьте функции:

- `vectorVoid createVectorV(size_t n, size_t baseTypeSize)`
- `void reserveV(vectorVoid *v, size_t newCapacity)`
- `void shrinkToFitV(vectorVoid *v)`
- `void clearV(vectorVoid *v)`
- `void deleteVectorV(vectorVoid *v)`

В большей части они совпадают реализациями функций `vector`. Поэтому 'хорошой' стратегией было бы копирование кода из `vector.h` / `vector.c` с незначительной модификацией.

Сделайте коммит `"memory usage of vectorVoid"`.

4. Добавьте функции:

- `bool isEmptyV(vectorVoid *v);`
- `bool isFullV(vectorVoid *v);`
- `void getVectorValueV(vectorVoid *v, size_t index, void *destination)`  
– записывает по адресу `destination` `index`-ый элемент вектора `v`. Вычисление адреса местоположения элемента и его копирование:

---

```

char *source = (char *) v->data + index * v->baseTypeSize;
memcpy(destination, source, v->baseTypeSize);

```

---

- `void setVectorValueV(vectorVoid *v, size_t index, void *source)` – записывает на `index`-ый элемент вектора `v` значение, расположенное по адресу `source`;
- `void popBackV(vectorVoid *v);`
- `void pushBackV(vectorVoid *v, void *source).`

Проверьте реализацию на тесте:

---

```

int main() {
    size_t n;
    scanf("%zd", &n);

    vectorVoid v = createVectorV(0, sizeof(int));
    for (int i = 0; i < n; i++) {
        int x;
        scanf("%d", &x);

        pushBackV(&v, &x);
    }

    for (int i = 0; i < n; i++) {
        int x;
        getVectorValueV(&v, i, &x);

        printf("%d ", x);
    }

    return 0;
}

```

---

и в варианте:

---

```

int main() {
    size_t n;
    scanf("%zd", &n);

    vectorVoid v = createVectorV(0, sizeof(float));
    for (int i = 0; i < n; i++) {
        float x;
        scanf("%f", &x);

        pushBackV(&v, &x);
    }

    for (int i = 0; i < n; i++) {
        float x;
        getVectorValueV(&v, i, &x);

        printf("%f ", x);
    }

    return 0;
}

```

---

Отметьте для себя, что наш вектор неплохо работает как с целыми, так с вещественными значениями. При желании вы могли бы использовать структуры и другие типы данных. Если ошибки не были получены, делаем коммит: "append push / pop functions".

5. Выполните ручное тестирование разработанного решения, устранение багов, рефакторинг и сделайте итоговый коммит "refactoring / bug fix"

6. Выполните команду

---

```
git log --stat -- libs/data_structures/vectorVoid/
```

---

и приложите в отчёт результат выполнения.

## 18.13 Лабораторная работа «Работа с многомерными массивами»

**Цель работы:** получение навыков работы с многомерными массивами.

**Содержание отчета:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Решения задач:
  - Текст задания.
  - Исходный код (в том числе и тестов).
  - Задания со звездочкой не являются обязательными, но их решение требуется для получения максимального балла.
- Ссылка на открытый репозиторий с решением.
- Скриншот с историей коммитов.

**Требования:**

- После решения каждой задачи из второго блока должен делаться коммит. **В случае нарушения данного требования работа будет выполняться заново.**
- Выполняйте грамотное именование тестов. Это позволит более тщательно подходить к наборам тестовых данных.
- Не создавайте наборы тестовых данных, которые проверяют один и тот же случай. Тестовые случаи должны отличаться между собой концептуально (вспомните лабораторную За).
- Одного теста недостаточно для проверки работоспособности. Обдумывайте крайние случаи.

**Рекомендации:**

- Возможно, для ваших решений потребуется доступ к библиотеке `algorithms`, реализованной ранее. Чтобы сборка прошла успешно, для `CMakeLists.txt` в папке `data_structures` используйте:

```
add_library(data_structures
            matrix/matrix.c
            )

target_link_libraries(data_structures algorithms)
```

- Всегда используйте цикл `for`, если заранее можно сказать точное количество итераций цикла.

### Задания к лабораторной работе:

1. В заголовочном файле `libs\data_structures\matrix\matrix.h` объявите структуру 'матрица' и 'позиция':

---

```

typedef struct matrix {
    int **values; // элементы матрицы
    int nRows; // количество рядов
    int nCols; // количество столбцов
} matrix;

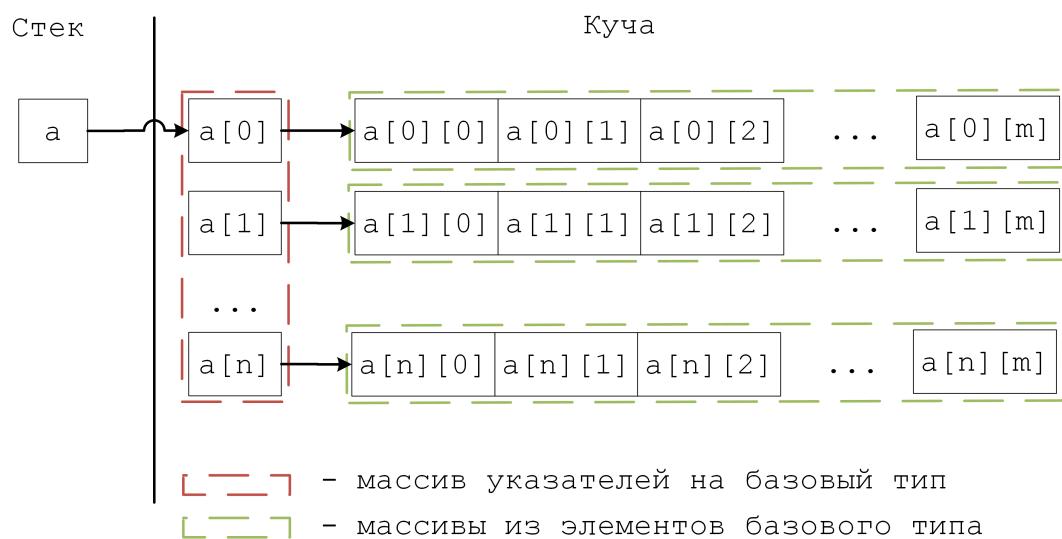
typedef struct position {
    int rowIndex;
    int colIndex;
} position;

```

---

2. В библиотеке `matrix` реализуйте функции для размещения в динамической памяти матриц:

- (а) `matrix getMemMatrix(int nRows, int nCols)` – размещает в динамической памяти матрицу размером `nRows` на `nCols`. Возвращает матрицу. Используется следующая схема размещения:



Размещение матрицы состоит из двух этапов:

- Размещается в памяти массив указателей на целое.
- Для каждого указателя из массива указателя размещается массив базового типа.

Реализация:

---

```

matrix getMemMatrix(int nRows, int nCols) {
    int **values = (int **) malloc(sizeof(int*) * nRows);
    for (int i = 0; i < nRows; i++)
        values[i] = (int *) malloc(sizeof(int) * nCols);
    return (matrix){values, nRows, nCols};
}

```

---

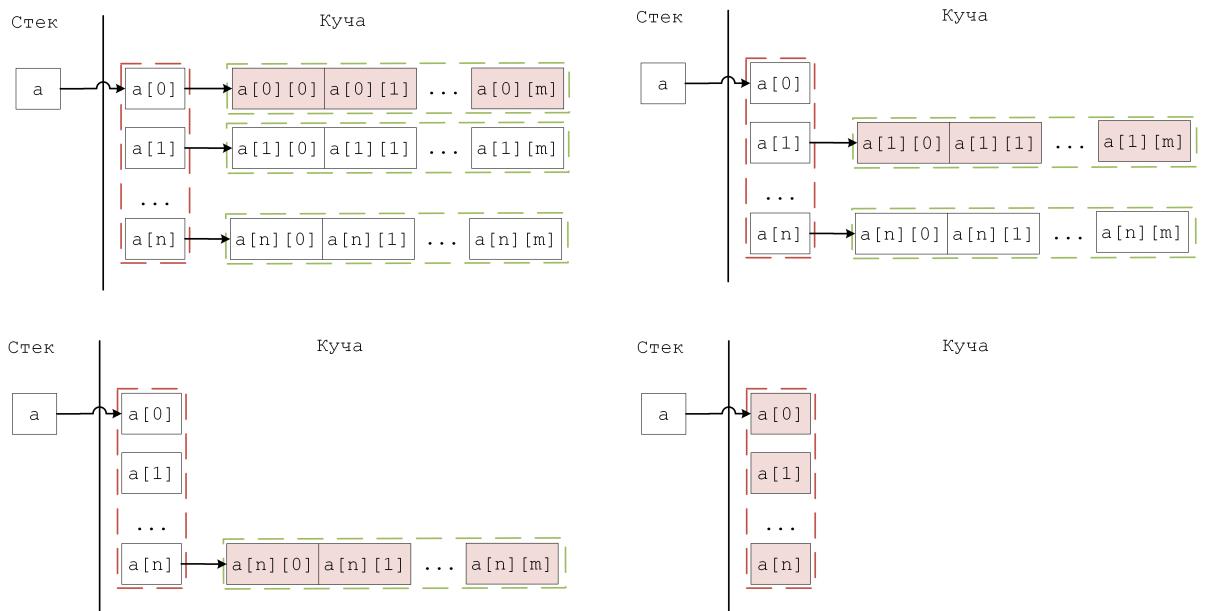


Рис. 18.18: Освобождение динамической памяти

- (b) `matrix *getMemArrayOfMatrices(int nMatrices, int nRows, int nCols)`  
 – размещает в динамической памяти массив из `nMatrices` матриц размером `nRows` на `nCols`. Возвращает указатель на нулевую матрицу.  
 Реализация:

```
matrix *getMemArrayOfMatrices(int nMatrices,
                               int nRows, int nCols) {
    matrix *ms = (matrix*) malloc(sizeof(matrix) * nMatrices);
    for (int i = 0; i < nMatrices; i++)
        ms[i] = getMemMatrix(nRows, nCols);
    return ms;
}
```

- (c) `void freeMemMatrix(matrix *m)` – освобождает память, выделенную под хранение матрицы `m`. Процесс освобождения изображен на рисунке 18.18.  
 (d) `void freeMemMatrices(matrix *ms, int nMatrices)` – освобождает память, выделенную под хранение массива `ms` из `nMatrices` матриц.

3. В библиотеке `matrix` реализуйте функции для ввода и вывода матриц:

Выдержка из Three Star Programmer:

“Система ранжирования С-программистов.

Чем выше уровень косвенности ваших указателей (т. е. чем больше \* перед вашими переменными), тем выше ваша репутация. Беззвёздочных С-программистов практически не бывает, так как практически все нетривиальные программы требуют использования указателей. Большинство являются однозвёздочными программистами. В старые времена (ну хорошо, я молод, поэтому это старые времена на мой взгляд) тот, кто случайно сталкивался с кодом, созданный трёхзвёздочным программистом, приходил в благоговейный трепет.

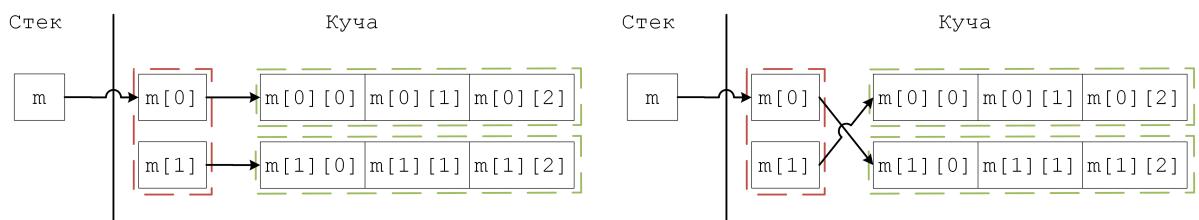
Некоторые даже утверждали, что видели трёхзвёздочный код, в котором указатели на функции применялись более чем на одном уровне косвенности. Как по мне, так эти рассказы столь же правдивы, сколь рассказы об НЛО.

Просто чтобы было ясно: если вас назвали Трёхзвёздочным Программистом, то обычно это не комплимент."

- (a) `void inputMatrix(matrix *m)` – ввод матрицы `m`.
- (b) `void inputMatrices(matrix *ms, int nMatrices)` – ввод массива из `nMatrices` матриц, хранящейся по адресу `ms`.
- (c) `void outputMatrix(matrix m)` – вывод матрицы `m`.
- (d) `void outputMatrices(matrix *ms, int nMatrices)` – вывод массива из `nMatrices` матриц, хранящейся по адресу `ms`.

4. В библиотеке `matrix` реализуйте функции для обмена строк и столбцов:

- (a) `void swapRows(matrix m, int i1, int i2)` – обмен строк с порядковыми номерами `i1` и `i2` в матрице `m`. Помните, что для этого достаточно обменять указатели соответствующих строк:



Сложность алгоритма  $O(1)$ . Дополнительно проверьте, что индексы не выходят за границы посредством `assert`.

- (b) `void swapColumns(matrix m, int j1, int j2)` – обмен колонок с порядковыми номерами `j1` и `j2` в матрице `m`.  
Обмен колонок будет заключаться в обмене `a[i][j1]` и `a[i][j2]` для всех `i` от `0` до `n - 1`. Сложность алгоритма  $O(n)$ .

5. В библиотеке `matrix` реализуйте функции для упорядочивания строк и столбцов:

- (a) `void insertionSortRowsMatrixByRowCriteria(matrix m, int (*criteria)(int*, int))` – выполняет сортировку вставками строк матрицы `m` по неубыванию значения функции `criteria` применяемой для строк<sup>92</sup>. Рассмотрим, где бы такая функция могла быть использована. Например, необходимо отсортировать строки матрицы по неубыванию сумм элементов строк. Действия следующие:

- Создаётся функция для вычисления суммы для одномерного массива.

---

```
getSum(int *a, int n);
```

---

- Данная функция передаётся как критерий в функцию сортировки матрицы:

---

```
insertionSortRowsMatrixByRowCriteria(m, getSum);
```

---

<sup>92</sup>В процессе сортировки создайте вспомогательный массив из `nRows` элементов, найдите значение функции для каждого ряда и выполните сортировку данного массива. В процессе обмена элементов полученного массива производите обмен строк при помощи `swapRows`.

- Функция `insertionSortRowsMatrixByRowCriteria` считает значение функции `getSum` для каждой строки матрицы. Результаты сохраняются в промежуточный массив.
- Выполняется сортировка промежуточного массива. Но когда обмениваются *i*-ый и *j*-ый элемент промежуточного массива, обмениваются и соответствующие им строки.

Действия изображены на схеме на рисунке 18.19.

Выполним оценку сложности процесса, исходя из того, что  $n$  – количество строк матрицы,  $m$  – количество столбцов. Пусть сложность вычисления критерия для одной строки имеет сложность  $O(x)$ . Тогда сложность алгоритма вычисления массива из значений критерия  $O(nx)$ . Сортировка вставками имеет сложность  $O(n^2)$ . Но в процессе сортировки будут обменяться не только элементы массива со значением критерия, но и строки матрицы, которые можно обменять за  $O(1)$ . Тогда сложность сортировки матрицы  $O(n^2)$ . Суммарная сложность процесса  $O(nx + n^2)$ .

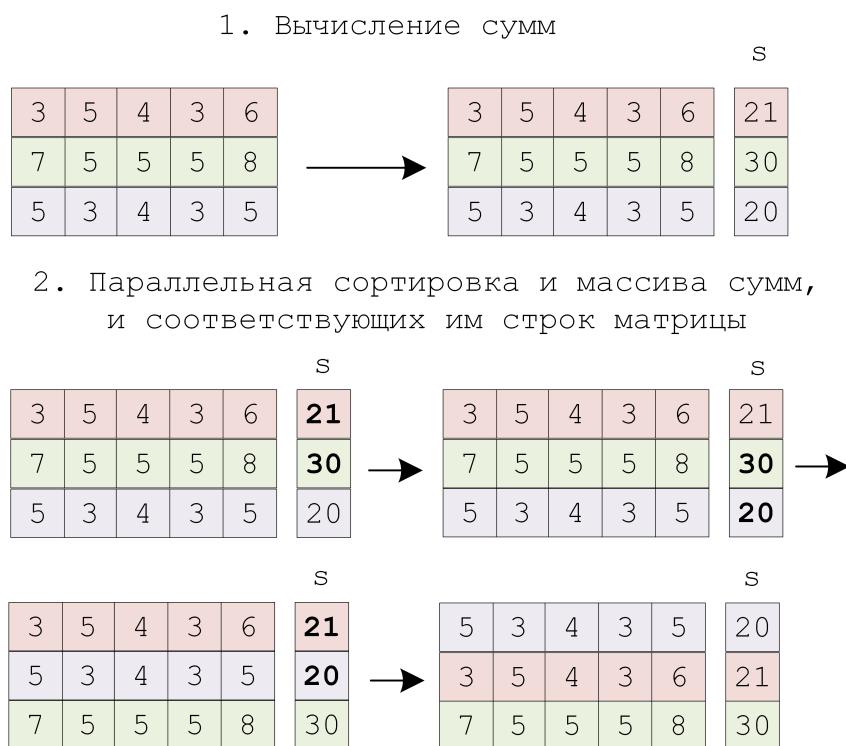


Рис. 18.19: Сортировка строк матрицы по критерию (неубывание сумм строк)

- (b) `void selectionSortColsMatrixByColCriteria(matrix m, int (*criteria)(int*, int))` – выполняет сортировку выбором столбцов матрицы  $m$  по неубыванию значения функции `criteria` применяемой для столбцов<sup>93</sup>.

Выполним оценку сложности процесса, исходя из того, что  $n$  – количество строк матрицы,  $m$  – количество столбцов. Пусть сложность вычисления

<sup>93</sup>В отличие от прошлого случая, функция применяется к столбцам матрицы. Однако функция-критерий ожидает указатель на последовательный участок памяти. Но в силу такого размещения матриц это невозможно, поэтому придётся выполнить копирование столбца в промежуточный массив и только потом вычислять значение критерия. Таким образом допускается, что будет использована дополнительная память для значений критерия и столбца матрицы.

критерия для одного столбца имеет сложность  $O(x)$ . Тогда сложность алгоритма вычисления массива из значений критерия  $O(mx)$ . Сортировка выбором имеет сложность  $O(m^2)$ . Но в процессе сортировки будут обменяться не только элементы массива со значением критерия, но и столбцы матрицы  $O(n)$ , тогда сложность сортировки матрицы  $O(m^2n)$ . Суммарная сложность процесса  $O(mx + m^2n)$ .

Вопрос на обсуждение: почему для обмена строк использовалась сортировка вставками, а для обмена столбцов сортировка выбором?

6. В библиотеке `matrix` реализуйте следующие функции-предикаты:

- `bool isSquareMatrix(matrix *m)` – возвращает значение 'истина', если матрица `m` является квадратной, ложь – в противном случае<sup>94</sup>.
- `bool areTwoMatricesEqual(matrix *m1, matrix *m2)` – возвращает значение 'истина', если матрицы `m1` и `m2` равны, ложь – в противном случае<sup>95</sup>.
- `bool isEMatrix(matrix *m)` – возвращает значение 'истина', если матрица `m` является единичной, ложь – в противном случае.
- `bool isSymmetricMatrix(matrix *m)` – возвращает значение 'истина', если матрица `m` является симметричной, ложь – в противном случае<sup>96</sup>.

7. В библиотеке `matrix` реализуйте следующие функции преобразования матриц:

- `void transposeSquareMatrix(matrix *m)` – транспонирует квадратную матрицу `m`.
- `void transposeMatrix(matrix *m)` – транспонирует матрицу `m`.

8. В библиотеке `matrix` реализуйте функции для поиска минимального и максимального элемента матрицы:

- `position getMinValuePos(matrix m)` – возвращает позицию минимального элемента матрицы `m`.
- `position getMaxValuePos(matrix m)` – возвращает позицию максимального элемента матрицы `m`.

9. Дополните библиотеку функциями для тестирования:

- `matrix createMatrixFromArray(const int *a, size_t nRows, size_t nCols)` – возвращает матрицу размера `nRows` на `nCols`, построенную из элементов массива `a`:

---

```
matrix createMatrixFromArray(const int *a,
                           int nRows, int nCols) {
    matrix m = getMemMatrix(nRows, nCols);

    int k = 0;
    for (int i = 0; i < nRows; i++)
```

<sup>94</sup>Функция должна содержать не более одной строки.

<sup>95</sup>Функция должна содержать вызов `memcmp`. Следовательно функция не должна содержать более одного `for`. Цикл `for` не должен быть вложен в `if`.

<sup>96</sup>Функция не должна содержать лишних сравнений элементов друг с другом. Например, элементы диагонали не должны оцениваться как таковые.

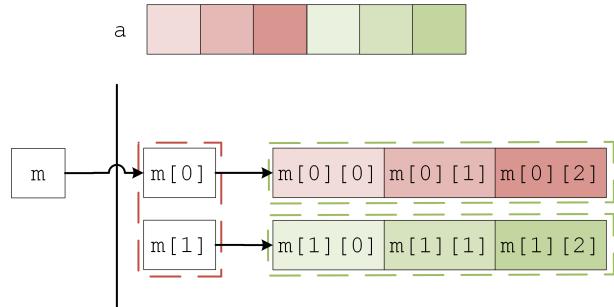
```

    for (int j = 0; j < nCols; j++)
        m.values[i][j] = a[k++];

    return m;
}

```

Опишем более подробно механизм работы данной функции. В третьей строке происходит выделение памяти под матрицу. Затем выполняется копирование элементов одномерного массива в строки матрицы:



Благодаря этому, можно осуществлять тестирование функций, принимающих матрицы:

```

void test_countZeroRows() {
    matrix m = createMatrixFromArray(
        (int[]) {
            1, 1, 0,
            0, 0, 0,
            0, 0, 1,
            0, 0, 0,
            0, 1, 1,
        },
        5, 3
    );

    assert(countZeroRows(m, 5, 3) == 2);

    freeMemMatrix(m, 5);
}

```

- `matrix *createArrayOfMatrixFromArray(const int *values, size_t nMatrices, size_t nRows, size_t nCols)` – возвращает указатель на нулевую матрицу массива из `nMatrices` матриц, размещенных в динамической памяти, построенных из элементов массива `a`:

```

matrix *createArrayOfMatrixFromArray(const int *values,
                                     size_t nMatrices, size_t nRows,
                                     ← size_t nCols) {

    matrix *ms = getMemArrayOfMatrices(nMatrices, nRows, nCols);

```

```

    int l = 0;
    for (size_t k = 0; k < nMatrices; k++)
        for (size_t i = 0; i < nRows; i++)
            for (size_t j = 0; j < nCols; j++)
                ms[k].values[i][j] = values[l++];
}

return ms;
}

```

При помощи реализованных функций выполнить решения следующих задач<sup>97</sup>:

1. Данна квадратная матрица, все элементы которой различны. Поменять местами строки, в которых находятся максимальный и минимальный элементы.
2. Упорядочить строки матрицы по неубыванию наибольших элементов строк:

7	1	2
1	8	1
3	2	3

→

3	2	3
7	1	2
1	8	1

- (a) `int getMax(int *a, int n)`  
(b) `void sortRowsByMinElement(matrix m)`
3. Данна прямоугольная матрица. Упорядочить столбцы матрицы по неубыванию минимальных элементов столбцов:

3	5	2	4	3	3
2	5	1	8	2	7
6	1	4	4	8	3

→

5	2	3	3	3	4
5	1	2	2	7	8
1	4	6	8	3	4

- (a) `int getMin(int *a, int n)`  
(b) `void sortColsByMinElement(matrix m)`
4. Если данная квадратная матрица  $A$  симметрична, то заменить  $A$  ее квадратом  $(A^2)$ <sup>98</sup>.

- `matrix mulMatrices(matrix m1, matrix m2)`
- `void getSquareOfMatrixIfSymmetric(matrix *m)`

<sup>97</sup> Каждое из заданий снабжено рекомендуемыми для выделения функциями (сверх тех, которые были реализованы ранее). Дополнительно содержится информация, на что следует обратить внимание.

<sup>98</sup> Симметричной называют квадратную матрицу, элементы которой симметричны относительно главной диагонали. Функция произведения возвращает новую матрицу. Если вы просто перезапишете старую матрицу новой, произойдёт утечка памяти (не освобождены ресурсы, отводимые старой матрице).

5. Данна квадратная матрица. Если среди сумм элементов строк матрицы нет равных, то транспонировать матрицу.

- `bool isUnique(long long *a, int n)`
- `long long getSum(int *a, int n)`
- `void transposeIfMatrixHasNotEqualSumOfRows(matrix m)`

6. Даны две квадратные матрицы  $A$  и  $B$ . Определить, являются ли они взаимно обратными ( $A = B^{-1}$ ).<sup>99</sup>

- (a) `bool isMutuallyInverseMatrices(matrix m1, matrix m2)`

7. Данна прямоугольная матрица. Назовем псевдодиагональю множество элементов этой матрицы, лежащих на прямой, параллельной прямой, содержащей элементы  $a_{i,i}$ . Найти сумму максимальных элементов всех псевдодиагоналей данной матрицы. На рисунке ниже все псевдодиагонали выделены различными цветами:

3	2	5	4
1	3	6	3
3	2	1	2

Значение суммы для примера выше:

$$s = 3 + 2 + 6 + 5 + 4 = 20$$

Решение данной задачи может быть осуществлено с использованием дополнительной памяти из  $n + m - 1$  элементов<sup>100</sup>.

- (a) `int max(int a, int b)`  
 (b) `long long findSumOfMaxesOfPseudoDiagonal(matrix m)`

8. Данна прямоугольная матрица, все элементы которой различны. Найти минимальный элемент матрицы в выделенной области:

---

<sup>99</sup>Проверьте, что при работе функции не происходит утечек памяти.

<sup>100</sup>Попробуйте выполнить решение, при котором осуществляется проход по матрице слева направо и сверху вниз, а не по псевдодиагоналям. При этом однозначно потребуется вспомогательный массив.

				max	

10	7	<b>5</b>	6
3	11	8	9
4	1	12	2

<b>6</b>	8	9	2
7	12	3	4
10	11	5	1

Нижний элемент области – максимальный элемент матрицы<sup>101</sup>.

(a) `int getMinInArea(matrix m)`

9. Дано  $n$  точек в  $m$ -мерном пространстве. Упорядочить точки по неубыванию их расстояний до начала координат. Расстояние до начала координат находится как

$$d = \sqrt{x_1^2 + x_2^2 + x_3^2 + \dots + x_m^2}$$

- `float getDistance(int *a, int n)`
- `void insertionSortRowsMatrixByRowCriteriaF(matrix m, float (*criteria)(int *, int))`
- `void sortByDistances(matrix m)`

10. Определить количество классов эквивалентных строк данной прямоугольной матрицы. Строки считать эквивалентными, если равны суммы их элементов.

Указание: задача сводится к тому, чтобы подсчитать количество уникальных сумм строк матрицы. При проверке на уникальность предварительно отсортируйте массив быстрой сортировкой из стандартной библиотеки.

7	1
2	7
5	4
4	3
1	6
8	0

(a) `int cmp_long_long(const void *pa, const void *pb)`

---

<sup>101</sup>Проверять элементы матрицы на уникальность не требуется. Это гарантируется условием задачи.

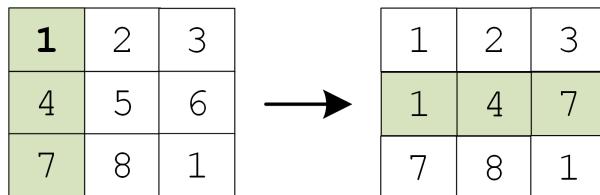
- (b) `int countNUnique(long long *a, int n)`<sup>102</sup>  
 (c) `int countEqClassesByRowsSum(matrix m)`

11. Данна матрица. Определить  $k$  – количество "особых" элементов матрицы, считая элемент "особым", если он больше суммы остальных элементов своего столбца.

3	5	5	4
2	3	6	<b>7</b>
<b>12</b>	2	1	2

- (a) `int getNSpecialElement(matrix m)`

12. Данна квадратная матрица. Заменить предпоследнюю строку матрицы первым из столбцов, в котором находится минимальный элемент матрицы.



<b>1</b>	2	3
4	5	6
7	8	1

1	2	3
<b>1</b>	<b>4</b>	<b>7</b>
7	8	1

- (a) `position getLeftMin(matrix m)`  
 (b) `void swapPenultimateRow(matrix m, int n)`

13. Дан массив матриц одного размера. Определить число матриц, строки которых упорядочены по неубыванию элементов (подходящие матрицы выделены зеленым):

7	1
1	1

1	6
2	2

5	4
2	3

1	3
7	9

- (a) `bool isNonDescendingSorted(int *a, int n)`  
 (b) `bool hasAllNonDescendingRows(matrix m)`  
 (c) `int countNonDescendingRowsMatrices(matrix *ms, int nMatrix)`

14. Дан массив целочисленных матриц. Вывести матрицы, имеющие наибольшее число нулевых строк<sup>103</sup>:

0	1
1	0
0	0

1	1
2	1
1	1

0	0
0	0
4	7

0	0
0	1
0	0

0	1
0	2
0	3

<sup>102</sup>Функции должны хорошо работать в изоляции от решаемой задачи. Проверьте, что она работает для массивов размера 0.

<sup>103</sup>Функцию вывода протестировать невозможно, если матрицы заранее не собираются в какой-нибудь массив. Достаточно приложить тесты только для `countZeroRows`.

- (a) `int countValues(const int *a, int n, int value)`
- (b) `int countZeroRows(matrix m)`
- (c) `void printMatrixWithMaxZeroRows(matrix *ms, int nMatrix)`

15. Дан массив целочисленных квадратных матриц. Вывести матрицы с наименьшей нормой. В качестве нормы матрицы взять максимум абсолютных величин ее элементов.
16. \*Дана матрица. Определить  $k$  – количество "особых" элементов данной матрицы, считая элемент "особым" если в строке слева от него находятся только меньшие элементы, а справа – только большие:

<b>2</b>	<b>3</b>	5	5	4
6	2	3	<b>8</b>	<b>12</b>
12	12	2	1	2

- (a) `int min2(int a, int b)`
  - (b) `int getNSpecialElement2(matrix m)`
17. \*Каждая строка данной матрицы представляет собой координаты вектора в пространстве. Определить, какой из этих векторов образует максимальный угол с данным вектором  $v$ .
- (a) `double getScalarProduct(int *a, int *b, int n)`
  - (b) `double getVectorLength(int *a, int n)`
  - (c) `double getCosine(int *a, int *b, int n)`
  - (d) `int getVectorIndexWithMaxAngle(matrix m, int *b)`
18. \*Дана целочисленная квадратная матрица, все элементы которой различны. Найти скалярное произведение строки, в которой находится наибольший элемент матрицы, на столбец с наименьшим элементом.
- (a) `long long getScalarProductRowAndCol(matrix m, int i, int j)`
  - (b) `long long getSpecialScalarProduct(matrix m, int n)`

## 18.14 Лабораторная работа «Работа со строками»

**Цель работы:** получение навыков работы со строками в стиле С.

**Содержание отчёта:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Исходный код `string_.h` / `string_.c` и решения задач. Фрагменты кода разделяйте по задачам и пропишите условия (допустимо комментариями).
- Ссылка на открытый репозиторий с решением.
- Скриншот с историей коммитов.

**Требования:**

- После решения задачи из второго блока должен выполняться коммит.
- Запрещено использование `string.h`.
- Запрещены операции обращения к элементу по индексу, а также замена:

---

`a[i] → *(a + i)`

---

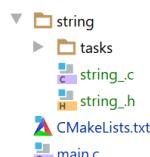
Однако в некоторых задачах допускается такая замена, но только для `i = 1`.

**Задания к лабораторной работе:**

Если вы желаете извлечь максимум из данной лабораторной работы, постарайтесь следовать требованиям в изложении. Вы можете отходить от них, если посчитаете нужным (например, вы можете при необходимости создавать вспомогательные файлы, закидывать решения в другие директории. Отхождения в пределах разумного поощряются).<sup>104</sup> Разработку организуйте через тестирование. Пишите тесты до написания функций.

Займёмся созданием библиотеки для работы со строками<sup>105</sup>:

1. Создайте файлы `string_.c` и `string_.h` для функций, которые будут использоваться при решении задач и папку `tasks`, в которой будут собираться решения задач:



<sup>104</sup>По правде говоря, это одна из моих нелюбимых лабораторных. Концентрат боли как при написании, так и при проверке. Я обещаю, что вам будет проще по мере продвижения дальше. Если какая-то задача вам не дается, пропустите её (не пропускайте только все). Постарайтесь решить максимальное количество. Должно получиться как с лабораторными по побитовым операциям: сначала не всё понятно, но по мере движения вы будете чувствовать большую уверенность в этом.

Заведомо предвкушаю возражения "А в моём языке X это делается в одну строчку. Да и вообще там есть тип `string`". Эта лабораторная должна улучшить ваши навыки работы с указателями. Постарался сделать всё возможное, чтобы это прошло максимально безболезненно для вас.

<sup>105</sup>Вы можете дополнить библиотеку своими функциями. Вероятно некоторые задачи не потребуют никаких функций из библиотеки.

И обновите CMakeList:

---

```

cmake_minimum_required(VERSION 3.16)
project(project C)

set(CMAKE_C_STANDARD 11)

# определение точки входа. Будет запущен файл main.c.
# указывается произвольная метка, в данном случае - project
add_executable(project main.c)

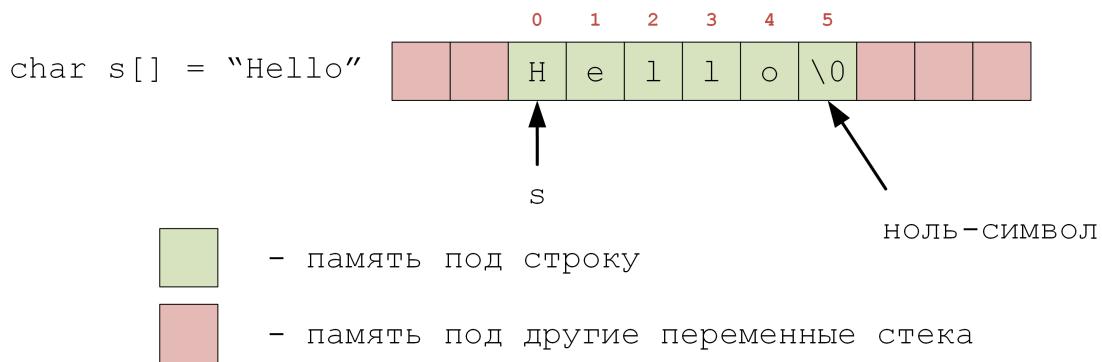
# создаём библиотеку
add_library(str string/string_.h string/string_.c
            # string/tasks/digitToStartTransform.h
            # string/tasks/reverseWords.h
            # string/tasks/replaceDigitsBySpaces.h
            # ...
            # < файл с решением задачи >
            # ...
            # string/tasks/hasPairOfWordsWithEqualLetterSet.h
            # string/tasks/printWordsNonEqualLastWord.h
        )

# описываем, что для запуска project потребуется сборка tasks
target_link_libraries(project str)

```

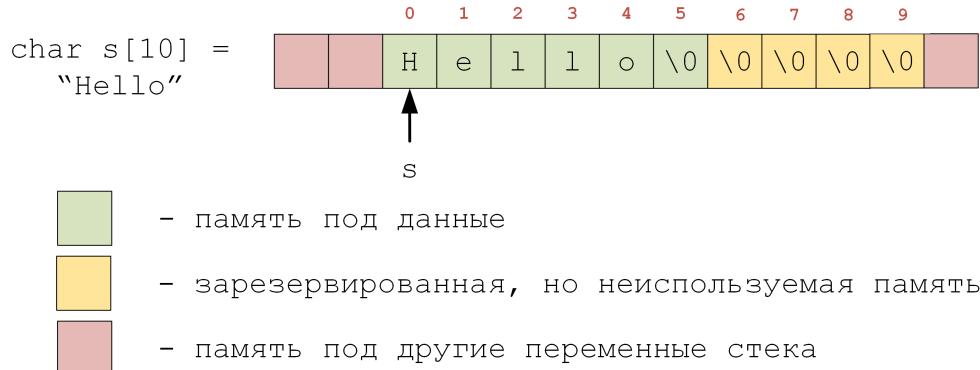
---

2. Начнём с чего-нибудь простого, например, с поиска длины строки. Когда вы объявляете строку с инициализацией, она размещается в стеке на ленте памяти:



Она занимает на один символ больше, так как требуется хранение ноль-символа. Именно по нему мы можем понять, что строка закончилась. Говорят, что длина строки равняется пяти, но на физическом уровне потребуется для представления 6 байт памяти.

Можно встретить и такое объявление массива типа `char`:



Оно отличается тем, что если строка, которой мы инициализируем переменную `s` более короткая, в свободные ячейки будут записаны ноль-символы, и тот фрагмент памяти может быть потенциально использован при решении задач.

3. Реализуем функцию `strlen`. Она возвращает количество символов в строке (не считая ноль-символа). Например:

---

```
// требуем #include <string.h>

char *s1 = "Hi";
char s2[10] = "\tHello\t";

printf("%u\n", strlen(s1)); // 2
printf("%u\n", strlen(s2)); // 7
```

---

Наша задача заключается в том, чтобы определить, сколько символов имеется от начала строки до первого ноль-символа. Опишу три способа сделать это. Пойдём от худшего к лучшему. Первый вариант – используем индексы:

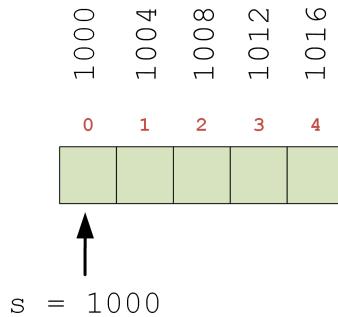
---

```
size_t strlen1(char *s) {
    int i = 0;
    while (s[i] != '\0')
        i++;

    return i;
}
```

---

Недостаток: наличие операции обращения к элементу по индексу. Компилятор, конечно, может оптимизировать данную функцию. Но давайте разберёмся, что происходит, когда используется операция обращения к элементу по индексу. Объявляя переменную `s`, которая является указателем, в ней записывается адрес нулевого элемента. Предположим, что в нашем примере он равен 1000:



При попытке обратиться к  $i$ -ому элементу массива требуется вычислить адрес нужной ячейки. Адрес вычисляется так:

$$\&s[i] = \&s[0] + \text{sizeof}(basetype) * i$$

Например, если работа идёт с массивом `int` и индекс  $i$  равен трём, тогда адрес `s[3]`:

$$\&s[3] = \&s[0] + \text{sizeof}(int) * i = 1000 + 4 * 3 = 1012$$

Если бы нумерация элементов в массиве велась с единицы, тогда потребовалось бы дополнительное вычитание из  $i$  значения 1:

$$\&s[4] = \&s[1] + \text{sizeof}(int) * (i - 1) = 1000 + 4 * (4 - 1) = 1012$$

Избавиться от обращений к элементам по индексу можно так:

---

```
size_t strlen2(char *s) {
    int i = 0;
    while (*s != '\0') {
        i++;
        s++;
    }

    return i;
}
```

---

однако имеется избыточное количество сложений с  $i$ . В окончательном варианте вам стоит использовать арифметику указателей:

---

```
size_t strlen_(const char *begin) {
    char *end = begin;
    while (*end != '\0')
        end++;
    return end - begin;
}
```

---

Как это работает? Адрес  $i$ -го элемента массива может быть найден так:

$$\&s[i] = s + i$$

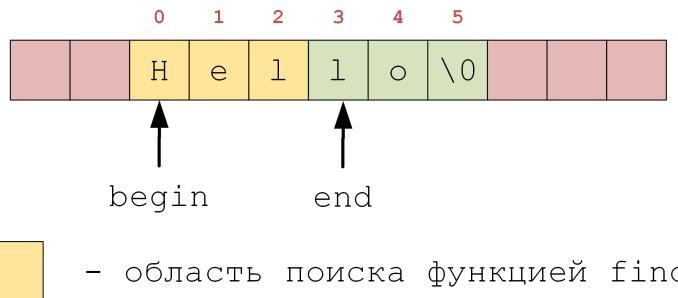
Выразим  $i$  из предыдущего уравнения:

$$i = \&s[i] - s$$

В уравнении выше происходит вычитание двух адресов. Но неочевидным является момент, что разность адресов и даст некоторый индекс. Но если на  $s[i]$  располагается символ конца строки, то  $i$  – длина строки. Следовательно, через вычитание можно найти количество элементов, которое располагается между парой указателей. Переместите в библиотеку `string_`.

4. Реализуем функции поиска:

- (a) `char* find(char *begin, char *end, int ch)` – возвращает указатель на первый элемент с кодом `ch`, расположенным на ленте памяти между адресами `begin` и `end` не включая `end`. Если символ не найден, возвращается значение `end`. Функция реализуется аналогично `strlen_`, только возвращает указатель и имеет чуть более сложное условие возобновления цикла<sup>106</sup>:




---

```
char* find(char *begin, char *end, int ch) {
    while (begin != end && *begin != ch)
        begin++;

    return begin;
}
```

---

- (b) `char* findNonSpace(char *begin)` – возвращает указатель на первый символ, отличный от пробельных<sup>107</sup>, расположенный на ленте памяти, начиная с `begin` и заканчивая ноль-символом. Если символ не найден, возвращается адрес первого ноль-символа<sup>108</sup>.

<sup>106</sup>Если бы данная задача решалась на массиве с использованием индексов, сначала проверялось значение индекса, и только потом значение по этому индексу. Аналогично и здесь: сначала проверяется не достиг ли указатель конца, и только потом значение по указателю. Несмотря на то, что адреса можно сравнивать и в условии цикла написать `begin < end`, отдайте предпочтение варианту `begin != end`.

<sup>107</sup>Не только пробел относится к пробельным символам. Используйте функцию `isspace` из `<ctype.h>`.

<sup>108</sup>У вас может появиться вопрос, почему функция `findNonSpace` не имеет параметр `end`. Ответ заключается в том, чтобы знать `end`, надо знать размер строки, а это лишний проход. В силу того, что оптимизация по времени в данной лабораторной имеет крайне высокий приоритет, было принято такое решение по интерфейсам. Можно было написать какой-нибудь `findIf` и передавать функцию-предикат `isspace` и прочие, но наличие отдельных функций с именами `findNonSpace` и `findSpace` упрощает чтение кода в других функциях.

- (c) `char* findSpace(char *begin)` – возвращает указатель на первый пробельный символ, расположенный на ленте памяти начиная с адреса `begin` или на первый ноль-символ.
- (d) `char* findNonSpaceReverse(char *rbegin, const char *rend)` – возвращает указатель на первый справа символ, отличный от пробельных, расположенный на ленте памяти, начиная с `rbegin` (последний символ строки, за которым следует ноль-символ) и заканчивая `rend` (адрес символа перед началом строки). Если символ не найден, возвращается адрес `rend`.<sup>109</sup>.
- (e) `char* findSpaceReverse(char *rbegin, const char *rend)` – возвращает указатель на первый пробельный символ справа, расположенный на ленте памяти, начиная с `rbegin` и заканчивая `rend`. Если символ не найден, возвращается адрес `rend`. Пример работы на рисунке 18.20:

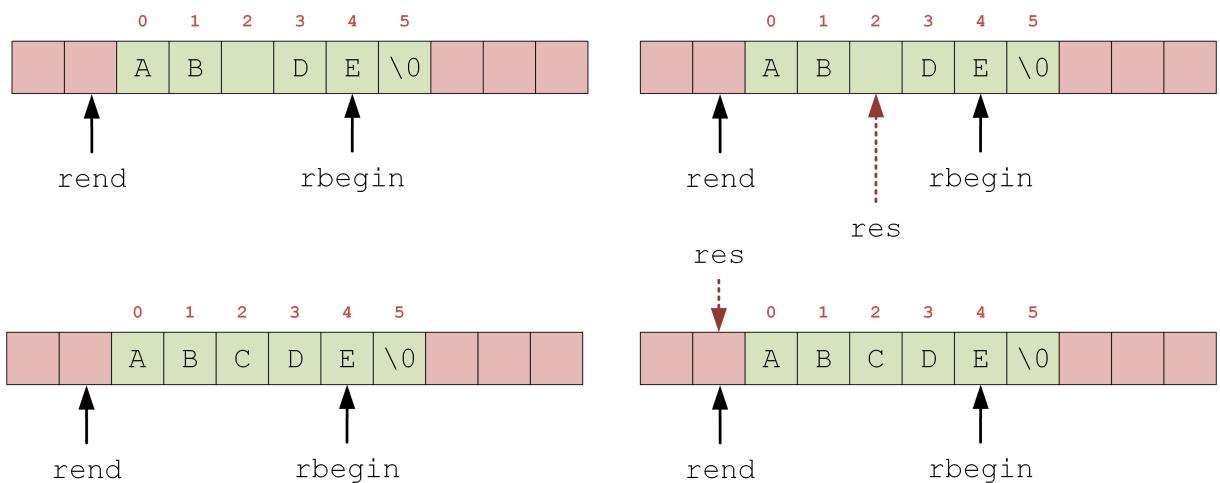


Рис. 18.20: Работа функции `findSpaceReverse` для случаев, когда пробел имелся в исходной строке и нет

5. Опишем функцию, которая часто используется для проверки строк на равенство. Начинающие программисты знают, что для этого используется функция `strcmp`, однако она работает не так, как они ожидают на первый взгляд:

<sup>109</sup>В данном случае только по `rend` мы можем понять, где располагается начало строки



Документация гласит следующее:

---

```
int strcmp(const char *lhs, const char *rhs)
```

---

Функция возвращает отрицательное значение, если `lhs`<sup>110</sup> располагается до `rhs` в лексикографическом порядке (как в словаре), значение 0, если `lhs` и `rhs` равны, иначе – положительное значение.

Что же в назначении подразумевается под положительным и отрицательным значениями? Разница символов, на котором остановилось сравнение двух строк. Попробуйте выполнить реализацию функции самостоятельно. Тело функции без пустых строк занимает 3 строчки кода.

## 6. Функции для копирования:

- `char* copy(const char *beginSource, const char *endSource, char *beginDestination)` – записывает по адресу `beginDestination` фрагмент памяти, начиная с адреса `beginSource` до `endSource`<sup>111</sup>. Возвращает указатель на следующий свободный фрагмент памяти в `destination`<sup>112</sup>:

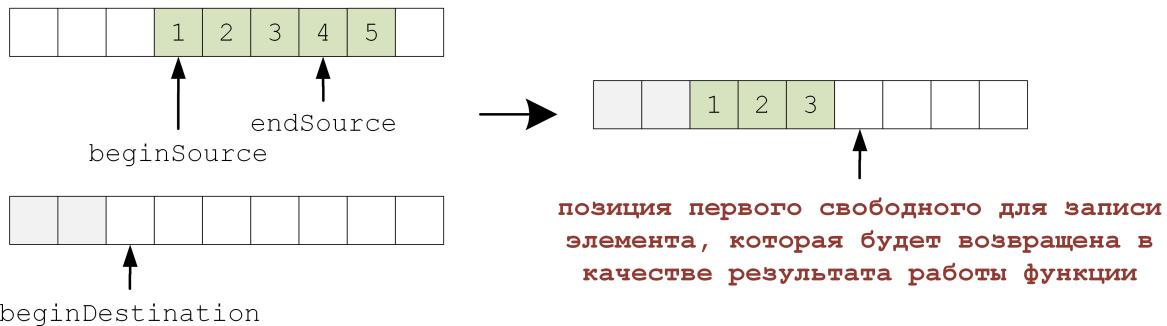
---

<sup>110</sup> `lhs` – *left hand size* – аргумент левой руки, `rhs` – *right hand size* – аргумент правой руки. В стандартных библиотеках часто используется такая нотация для функций из двух однородных аргументов.

<sup>111</sup> При реализации используйте `memcpuy` из `<memory.h>`

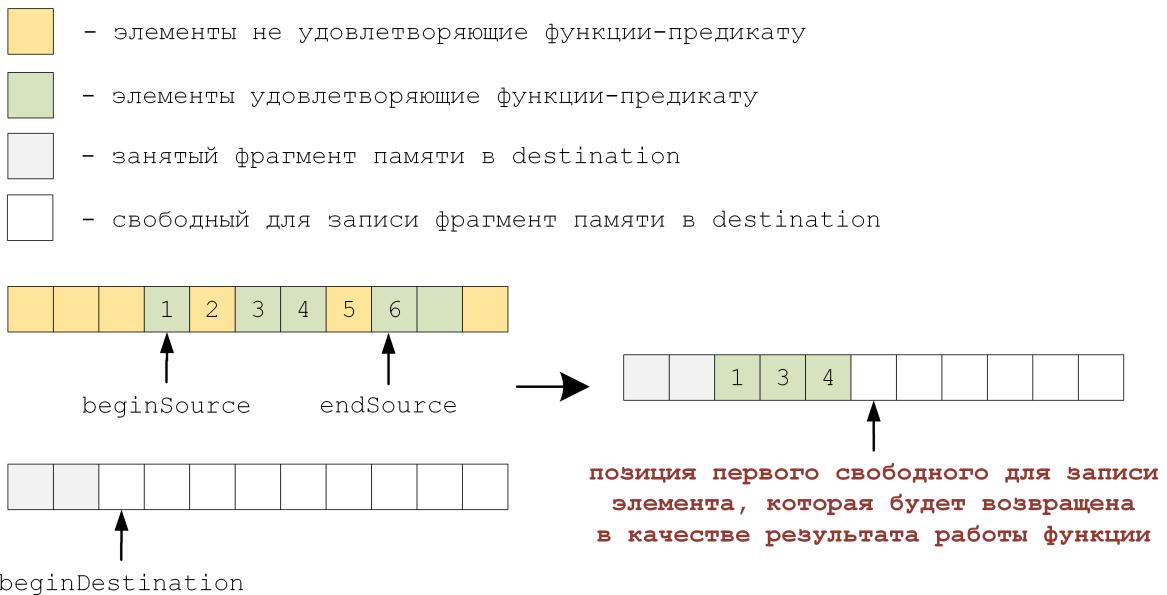
<sup>112</sup> Приём с возвратом следующей свободной для записи позиции часто используется при решении практических задач. Заголовки функций подобраны таким образом, чтобы иметь наибольшее количество параллелей со стандартной библиотекой `<algorithm>` C++.

-  - копируемый фрагмент
-  - занятый фрагмент памяти в destination
-  - свободный для записи фрагмент памяти в destination



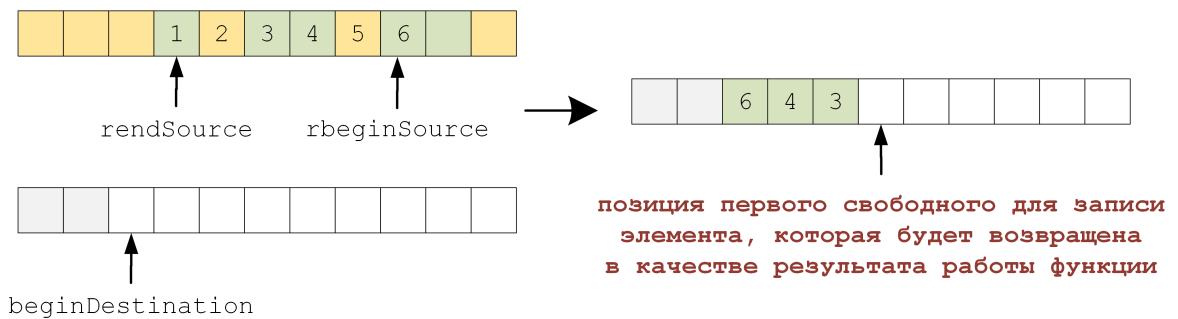
По окончанию работы функции ноль-символ не записывается.

- `char* copyIf(char *beginSource, const char *endSource, char *beginDestination, int (*f)(int))` – записывает по адресу `beginDestination` элементы из фрагмента памяти начиная с `beginSource` заканчивая `endSource`, удовлетворяющие функции-предикату `f`. Функция возвращает указатель на следующий свободный для записи фрагмент памяти.



По окончанию работы функции ноль-символ не записывается.

- `char* copyIfReverse(char *rbeginSource, const char *rendSource, char *beginDestination, int (*f)(int))` – записывает по адресу `beginDestination` элементы из фрагмента памяти начиная с `rbeginSource` заканчивая `rendSource`, удовлетворяющие функции-предикату `f`. Функция возвращает значение `beginDestination` по окончанию работы функции.

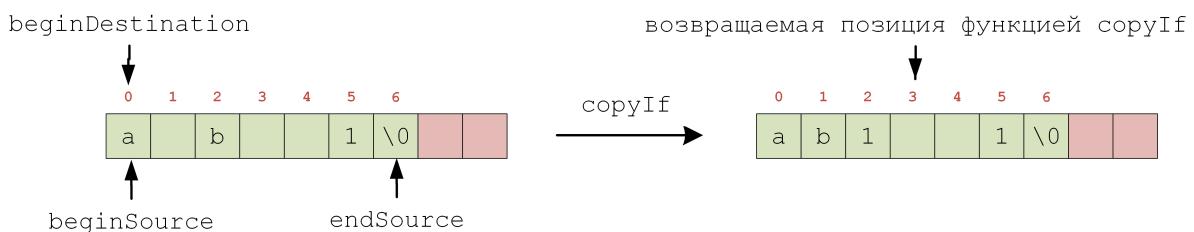


По окончанию работы функции ноль-символ не записывается.

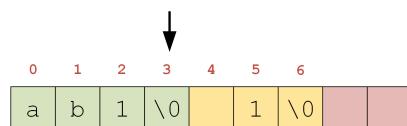
## Список задач:

1. Выполним разбор задачи: удалить из строки все пробельные символы. Решение задачи сводится к вызову `copyIf`:

1. Выполняем копирование необходимых символов.  
Функция `copyIf` вернёт позицию первого символа,  
на котором закончилась запись:



2. Записываем ноль-символ, чтобы обозначить конец строки:



- память под данные
- зарезервированная, но неиспользуемая память
- память под другие переменные стека

Функция для решения задачи:

```
void removeNonLetters(char *s) {
    char *endSource = getEndOfString(s);
    char *destination = copyIf(s, endSource, s, isgraph);
    *destination = '\0';
}
```

Важно не забыть поставить ноль-символ. Иначе при выводе строки выведутся символы: "ab1 1".

Перейдём к тестированию. Было бы неплохо иметь такую функцию тестирования, которая не 'ложила' бы наше приложение как `assert`, но давала бы информацию о том, а где именно произошла ошибка. Опишем функцию `assertString`:

```
void assertString(const char *expected, char *got,
                  char const *fileName, char const *funcName,
                  int line) {
    if (strcmp_(expected, got)) {
        fprintf(stderr, "File %s\n", fileName);
        fprintf(stderr, "%s - failed on line %d\n", funcName, line);
        fprintf(stderr, "Expected: \"%s\"\n", expected);
        fprintf(stderr, "Got: \"%s\"\n", got);
    } else
```

```

        fprintf(stderr, "%s - OK\n", funcName);
}

```

Вы уже тоже чувствуете удобство этого интерфейса. Теперь можно писать:

```

assertString(s1, s2, "digitToStartTransform.h",
             "test_digitToStartTransform_oneWord", 30);

```

```

File C:\Users\John\CLionProjects\course\string\tasks\digitToStartTransform.h
test_digitToStartTransform_oneWord - failed on line 30
Expected: "321Hi"
Got: "321Hi "

```

Самое приятное в этой истории, что если поменяется исходник, название функций, строка, из которой происходит вызов, придётся перелопатить все тесты. Вы должны хорошо понимать, что сопровождение тестов – это тоже требует затрат. И предложение выше одно из худших, которое вы могли прочитать. Но можно его немного доработать. Будем руководствоваться желаемым интерфейсом: если кто-то хочет сравнить строки, он просто желает написать `assertString(s1, s2)`. Но при этом хотелось бы сохранить функционал от прошлого варианта. На помощь приходят макрофункции:

```

#define ASSERT_STRING(expected, got) assertString(expected, got, \
                                                 __FILE__, __FUNCTION__, __LINE__)

```

Что нам это даёт? Предположим, что вы написали такой тест:

```

void test_digitToStartTransform_oneWord() {
    char s[] = "Hi123 ";
    digitToStartTransform(s);
    ASSERT_STRING("321Hi", s);
}

```

На этапе препроцессирования будет выполнена макроподстановка:

```

void test_digitToStartTransform_oneWord() {
    char s[] = "Hi123 ";
    digitToStartTransform(s);
    assertString("321Hi", s,
                "C:\Users\...\digitToStartTransform",
                "test_digitToStartTransform_oneWord", 4);
}

```

Препроцессор вместо `__FILE__` подставит полный путь к файлу с его именем в виде строки, `__FUNCTION__` изменится на строковый литерал с именем функции, вместо `__LINE__` подставится целое число с номером строки.

Мы решили проблему с некрасивым вызовом.

2. Решите любую из задач:<sup>113</sup>

- Преобразовать строку, оставляя только один символ в каждой последовательности подряд идущих одинаковых символов (`void removeAdjacentEqualLetters(char *s)`).
- Сократить количество пробелов между словами данного предложения до одного (`void removeExtraSpaces(char *s)`).

3. Выполним разбор задачи: преобразовать строку таким образом, чтобы цифры каждого слова были перенесены в начало слова и изменить порядок следования цифр в слове на обратный, а буквы – в конец слова, без изменения порядка следования.

Потребуется функция для считывания слов. Спроектируем её с таким интерфейсом, чтобы позволить следующий приём:

---

```
char *beginSearch = beginString;

WordDescriptor word; // считываемое слово
while (getWord(beginSearch, &word)) {
    // обработка слова
    // ...
    beginSearch = word.end;
}
```

---

Функция `getWord` вернёт значение 0, если слово не было считано, в противном случае будет возвращено значение 1 и в переменную `word` типа `WordDescriptor` будут записаны позиции начала слова, и первого символа после конца слова:

---

```
typedef struct WordDescriptor {
    char *begin; // позиция начала слова
    char *end;   // позиция первого символа, после последнего символа
    ↪ слова
} WordDescriptor;
```

---

Предлагаемый заголовок:

---

```
int getWord(char *beginSearch, WordDescriptor *word);
```

---

Продолжим рассуждения. Что значит 'найти слово'? И что является словом? Под словом будем считать произвольную последовательность непробельных символов. Примеры: "a", "hello", "123". Мы хотели бы научиться считывать слова с произвольной позиции в строке (рисунок 18.21).

<sup>113</sup>При написании тестов, старайтесь постепенно усложнять тестируемые случаи. Например, если вы реализуете функцию для удаления повторяющихся символов, порядок проверки мог бы быть таким: "", "a", "aa", "aaa", "aabbb", " aa bbb c" и так далее.

Классификация задач заняла львиную долю создания работы.

При тестировании функций на строки обязательно проверяйте случай с пустой строкой.

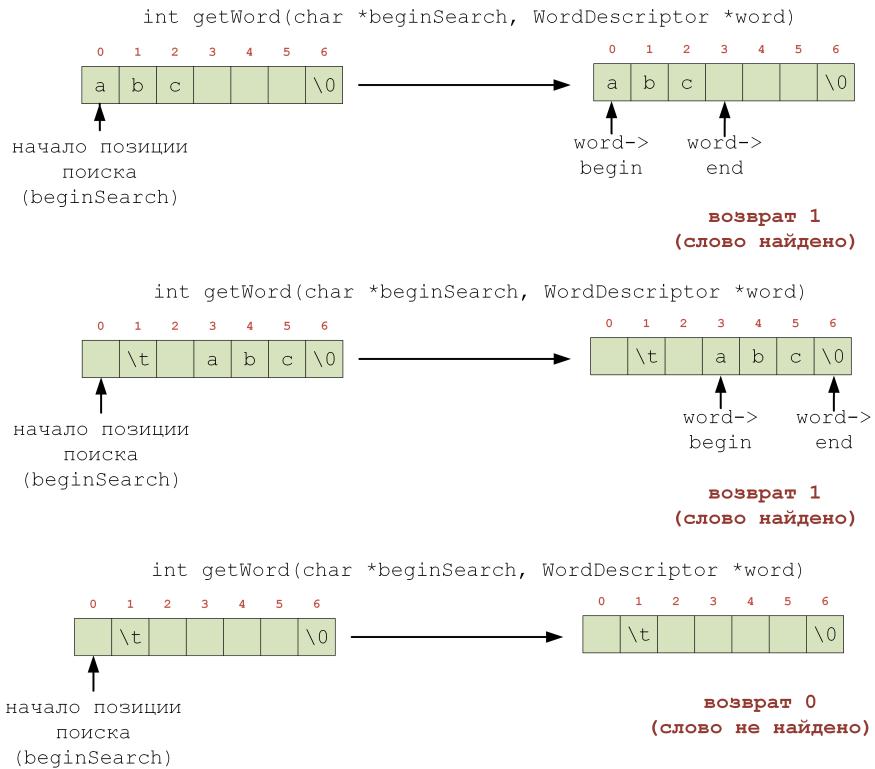
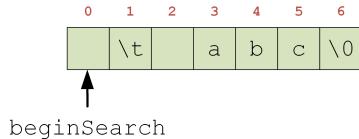


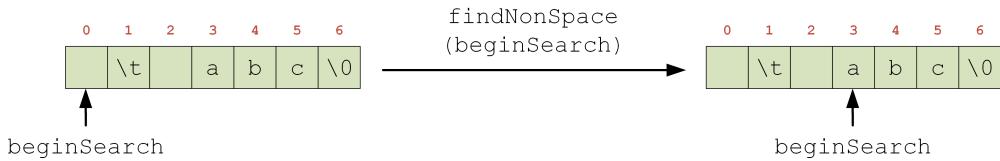
Рис. 18.21: Результат работы функции `getWord`

Если для вас очевидна мысль, что `word->begin` позиция первого символа, который не является пробельным, а `word->end` позиция первого пробельного символа начиная с позиции `beginWord` или ноль-символ, то всё, что происходит дальше, будет довольно просто понять:

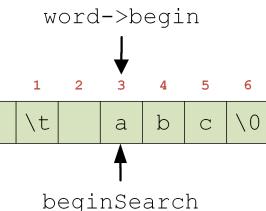
Пусть дана строка:



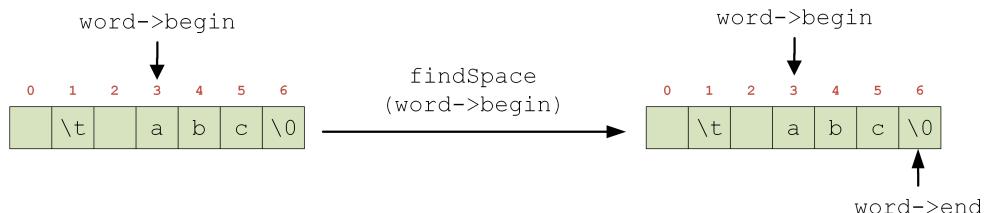
1. Выполняем поиск первого непробельного символа:



2. Сохраняем позицию начала слова (если, конечно не встретили '\0'):



3. Выполняем поиск конца слова, начиная с beginWord:



Если выразить кодом:

---

```
int getWord(char *beginSearch, WordDescriptor *word) {
    word->begin = findNonSpace(beginSearch);
    if (*word->begin == '\0')
        return 0;

    word->end = findSpace(word->begin);

    return 1;
}
```

---

Если функция не найдёт слово, она вернёт значение 0. В функции, которая обрабатывает строки важно не забыть переместить позицию начала поиска:

---

```
char *beginSearch = beginString;
WordDescriptor word;
while (getWord(beginSearch, &word)) {
    // обработка слова
```

---

```
beginSearch = word.end;
}
```

---

Ну всё. Теперь у нас есть позиции слов. Осталось выполнить преобразование каждого слова. По условию задачи: требуется преобразовать строку таким образом, чтобы цифры каждого слова были перенесены в начало слова и изменить порядок следования цифр в слове на обратный. Опишем функцию обработки одного слова. Схематично работа функции представлена на рисунке 18.22.

```
void digitToStart(WordDescriptor word) {
    char *endStringBuffer = copy(word.begin, word.end,
                                  _stringBuffer);
    char *recPosition = copyIfReverse(endStringBuffer - 1,
                                       _stringBuffer - 1,
                                       word.begin, isdigit);
    copyIf(_stringBuffer, endStringBuffer, recPosition, isalpha);
}
```

---

В `string_.h` объявитите следующую глобальную переменную под буфер:

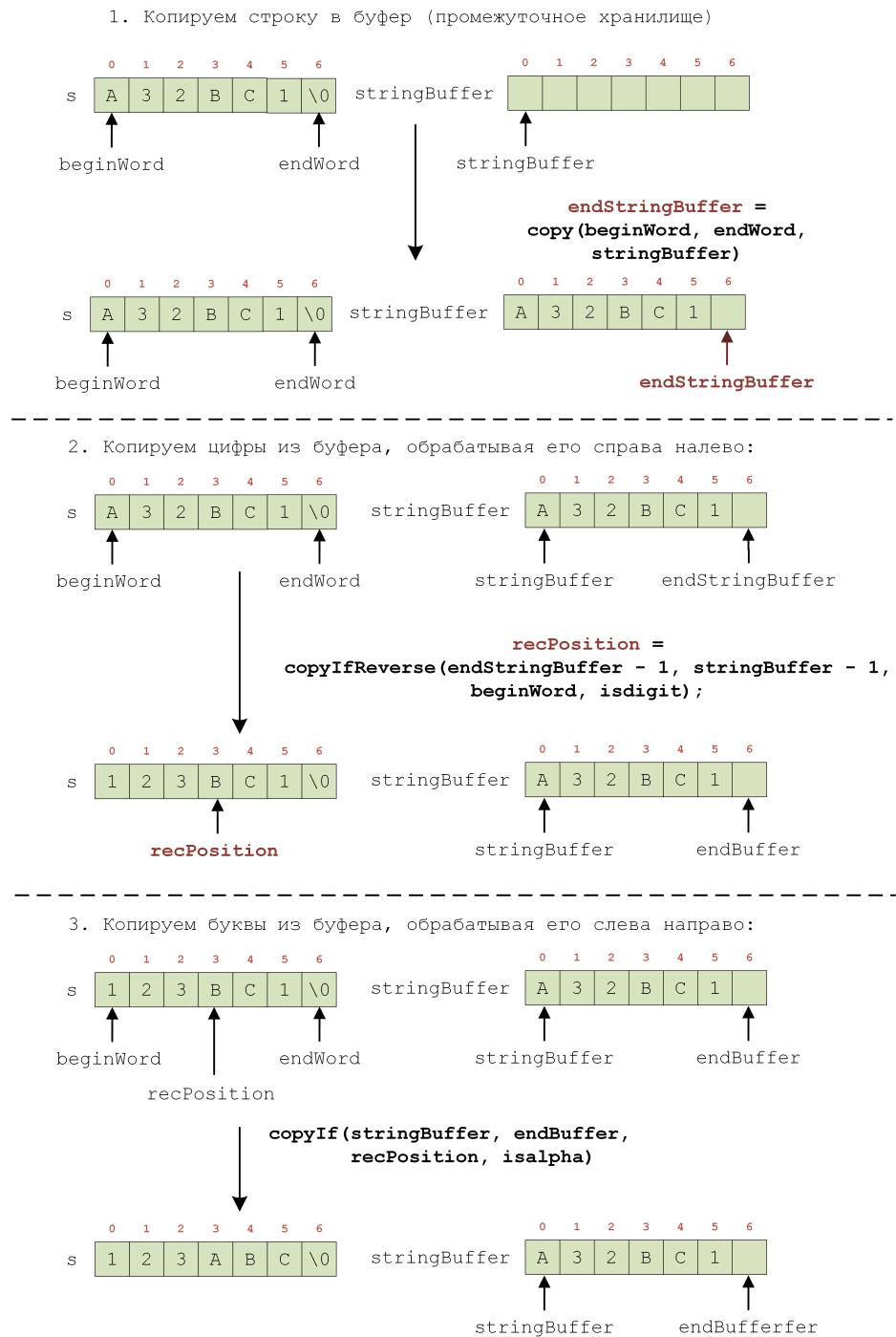
```
char _stringBuffer[MAX_STRING_SIZE + 1];
```

---

Для закрепления материала решите любую задачу из предложенных ниже:

- Преобразовать строку таким образом, чтобы цифры каждого слова были перенесены в конец слова без изменения порядка следования их в слове, а буквы – в начало.
- Преобразовать строку таким образом, чтобы цифры каждого слова были перенесены в конец слова, и изменить порядок следования цифр в слове на обратный, буквы перенести в начало слова.
- Преобразовать строку таким образом, чтобы цифры каждого слова были перенесены в начало слова без изменения порядка следования их в слове, буквы перенести в конец слова.
- Преобразовать строку, обратив каждое слово этой строки. Буфер не использовать.

Дополнительно к этому реализуйте функцию  
`bool getWordReverse(char *rbegin, char *rend, WordDescriptor *word)`  
для считывания слова с конца строки. Механизм работы должен быть схож с `getWord`.

Рис. 18.22: Работа функции *digitToStart*

4. Преобразовать строку, заменяя каждую цифру соответствующим ей числом пробелов. Имеются такие задачи, при которых размер строки может увеличиться. Мы пока что будем исходить из предположения, что размер итоговой строки не превысит некоторого `MAX_STRING_SIZE` определенного в `string_.h`:

---

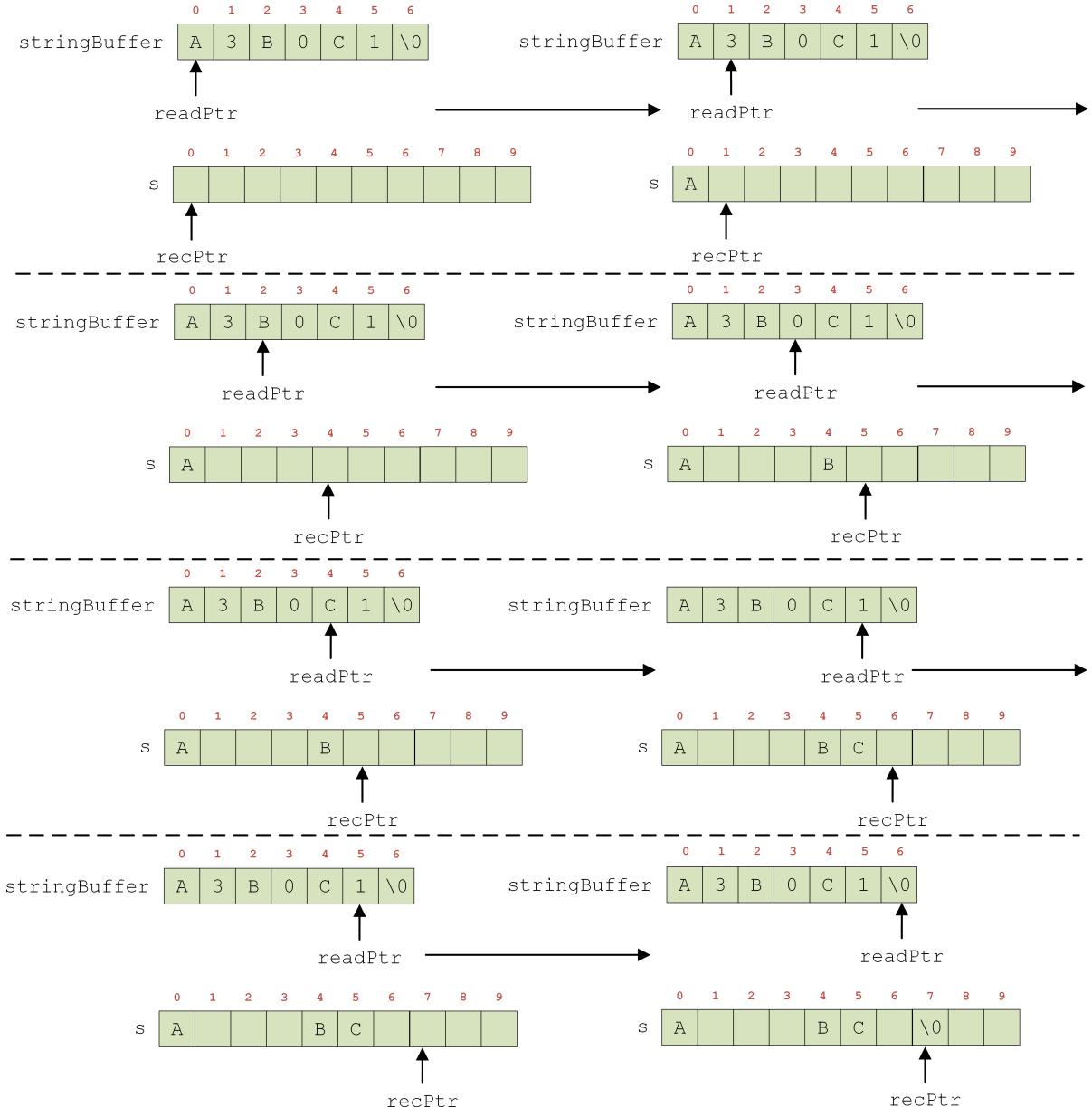
```

#define MAX_STRING_SIZE 100
#define MAX_N_WORDS_IN_STRING 100
#define MAX_WORD_SIZE 20
  
```

---

Решение организуйте так: скопируйте исходную строку в буфер, а потом полу-

чайте ответ на строке `s`:



5. Заменить все вхождения слова  $w_1$  на слово  $w_2$ .

Порассуждаем о задаче. Если слово  $w_1$  будет более длинным, чем  $w_2$ , тогда все преобразования можно сделать на исходной строке. Однако если это не так, возможно (если встретим  $w_1$ ), придется задействовать буфер. Для упрощения задачи будем считать, что слово  $w_1$  всегда присутствует в строке, следовательно, копирование в буфер будет оправдано. Напишите функцию, которая решает данную задачу. Я помогу лишь с её началом:

```
void replace(char *source, char *w1, char *w2) {
    size_t w1Size = strlen_(w1);
    size_t w2Size = strlen_(w2);
    WordDescriptor word1 = {w1, w1 + w1Size};
    WordDescriptor word2 = {w2, w2 + w2Size};

    char *readPtr, *recPtr;
```

```

if (w1Size >= w2Size) {
    readPtr = source;
    recPtr = source;
} else {
    copy(source, getEndOfString(source), _stringBuffer);
    readPtr = _stringBuffer;
    recPtr = source;
}

// продолжение функции
}

```

---

Указание: выполните копирование строки в буфер и записывайте результат в исходную строку.

6. Определить, упорядочены ли лексикографически<sup>114</sup> слова данного предложения.

Задачи, которые работают с обычными последовательностями и строками много общего. Подумайте, как бы вы решали задачу о проверки последовательности на неубывание. Примерно схожая структура решения ожидается и здесь:

- (a) Пытаемся считать первое слово. Если слово найдено, продолжаем работу функции, иначе – возвращаем значение 'истина'.
- (b) Считываем следующее слово. Если оно не нарушает требуемую упорядоченность, то присвоить прошлому слову текущее и перейти дальше, иначе вернуть 'ложь'. Второй пункт выполняется до тех пор, пока не будет обработана вся строка или найдена последовательность слов, нарушающая лексикографический порядок.

Предварительно реализуйте функцию `int areWordsEqual(WordDescriptor w1, WordDescriptor w2);` для сравнения двух слов.

7. Вывести слова данной строки в обратном порядке по одному в строке экрана.

Указание: заведём вспомогательную структуру, которая будет хранить начало и конец каждого слова исходной строки:

```

typedef struct BagOfWords {
    WordDescriptor words[MAX_N_WORDS_IN_STRING];
    size_t size;
} BagOfWords;

```

---

Опишите функцию `void getBagOfWords(BagOfWords *bag, char *s)`, которая получает позиции начала и конца каждого слова строки:

<sup>114</sup>Если простыми словами, проверьте, соответствует ли порядок слов с порядком их следования в толковом словаре.

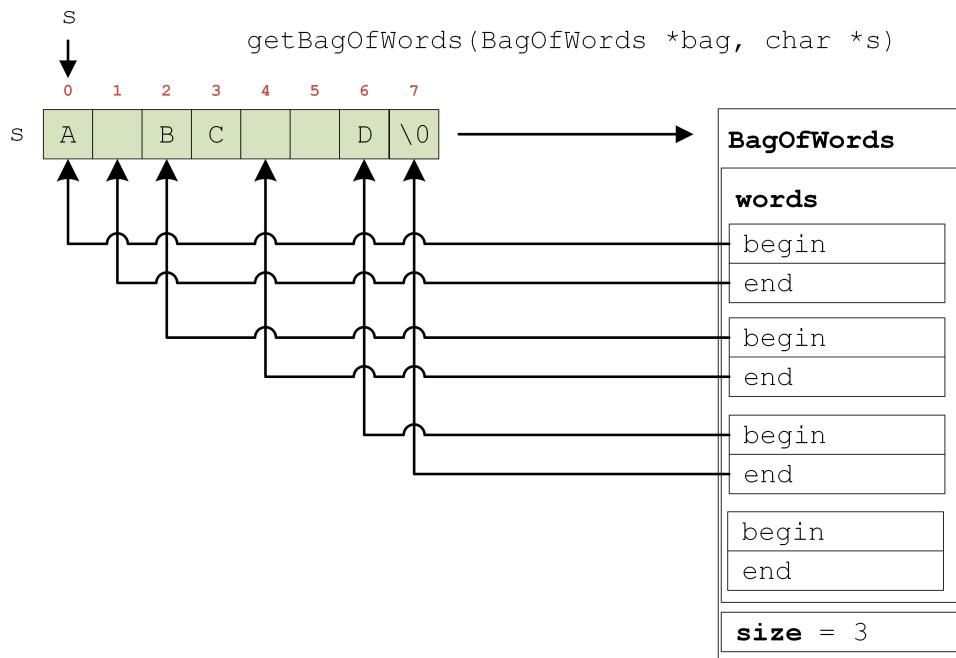


Рис. 18.23: В условиях данной задачи программа должна вывести слова "D", "BC", "A".

Чтобы не создавать переменные `BagOfWords` внутри функций (и экономить на памяти), заведём в `string_.h` две глобальные переменные:

```
BagOfWords _bag;  
BagOfWords _bag2;
```

8. В данной строке соседние слова разделены запятыми. Определить количество слов-палиндромов.

Указание: попробуйте решить задачу без использования BagOfWords.

9. Даны две строки. Получить строку, в которой чередуются слова первой и второй строки. Если в одной из строк число слов больше, чем в другой, то оставшиеся слова этой строки должны быть дописаны в строку-результат. В качестве разделителя между словами используйте пробел.

Указание: постарайтесь обойтись одним циклом. Вам может быть полезен оператор запятая в таком исполнении:

```
WordDescriptor word1, word2;
bool isW1Found, isW2Found;
char *beginSearch1 = s1, *beginSearch2 = s2;
while ((isW1Found = getWord(beginSearch1, &word1)),
       (isW2Found = getWord(beginSearch2, &word2)),
       isW1Found || isW2Found) {
    //...
}
```

10. Преобразовать строку, изменив порядок следования слов в строке на обратный.

Указание: скопируйте строчку на буфер, считывайте по одному слову с конца и записывайте результат на исходной строке.

11. Вывести слово данной строки, предшествующее первому из слов, содержащих букву "а". Регистр значения не имеет.

Указание: можно описать функцию `void printWordBeforeFirstWordWithA(char *s)`. Возможны следующие случаи:

- В строке нет слов.
- В строке нет слов с 'а'.
- Первое слово с 'а' является первым в строке.
- Имеется слово перед словом с 'а'.

Каждому исходу будет соответствовать сообщение. Но как протестировать функцию, выводящую сообщение? В такой вариации никак. Можно несколько улучшить подход. Первая приходящая идея состоит в том, чтобы возвращать некоторое числовое значение от 0 до 3 (по одному значению на случай). Но это крайне неинформативно. Значения 0 и 3 будут восприниматься как магические константы. Однако лучше создать группу именованных констант при помощи `enum`:

---

```
typedef enum WordBeforeFirstWordWithAReturnCode {
    FIRST_WORD_WITH_A,
    NOT_FOUND_A_WORD_WITH_A,
    WORD_FOUND,
    EMPTY_STRING
} WordBeforeFirstWordWithAReturnCode;

// заголовок функции
WordBeforeFirstWordWithAReturnCode getWordBeforeFirstWordWithA(
    char *s, WordDescriptor *w)
```

---

Вместо возврата непонятных кодов можно будет написать:

---

```
return FIRST_WORD_WITH_A;
```

---

А тесты могут быть оформлены так:

---

```
void testAll_getWordBeforeFirstWordWithA() {
    WordDescriptor word;

    char s1[] = "";
    assert(
        getWordBeforeFirstWordWithA(s1, &word)
        == EMPTY_STRING
    );

    char s2[] = "ABC";
```

---

```

    assert(
        getWordBeforeFirstWordWithA(s2, &word)
        == FIRST_WORD_WITH_A
    );

    char s3[] = "BC A";
    assert(
        getWordBeforeFirstWordWithA(s3, &word)
        == WORD_FOUND
    );
    char got[MAX_WORD_SIZE];
    copy(beginWord, endWord, got);
    got[endWord - beginWord] = '\0';
    ASSERT_STRING("BC", got);

    char s4[] = "B Q WE YR OW IUWR";
    assert(getWordBeforeFirstWordWithA(s4, &word) ==
        NOT_FOUND_A_WORD_WITH_A);
}

```

---

12. Даны две строки. Определить последнее из слов первой строки, которое есть во второй строке.

Указание: используйте `BagOfWords`. Для тестирования было бы крайне удобно иметь функцию, переводящую `WordDescriptor` в `char*`. Реализуйте для этого функцию `void wordDescriptorToString(WordDescriptor word, char *destination)`.

Тогда фрагмент теста будет выглядеть так:

---

```

WordDescriptor word =
    lastWordInFirstStringInSecondString(s1[i], s2[i]);
wordDescriptorToString(word, string);
ASSERT_STRING(expected, string);

```

---

13. Определить, есть ли в данной строке одинаковые слова.

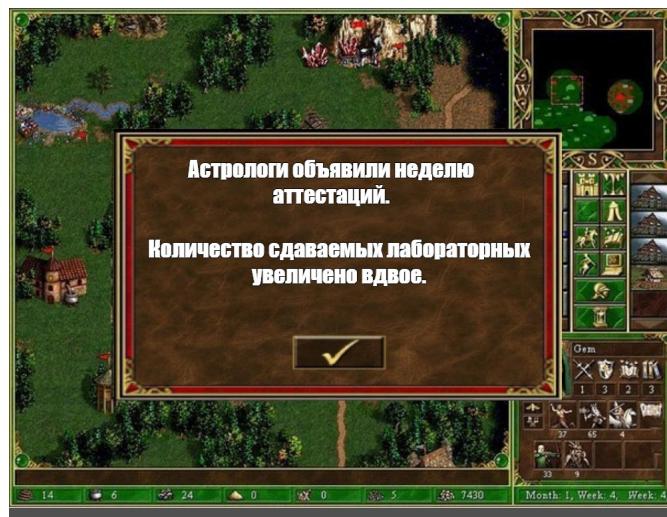
14. Определить, есть ли в данной строке пара слов, составленных из одинакового набора букв.

Будем исходить из следующих предположений: исходная строка может быть скопирована в буфер. Каким-то образом хочется уменьшить количество проводимых операций. Существенным фактором в выборе подхода будет вероятностный момент, а именно насколько вероятно, что такая пара имеется. Предположим, что такая вероятность мала.

Тогда было бы логично организовать следующую цепочку преобразований:

- Скопируйте строку в буфер.
- Найдите все слова строки.
- Отсортируйте буквы каждого слова.

- Проверьте, имеются ли в буфере два одинаковых слова.
15. Получить строку из слов данной строки, которые отличны от последнего слова.
16. Даны две строки  $s_1$  и  $s_2$ . Пусть  $w$  – первое из слов строки  $s_1$ , которое есть и в строке  $s_2$ . Найти слово, предшествующее первому вхождению  $w$  в  $s_1$ .
17. Задача на удаление слов из строки (любую одну на выбор):
- Удалить из данной строки слова-палиндромы.
  - Удалить из данной строки слова, содержащие заданную последовательность символов.
  - Удалить из строки слова, содержащие повторяющиеся символы.
18. Даны две строки. Пусть  $n_1$  – число слов в первой строке, а  $n_2$  – во второй. Требуется дополнить строку, содержащую меньшее количество слов, последними словами строки, в которой содержится большее количество слов.
19. Определить, входит ли в данную строку каждая буква данного слова.  
Указание: алгоритм должен иметь линейную сложность.



## 18.15 Лабораторная работа «Оценка сложности алгоритмов сортировки по времени»

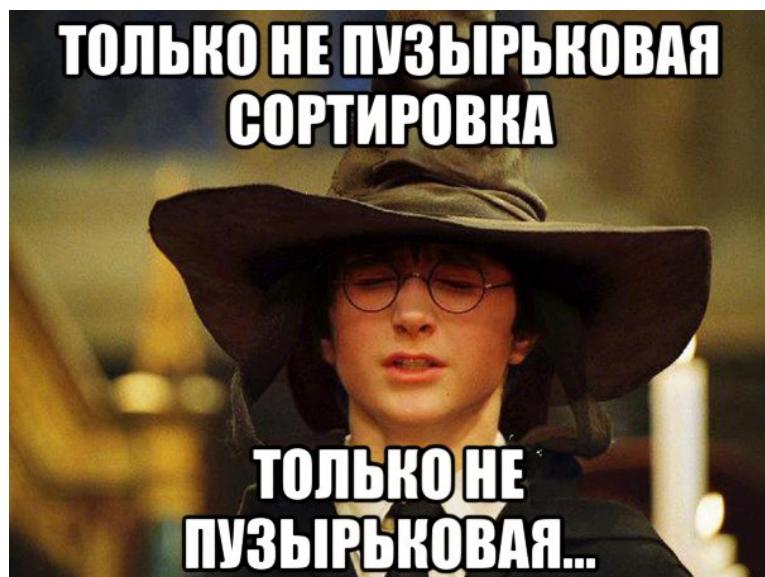
**Цель работы:** получение навыков организации эксперимента и базового анализа результатов для оценки сложности алгоритмов по времени. Закрепление знаний о сортировках.

### Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Результаты замеров (таблицы по заданию)
- Графики полученных зависимостей.
- Исходный код экспериментов.
- Ссылка на открытый репозиторий с решением и полученными замерами и история коммитов (допускается текстом).

### Рекомендации:

- Внимательно читайте задания. Например, в последнем задании первой части вам так же необходимо приложить скриншоты полученных графиков.
- Хорошо работающая программа и хороший код – это важно, но если вывод программы не отформатирован в удобной форме, то мало кому она понадобится. В конце концов, мы пишем программы не для компьютера, а для людей.
- Допускается использование более сложных сортировок вместо простых. Если вам наскучил 'пузырёк', замените на сортировку кучей и т. п.. Это сделает работу интереснее.



Лабораторная в основной части составлена Морозовым Данилой. С определенными редакционными правками представлена для решения.

С похожей лабораторной работой мне приходилось сталкиваться в курсе "Алгоритмы и структуры данных". Хорошо, что пока студенты не могут отправиться в прошлое и посмотреть моё решение. Оно работало, но могло выглядеть элегантнее. Дабы вы не повторяли плохие решения, вот ещё одно (на этот раз, надеемся, хорошее).

## Задания к лабораторной работе:

Можно сказать, что работа состоит из двух частей.

- Первая – вы реализовываете алгоритмы сортировки массивов целых чисел, проводите измерения, анализируете результаты и составляете таблицу.
- Вторая – вопросы и задания, направленные на закрепления знаний о самих сортировках, вашей изобретательности. Если в задании спрашивается «как вы думаете?..», «что нужно?..» и т.п. – это значит, что необходимо дать развернутый ответ.

Насколько развернутый? Представьте, что вам необходимо объяснить данный вопрос программисту вашего уровня, затем прочтайте свой ответ вслух – и ваше программистское чутье подскажет, достаточно ли развернут ваш ответ.

И хотя не запрещается изучать другие источники для ответа, но наибольшую ценность эти задания представляют не для преподавателя, а для **вас**, как возможность самостоятельно дойти до решения, поэтому постарайтесь сперва ответить на них самостоятельно.

О некоторых функциях ничего не было рассказано ранее. Вам стоит привыкать к этому. Часто придётся работать с кодом, который кто-то до вас написал, причём важно разобраться, что именно там происходит.

1. **Обратите внимание:** для заданий первой части ваши функции сортировки должны иметь интерфейс `void sort(int *, size_t)`.

**Все функции сортируют в порядке неубывания.**

Основная цель – научиться организовывать инфраструктуру для измерений. Все должно быть удобным для вас.

Сейчас будет рассказано о том, как это было организовано нами, но вы можете реализовать любой другой вариант, если он окажется лучше<sup>115</sup>.

Для организации инфраструктуры потребовались следующие библиотеки:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
```

Выделим структуры, описывающие сортировки:

```
// функция сортировки
typedef struct SortFunc {
    void (*sort)(int *a, size_t n);      // указатель на функцию
                                         // сортировки
    char name[64];                      // имя сортировки,
                                         // используемое при выводе
} SortFunc;

// функция генерации
```

<sup>115</sup> Возможно он и не окажется лучше, но самостоятельность в организации решения приветствуется

```

typedef struct GeneratingFunc {
    void (*generate)(int *a, size_t n); // указатель на функцию
                                         // генерации последоват.
    char name[64];                      // имя генератора,
                                         // используемое при выводе
} GeneratingFunc;

```

Решим вопрос для измерения времени для некоторого фрагмента кода:

```

double getTime() {
    clock_t start_time = clock();
    // фрагмент кода
    // время которого измеряется
    clock_t end_time = clock(); \
    clock_t sort_time = end_time - start_time;
    return (double) sort_time / CLOCKS_PER_SEC;
}

```

Но если придётся делать множество разных замеров, потребуется не одна такая функция. На самом деле, мы бы хотели, чтобы нужные нам фрагменты кода были обёрнуты процессом вычисления времени. И результат замеров куда-то записывался. Возможное решение лежит в макрофункциях<sup>116</sup>:

```

#define TIME_TEST(testCode, time) { \
    clock_t start_time = clock(); \
    testCode \
        clock_t end_time = clock(); \
    clock_t sort_time = end_time - start_time; \
    time = (double) sort_time / CLOCKS_PER_SEC; \
}

```

Если вы хорошо помните макрофункции, вы отметите, что никакой магии тут нет. Во время компиляции:

```

double time;
TIME_TEST({
    sortFunc(innerBuffer, size);
}, time);

```

будет заменён на

```

{
    clock_t start_time = clock();
    {
        sortFunc(innerBuffer, size);
    }
}

```

<sup>116</sup>Макрофункции очень капризны. Пожалуйста, не оставляйте лишних пробелов (компилятор не напишет "пожалуйста"). Это может повлиять на возможность компиляции.

```

    clock_t end_time = clock();
    clock_t sort_time = end_time - start_time;
    time = (double) sort_time / CLOCKS_PER_SEC;
}

```

---

В силу наличия фигурных скобок будет создана новая область видимости, и создаваемые в макросе переменные не будут засорять пространство имён. Пожалуйста, убедитесь, что этот приём вам понятен. Макрофункции довольно сильный инструмент, который иногда может крайне удачно использоваться.

Выделим также макрофункцию, которая позволит определить размер массива, если он объявлен в данной функции:

```
#define ARRAY_SIZE(a) sizeof(a) / sizeof(a[0])
```

---

Перейдём к функции `timeExperiment`, в которой и будет организовано ядро эксперимента по временным замерам:

```

void timeExperiment() {
    // описание функций сортировки
    SortFunc sorts[] = {
        {selectionSort, "selectionSort"}, 
        {insertionSort, "insertionSort"}, 
        // вы добавите свои сортировки
    };
    const unsigned FUNCS_N = ARRAY_SIZE(sorts);

    // описание функций генерации
    GeneratingFunc generatingFuncs[] = {
        // генерируется случайный массив
        {generateRandomArray, "random"}, 
        // генерируется массив 0, 1, 2, ..., n - 1
        {generateOrderedArray, "ordered"}, 
        // генерируется массив n - 1, n - 2, ..., 0
        {generateOrderedBackwards, "orderedBackwards"}
    };
    const unsigned CASES_N = ARRAY_SIZE(generatingFuncs);

    // запись статистики в файл
    for (size_t size = 10000; size <= 100000; size += 10000) {
        printf("-----\n");
        printf("Size: %d\n", size);
        for (int i = 0; i < FUNCS_N; i++) {
            for (int j = 0; j < CASES_N; j++) {
                // генерация имени файла
                static char filename[128];
                sprintf(filename, "%s_%s_time",
                        sorts[i].name, generatingFuncs[j].name);
                checkTime(sorts[i].sort,
                          generatingFuncs[j].generate,
                          size, filename);
            }
        }
    }
}

```

```

        }
    }
    printf("\n");
}
}

int main() {
    timeExperiment();

    return 0;
}

```

---

Для этого использовался следующая функция `checkTime`:

```

void checkTime(void(*sortFunc)(int *, size_t),
               void (*generateFunc)(int *, size_t),
               size_t size, char *experimentName) {
    static size_t runCounter = 1;

    // генерация последовательности
    static int innerBuffer[100000];
    generateFunc(innerBuffer, size);
    printf("Run #%zu | ", runCounter++);
    printf("Name: %s\n", experimentName);

```

---

В фрагменте выше происходит генерация последовательности, выводится сообщение с номером и информацией о замере, например: "Run #1 | Name: selectionSort\_random".

```

// замер времени
double time;
TIME_TEST({
    sortFunc(innerBuffer, size);
}, time);

```

---

В фрагменте ниже выполняется проверка, а действительно ли массив отсортировался (для этого придётся реализовать функцию `isOrdered`).

```

// результаты замера
printf("Status: ");
if (isOrdered(innerBuffer, size)) {
    printf("OK! Time: %.3f s.\n", time);

    // запись в файл
    char filename[256];
    sprintf(filename, "./data/%s.csv", experimentName);
    FILE *f = fopen(filename, "a");
    if (f == NULL) {
        printf("FileOpenError %s", filename);
        exit(1);

```

```

    }
    fprintf(f, "%zu; %.3f\n", size, time);
    fclose(f);
} else {
    printf("Wrong!\n");

    // вывод массива, который не смог быть отсортирован
    outputArray_(innerBuffer, size);

    exit(1);
}
}

```

---

Вы можете заметить работу с файлами. Это необходимо нам для фиксирования данных<sup>117</sup>. Для открытия файла используется функция `fopen`:

---

```
FILE *f = fopen(filename, "a");
```

---

Первый аргумент функции `fopen` – имя файла, второй аргумент – режим, в котором его необходимо открыть. В данном случае мы используем `"a"` – добавление в конец файла. Если файл не существует – он будет создан. Файлы с замерами будут создаваться в папке `data`. Если вы используете *CLion* потребуется в папке `cmake-build-debug` создать папку `data`.

Далее идет проверка, открылся ли файл с именем `filename`:

---

```
if (f == NULL) {
    printf("FileOpenError %s", filename);
    exit(1);
}
```

---

И затем мы используем функцию `fprintf` – аналогичная `printf`, при помощи которой мы записываем результат замера в файл:

---

```
fprintf(f, "%zu; %f\n", size, time);
```

---

Отличие данной функции от `printf` заключается в том, что первым аргументом указывается поток, в который осуществляется вывод. И последним шагом **обязательно** закрываем файл:

---

```
fclose(f);
```

---

Если вы вчитывались внимательно, вы должны были заметить, что мы создаём файлы формата `.csv` в папке `data`. Рекомендуется прочитать о нём подробнее. Формат `.csv` поддерживается редакторами электронных таблиц, и на выходе получается такая таблица:

---

<sup>117</sup>Обращу внимание, что хотя вы можете получить доступ к полям структуры `FILE`, лучше их не трогать. В C++, к счастью, появился механизм, позволяющий регулировать доступ к полям

	A	B
1	10000	0.179000
2	20000	0.715000
3	30000	1.617000
4	40000	2.896000
5	50000	4.593000
6	60000	6.566000
7	70000	8.860000

2. Реализуйте алгоритм сортировки обменом (пузырьковую).
3. Реализуйте алгоритм сортировки выбором.
4. Реализуйте алгоритм сортировки вставками.
5. Реализуйте алгоритм сортировки расческой.
6. Реализуйте алгоритм сортировки Шелла.
7. У данного задания 3 варианта выполнения:
  - (а) Не делать его вовсе;
  - (б) Реализовать только для положительных;
  - (с) Реализовать для отрицательных и положительных.

Реализуйте алгоритм цифровой сортировки. В пособии был приведен пример, когда для каждого шага бралось по 2 бита, в данном случае возьмите для сравнения на каждый шаг по 8 бит – сперва вы сравниваете числа по первому байту, затем по второму, третьему и четвертому. При реализации постарайтесь применять побитовые как можно больше.

**Запрещено** реализовывать цифровую сортировку по основанию 10, в которой сравниваются цифры чисел, а не байты.

8. Проведите замеры времени для всех реализованных алгоритмов с упорядоченными, неупорядоченными и упорядоченными в обратном направлении массивами размеров от 10000 до 100000 с шагом 10000. Создайте соответствующий коммит на гите (вы можете использовать меньшие объемы данных, например от 5000 до 50000 с шагом 5000)<sup>118</sup>. Если для некоторых сортировок вы получаете значения времени близкие к нулю (например 0,001), требуется увеличить размеры массивов и шаг (например, с 500000, 1000000, 1500000, 2000000, ..., 5000000).
9. Выполните модификацию проекта таким образом (посредством добавления новых функций, объявлением вспомогательных структур), чтобы можно было считать как и время, так и количество проводимых внутри операций сравнения, и так же проведите эксперимент с записью данных в файлы, только вместо времени выполнения записываете количество операций сравнения. По окончанию замеров сделайте коммит.

---

<sup>118</sup>В силу того, что при новом запуске программы происходит дозапись в существующий файл, вам придётся очистить папку `data` перед контрольными замерами, чтобы в файлах не содержались записи с одинаковыми N.

Например реализация сортировки выбором, в которой выполняется подсчёт операций сравнения:

```
long long getSelectionSortNComp(int *a, size_t n) {
    long long nComps = 0;
    for (int i = 0; ++nComps && i < n; i++) {
        int min = a[i];
        int minIndex = i;
        for (int j = i + 1; ++nComps && j < n; j++)
            if (++nComps && a[j] < min) {
                min = a[j];
                minIndex = j;
            }
        if (++nComps && i != minIndex)
            swap(&a[i], &a[minIndex]);
    }

    return nComps;
}
```

10. Проанализируйте результаты с помощью инструмента, заполните таблицы, представленные на следующей странице.
11. Для первой таблицы приложите графики полученных зависимостей (пример на рисунке 18.24)<sup>119</sup>



<sup>119</sup>Точки, получаемые на графиках, могут отклоняться от аппроксимирующей кривой. Чтобы уменьшить подобные отклонения рекомендуется вычислять время несколько раз и усреднять результаты измерений.

Таблица 18.2: Затрачиваемое время сортировкой, чтобы упорядочить  $N$  элементов упорядоченного массива

Тип сортировки	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	$T(N)$
Пузырьком											
Выбором	0.085	0.344	0.752	1.339	2.090	3.009	4.096	5.345	6.763	8.364	$T(N) = 8.36 * 10^{-10} N^2$

Таблица 18.3: Затрачиваемое время сортировкой, чтобы упорядочить  $N$  элементов упорядоченного массива

Тип сортировки	500000	1000000	1500000	2000000	2500000	3000000	3500000	4000000	4500000	5000000	$T(N)$
Вставками											
Расческой											
Шелла											
Цифровая											

Самостоятельно определите определите требуемые для отображения результатов экспериментов таблицы.

Таблица 18.4: Затрачиваемое время сортировкой, чтобы упорядочить  $N$  элементов массива упорядоченного в обратном порядке

Тип сортировки	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	$T(N)$
...	...	...	...	...	...	...	...	...	...	...	...

Таблица 18.5: Затрачиваемое время каждой из сортировок для того, чтобы упорядочить  $N$  элементов неупорядоченного массива

Тип сортировки	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	$T(N)$
...	...	...	...	...	...	...	...	...	...	...	...

Таблица 18.6: Зависимость количества сравнений, производимых сортировкой от количества элементов в массиве ( $N$ )

Тип сортировки	Упорядоченный (1)	В обратном порядке (2)	Неупорядоченный (3)	Порядок ФВС (1)	Порядок ФВС (2)	Порядок ФВС (3)
Пузырьком						
Выбором	$N_{comp} = 1.024N^2$	$N_{comp} = 1.024N^2$	$N_{comp} = 1.024N^2$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Вставками						
Расческой						
Шелла						
Цифровая						

Функции  $T(N)$  и  $N_{comp}(N)$  должны иметь один и тот же порядок. Если смотреть на сортировку выбором, формулы следующие:

$$T(N) = 8.36 * 10^{-10} N^2 \quad N_{comp}(N) = 1.024N^2$$

Заметьте, что и в первом, и во втором случае порядок функции  $N^2$  (совпадают между собой).

Разберемся, как пользоваться данным калькулятором для заполнения таблицы. Перейдите на [сайт](#). И выберете обобщенный метод:

## Метод наименьших квадратов

**Метод:** Обобщённый (нелинейный)

**Функция:**

Используется для обобщённого метода наименьших квадратов. Введите кастомную функцию аппроксимации - зависит от x., все остальные переменные (кроме x) будут считаться параметрами, например a, b, z, k1, k2, v

**Данные:**

10000; 0.085000
20000; 0.344000
30000; 0.752000
40000; 1.339000
50000; 2.090000
60000; 3.009000
70000; 4.096000
80000; 5.345000
90000; 6.763000
100000; 8.364000

Введите данные в эту форму

**Или CSV-файл с данными:**  Файл не выбран.

Затем мы предполагаем, по какой функции распределяются наши данные и пишем её поле "Функция". Возможные варианты:

- $k1 * x$
- $k1 * x^2$
- $k1 * x * \log(x)$
- $k1 * x^{120}$

Потом вносим данные (в примере занесена информация о времени выполнения сортировки выбором от количества элементов). Затем мы нажимаем «аппроксимировать»<sup>121</sup>, после чего будет отражена полученная зависимость:

Обобщенная аппроксимация:

Найденные аппроксимированные параметры:  
Получается график функции:

$f(x) = k1*x^2 =$

$8.35753365174184e-10*x$

<sup>120</sup> Для зависимости времени от количества элементов для сортировки выбором при таком варианте аппроксимации было получено  $8.3682 * 10^{-10} N^{1.999888}$

<sup>121</sup> Аппроксимация (от лат. рохіма — ближайшая) или приближение — научный метод, состоящий в замене одних объектов другими, в каком-то смысле близкими к исходным, но более простыми.

Ниже вы увидите график, по которому следует определить, насколько точной получилась аппроксимация:

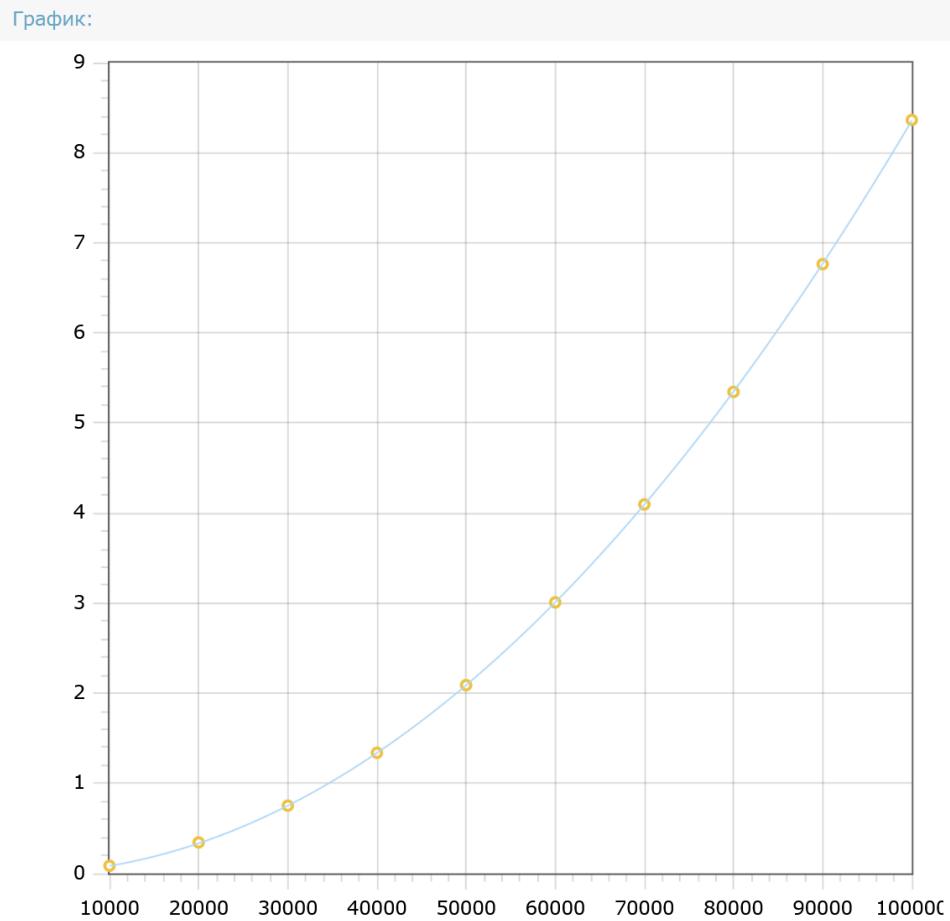


Рис. 18.24: Аппроксимация полученных данных в ходе эксперимента

Как мы видим, функция была подобрана правильно, ведь её график (синяя линия) проходит там же, где и реальные данные (желтые точки). Таким образом, мы можем утверждать, что время работы функции сортировки может быть найдено по формуле:

$$T(N) = 8.36 * 10^{-10} N^2 \text{ секунд}$$

Например, если потребуется отсортировать массив из миллиона элементов на моей ЭВМ, потребуется:

$$T(1000000) = 8.36 * 10^{-10} 1000000^2 = 8.36 * 10^2 = 836 \text{ секунд}$$

Имея формулу можно решить и обратную задачу, какой объем данных может быть отсортирован за 1 год:

$$N^2 = \frac{365 * 24 * 60 * 60}{8.36 * 10^{-10}} \approx 3.772248 * 10^{16}$$

$$N \approx 194222779$$

Рассмотрим, как будет выглядеть график, если вы подберете неподходящую функцию:

## Метод наименьших квадратов

**Метод:** Обобщённый (нелинейный) ▾

**Функция:**

Используется для обобщённого метода наименьших квадратов. Вводите кастомную функцию аппроксимации - зависит от x., все остальные переменные (кроме x) будут считаться параметрами, например a, b, z, k1, k2, v

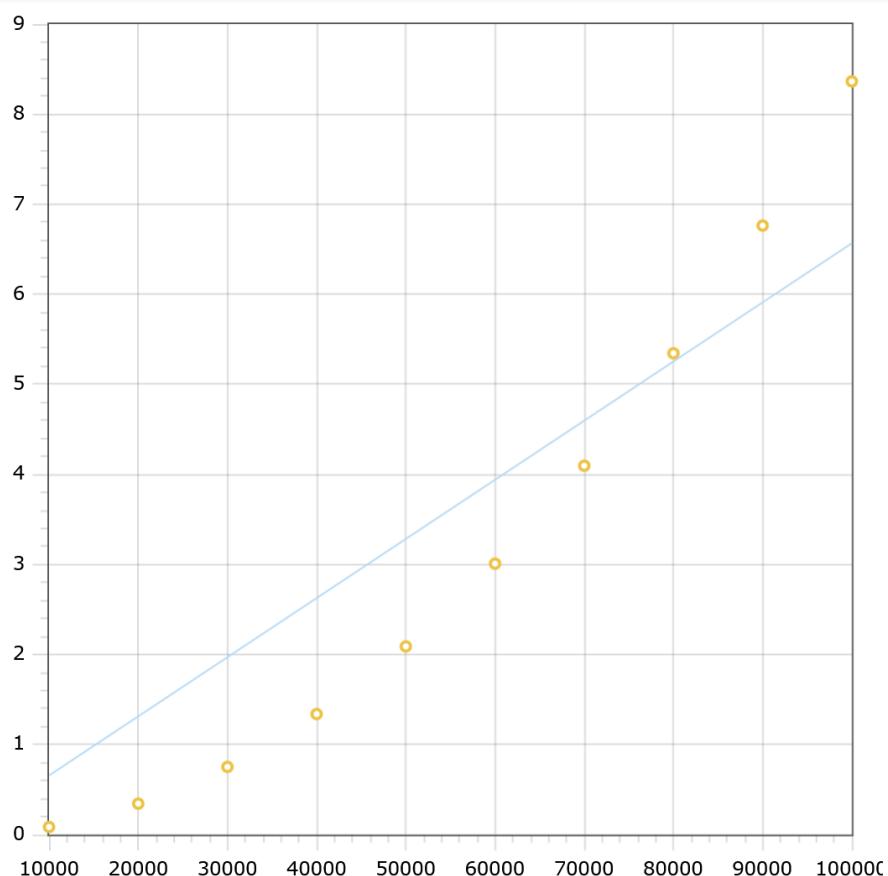
**Данные:**

```
10000.0;0.085
20000.0;0.344
30000.0;0.752
40000.0;1.339
50000.0;2.09
60000.0;3.009
70000.0;4.096
80000.0;5.345
90000.0;6.763
100000.0;8.364
```

Введите данные в эту форму

**Или CSV-файл с данными:**  Файл не выбран.

График:



Как мы видим, аппроксимация довольно далека от реальных данных. Следовательно время работы от количества данных не может быть выражено линейной зависимостью.

Дайте развернутый ответ на следующие вопросы:

1. При измерении времени выполнения, конечно же, влияет не только количество сравнений, но и операции обмена. Если бы вам было необходимо сортировать обменной сортировкой массив из структур большого размера (скажем, 8096 байт) – как можно было бы избежать «просадок» в скорости сортировки, не меняя алгоритм сортировки<sup>122</sup>? Помните, что нам важно получить *правильный порядок* элементов, но для этого они сами *не обязаны располагаться в правильном порядке*<sup>123</sup>.
2. Возьмите произвольную сортировку. Оцените, сколько времени потребуется ей для упорядочения массива из  $N$  ( $N \geq 200000$ ) элементов. Все вычисления приложите к отчёту. Выполните запуск выбранной сортировки на вашей ЭВМ. Оцените погрешность измерений (укажите отклонение в процентах).
3. \* Если бы мы использовали сортировку вставками для сортировки двусвязного линейного списка, а не массива, то от какого недостатка этого алгоритма мы бы избавились, по сравнению с сортировкой массива?
4. \* Подумайте, можно ли применить вашу реализацию цифровой сортировки к дробным числам? Что нужно изменить, чтобы она работала? Не обязательно реализовывать что-то, главное - описать идею. Ответить на эти вопросы поможет стандарт IEEE-754.

---

<sup>122</sup>Имеется в виду, что вы не можете выбрать другую сортировку взамен обменной, но можете вносить в нее изменения

<sup>123</sup>К примеру, вам нужно пройти медосмотр. Вместо того, чтобы перестроить больницу для вас, дабы вы прошли врачей в правильном порядке, вам выдают список, указывающий правильный порядок обхода. Как эту идею применить в данном случае?

## 18.16 Лабораторная работа «Потоки. Ссылки»

**Цель работы:** получение навыков работы с потоками, ссылками, управляющими конструкциями; осознание необходимости появления данных языковых средств.

### Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Тексты заданий с набранными фрагментами кода к каждому из пунктов и ответами на вопросы по ходу выполнения работы.

### Требования:

- При выполнении работы не используйте `printf` и `scanf`.
- Предпочитайте цикл *range-based for* циклу *for*.

### Задания к лабораторной работе:

Задания к данной лабораторной работе представлены множеством небольших пунктов, выполнение которых позволит лучше понять аспекты языка. Настоятельно рекомендуется делать отчёт на этапе выполнения лабораторной.

Вероятно, поведение, которое вы будете наблюдать, будет отличаться от описанных в теории. Буферизация потоков может зависеть от многих факторов. Однако многие источники будут указывать на буферизацию `std::cout` и `std::clog`.

Библиотека `<iostream>` предоставляет множество классов для работы с вводом / выводом:

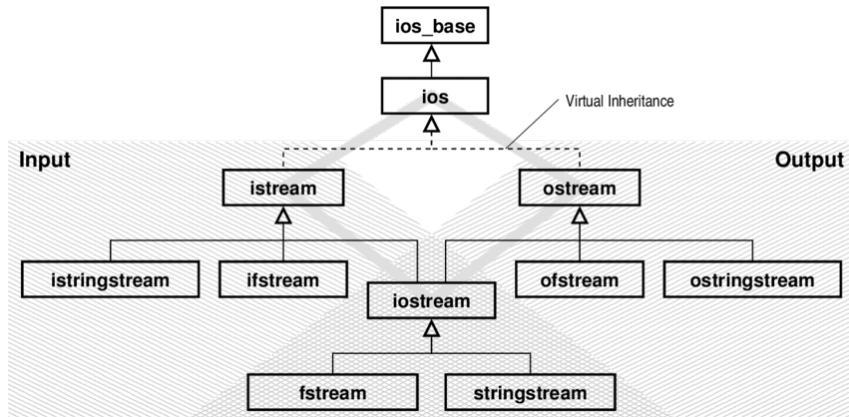


Рис. 18.25: Иерархия классов ввода / вывода

Одна из целей лабораторной заключается в том, чтобы на практике поработать с данными средствами языка.

### 18.16.1 Буферизация в выходных потоках

1. Создайте функцию `infinitivePause`, которая в своём теле будет воспроизводить бесконечный цикл.
2. Отправьте в объект `cout`, связанный со стандартным потоком вывода, произвольное сообщение без использования `endl`. После чего сделайте вызов `infinitivePause`. Запустите программу. Опишите<sup>124</sup> наблюдаемое поведение.
3. Выполните аналогичные пункту 2 действия только с использованием `endl`. Опишите наблюдаемое поведение.
4. Для случая 2, после вывода сообщения вставьте инструкцию ввода какой-нибудь переменной. Запустите приложение. Опишите наблюдаемое поведение.
5. Выполните создание строки, состоящей из 10000 символов 'а':

```
std::string s(10000, 'a');
```

Осуществите её вывод перед вызовом `infinitivePause`. Опишите наблюдаемое поведение.

6. На основании прошлых трёх пунктов выделите 3 случая, когда очищается буфер вывода. Дополнительно добавьте четвертый: буфер очищается по окончанию работы программы.
7. Говорят, что частый сброс буфера способен повлиять на производительность приложения<sup>125</sup>. Проведите эксперимент, доказывающий это. Например, он мог выглядеть так: выполним создание файла, в который будем записывать данные. В одном случае будем сбрасывать буфер, в другом не будем:

```
#include <iostream>
#include <fstream> // для работы с файлами
#include <ctime>

// вывод осуществляется в поток, связанный с логами
#define TIME_TEST(testCode, message) { \
    clock_t start_time = clock () ; \
    testCode \
    clock_t end_time = clock () ; \
    clock_t sort_time = end_time - start_time ; \
    std::clog << message << ":" " \
                << (double) sort_time/CLOCKS_PER_SEC << std::endl; \
}

int main() {
    const char *filename = "tmp.txt";
    std::ofstream file(filename); // выполняем создание файла
    TIME_TEST({
```

<sup>124</sup>Когда вас просят описать поведение, напишите то, что увидели, в процессе запуска приложения.

<sup>125</sup>В примере ниже вы можете заметить использование макрофункций. Найдите для себя ответ на вопрос: "Для каких случаев макрофункции не могут быть заменены обычными функциями?"

```

        for (int i = 0; i < 1000000; i++)
            file << 'a' << std::endl;
    }, "Often buffer reset");

TIME_TEST({
    for (int i = 0; i < 1000000; i++)
        file << 'a' << '\n';
}, "Buffer opt");

file.close();           // закрываем файл
std::remove(filename); // удаляем временный файл

return 0;
}

```

Вставьте код вашего эксперимента, а также выводимые программой значения.

8. Рассмотрим случай, когда частый сброс буфера вывода из-за чередования ввода / вывода оказывает влияние на производительность. Выполните решение задачи Минимальная OR сумма (1635A) на *codeforces* и приложите вердикт тестирующей системы.
9. В условиях задачи 4 добавьте

---

```
std::cin.tie(nullptr);
```

---

Опишите наблюдаемое поведение.

10. В решение вашей задачи для пункта 8 добавьте строки:

---

```

std::cin.tie(nullptr);           // разрывает связь с потоком
                                // вывода
std::ios_base::sync_with_stdio(false); // устанавливает,
                                // синхронизируются ли
                                // стандартные потоки C++
                                // со стандартными потоками C
                                // после каждой операции
                                // ввода/вывода.

// последняя команда обязана быть выполнена до первой операции
// ввода / вывода

```

---

И снова приложите вердикт тестирующей системы.

11. Существуют объекты, связанные с потоками вывода ошибок (**cerr**) и логов (**clog**). Приведите код, который может проверить, является ли вывод в них буферизированным. Как вы считаете, почему было принято именно такое решение по буферизации данных потоков? Ответ обоснуйте.

## 18.16.2 Некоторые особенности вывода

- При помощи `cout` можно выводить в поток вывода значения переменных и адреса:

---

```
int i = 10;
std::cout << i << " " << &i;
```

---

Создайте переменную `s` типа `const char*`, выполните её инициализацию и попробуйте вывести её в поток вывода. Получившееся поведение опишите.

- Несмотря на то, что переменная типа `const char*` и так является адресом, вы не должны были его увидеть. Чтобы увидеть адрес, необходимо такую переменную привести к типу `void*`<sup>126</sup>. Однако одним приведением типа вы не отделаетесь, так как мешает `const`. Цепочка преобразований будет такова:

*`const char* → char* → void*`*

---

```
std::cout << static_cast<void*>(const_cast<char*>(s));
```

---

- Выведите литералы `true` и `false` в поток вывода. Какие значения были отображены на экране?
- Добавьте манипулятор `std::boolalpha`.

---

```
#include <iostream>

int main() {
    std::cout << std::boolalpha << true << " " << false;

    return 0;
}
```

---

Опишите отображаемый вывод. Проверьте, распространяется ли действие манипулятора на последующие выводы.

- Чтобы переключиться на вывод логических значений 1 и 0 используйте манипулятор `std::noboolalpha`.

## 18.16.3 Ввод

- Создайте переменную целочисленного типа, но при осуществлении ввода введите строку, состоящую из символов-букв. Опишите наблюдаемое поведение.
- Переменная, связанная с потоком ввода будет возвращать значение "истина", если поток ввода не находится в состоянии ошибки. Попробуйте запустить пример из прошлого пункта, добавив обработку:

---

<sup>126</sup>На самом деле, вы могли бы использовать приведение в стиле C, но оно не рекомендуется.

---

```

int a;
std::cin >> a;

if (std::cin)
    std::cout << "Success";
else
    std::cout << "Error";

```

---

Поэкспериментируйте с разными вариантами ввода. Приведите пример такого ввода, который выдаёт "Success", но содержит символы, отличные от цифр.

3. Создайте переменную типа `char` и попробуйте выполнить ввод пробельного символа. Опишите наблюдаемое поведение.
4. Проделайте то же самое но при считывании осуществляйте ввод через `cin.get()`:

---

```

#include <iostream>

int main() {
    char c;
    std::cin.get(c);

    std::cout << c;
}

```

---

Сделайте выводы.

5. Манипулятор `std::noskipws` заставит выполнить ввод и пробельного символа. Запустите пример:

---

```

#include <iostream>

int main() {
    char c;
    std::cin >> std::noskipws >> c;

    std::cout << c;

    return 0;
}

```

---

Проверьте, распространяется ли действие манипулятора на последующие вводы. Сделайте выводы.

6. Иногда возникает потребность пропустить некоторые символы в потоке ввода. Например, вводится дата в формате "DD.MM.YYYY" (например 19.02.2003). И стоит цель вычленить отдельно день, месяц и год в переменные типа `int`. Это можно сделать так:

---

```
#include <iostream>

int main() {
    int day;
    std::cin >> day;
    std::cin.ignore(1);

    int month;
    std::cin >> month;
    std::cin.ignore(1);

    int year;
    std::cin >> year;

    // при вводе 19.02.2003 выдаст 19/02/2003
    std::cout << day << '/' << month << '/' << year;

    return 0;
}
```

---

Выполните чтение числа до точки и после точки. Например "14.341" должно быть разбито на 14 и 341.

7. Обсудим строковый ввод. Несмотря на возможность осуществлять ввод как строк в стиле C, так и за счёт класса `std::string` в C++, вам следует остановиться на последнем, так как происходит автоматическое управление памятью. Идеальный ввод на C++ строк в стиле C выглядел бы так:

---

```
#include <iostream>
#include <cstring>

int main() {
    char buf[1000];
    std::cin >> buf;

    char *s = new char[strlen(buf) + 1]; // выделение памяти под строку
    strcpy(s, buf); // копирование из буфера
    // в строку

    // что-то делаем со строкой, например, выводим
    std::cout << s;

    delete[] s; // освобождение памяти

    return 0;
}
```

---

Внедрение работы с памятью в алгоритм вовсе не улучшает его читаемость. Тот же фрагмент на C++:

---

```
#include <iostream>

int main() {
    std::string s;
    std::cin >> s;

    std::cout << s;

    return 0;
}
```

---

Просто откажитесь от соблазна использовать строки в стиле C, и с этого пункта достаточно.

#### 18.16.4 Файловый ввод / вывод

Библиотека *iostream* поддерживает чтение и запись в файлы. Для этого предназначены следующие классы:

- *ifstream* – связывает ввод программы с файлом
- *ofstream* – связывает вывод программы с файлом
- *fstream* – связывает как ввод, так и вывод программы с файлом

1. Очень часто приходится оперировать работой с **файлами** – именованной областью данных на носителе информации. В рамках данной лабораторной разберёмся, как осуществляется работа с текстовыми файлами. Создайте вручную (не при помощи кода) файл `input.txt` содержащий несколько строк, по одному числу в каждой.
2. Убедитесь, что можете использовать оператор `>>` для считывания строк из файла:

---

```
#include <fstream>

int main() {
    // чтение из файла
    std::ifstream inputFile("input.txt");

    std::string s;
    while (inputFile >> s)
        std::cout << s << '\n';

    return 0;
}
```

---

Если файл будет считан полностью, в результате выполнения выражения `inputFile >> s` будет возвращено значение 'ложь'.

3. Код описанный выше несколько небезопасен. Файл с таким именем мог и не существовать. Попробуйте запустить прошлый фрагмент для файла, который не был создан до этого и опишите наблюдаемое поведение.

4. В прошлом примере для проверки на то, открыт ли файл, мог использоваться функция-член `.is_open()`:

---

```
std::ifstream inputFile("notExists.txt");

if (inputFile.is_open())
    std::cout << "Exists";
else
    std::cout << "Not exists";
```

---

однако и сама переменная, ассоциированная с файлом может быть использована для проверки на то, прошла ли операция открытия успешно:

---

```
std::ifstream inputFile("notExists.txt");

if (inputFile)
    std::cout << "Exists";
else
    std::cout << "Not exists";
```

---

5. В примере выше мы осуществляли ввод в строку, но с таким же успехом вы можете считывать данные в переменные, например, целочисленного типа.

---

```
int x;
while (inputFile >> x) {
```

}

---

Имея данные сведения, реализуйте функцию `long long getSum(const std::string &filename)` для вычисления суммы чисел, записанных в файл с именем `filename`. Если файла с таким именем нет, функция должна возвращать -1.

6. История с возвратом значения -1, если файл не был найден, смотрится несколько нелепо (ответьте для себя, а почему?). В C++ появились другие средства для сигнализирования об ошибках, называемых исключениями. Мы не будем вдаваться в подробности сейчас. Если вы хотите сигнализировать, что возникла некоторая ошибка времени исполнения, можно использовать такой подход:

---

```
#include <stdexcept>

long long getSum(const std::string &filename) {
    std::ifstream inputFile(filename);

    if (!inputFile)
        throw std::runtime_error("File doesn't exist");

    // работа с файлом; ветку else стоят опустить
}
```

---

Модифицируйте программу из пункта 5, запустите её для несуществующего файла. Вставьте в отчёт полученное сообщение.

7. Пусть в файле `inputFile` записаны размеры матрицы, а далее записаны непосредственно элементы матрицы, например:

---

```
2 4
1 2 3 4
5 3 4 2
```

---

Функция `long long getSumOfMaxesInRows(const std::string &filename)` должна выполнять поиск суммы максимальных элементов строк. Для примера выше сумма равняется 9. Реализуйте её.

### 18.16.5 *sstream*

Библиотека *iostream* поддерживает также ввод / вывод в область памяти, при этом поток связывается со строкой в памяти программы. С помощью потоковых операторов ввода / вывода мы можем записывать данные в эту строку и считывать их оттуда. Определены следующие классы:

- *istringstream* – чтение из строки
- *ostringstream* – запись в строку
- *stringstream* – чтение в строку и запись из строки

Например, можно крайне комфортно сконструировать сообщения об ошибках:

---

```
#include <iostream>
#include <sstream>

int main() {
    std::cout << "Input x > 0." << '\n';

    int x;
    std::cin >> x;

    if (x <= 0) {
        std::ostringstream errorMessage;
        errorMessage << "Expected x < 0, "
                     << "Got: " << x << '\n';

        throw std::runtime_error(errorMessage.str());
    } else {
        std::cout << "Success";
    }

    return 0;
}
```

---

Возможное решение задачи о выводе слов с чередованием из двух строк:

---

```
#include <iostream>
#include <sstream>

int main() {
    std::string s1, s2;
    std::getline(cin, s1);
    std::getline(cin, s2);

    std::istringstream ss1(s1);
    std::istringstream ss2(s2);

    std::string word;
    // ss1.eof() возвращает значение истине, если буфер пуст
    while (!ss1.eof() || !ss2.eof()) {
        if (!ss1.eof()) {
            ss1 >> word;
            std::cout << word << ' ';
        }
        if (!ss2.eof()) {
            ss2 >> word;
            std::cout << word << ' ';
        }
    }

    return 0;
}
```

---

## 18.16.6 Определение операций ввода / вывода для произвольных типов

- Опишите произвольную (но отличную от примера) структуру. Например `Person` с полями `name` и `age`<sup>127</sup>:

---

```
struct Person {
    std::string name;
    int age;
};
```

---

- Реализуйте функции ввода и вывода<sup>128</sup> структуры:

---

```
void inputPerson(Person &p);
void outputPerson(const Person &p)
```

---

<sup>127</sup>Имя структуры обязано быть написано в `PascalCase` (с заглавной буквы и каждое последующее слово с заглавной).

<sup>128</sup>Операция вывода никогда не должна выводить символа переноса на новую строку. Оставьте возможность определить данный момент пользователю библиотеки.

3. В `main` создайте переменную данного типа и выполните ввод и вывод структуры на экран:

---

```
int main() {
    Person p;

    inputPerson(p);
    outputPerson(p);

    return 0;
}
```

---

Такой подход немного неудобен. Ведь когда мы вводим переменные, нам бы хотелось использовать `cin`, а для вывода – `cout`. Для этого придётся для нашей структуры определить операцию ввода и вывода:

---

```
void operator>>(std::istream &in, Person &p) {
    in >> p.name >> p.age;
}

void operator<<(std::ostream &out, Person &p) {
    out << p.name << " is " << p.age << " years old";
}
```

---

Переменная `in` выше имеет тип `std::istream&`. Функция принимает объект типа `std::istream&`. Описание функции в таком ключе позволит осуществлять ввод структуры для произвольного объекта, ассоциированных с вводом хоть с клавиатуры, хоть с файла.

4. Создайте файл `input.txt`, и выполните чтение структуры и с клавиатуры, и из файла:

---

```
Person p;

// ввод с клавиатуры
std::cin >> p;
// вывод в консоль
std::cout << p;

// ввод с файла
std::ifstream inputFile("input.txt");
// вывод в файл
std::ofstream outputFile("output.txt");

// ввод с файла
inputFile >> p;
outputFile << p;
```

---

5. Попробуйте написать цепочку вида:

---

```
Person p1, p2;
std::cin >> p1 >> p2;
```

---

Опишите наблюдаемое поведение.

6. Чтобы двигаться дальше, необходимо понять, почему наблюдалось такое поведение. Когда вы пишите:

---

```
std::cin >> p;
```

---

неявно происходит вызов функции:

---

```
operator>>(std::cin, p);
```

---

Для ввода одного значения такой вызов хорошо работает. Но если будет цепочка:

---

```
cin >> p1 >> p2;
```

---

Это заменяется на

---

```
operator>>(operator>>(std::cin, p1), p2);
```

---

Но в силу того первый вызов вернёт `void` произойдет ошибка компиляции. Внимательно почитайте об ошибке, например, для *CLion*:

```
Person p1, p2;
std::cin >> p1 >> p2;
```

Invalid operands to binary expression ('void' and 'Person')
 candidate function not viable: cannot convert argument of incomplete type 'void' to 'std::istream &' (aka
 'basic\_istream<char> &') for 1st argument

что даст понимание того, что данное рассуждение верно.<sup>129</sup>

## Как выглядят ошибки в Python



## Как выглядят ошибки в C++



<sup>129</sup>Читать ошибки, выдаваемые плюсами, довольно тяжело.

Решается данная проблема просто, изменением сигнатуры функции:

---

```
std::istream& operator>>(std::istream &in, Person &p) {
    in >> p.name >> p.age;
    return in;
}

std::ostream& operator<<(std::ostream &out, Person &p) {
    out << p.name << " is " << p.age << " years old\n";
    return out;
}
```

---

Выполните такую замену для своей структуры, убедитесь, что всё работает:

---

```
int main() {
    Person p1, p2;
    std::cin >> p1 >> p2;

    std::cout << p1 << p2;

    return 0;
}
```

---

### 18.16.7 Ссылки

Предпосылками к созданию ссылок было определенное неудобство при работе с указателями. Напомню, что в языке программирования С они решали следующие 5 задач:

- как способ передачи объектов в функции для их изменения;
- как способ передачи объектов без копирования;
- как необязательный аргумент функции, если функция имела несколько выходных параметров (когда было недостаточно одного возвращаемого значения);
- как половина пары указатель/длина, используемой для представления массивов;
- как средство управления памятью в куче.

Первые три задачи были отданы ссылкам.

1. Реализуйте функцию `sort2` с использованием указателей. Введите два значения в `main` и выполните вызов функции `sort2`.
2. Реализуйте функцию `sort2` с использованием ссылок. Введите два значения в `main` и выполните вызов функции `sort2`.
3. Создайте в функции `main` вектор из 100000000 целых значений:

---

```
std::vector<int> v(100000000);
```

---

Напишите функции которые в своём теле ничего не делают, но принимают вектор следующими способами:

- По значению (`vector<int> v`)
- По значению указателя (`vector<int> *v`)
- По ссылке (`vector<int> &v`)

Замерьте время вызова каждой из функций и приложите к отчёту вместе с кодом эксперимента. Сделайте выводы.

4. Напишите следующие функции:

- Ввод вектора целых чисел
- Вывод вектора целых чисел
- Поиск минимального значения среди элементов вектора
- Вводится последовательность с клавиатуры, признак конца ввода – 0. Изменить порядок следования элементов в векторе на обратный<sup>130</sup>. Индексы в функции изменения порядка должны иметь тип `size_t`. Убедитесь, что функция корректно работает для пустого вектора (проще обработать данный случай до тела цикла).

**Убедитесь, что используете оптимальный способ передачи вектора в функцию (по ссылке или по ссылке на константу).**

5. Решите следующую задачу, используя указатели: даны коэффициенты  $a$ ,  $b$ ,  $c$  квадратного уравнения. Гарантируется, что количество корней равняется двум. Вывести полученные корни. Корни должны выводиться в функции `main`, а их вычисление – происходить в `void getRoots(int a, int b, int c, double *x1, double *x2)`.

6. Для условия задачи 5, выполните переход на ссылки: `void getRoots(int a, int b, int c, double &x1, double &x2)`

7. При необходимости проведите эксперименты и заполните следующую таблицу:

Аспект	Указатель	Ссылка
Обязан быть инициализирован		
Особенности обращения к элементам		
Возможность перенацеливания		
Возможность получения адреса переменной (адреса указателя или адреса ссылки)		
Возможность непосредственной работы с динамической памятью		
Возможность создавать массивы		

<sup>130</sup> Для добавления элементов в конец вектора используется метод `.push_back()`. Для данной задачи весьма красиво будет смотреться использование бесконечного цикла.

## 18.17 Лабораторная работа «Исключения»

**Цель работы:** получение навыков работы с исключениями, осознание необходимости данных языковых средств.

### Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Тексты заданий с набранными фрагментами и ответами на необходимые вопросы. Даже если пункт сделан в ознакомительных целях, в процессе набора вы акцентируетесь на нужных моментах.

### Задания к лабораторной работе:

В процессе запуска программ, взаимодействуя с ней, могут происходить различные нештатные ситуации: выход за пределы массива, некорректные аргументы, получаемые функциями и т. п.. Данные ситуации называются **исключительными**<sup>131</sup>. Есть несколько способов, как вы можете реагировать на это:

1. Не реагировать. Умолчать об ошибке. Что будет, если мы умолчим о выходе за пределы массива? Это является неопределенным поведением и может произойти что угодно:
  - (a) Программа завершится с ошибкой.
  - (b) Программа благополучно отработает и выдаст правильный результат.
  - (c) Программа отработает и выдаст какой-нибудь результат: число, строку, может изобрести искусственный интеллект, активировать *SkyNet* и уничтожить человечество (да, у нас опасная работа!).
2. Остановить работу программы во избежание рисков, описанных выше. Использовать `exit`.
3. Вывести сообщение об ошибке (если кому-то от этого легче).
4. Использовать специальные языковые средства, для обработки исключений.

Ещё примеры исключительных ситуаций:

- деление на ноль;
- нехватка оперативной памяти при использовании оператора `new` для ее выделения (или другой функции);
- доступ к элементу массива за его пределами (ошибочный индекс);
- переполнение значения для некоторого типа;
- взятие корня из отрицательного числа.

---

<sup>131</sup> Впервые с исключениями я познакомился на втором курсе. При рассмотрении объектно-ориентированного программирования, данная тема отодвигается к середине курса. Однако мне не видится в ней ничего сложного, и это может быть понято на начальном уровне.

В языке C++ **исключения** – это специальные объекты класса или значения базового типа, которые описывают (определяют) конкретную исключительную ситуацию и соответствующим образом обрабатываются.

- Рассмотрим всё же плохой сценарий обработки (вывод сообщения). Самостоятельно наберите пример ниже и опишите его поведение при `b` равном и неравном нулю.

---

```
#include <iostream>

int division(int a, int b) {
    if (b != 0)
        return a / b;
    else
        std::cerr << "b must not be equal to 0" << std::endl;
}

int main() {
    int a, b;
    std::cin >> a >> b;

    auto res = division(a, b);
    std::cout << res;

    return 0;
}
```

---

Какие недостатки вы можете выделить?

- Каким образом функция `main` из прошлого примера поймёт, что что-то в функции `division` пошло не так? Да, вы можете сказать, что это было известно ещё до вызова функции. Однако заранее знать ситуации, при которых могут возникнуть исключительные ситуации не всегда возможно. Функция `division` хотела бы как-нибудь информировать `main` о том, что произошла проблема. Она могла бы делать это при помощи возвращаемого значения:

---

```
#include <iostream>

bool division(int a, int b, int &res) {
    if (b != 0) {
        res = a / b;
        return true;
    } else {
        return false;
    }
}

int main() {
    int a, b;
    std::cin >> a >> b;

    int res;
```

```

    if (division(a, b, res)) {
        std::cout << res;
    } else {
        std::cerr << "b must not be equal to 0" << std::endl;
    }

    return 0;
}

```

---

Выделите недостатки.

- Для сигнализирования исключительной ситуации в языке программирования C++ используется ключевое слово `throw`:

```

#include <iostream>

int division(int a, int b) {
    if (b == 0)
        throw std::string("Division by zero");

    return a / b;
}

```

---

за которым следует исключение<sup>132</sup>.

Проверьте работу функции `main` с данным примером. Результаты приложите.

- `Throw` прокидывает исключение вызывающей стороне. Считается, что она (вызывающая сторона) каким-то образом знает, как обработать данную исключительную ситуацию на основании полученного исключения. Для перехвата исключения используется конструкция вида:

```

try {
    // тело блока try.
    // В нём могут располагаться произвольные операторы

}
catch( тип1 идентификатор1 )
{
    // тело блока catch
}
catch( тип2 идентификатор2 )
{
    // тело блока catch
}
...
catch( типN идентификаторN )
{
    // тело блока catch
}

```

---

<sup>132</sup>Пожалуйста, разделяйте понятия "Исключение" и "Исключительная ситуация"

Если в блоке `try` генерируется соответствующая исключительная ситуация, то она перехватывается подходящим блоком `catch`. Выбор того или иного блока `catch` осуществляется в зависимости от типа исключительной ситуации. После возникновения исключительной ситуации определенного типа, вызывается блок `catch` с соответствующим типом аргумента.

Если в блоке `try` возникнет исключительная ситуация, которая не предусмотрена блоком `catch`, то вызывается стандартная функция `terminate()`, которая по умолчанию вызовет функцию `abort()`. Эта стандартная функция останавливает выполнение программы.

В блоке `catch` часто выполняются следующие действия:

- Выводится сообщение об ошибке в поток.
- Ошибка может быть проинута дальше в вызывающий код.
- Осуществлен возврат значения для вызывающей функции.

Ознакомьтесь с примерами (наберите, опишите поведение):

---

```
#include <iostream>

int division(int a, int b) {
    if (b == 0)
        // выкидываем строку, по которой можно будет понять
        // и при необходимости устранить проблему
        throw std::string("Division by zero");

    return a / b;
}

int main() {
    int a, b;
    std::cin >> a >> b;

    try {
        // блок кода, в котором потенциально могут возникнуть исключения
        auto res = division(a, b);
        std::cout << res;
    } catch (const std::string &s) { // если ошибка произойдёт
        // перехваченное значение
        // запишется в s
        // переменной s можно как-то оперировать
        std::cerr << s;
    }

    return 0;
}
```

---

На самом деле, вы можете использовать более короткий пример, чтобы убедиться в том, что исключения перехватываются блоком `catch` (наберите, опишите поведение):

---

```
#include <iostream>

int main() {
    try {
        throw 42;
        // код ниже не выполнится
        // если код выше возбудил исключение
        std::cout << "Success";
    } catch (int a) {
        std::cerr << a;
    }

    return 0;
}
```

---

Механизм перехвата исключения следующий:

- Генерируется исключение.
- Точка выполнения немедленно переходит к ближайшему блоку `try` внутри которого располагался код, в результате которого произошла исключительная ситуация.
- Затем проверяются обработчики `catch` на соответствие типу исключения. Если обработчик найден, исключение обрабатывается, иначе передаётся дальше (функции, которая вызвала эту функцию, вдруг у неё есть соответствующий обработчик `catch`).
- На основании описания поведения абзацем выше объясните, каким образом работает данный фрагмент:

---

```
#include <iostream>

void f1() {
    throw std::string("ERROR!!!!");
}

void f2() {
    try {
        f1();
    } catch (int a) {
        std::cerr << "CATCH IN F2";
        std::cerr << "(" << a << ")";
    }
}

int main() {
    try {
        f2();
    } catch (std::string &s) {
        std::cerr << "CATCH IN MAIN";
        std::cerr << "(" << s << ")";
    }
}
```

---

```

    }

    return 0;
}

```

---

- Напишите фрагмент кода, который выбирает необходимый `catch` в зависимости от типа исключения (конструкцию с несколькими блоками `catch`).
- Напишите фрагмент кода в котором выкидывается исключение, но никак не обрабатывается. Опишите наблюдаемое поведение.
- Можно перехватывать исключения любого типа используя такой синтаксис:

---

```

try {
    // блок кода с потенциальной ошибкой
} catch (...) {
}

```

---

Сгенерируйте какое-нибудь исключение и проверьте, что решение через `...` работает.

- Оператор `throw` может и не содержать параметров. Он используется тогда, когда возникает необходимость прокинуть исключение дальше. В примере ниже функция `f()` с какой-то вероятностью не может выделить память. Предположим, если память не смогла быть выделена, наша программа должна освободить память под все ранее выделенные объекты, и при этом `main` должен узнать, что произошла ошибка:

---

```

#include <iostream>
#include <vector>
#include <ctime>

using namespace std;

int* getmem(float successProbability) {
    float r = static_cast <float> (rand()) / static_cast <float>
        (RAND_MAX);
    // имитируем работу системы:
    // с вероятностью 1 - successProbability выкидывается ошибка
    if (r > successProbability)
        // прокидываем исключение bad_alloc()
        throw bad_alloc();

    return new int[10];
}

// возвращает вектор из nParts фрагментов динамической памяти
// в случае возникновения ошибки освобождает память всем выделенным
// фрагментам;
// прокидывает исключение

```

---

```

vector<int*> getParts(int nParts, float successProbability) {
    vector<int*> parts;
    for (int i = 0; i < nParts; i++) {
        try {
            parts.push_back(getmem(successProbability));
        } catch (const bad_alloc &e) {
            clog << "free dynamic memory" << '\n';
            for (auto &part : parts)
                delete[] part;
            throw; // перекидываем исключение дальше
        }
    }
    return parts;
}

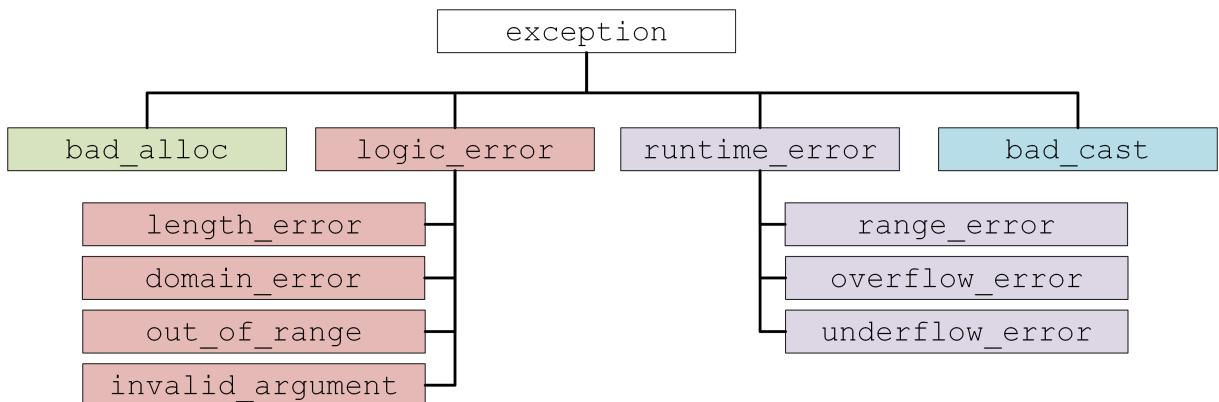
int main() {
    srand(time(nullptr));

    try {
        auto parts = getParts(10, 0.95);
        cout << "Success!";
    } catch (const exception &e) {
        // перехватываем исключение типа exception
        // - стандартный класс исключений C++
        // у стандартных исключений C++ имеется метод .what(),
        // позволяющий получить информацию от исключения
        cerr << e.what();
    }

    return 0;
}

```

- Как выше было замечено, перехватывалась переменная класса `exception`. Все стандартные исключения наследуются от него<sup>133</sup>:



<sup>133</sup> Приведенные на схеме исключения предоставляются языком программирования. При необходимости вы можете создавать и свои исключения (однако это выходит за рамки курса и будет рассматриваться в ООП).

Из этой схемы следует, что если вы выбросите исключение `length_error`, то оно может быть перехвачено в блоке `catch` переменными типа `logic_error` и `exception`:

---

```
#include <iostream>

int main() {
    try {
        throw std::logic_error("<Error description>");
    } catch (const std::exception &e) {
        std::cerr << e.what();
    }

    return 0;
}
```

---

Перехватите исключение `out_of_range` при помощи `logic_error`. Чтобы исключение сгенерировалось более осмысленно, создайте вектор размера `n`, и обратитесь к элементу вектора при помощи метода `.at()`, допуская выход за пределы массива. Напишите код таким образом, чтобы при каких-то значениях исходных данных исключение выбрасывалось, а при каких-то – нет.

Приложите написанный код, результат работы программы, когда исключение было и не было возбуждено.

В задании к пункту опишите, когда именно выбрасываются исключения классов, представленных на схеме.

- Пусть есть некоторая функция, которая возвращает фрагмент свободной динамической памяти:

---

```
int* getmem(size_t partSize) {
    return new int[partSize];
}
```

---

Напишите функцию

---

```
vector<int*> getParts(int nParts, size_t partSize)
```

---

которая по окончанию своей работы должна вернуть вектор указателей на выделенные фрагменты памяти. Если память выделить при помощи `new` не удалось, будет выброшен `bad_alloc`, который должен быть перехвачен в `main`.



5. Таблица умножения (39Н). Матрицу не хранить.
6. Рост прибыли компании (39В).
7. Фотограф (203С). Данная задача имеет тег "жадные алгоритмы". Из определения: **жадный алгоритм** – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным. Относительно этой задачи: обслуживайте клиентов по неубыванию затрат на их обслуживание. Прежде всего вычислите затраты на каждого из них, создайте вектор:

```
vector<pair<int, int>> clients(n);
```

где первым элементом пары будет выступать номер клиента, а вторым – затраты на него (несмотря на то, что лучше использовать структуры, при решении данной задачи используйте `std::pair`). В библиотеке `algorithm` определена функция `std::sort`, которая принимает итераторы на начало и конец последовательности и функцию-компаратор. В нашем случае будет полезен такой вариант использования:

```
sort(begin(clients), end(clients), [] (auto &x, auto &y){  
    return x.second < y.second;  
});
```

8. Часпитие (808С). Подход к решению данной задачи аналогичен прошлой задаче. Только в данном случае используйте структуру, похожую на:

```
struct Cup {  
    int index;           // номер кружки  
    int fill_in = 0;    // степень заполненности при создании  
                        // структуры (в данном случае 0)  
    int capacity;       // вместимость кружки  
};
```

9. Точка на спирали (279А). Промоделируйте процесс движения и считайте повороты по пути. Настоятельно рекомендуется нарисовать движение объекта на бумаге. Допускается сложность алгоритма по времени  $O(x \cdot y)$  (но может быть найдено решение за  $O(1)$ ).
10. Сделай квадрат (962С). Переберите все числа от 1 до  $\sqrt{n}$ . Найдите квадрат каждого числа и проверьте, может ли он быть получен путём удаления некоторых символов из исходной строки. Возможно, в решении задачи вам пригодится функция `to_string`.
11. Garbage Disposal (1070D). Лень – двигатель прогресса.
12. Котик-гурман (1154С).
13. Дилемма о доставке (1443С). Пригодится бинарный поиск в задачах оптимизации. Может быть решена и другим способом. Но если ответ легко может быть перебран бинарным поиском, пользуйтесь этим.

## 18.19 Лабораторная работа «Векторы»

**Цель работы:** получение навыков работы с векторами.

**Содержание отчета:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Исходный код, разработанных функций.
- Выводы.

### 18.19.1 Массивы в стиле C *std::array*

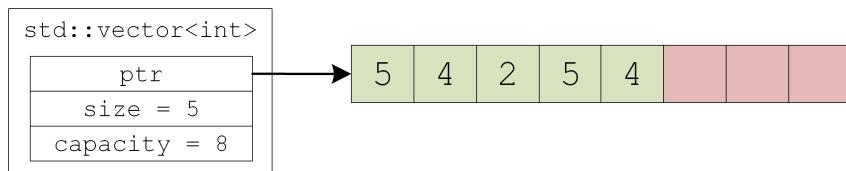
Простейший стандартный контейнер – класс `std::array<T, N>`, действующий как встроенный («C-подобный») массив, добавленный в C++11. Первый шаблонный параметр в `std::array` определяет тип, а второй – количество элементов массива. Это один из немногих классов в стандартной библиотеке, где шаблонный параметр является целым числом, а не именем типа:

```
std::array<int, 10> a{0}; // массив a из 10 элементов, инициализированных нулям
```

Основное отличие данного массива от встроенных, наличие большего количества синтаксического сахара (например, он имеет метод `size`, может быть возвращен из функции копированием, допустимо присваивание одного массива другому и т. п.). Но в связи с наличием более удачных альтернатив чаще используется `std::vector`.

### 18.19.2 Векторы *std::vector*

Вектор в C++ (C++03) — динамический массив, который может сам управлять выделенной себе памятью. Вектор может хранить произвольное количество значений базового типа. При необходимости он будет 'уничтожен' автоматически, и занимаемая им динамическая память освободится. Использование векторов делает код более безопасным, предотвращая утечки памяти.



Вектор состоит из трёх полей:

- `ptr` – указатель на нулевой элемент,
- `size` – фактическое количество полезных элементов в векторе,
- `capacity` – количество элементов, которые может хранить вектор без перераспределения динамической памяти.

Когда все элементы вектора заполнены, и возникает необходимость добавить ещё элемент, выполняется перераспределение памяти: выполняется попытка увеличить capacity в 1.5 / 2 раза.

Сложность операций над вектором:

- Произвольный доступ –  $O(1)$ ;
- Вставка в конец вектора –  $O(1)$ ;
- Удаление с конца вектора –  $O(1)$ ;
- Вставка или удаление элементов –  $O(n)$ .

### 18.19.3 Создание вектора

У вектора имеется несколько конструкторов:

---

```
#include <iostream>
#include <vector> // заголовочный файл, необходимый для работы с векторами

int main() {
    // Вектор из 10 элементов типа int
    // Элементы вектора будут инициализированы
    // значением по умолчанию
    std::vector<int> v1(10);

    // Пустой из элементов типа string
    std::vector<std::string> v2;

    // Вектор, состоящий из 10 элементов типа int
    // Все элементы заполняются нулями
    std::vector<int> v3(10, 0);

    // Вектор, состоящий из 5 элементов типа int
    // Инициализация при помощи списка инициализации (C++11)
    std::vector<int> v4 = {1, 5, 4, 2, 4};

    return 0;
}
```

---

Вектор – шаблонный класс. Для создания класса требуется указать базовый тип элементов в скобках `<>`.

В качестве базового типа вектора может выступать структура:

---

```
#include <iostream>
#include <vector> // заголовочный файл, необходимый для работы с векторами

struct Person {
    std::string name;
    std::string surname;
    int height;
    float weight;
};
```

---

```

int main() {
    // ...

    std::vector<Person> stuff = {
        {"Ivanov", "Ivan", 145, 45},
        {"Antonov", "Anton", 165, 55},
        {"Petrov", "Petr", 185, 75},
    };
}

```

---

или другой вектор:

```

std::vector<std::vector<int>> vs1;

std::vector<std::vector<int>> vs2 = {
    {1, 2, 3},
    {4},
    {5, 6, 7, 8, 9}
}

// создание вектора из 3 векторов размера 5,
// где каждый элемент проинициализирован нулём
std::vector<std::vector<int>> vs3(3, std::vector<int>(5));

// создание вектора из 3 векторов размера 20,
// где каждый элемент проинициализирован значением 10
std::vector<std::vector<int>> vs4(3, std::vector<int>(20, 10));

```

---

Векторы могут быть сравнены с использованием операции `==`:

```

std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = {3, 2, 1};
std::vector<int> v3 = v1;

bool f1 = v1 == v2; // false;
bool f2 = v1 == v3; // true;

```

---

Элементы одного вектора могут быть скопированы в другой вектор, при применении оператора присваивания:

```

std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2;
// v2 - копия вектора v1
v2 = v1;

```

---

Создать вектор можно из другого вектора или при помощи итераторов:

```

std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2(v1);
std::vector<int> v3(rbegin(v1), rend(v1)); // v3 = \{3, 2, 1\};

```

---

## 18.19.4 Ввод и вывод вектора

Ввод вектора можно реализовать через цикл `for`:

---

```
std::vector<int> v(5);
for (auto &x : v)
    std::cin >> x;
```

---

Цикл `for` пройдёт по всем элементам вектора, и так как `x` – ссылка на элемент массива, при осуществлении ввода `x` будут вводиться элементы вектора `v`. Данный способ необходим, если размеры вектора заранее известны. В противном случае можно использовать метод `push_back`, который выполнит вставку элемента в конец вектора. Ниже представлен листинг ввода вектора до первого нуля:

---

```
// ввод элементов до первого нуля
std::vector<int> v;
while (true) {
    int x;
    std::cin >> x;

    if (x != 0)
        v.push_back(x);
    else
        break;
}

// вывод элементов вектора
for (const auto &x : v)
    std::cout << x << ' ';
```

---

Вернувшись к примеру со студентами:

---

```
std::vector<Person> stuff = {
    {"Ivanov", "Ivan", 145, 45},
    {"Antonov", "Anton", 165, 55},
    {"Petrov", "Petr", 185, 75},
};

for (auto &person : stuff)
    std::cout << person.surname << ' ' << person.name << '\n' ;
```

---

Мы не можем выводить элементы вектора `v` как-нибудь так:

---

```
std::cout << v;
```

---

потому что неопределено, каким образом данные значения должны выводиться в поток `cout`.

Обращаться к элементам вектора можно по индексу:

---

```
std::vector<int> v = {5, 3, 5, 6};
std::cout << v[1]; // 3
```

---

```
v[0] = 0;
v[3] = 10; // v = {0, 3, 5, 10}
```

Если требуется выполнять проверку границ диапазона при обращении по индексу, используйте метод `at`. Он возвращает ссылку на элемент:

```
std::vector<int> v = {1, 2, 3};
v.at(10) = 15; // попытка присвоить 10-ому элементу значение 15
```

Посредством языка будет 'выкинуто' исключение:

```
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check: __n (which is 10) >= this->size() (which is 3)
```

## 18.19.5 Методы для работы с вектором

Методы для получения информации о размере и ёмкости:

- `size` – получение количества элементов в векторе:

```
std::vector<int> v = {1, 5, 4};
std::cout << v.size(); // 3
```

- `empty` – возвращает значение 'истина', если вектор пуст, иначе – 'ложь':

```
std::vector<int> v = {1, 5, 4};
std::cout << v.empty(); // 0
```

Вместо:

```
if (!v.size()) {
}
```

используйте:

```
if (v.empty()) {
}
```

- `capacity` – получение ёмкости вектора (количество элементов в выделенном динамическом массиве для его хранения):

```
std::vector<int> v = {1, 5, 4};
v.push_back(10);
std::cout << v.size() << '\n'; // 4
std::cout << v.capacity() << '\n'; // 6
```

Можно заметить, что выделение происходит 'с запасом', чтобы исключить по-элементное выделение памяти.

- `shrink_to_fit` - удаляет лишнюю ёмкость (обрезает динамический массив под хранение необходимых элементов).
- `resize` – изменяет размер вектора. Если новый размер больше старого – инициализирует новые значения вектора значениями по умолчанию, если размер меньше – обрезает существующий вектор:

```
std::vector<int> v = {1, 5, 4};
v.resize(2); // Удалён последний элемент: v = \{1, 5\}
v.resize(4); // Добавлено два элемента,
             // инициализированных по умолчанию: v = \{1, 5, 0, 0\}
```

Важно отметить, что старые элементы остались без изменений.

- `reserve` – резервирует под вектор определённое количество элементов **без инициализации**:

```
std::vector<int> v;
v.reserve(10);
```

Методы для модификации вектора:

- `push_back` – добавление элемента в конец вектора:

```
std::vector<int> v = {1, 5, 4};
v.push_back(10); // v = \{1, 5, 4, 10\}
```

- `emplace_back` – шаблонная функция идеальной передачи с переменным числом аргументов, которая действует подобно `push_back(t)`, но помещает в конец вектора не копию `t`, а создает объект типа `T`, как если бы был вызван конструктор `T(args...)`.

- `pop_back` – удаляет последний элемент вектора:

```
std::vector<int> v = {1, 5, 4};
v.pop_back(); // v = \{1, 5\}
```

- `insert` – вставка элементов в вектор:

```
std::vector<int> v = {1, 5, 4};
// вставка на первую позицию значения 6
v.insert(begin(v) + 1, 6);
// \{1, 5, 4\} -> \{1, 6, 5, 4\}

// вставка на нулевую позицию значений 1, 2, 3
v.insert(begin(v), {1, 2, 3});
```

```
// \{1, 6, 5, 4\} -> \{1, 2, 3, 1, 6, 5, 4\}

// вставка на нулевую позицию элементов v2[1..3]
std::vector<int> v2 = {5, 4, 3, 2, 1};
v.insert(begin(v), begin(v2) + 1, end(v2) - 1);
// \{1, 2, 3, 1, 6, 5, 4\} -> \{4, 3, 2, 1, 2, 3, 1, 6, 5, 4\}

// вставка на первую позицию трёх нулей
v.insert(begin(v) + 1, 3, 0);
// \{4, 3, 2, 1, 2, 3, 1, 6, 5, 4\} -> \{4, 0, 0, 0, 3, 2, 1, 2, 3, 1, 6,
→ 5, 4\}
```

По возможности старайтесь избегать `insert`, так как это приведёт к сдвигу элементов вектора для освобождения места для вставляемых элементов.

- `erase` – удаление элементов из вектора:

```
std::vector<int> v = {6, 5, 4, 3, 2, 1};

// Удаление второго элемента массива
// v = \{6, 5, 3, 2, 1\}
v.erase(begin(v) + 2);

// Удаление элементов с 2 по 4 (не включая)
// v = \{6, 5, 1\}
v.erase(begin(v) + 2, begin(v) + 4);
```

- `clear` – удаляет все элементы из вектора;
- `swap` – обмен значениями между двумя векторами:

```
std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = {3, 2, 1};
v1.swap(v2); // v1 = \{3, 2, 1\}, v2 = \{1, 2, 3\}
```

Методы для осуществления доступа к элементам:

- `front` – возвращает ссылку на первый элемент вектора:

```
std::vector<int> v = {1, 5, 4};
std::cout << v.front(); // 1
v.front() = 100; // v = \{100, 5, 4\}
```

- `back` – возвращает ссылку на последний элемент вектора:

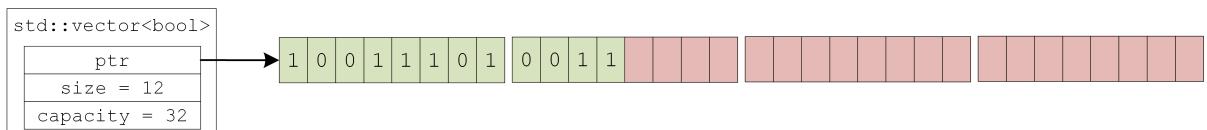
```
std::vector<int> v = {1, 5, 4};
std::cout << v.back(); // 4
v.back() = 100; // v = \{1, 5, 100\}
```

- `at` – возвращает ссылку на  $i$ -ый элемент вектора. Дополнительно проверяется корректность индекса.
- `data` – возвращает указатель на нулевой элемент вектора.

Итераторы: `begin / end`, `cbegin / cend`, `rbegin / rend`, `crbegin / crend`.

### 18.19.6 `std::vector<bool>`

В силу того, что для представления логического типа достаточно одного бита, набор значения из `n` переменных `bool` потребует `n` бит:




---

```
std::vector<bool> a = {
    true, false, false, true, true, true, false, true,
    false, false, true, true
};

std::cout << a.size();      // 12
std::cout << a.capacity(); // 32
```

---

#### Задания к лабораторной работе:

Разминочные задачи<sup>134</sup>:

1. Еда в самолёте (725B).
2. Робот-пылесос (1623A). Смоделируйте движение робота. Вероятно, такое решение будет проще, чем через набор `if-else`.
3. Спасите задачу! (865A).
4. Происшествие в клубе (245E).
5. Габриэл и гусеница (652A). Допускается решение через моделирование процесса движения.

Векторы:

1. Сортировка частей (1637A).
2. Каникулы (44C).
3. Гадание (59B).
4. Социальная дистанция (1668B). Изначально рассаживайте людей, которые требуют большей изоляции. Можно найти решение, которое не использует векторы. Сложность по времени:  $O(n)$ . Сложность по памяти:  $O(1)$ .

<sup>134</sup>Только некоторые связаны с темой лабораторной работы. Руководствуйтесь принципом *KISS*

5. Деревья в ряду (402В). Перебирайте величину первого дерева. Высоты оставшихся деревьев можно вычислить. Найдите оптимальную высоту первого дерева. Это поможет получить требуемый ответ.
6. Два массива (1584С).
7. Красные против синих (1659А). Для вывода  $n$  одинаковых символов можно использовать следующий конструктор строк:

---

```
// выведем 10 символов R
cout << string(10, 'R');
```

---

Строки и векторы имеют схожий интерфейс. Многое, что вы можете делать со векторами, вы можете делать со строками.

8. Считалка (792В). Задача на использование функции `erase`.
9. Om Nom and Spiders (436В). Потребует использования вектора строк. Например, чтение исходных данных:

---

```
int n, m, k;
cin >> n >> m >> k;

vector<string> a(n);
for (auto &x : a)
    cin >> x;
```

---

10. Простая матрица (271В). В реализации потребуется использовать алгоритм решета Эратосфена. Изучите и реализуйте алгоритм. Получите вектор из логических значений, где  $i$ -ый элемент является истиной, если  $i$  – простое число.
11. Очередь за чаем (920В)

## 18.20 Лабораторная работа «Стек, очередь, дек»

**Цель работы:** получение навыков работы со стеком, очередью, деком. Изучение сторонних источников стандартной библиотеки *STL*.

**Содержание отчета:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Тексты заданий и их решения.
- Словесное описание решения каждой из задач.
- Выводы.

### 18.20.1 Деки `std::deque`

Дек – структура данных, двусторонняя очередь (*double – ended – queue*), позволяющая осуществлять вставку и удаление как с начала коллекции, так и с её конца.

Хранилище дека автоматически расширяется и сокращается по мере необходимости. Расширение двухсторонней очереди дешевле, чем расширение `std::vector`, потому что оно не требует копирования существующих элементов в новое место в памяти.

Сложность операций над деком `deque<T>` библиотеки *STL*:

- Произвольный доступ –  $O(1)$ ;
- Вставка или удаление элементов в конце или начале –  $O(1)$ ;
- Вставка или удаление элементов в отличных от крайних элементов –  $O(n)$ .

#### Методы для работы с деком

Методы для получения информации о размере и ёмкости:

- `size` – получение количества элементов в деке.
- `empty` – возвращает значение 'истина', если вектор пуст, иначе – 'ложь'.
- `shrink_to_fit` – освобождает лишнюю память, выделенную под хранение потенциальных элементов.
- `resize` – изменяет размер дека.

Методы для модификации дека:

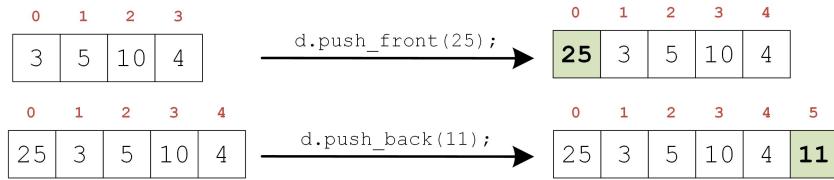
- `push_front, emplace_front / push_back, emplace_back` – добавление элемента в начало / конец дека:

---

```
std::deque<int> d = {3, 5, 10, 4};

d.push_front(25);
d.push_back(11);
```

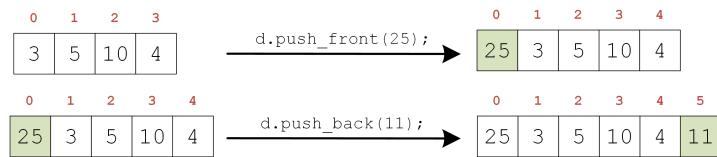
---



- `pop_front` / `pop_back` – удаляет первый / последний элемент дека:

```
std::deque<int> d = {3, 5, 10, 4};

d.pop_back();
d.pop_front();
```



- `insert` – вставка элементов в дека.
- `erase` – удаление элементов из дека.
- `clear` – удаляет все элементы из дека.
- `swap` – обмен значениями между двумя деками.

Методы для осуществления доступа к элементам:

- `front` / `back` – возвращает ссылку на первый / последний элемент дека.
- `at` – возвращает ссылку на  $i$ -ый элемент дека. Дополнительно проверяется корректность индекса.

Итераторы: `begin` / `end`, `cbegin` / `cend`, `rbegin` / `rend`, `crbegin` / `crend`.

Ссылки на документацию: EN, RU.

### 18.20.2 Списки `std::list`, `std::forward_list`

Списки являются концептуально интересной, но редко используемой структурой данных из-за своей недружественности к кешу процессора. Рассмотрим идею списка. Для примера возьмём следующий:

```
std::list<int> x = {5, 14, 552};
```

Список представляет из себя множество узлов, расположенных в различных участках памяти (в куче):

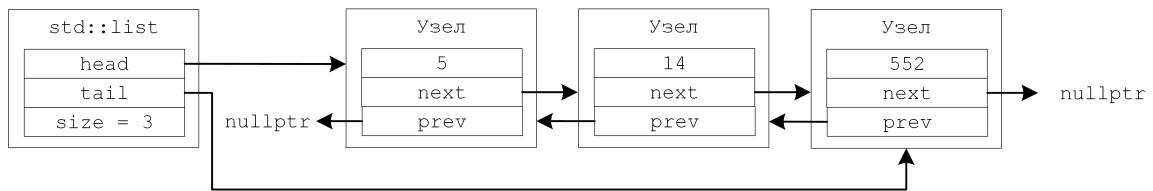
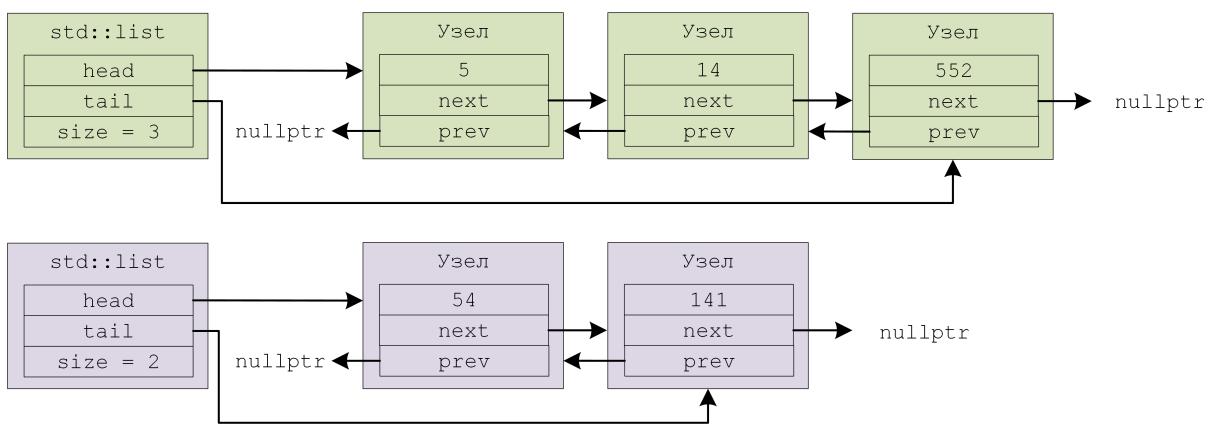
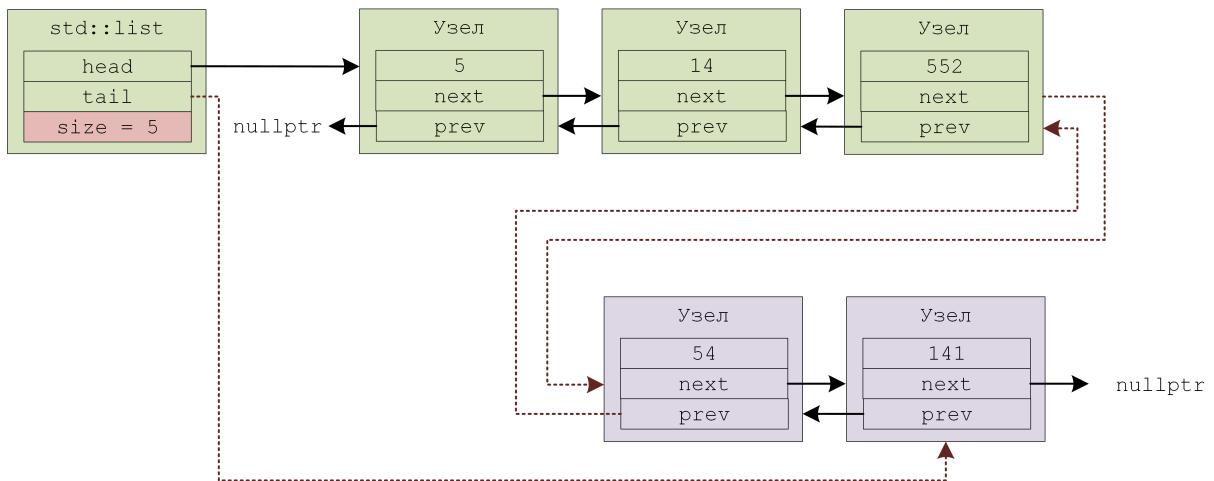


Рис. 18.26: Двусвязный список

Обращение к необходимому элементу за  $O(1)$  невозможно, если это не хвост или голова. Во всех остальных случаях потребуется время, чтобы от одного элемента переходить к другому. Скачки по разным фрагментам динамической памяти делают алгоритм недружественным к кешу. Следовательно время работы заметно увеличивается при оперировании с элементами списка. Более того, требуется дополнительная память под хранение указателей. Преимуществом данной структуры является быстрая процедура объединения двух списков. Пусть даны списки:



При объединении достаточно лишь перенацелить указатели и просуммировать размеры:



Её сложность составляет  $O(1)$ :

---

```

std::list<int> x = {5, 14, 552};
std::list<int> y = {54, 141};

x.splice(x.end(), y);
    
```

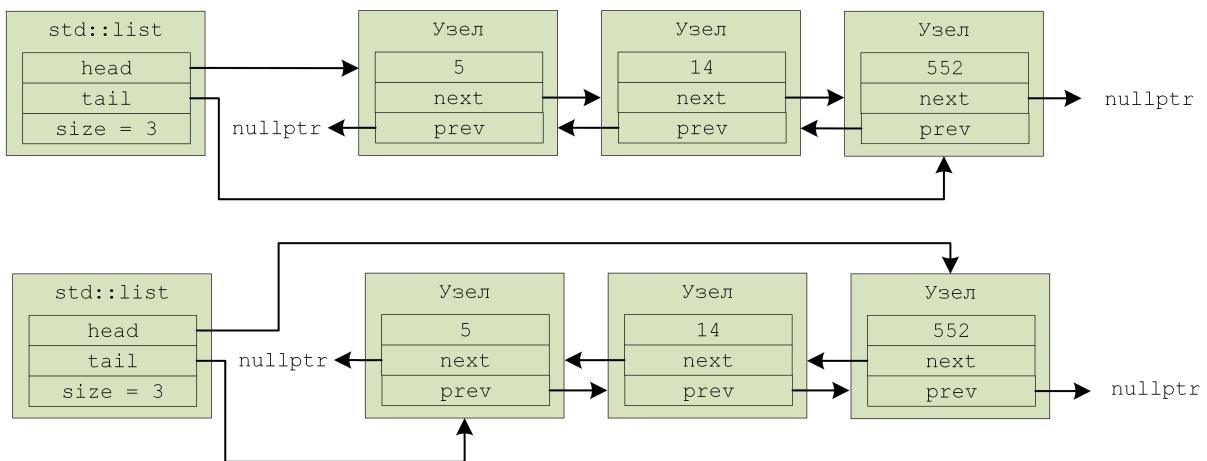
---

---

```
std::cout << x.size() << '\n'; // 5
std::cout << y.size() << '\n'; // 0
```

---

Процедура изменения порядка элементов двусвязного списка на обратный тоже проста: если изменить для каждого элемента указатель на следующий и предыдущий значениями и обменять голову и хвост – список будет перевёрнут:




---

```
std::list<int> x = {5, 14, 552};
x.reverse();
```

---

Операция удаления элементов стоит недорого: как минимум она не требует дополнительных перемещений неудаляемых элементов.

### Методы для работы с двусвязным списком

- `lst.remove(v)` – удаляет и стирает все элементы, равные `v`.
- `lst.remove_if(p)` – удаляет и стирает все элементы `e`, удовлетворяющие предикату `p(e)` с одним аргументом.
- `lst.unique()` удаляет и стирает все элементы, кроме первого в каждой группе идущих подряд уникальных элементов. Существует версия с компаратором: `lst.unique(p)`, удаляющая и стирающая элементы `e2`, для которых `p(e1, e2)` возвращает `true`.
- `lst.sort()` сортирует список на месте. Это особенно полезная операция, потому что алгоритм `std::sort(ctr.begin(), ctr.end())` не работает с контейнерами не поддерживающим произвольный доступ. Недоступной на данный момент является сортировка поддиапазона. Приходится выполнять следующий трюк:

---

```
#include <iostream>
#include <list>
#include <algorithm>

int main()
{
    std::list<int> x = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

---

```

        auto startSortIterator = std::next(begin(x), 2);
        auto endSortIterator = std::prev(end(x), 2);

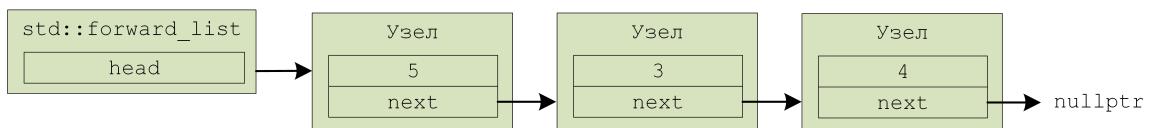
        std::list<int> tmpList;
        tmpList.splice(begin(tmpList), x, startSortIterator,
                       endSortIterator);
        tmpList.sort();
        // вставка в splice осуществляется до позиции endSortIterator
        x.splice(endSortIterator, tmpList);

        for (auto &a : x)
            std::cout << a << ' ';
    }

    return 0;
}

```

`std::forward_list` представляет собой односвязный список, но у него нет возможности получить размер и выполнить обход элементов в обратном направлении в силу внутренней структуры:



Функции `push_back` и `splice` недоступны, так как для первой нужно постоянно искать хвост, в функция `splice` выполняет вставку до переданного ей итератора, а элемент списка не знает, что именно находится до него. Добавлены альтернативные функции:

- `flst.erase_after(it)` – стирает элемент, следующий за указанным;
- `flst.insert_after(it, v)` – вставляет новый элемент вслед за указанным;
- `flst.splice_after(it, otherflst)` – вставляет элементы из `otherflst` вслед за указанным.

Ссылки на документацию: EN, RU.

### 18.20.3 Стек `std::stack`

Стек – структура данных, поддерживающая следующий набор операций:

- `top()` – получение элемента с 'вершины' стека;
- `pop()` – удаление элемента с 'вершины' стека;
- `push(v)` – добавление элемента на 'вершину' стека.

Уже было рассказано о трёх структурах данных, которые и так умеют делать данные операции:

- вектор (`std::vector`);

- дек (`std::deque`);
- двусвязный список (`std::list`).

Поэтому при создании стека можно задать структуру данных, на которой будет реализован стек:

---

```
#include <iostream>
#include <stack>
#include <vector>
#include <list>

int main()
{
    // стек на векторе
    std::stack<int, std::vector<int>> vectorStack;

    // стек на двусвязном списке
    std::stack<int, std::list<int>> listStack;

    // стек на деке
    std::stack<int, std::deque<int>> dequeStack;

    return 0;
}
```

---

Если вариант явно не указан, в качестве контейнера будет выбран дек<sup>135</sup>.

Решим какую-нибудь небольшую задачу с использованием стека. Например вводится строка, состоящая из различных скобок. Требуется проверить, действительно ли последовательность скобок корректна. Например, `()[(()[])]` – корректная скобочная последовательность, а вот `([]())` – нет. В решении будем исходить из предположения, что других скобок в записи нет:

---

```
#include <iostream>
#include <stack>

int main() {
    std::string bracketsSequence;
    std::cin >> bracketsSequence;

    std::stack<char> charStack;
    bool isGoodBracketsSequence = true;
    for (auto c : bracketsSequence) {
        if (c == '(' || c == '[')
            charStack.push(c);
        else if (!charStack.empty() && (
                    charStack.top() == '(' && c == ')') ||
                  charStack.top() == '[' && c == ']')) {
            charStack.pop();
        } else {
            isGoodBracketsSequence = false;
        }
    }
}
```

---

<sup>135</sup>Как по мне – гениальный ход. Берём дек, запрещаем ему вставку и удаление с начала контейнера и получили стек!

```

        break;
    }
}

isGoodBracketsSequence &= charStack.empty();

std::cout << isGoodBracketsSequence;

return 0;
}

```

---

Правила обработки простые:

- Если скобка открывающаяся, помещаем на вершину стека.
- Если скобка закрывающаяся, то возможны следующие исходы:
  - Стек пустой. Следовательно, не было открывающей скобки к этой закрывающей (пример последовательностей: `]`, `[]()`).
  - Стек непустой, но там лежит открывающая скобка другого типа (пример последовательностей: `[]`, `[]([])`).
  - Стек непустой, и на вершине лежит открывающая скобка такого же типа. Удаляем скобку из стека.

В конце последовательности надо убедиться, что стек пуст. Иначе валидными окажутся последовательности: `(`, `(([])` и другие.

Стек является одной из наиболее часто используемых структур данных на практике. Уделите больше времени его пониманию.

Ссылки на документацию: EN, RU.

#### 18.20.4 Очередь `std::queue`

Принцип работы данной структуры интуитивно понятен, ввиду хорошего отображения имени структуры данных на процессы, которые там происходят. Представим себе очередь из людей. Если в очередь кто-то добавится, он встанет в конец очереди (в 'хвост')<sup>136</sup>. Если человек обслужен кассой, он покидает очередь с 'головы'<sup>137</sup>.

Запросы используемые в очереди:

- Удаление с начала –  $O(1)$ ;
- Встака в конец –  $O(1)$ .

В качестве контейнера, на котором будет реализована очередь без указания шаблонного параметра типа контейнера в библиотеке *STL*, будет использован дек.

Ссылки на документацию: EN, RU.

Слово *queue* иронично: это простое '*Q*' с группой непроизносимых букв, которые ожидают в очереди.

<sup>136</sup>Тут не действует правило 'Я только спросить'.

<sup>137</sup>Фраза "Свободная касса!" в Пятёрочке превращает очередь в дек.

### 18.20.5 Приоритетные очереди `std::priority_queue`

Теперь предположим, что элементы, поступающие в очередь, неравнозначны. Обслуживать мы должны не в порядке прихода, а на основании какого-то показателя. Например, имеется список дел. Есть дела более важные. Есть менее важные. Если поступает более важное дело, нам следует выполнить его раньше. Приоритетная очередь под капотом реализована на СД кучи. Следовательно, все операции, такие как вставка и удаление, работают со сложностью  $O(\log n)$ .

---

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(1);
    pq.push(3);
    pq.push(2);

    std::cout << pq.top() << '\n'; // 3;
    pq.pop();
    std::cout << pq.top() << '\n'; // 2;
    pq.pop();
    std::cout << pq.top() << '\n'; // 1;
    pq.pop();

    return 0;
}
```

---

Можно реализовать и другой компаратор, и создавать приоритетные очереди, использующие произвольный тип данных:

---

```
#include <iostream>
#include <queue>

struct Student {
    std::string name;
    int score;
};

int main() {
    auto compFunc = [] (Student &s1, Student &s2){
        return s1.score > s2.score;
    };

    std::priority_queue<Student, std::vector<Student>,
        decltype(compFunc)> pq(compFunc);

    pq.push({"Ivanov", 5});
    pq.push({"Petrov", 4});
    pq.push({"Sidorov", 3});

    auto toExpel = pq.top(); // получение наибольшего элемента
    pq.pop(); // выталкивает наибольший элемент
```

---

```
// на основании функции сравнения
std::cout << toExpel.name << ' ' << toExpel.score << '\n';

pq.push({"Antonov", 3});
while (!pq.empty()) {
    toExpel = pq.top();
    pq.pop();
    std::cout << toExpel.name << ' ' << toExpel.score << '\n';
}
std::cout << "Hooray! All are expelled!";

return 0;
}
```

Программа выведет:

---

```
Sidorov 3
Antonov 3
Petrov 4
Ivanov 5
Hooray! All are expelled!
```

---

Ссылки на документацию: EN, RU.

## Задания к лабораторной работе

- Стек

- Игра со строкой (1104В) Вам может быть полезен стек символов. Считайте исходные данные в строку и оперируйте стеком далее:

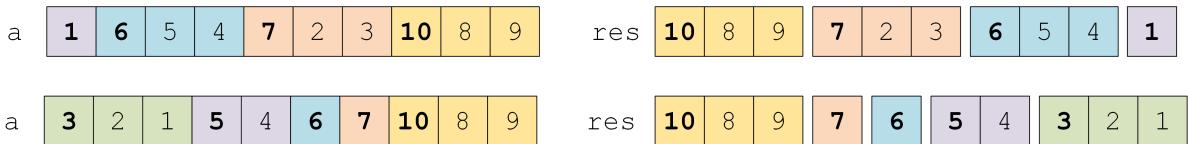
---

```
string s;
cin >> s;

stack<char> st;
```

---

- ABBB (1428С)
- Спасти от камней (264А). Вектор не использовать. Минимизировать расходы по памяти.
- Колода карт (1492В). Описание задачи довольно тяжелое. Попытаюсь объяснить чуть проще, что именно от вас хотят: необходимо получить такую последовательность карт, чтобы как можно больше элементов шли в последовательности раньше. Однако брать сверху колоды вы можете 'кусками' последовательности. Для лучшего понимания обратитесь к рисунку:



- Дек

- Социальная сеть (1234В1) При реализации алгоритма вам потребуется проверить, имеется ли некоторый элемент в деке. Для этого потребуется использование функции `find` из стандартной библиотеки `<algorithm>`:

---

```
// возвращает значение 'истина', если элемент не найден в деке
find(a.begin(), a.end(), value) == a.end()
```

---

При желании вы можете решить более сложную версию данной задачи: Социальная сеть (1234В2), однако узким местом нашего алгоритма будет выступать поиск в деке, который работает за линейное время. Решить данную проблему можно с использованием контейнера `set`, поиск в котором работает за  $O(\log N)$ .

- Настольный теннис (879В) Промоделируйте процесс. Создайте структуру игрок и дек, содержащий игроков:

---

```
struct Gamer {
    int power;
    int nWins;
};

int main() {
```

---

```
// ...
deque<Gamer> D;
// ...
}
```

---

- Приоритетная очередь

1. Снековик (767A)
2. Фото на память (522B). На самом деле, буду честен: задача решается без всяких приоритетных очередей. Там потребуется поиск двух максимальных значений. Но давайте попробуем обобщить частный случай. Предположим, имеется вектор, содержащий элементы произвольного типа. Необходимо из вектора выбрать  $K$  наибольших элементов. Решение в лоб: отсортировать коллекцию размера  $N$  и выбрать  $K$  наибольших. Сложность решения:  $O(N \log N)$ . Но можно поступить иначе: будем поддерживать  $K$  элементов в приоритетной очереди. Сложность вставки элемента в приоритетную очередь и удаления из неё  $O(\log N)$ , где  $N$  – количество элементов в очереди.

Рассмотрим худший случай, массив был изначально отсортирован. Тогда общие затраты на обработку  $N$  элементов для поиска  $K$  максимальных составят:  $O(N \log K)$ . Для случаев, когда  $K \ll N$ <sup>138</sup> это может давать ощутимый выигрыш.

Опишем заголовок функции:

---

```
template<typename T>
vector<T> getNMaxInSequence(const vector<T> &a, size_t k) {
    assert(k <= a.size());

    // создание приоритетной очереди, в которой
    // выталкивается элемент с наименьшим значением
    priority_queue<int, vector<int>, greater<> q;

    // замалкиваем первые k элементов вектора
    // для всех i ∈ {k, k + 1, ..., n - 2, n - 1}
    // - сравнить наименьший элемент кучи с a[i]
    // - если a[i] окажется больше, 'вытащить'
    //   наименьший элемент кучи и вставить текущее a[i]

    // пока куча не пуста
    // - вытолкнуть все элементы из кучи в вектор
}
```

---

3. Автобус характеров (982B) Вы можете создавать приоритетные очереди, элементами которых являются структуры:

---

<sup>138</sup>  $a \ll b$  — означает, что величина  $a$  намного меньше, чем  $b$ .

---

```

struct Place {
    int width;
    int number;
};

int main() {
    // требует задания функции-компаратора
    // (каким образом очередь будет понимать, какое место
    // → приоритетнее)
    auto compare = [](Place a, Place b) {
        return a.width > b.width;
    };

    priority_queue<Place, std::vector<Place>,
                  decltype(compare)>
        introvert_places(compare);
}

```

---

4. Феникс и башни (1515С). Алгоритм решения данной задачи относится к классу жадных. Нам достаточно собирать башни из предварительно отсортированных блоков. Создайте структуры:

---

```

struct Block {
    size_t index;
    int value;
};

struct Tower {
    size_t index;
    int size = 0;
};

```

---

Выстроим блоки по убыванию высот. Поочередно будем добавлять блок к той башне, высота которой минимальна. Ознакомьтесь с примером и поймите, как формируется результат (рисунок 18.27).

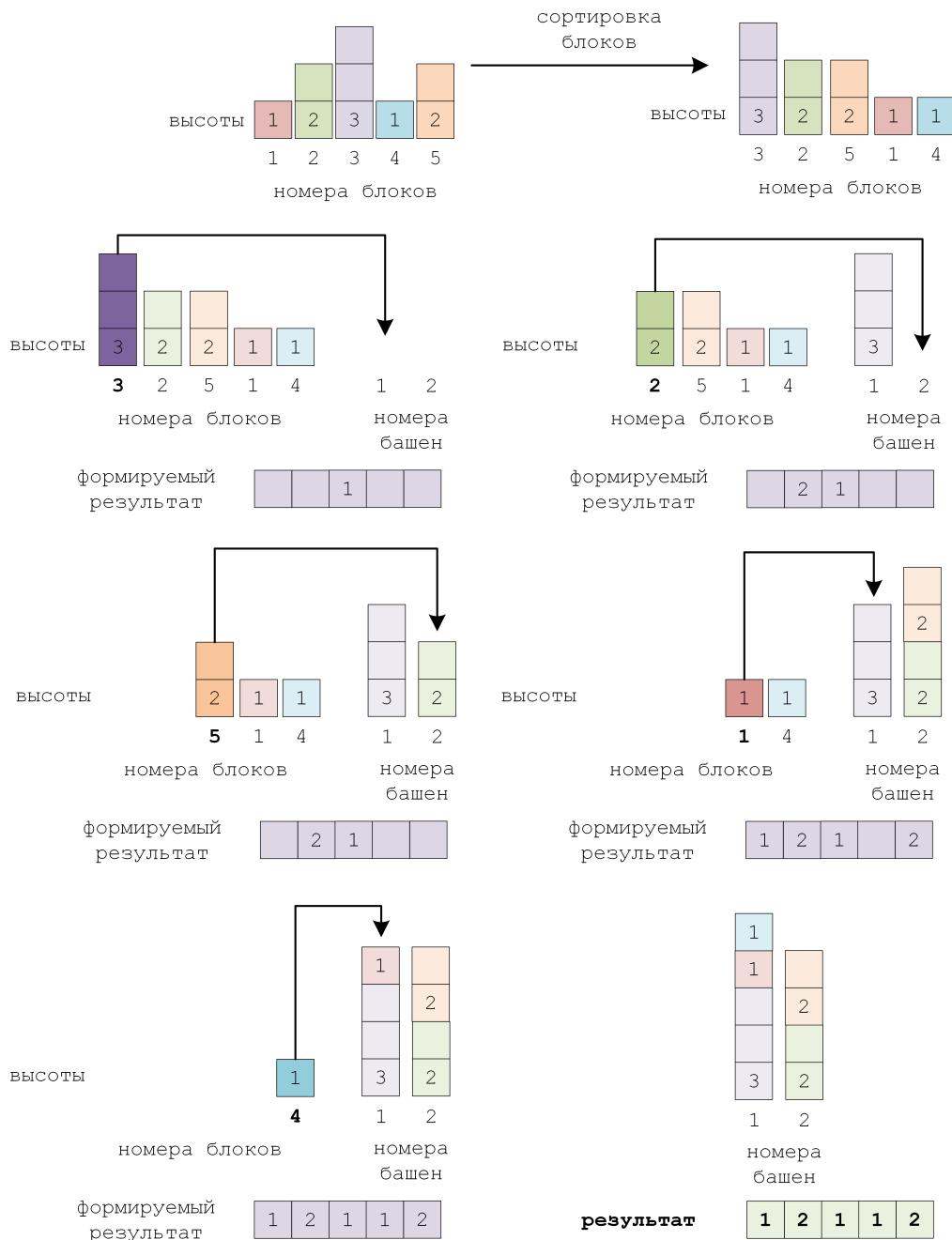


Рис. 18.27: Решение задачи о башнях

Вам может быть полезен фрагмент:

```

vector<Block> a(n);
for (size_t i = 0; i < n; i++) {
    cin >> a[i].value;

    a[i].index = i;
}

// сортировка блоков в
auto blockCmp = [] (const Block& lhs, const Block& rhs) {
    return lhs.value < rhs.value;
};

```

```
sort(rbegin(a), rend(a), blockCmp);

// данный компаратор будет выталкивать из очереди
// наибольший элемент
auto towerCmp = [] (Tower a, Tower b) {
    return a.size > b.size;
};

priority_queue<Tower, vector<Tower>,
               decltype(towerCmp)> p(towerCmp);
```

---

## 18.21 Лабораторная работа «Структуры в стиле C++»

**Цель работы:** получение навыков работы со структурами.

**Содержание отчета:**

- Тема лабораторной работы.
- Цель лабораторной работы.
- Исходный код, разработанных функций.
- Выводы.

**Задания к лабораторной работе:**

Одна из самых фундаментальных работ. Качество её понимания критически скажется на усвоении курса далее.

### 18.21.1 Причины появления структур в языках программирования

Рассмотрим следующую задачу. Требуется написать функцию для вывода информации о городе. Пусть каждый город характеризуется своим именем (`std::string`) и численностью населения (`unsigned long long`). Опишем функцию вывода:

---

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

void printCity(const std::string &name, unsigned long long population) {
    std::cout << "Name of city: " << name << "\n"
                  "Population: " << population;
}

int main() {
    printCity("Veliky Ustyug", 31078);

    return 0;
}
```

---

На экране отобразится:

---

```
Name of city: Veliky Ustyug
Population: 31078
```

---

Усложним задачу, теперь бы мы хотели выводить информацию о городах. Добавится функция `printCities`:

---

```

void printCities(const std::vector<std::string> &names,
                const std::vector<unsigned long long> &populations) {
    for (int i = 0; i < names.size(); i++)
        printCity(names[i], populations[i]);
}

```

---

Основные недостатки:

- Если город будет характеризоваться ещё каким-нибудь параметром (например, координатой), придётся внести изменения в функции `printCity` и `printCities`. С ростом числа характеристик проблема усугубится.
- Ей можно 'подсунуть' два вектора разных размеров.

Эти недостатки решались при помощи структур:

---

```

#include <iostream>
#include <string>
#include <vector>

struct City {
    std::string name;
    unsigned long long population;
};

void printCity(const City &city) {
    std::cout << "Name of city: " << city.name << "\n"
            "Population: " << city.population << "\n";
}

void printCities(const std::vector<City> &cities) {
    for (auto &city : cities)
        printCity(city);
}

int main() {
    std::vector<City> cities = {
        {"Veliky Ustyug", 31078},
        {"Belgorod", 369815},
        {"Moscow", 12632409},
    };

    printCities(cities);

    return 0;
}

```

---

Структуры как раз и позволяют выполнить создание пользовательского типа данных (тип `City` в примере). Как и любой другой тип, они могут представлять возвращаемый тип функции:

---

```

City getCapitalOfRussia() {
    // конструирование структуры при возврате из функции
    return {"Moscow", 12632409};
}

int main() {
    City city = getCapitalOfRussia();

    return 0;
}

```

---

Структуры могут использоваться в качестве поля другой структуры:

---

```

struct Country {
    std::string name;
    std::vector<City> cities;
};

```

---

Задания по пункту:

- Создайте произвольную структуру.
- Реализуйте функции ввода и вывода вектора структур.
- Придумайте произвольную задачу, в которой потребовалось бы обработка вектора структур. В качестве ответа укажите задание и его решение.

### 18.21.2 Ограничение доступа к полям структуры

Рассмотрим такую задачу: пусть между некоторыми населенными пунктами проекладываются маршруты (мы несколько "оторвёмся" от прошлой задачи не будем использовать структуру `City`). А расстояние будет вычисляться как сумма длин названий городов. Объявим структуру `Route`, опишем функцию `getDistance` для вычисления расстояния между ними:

---

```

#include <iostream>
#include <string>

struct Route {
    std::string source;
    std::string destination;
    unsigned distance;
};

unsigned getDistance(const std::string &lhs, const std::string &rhs) {
    return lhs.size() + rhs.size();
}

int main() {
    auto aName = "Moscow";
    auto bName = "Belgorod";
    Route route = {aName, bName, getDistance(aName, bName)};

```

---

```

    std::cout << route.distance; // 14

    return 0;
}

```

---

Основной недостаток данного подхода – имеется возможность легко поменять любую из точек маршрута и при этом расстояние не будет пересчитано:

```

auto aName = "Moscow";
auto bName = "Belgorod";
Route route = {aName, bName, getDistance(aName, bName)};

std::cout << route.distance << '\n'; // 14
route.source = "Saint Petersburg";
std::cout << route.distance << '\n'; // 14

```

---

Таким образом, обращение к некоторым полям структуры является нежелательным. Не сохраняются некоторые ограничения относительно значений полей класса. В таком случае говорят, что нарушается **инвариантность структуры** (нам было обещано, что расстояние вычисляется как сумма длин названий городов, но на деле это не так.). Мы ещё вернёмся к этому проблеме.

Небольшое отступление к философии ООП: считается, что освоение объектно-ориентированных концепций не сводится к изучению конкретного метода разработки, языка программирования или набора инструментов проектирования. В качестве ключевого здесь использовано слово «концепции», а не «технологии». Технологии в индустрии программного обеспечения очень быстро изменяются, в то время как концепции эволюционируют.

Исторически сложилось так, что объектно-ориентированные языки определяются следующими концепциями: инкапсуляцией, наследованием и полиморфизмом. Поэтому если тот или иной язык программирования не реализует все эти концепции, то он, как правило, не считается объектно-ориентированным. Данные концепции будут описаны подробно позже.

Прежде чем мы углубимся в преимущества объектно-ориентированной разработки, рассмотрим более существенный вопрос: что такое объект? Например, когда вы смотрите на какого-то человека, вы видите его как объект. При этом объект определяется двумя компонентами: атрибутами и поведением. У человека имеются такие атрибуты, как цвет глаз, возраст, вес и т. д. Человек также обладает поведением, то есть он ходит, говорит, дышит и т. д. В соответствии со своим базовым определением, **объект** – это сущность, одновременно содержащая данные и поведение. Слово одновременно в данном случае определяет ключевую разницу между объектно-ориентированным программированием и другими методологиями программирования.

При структурном программировании данные зачастую отделяются от процедур и являются глобальными, благодаря чему их легко модифицировать вне области видимости вашего кода. Это означает, что доступ к данным неконтролируемый и непредсказуемый (то есть у множества функций может быть доступ к глобальным данным). Например, на языке C промоделировать поведение собаки можно было бы так:

---

```

#include <stdio.h>
#include <string.h>
#include <stdbool.h>

typedef struct Dog {
    char name[100];
    bool isAggressive;
} Dog;

Dog createDog(char *name, bool isAggressive) {
    Dog d;

    strcpy(d.name, name);
    d.isAggressive = isAggressive;

    return d;
}

void bark(Dog d) {
    printf("%s says: woof-woof\n", d.name);
}

void wag(Dog d) {
    printf("%s wags its tail\n", d.name);
}

void responseToFirstMeeting(Dog d) {
    if (d.isAggressive) {
        bark(d);
        bark(d);
        printf("Stranger runs away");
    } else {
        wag(d);
        bark(d);
        wag(d);
    }
}

int main() {
    char name[100];
    gets(name);

    Dog d1 = createDog(name, 0);
    responseToFirstMeeting(d1);

    gets(name);
    Dog d2 = createDog(name, 1);
    responseToFirstMeeting(d2);

    return 0;
}

```

---

Вы можете заметить, что в данном случае поведение (в виде функций) оторвано от некоторой абстрактной собаки, которая выполняет данное действие. Каждая из поведенческих функций явно ждёт от нас, что получит на вход некоторое животное, и будет им воспроизводить поведение.

Более естественным смотрится ООП подход, когда поведение атрибуты (имя и агрессивность) хранятся вместе с поведением (лаять, вилять хвостом, реагировать на знакомство):

---

```
#include <iostream>

using namespace std;

struct Dog {
    string _name;
    bool _isAggressive;

    Dog(const string &name, bool isAggressive = false) {
        _isAggressive = isAggressive;
        _name = name;
    }

    void bark() { cout << _name << " says: woof-woof" << '\n'; }

    void wag() { cout << _name << " wags its tail" << '\n'; }

    void responseToFirstMeeting() {
        if (_isAggressive) {
            bark();
            bark();
            cout << "Stranger runs away";
        } else {
            wag();
            bark();
            wag();
        }
    }
};

int main() {
    string name;
    cin >> name;

    Dog d1(name);
    d1.responseToFirstMeeting();

    cin >> name;
    Dog d2(name, true);
    d2.responseToFirstMeeting();

    return 0;
}
```

---

Объединив атрибуты и методы в одной сущности (это действие в объектноориентированной терминологии называется **инкапсуляцией**), мы можем управлять доступом к данным.

Несколько некорректно рассуждать, что современный мир использует исключительно ООП (а тем более и структурное программирование). Для многих методик разработки программного обеспечения характерны свойства объектноориентированной и структурной методологий.

В цель данного курса не входит сколько бы то ни было серьёзное ознакомление именно с ООП. Всё данное лирическое отступление нужно для того, чтобы у слушателей не возникал вопрос: "А почему для вектора мы пишем `a.clear()`?" Оказывается, что вектор способен себя очистить. Умение очистить себя от хранимых значений – элемент его поведения.

Подытожим написанное выше. В языке программирования C++ мы можем объявлять функции-члены<sup>139</sup> внутри структур. Они ничем не отличаются от прочих функций, только **имеют доступ ко всем полям структуры**:

---

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

struct Route {
    std::string _source;
    std::string _destination;

    int getDistance() {
        // функция не принимает явно никаких параметров,
        // но имеет доступ к полям Route
        return _source.size() + _destination.size();
    }
};

int main() {
    auto aName = "Moscow";
    auto bName = "Belgorod";
    Route route = {aName, bName};

    std::cout << route.getDistance() << '\n'; // 14
    route._source = "Saint Petersburg";
    std::cout << route.getDistance() << '\n'; // 24

    return 0;
}
```

---

Обратите внимание, чтобы вызвать такую функцию требуется указывать идентификатор созданного объекта, поставить точку, указать конкретную функцию (при необходимости передать параметры).

Решило ли это проблему? Частично. Представьте, что функция вычисления расстояния имеет большую сложность, и повторные расчёты создадут избыточную вычислительную нагрузку. Мы бы хотели минимизировать количество вычислений `getDistance`.

---

<sup>139</sup> Иногда вместо функций-членов можно услышать термин "методы", однако он не совсем является корректным. Если вы выступаете в сверхтребовательной (токсичной) к терминам среде, не используйте слово "метод".

Но когда расстояние должно вычисляться повторно? Когда изменяется один из населённых пунктов. Давайте разгранилим, что пользователю можно делать, а что ему делать нельзя. Мы запретим ему прямой доступ к полям извне (чтобы он не смог изменить значение хоть `source`, хоть `destination`), но вот доступ через функции-члены разрешим. Используются ключевые слова `public` для тех полей и функций-членов, к которым мы хотели бы давать доступ пользователю, и ключевое слово `private` для тех, к которым не хотели бы. Возможное решение наших проблем:

---

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

struct Route {
public: // public распространяется на все поля и функции-члены ниже
    // если не будет встречен другой модификатор доступа
    void setSource(const std::string &source) {
        _source = source;
        updateDistance();
    }

    std::string getSource() { return _source; }

    void setDestination(const std::string &destination) {
        _destination = destination;
        updateDistance();
    }

    std::string getDestination() { return _destination; }

    unsigned getDistance() { return _distance; }

private: // private распространяется на все поля и функции-члены ниже
    // если не будет встречен другой модификатор доступа
    std::string _source;
    std::string _destination;
    unsigned _distance;

    void updateDistance() {
        _distance = _source.size() + _destination.size();
    };
};

int main() {
    Route route;
    route.setSource("Moscow");
    route.setDestination("Belgorod");

    std::cout << route.getDistance() << '\n'; // 14
    route.setDestination("Saint Petersburg");
    std::cout << route.getDistance() << '\n'; // 22

    return 0;
}
```

---

На самом деле, если заменить слово `struct` на `class`<sup>140</sup> ничего не изменится:

---

```
class Route {
public:
    void setSource(const std::string &source) {
        _source = source;
        updateDistance();
    }

    //...
}
```

---

### 18.21.3 Константность функций-членов

Добавим функцию вывода для маршрута. Функции вывода определяют обычно все структуры / класса:

---

```
void printRoute(const Route &route) {
    std::cout << "From: " << route.getSource() << '\n';
    std::cout << "To: " << route.getDestination() << '\n';
    std::cout << "Distance = " << route.getDistance() << '\n';
}
```

---

При попытке скомпилировать код вы получите ошибку компиляции. Из заголовка функции видно, что функция ни в коем случае не должна менять каким-то образом структуру. Однако где гарантии, что функции `getSource`, `getDestination`, `getDistance` не изменяют значение структуры `route`? По умолчанию считается, что создаваемые функции-члены могут модифицировать класс (даже если они это не делают). Очень часто компилятору нужно дать точное указание на то, что поля структуры в процессе вызова функции никак не меняются. Это можно сделать при помощи ключевого слова `const`:

---

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

struct Route {
public:
    void setSource(const std::string &source) {
        _source = source;
        updateDistance();
    }

    std::string getSource() const { return _source; }

    void setDestination(const std::string &destination) {
        _destination = destination;
        updateDistance();
    }
```

---

<sup>140</sup>Никогда не понимал ажиотажа вокруг классов. Когда говорят слово класс, думайте о ней как о структуре. Это ровно такая же штука. Одно из отличий, о которых следует поговорить сейчас: у структуры все поля по умолчанию открыты (`public`) в отличие от класса. У него все поля закрыты (`private`).

```

    }

    std::string getDestination() const { return _destination; };

    unsigned getDistance() const { return _distance; };

private:
    std::string _source;
    std::string _destination;
    unsigned _distance;

    void updateDistance() {
        _distance = _source.size() + _destination.size();
    };
};

void printRoute(const Route &route) {
    std::cout << "From: " << route.getSource() << '\n';
    std::cout << "To: " << route.getDestination() << '\n';
    std::cout << "Distance = " << route.getDistance() << '\n';
}

int main() {
    Route route;
    route.setSource("Moscow");
    route.setDestination("Belgorod");

    std::cout << route.getDistance() << '\n'; // 14
    route.setDestination("Saint Petersburg");
    std::cout << route.getDistance() << '\n'; // 22

    return 0;
}

```

---

Ключевое слово `const` пишется в конце заголовка. Следовательно, **если написанная внутри класса функция не изменяет объект, объявит её константной**.

Запомните данный момент: константные функции могут вызывать только константные функции-члены, неконстантные – любые, но константные не могут вызвать неконстантные функции-члены.

Закрепим изученное парой контрольных вопросов:

- Дан класс `Office`:

---

```

class Office {
public:
    int countBrokenTables() const { /* ... */ }
    void burnBrokenTables() { /* ... */ }

private:
    bool isTableBroken(int tableIndex) const { /* ... */ }
    void burnTable(int tableIndex) { /* ... */ }
    // приватные поля
};

```

---

Кроме того, дана функция:

---

```
std::string inspectOffice(const Office& office) {
    // INSERT CODE HERE
    return "Office is OK";
}
```

---

Какие строки кода, будучи вставленными на место комментария **INSERT CODE HERE**, будут компилироваться?

---

```
office.burnTable(0);
office.burnBrokenTables();
int brokenTableCount = office.countBrokenTables();
bool firstBroken = office.isTableBroken(0);
```

---

Если данный вариант не компилируется, укажите причину.

#### 18.21.4 Конструкторы

Как вам уже известно, структуры/классы в языке программирования C++ могут содержать в себе функции-члены. В объектно-ориентированном программировании выделяют так называемые **конструкторы** – специальный блок инструкций, вызываемый при создании объекта. Цель конструктора **инициализация полей структуры и выделение необходимых ресурсов**.

Представьте, что вам нужна функция, которая создавала бы множества. Вот вам несколько вариантов названий, которые могли бы встретиться:

- `makeSet;`
- `buildSet;`
- `createSet;`
- `getNewSet` и др.

Чтобы несколько уменьшить разнообразие названий, в объектно-ориентированных языках имя функции, создающей объект и совпадающей с именем структуры/класса называется **конструктором**. Конструктор не имеет возвращаемого значения. Пример конструктора без параметров:

---

```
#include <string>

struct Route {
public:
    Route() {} // конструктор без параметров,
                // который не выделяет никаких ресурсов
                // и не инициализирует поля структуры
private:
    std::string _source;
    std::string _destination;
};

int main() {
    Route a; // при объявлении переменной неявно вызывается Route()
}
```

---

Если он не объявлен явно, он создаётся неявно (ровно таким же образом ничего не делает). Пример ещё одного конструктора без параметров:

---

```
struct Route {
public:
    Route() {
        setSource("Belgorod");
        setDestination("Krasnoborsk");
    }

    //...
};
```

---

Теперь если вы где-нибудь объявили маршрут, он будет проинициализирован именно этими населенными пунктами:

---

```
int main() {
    Route route;
    printRoute(route); // From: Belgorod
                      // To: Krasnoborsk
                      // Distance = 19
    return 0;
}
```

---

Типичной ошибкой является следующая:

---

```
int main() {
    Route a; // Маршрут "Belgorod - Krasnoborsk"
    Route b(); // объявление функции без параметров (!)
               // с возвращаемым значением типа Route.
}
```

---

Конструктор может содержать параметры (быть параметризованным). Пример:

---

```
Route(const std::string &source, const std::string &destination) {
    setSource(source);
    setDestination(destination);
}
```

---

Тогда объект может быть создан так:

---

```
Route a("Belgorod", "Paris");
```

---

Если при наличии параметризованного конструктора, вы не объявили непараметризованный, непараметризованный конструктор по умолчанию не будет создан:

---

```
#include <string>

struct Route {
public:
```

```

Route(const std::string &source, const std::string &destination) {
    // ...
};

// ...
};

int main() {
    Route a; // ошибка компиляции, так как имеется
             // параметризованный конструктор,
             // а непараметризованного нет

    return 0;
}

```

---

Если вам всё-таки нужен непараметризованный конструктор в дополнении к параметризованному, описать его лучше так:

```

#include <string>

struct Route {
public:
    Route() = default;

    Route(const std::string &source, const std::string &destination) {
        setSource(source);
        setDestination(destination);
    };
    // ...
};

```

---

Если вас смущает двойной перерасчёт длины, можно выполнить и такую реализацию<sup>141</sup>:

```

Route(const std::string &source, const std::string &destination) {
    _source = source;
    _destination = destination;
    updateDistance();
}

```

---

Пример создания объекта:

```

int main() {
    Route route("Belgorod", "Paris");
    printRoute(route); // From: Belgorod
                      // To: Paris
                      // Distance = 13

    return 0;
}

```

---

<sup>141</sup>Доступ к полям напрямую вполне допустим для конструкторов.

Для того, чтобы передать какие-то маршруты в функцию необязательно создавать переменную типа `Route` до вызова. Ознакомьтесь с примером:

---

```
printRoute(Route());
printRoute({}); // будет вызван Route();

printRoute(Route("Belgorod", "London"));
printRoute({"Belgorod", "London"}); // будет вызван Route(source, destination)
```

---

### 18.21.5 Инициализация полей и структур / классов по умолчанию

Поля структур и классов могут быть проинициализированы при объявлении. Пусть для создаваемых значений поле `destination` инициализируется значением `"Rome"`:

---

```
class Route {
// ...
private:
    std::string _source;
    std::string _destination = "Rome";
    int _distance;
};

int main() {
    Route route;
    route.setSource("Belgorod");
    printRoute(route); // From: Belgorod
                      // To: Rome
                      // Distance = 12
}
```

---

Задания:

1. `Fraction` – структура для работы с дробями.
  - Создайте структуру `Fraction` с двумя приватными полями для хранения числителя и знаменателя.
  - Конструктор по умолчанию должен создавать дробь с числителем 0 и знаменателем 1.
  - При конструировании объекта класса `Fraction` с параметрами `p` и `q` должно выполняться сокращение дроби  $p / q$ . Функция сокращения должна быть объявлена внутри класса и быть приватной.
  - Если дробь  $p / q$  отрицательная, то объект `Fraction(p, q)` должен иметь отрицательный числитель и положительный знаменатель.
  - Если дробь  $p / q$  положительная, то объект `Fraction(p, q)` должен иметь положительные числитель и знаменатель ( обратите внимание на случай `Fraction(-2, -3)`).
  - Если числитель дроби равен нулю, то знаменатель должен быть равен 1.

- Напишите тесты, которые проверяют, что поведение структуры корректно.
2. `MaterialPoint` – структура для работы с материальными точками.

- Объявите структуру `MaterialPoint` содержащую приватные поля позиция, скорость и ускорение.
- Поле "позиция" при создании экземпляра класса должна иметь значение 0. Должен поддерживаться конструктор:

---

```
MaterialPoint(double speed, double acceleration);
```

---

Спроектируйте его таким образом, чтобы можно было создать материальную точку

- без указания скорости и ускорения (должны равняться нулю),
- с указанием только скорости, а ускорение при этом равно нулю,
- с указанием и скорости, и ускорения

Приведите код для создания трёх материальных точек (по одной на каждый тип).

- Реализуйте функции `setSpeed` и `setAcceleration` для изменения скорости и ускорения материальной точки.
- Реализуйте функции `getPosition()` / `getSpeed()` для получения текущей позиции / скорости объекта.
- Реализуйте функцию `move(int duration)`, которая двигает материальную точку по числовой прямой на протяжении `duration` секунд. Если движение было равноускоренным, должна измениться и скорость.
- Проведите автоматизированное тестирование разработанного решения.

### 18.21.6 Перегрузка операций ввода и вывода

Ранее показывалось, как перегрузить операции ввода и вывода для структуры. Однако у вас возникнут проблемы при перегрузке операций ввода вывода для структур и классов с приватными полями. Ведь для того, чтобы вывести поле, к нему необходимо иметь доступ. А доступ к некоторым полям может не обеспечиваться интерфейсом класса. Довольно частым решением является использование ключевого слова `friend`:

---

```
struct Fraction {
public:
    friend std::ostream&
        operator<<(std::ostream &out, const Fraction &f);
private:
    int _numerator;
    int _denominator;
};
```

---

В данном случае оно позволяет определить вне класса функцию `std::ostream& operator<<(std::ostream &out, Fraction &f)`, которая будет иметь доступ ко всем полям структуры типа `Fraction` для вывода:

---

```

#include <iostream>

struct Fraction {
public:
    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }

    friend std::ostream&
        operator<<(std::ostream &out, const Fraction &f);
private:
    int _numerator;
    int _denominator;
};

std::ostream& operator<<(std::ostream &out, const Fraction &f) {
    if (f._denominator == 1)
        out << f._numerator;
    else
        out << f._numerator << "/" << f._denominator;

    return out;
}

int main() {
    Fraction f(1, 3);
    std::cout << f; // выведет "1/3"

    return 0;
}

```

---

При перегрузке операций ввода / вывода крайне часто используется такой прием: мы оставляем за пользователем нашей структуры определять, каким образом должна быть реализована операция вывода.

Задания: реализуйте ввод и вывод для структуры `Fraction`.

### 18.21.7 Перегрузка арифметических операций

При решении математических задач иногда используют перегрузку арифметических операций. Предположим, что в коде хочется писать что-то такое:

---

```

Fraction f1(1, 3);
Fraction f2(2, 3);

cout << f1 + f2;

```

---

Однако компилятор не знает, каким образом должно осуществляться сложение. Решим данную задачу:

---

```

Fraction operator+(const Fraction &other) const {
    Fraction res;

    int _lcm = std::lcm(_denominator, other._denominator);
    res._numerator = _numerator * (_lcm / _denominator) +
                      other._numerator * (_lcm / other._denominator);
    res._denominator = _lcm;
    fractionReduction(res);

    return res;
}

```

---

Задания:

- **Fraction:**

- Выполните перегрузку для операций +, -, \*, /.

- **BitSet.** Вам будет предложено реализовать похожую на ранее реализованную в рамках лабораторных работ структуру. Будет наблюдаться лишь несколько отличий: добавляются новые поля, разграничение доступа к ним. Мы попробуем собрать всё, что нам известно, а также изучим несколько новых техник.

1. Создайте структуру:

---

```

#include <iostream>

struct BitSet {
    BitSet() = default;

private:
    // максимальное поддерживаемое значение битсетом как таковым
    static const int _MAX_VALUE_SUPPORTED = 31;
    // максимальное поддерживаемое значение
    // для создаваемой структуры
    const int _maxValue = _MAX_VALUE_SUPPORTED;
    // поле для хранения значений
    uint32_t _values = 0;
    // мощность множества
    unsigned _power = 0;
};

```

---

Что нового тут можно увидеть:

- Объявлена константа `static const int _MAX_VALUE_SUPPORTED`, равная максимальному значению, которое может быть представлено множествами, посредством данной структуры.
  - Поля структуры начинаются с `_`, чтобы отделять переменные структуры от параметров функций.
2. Познакомися со списками инициализации. Конструкторы предназначены для инициализации полей структур:

---

```
struct A {
    A(int a, int b) {
        _a = a;
        _b = b;
    }

    int _a;
    int _b;
};
```

---

Вы действительно можете использовать такой подход, но он не сработает при необходимости инициализировать константные поля:

---

```
struct A {
    A(int a, int b, int c) {
        _a = a;
        _b = b;
        _c = c; // ошибка компиляции
    }

    int _a;
    int _b;
    const int _c;
};
```

---

Более верной техникой является использование списков инициализации:

---

```
struct A {
    A(int a, int b, int c) : _a(a), _b(b), _c(c) { }

    int _a;
    int _b;
    const int _c;
};
```

---

Создайте конструктор `Bitset(int maxValue)`, который создаёт пустое множество и инициализирует максимальное допустимое значение данного множества значением `maxValue`. Если пользователь указал значение большее, чем `MAX_VALUE_SUPPORTED`, должно выкидываться исключение `invalid_argument` с информативным сообщением об ошибке.

3. Реализуйте функцию `empty()`, возвращающую значение "истина", если множество пусто, в противном случае – "ложь".
4. Проверьте, что поставили ключевое слово `const` для функции в прошлом случае. Я не буду больше о нём писать. На том этапе, когда пишете функцию, всегда задумывайтесь о том, меняет ли данная функция объект или нет.
5. Реализуйте функцию `bool find(unsigned x)`, которая возвращает значение "истина", если элемент `x` присутствует в множестве, иначе – "ложь".

6. Реализуйте функцию вставки (`void insert(unsigned x)`) элемента в множество. Если осуществляется попытка вставить элемент больше максимального значения, представимого множеством, должно выбрасываться исключение `invalid_argument`.
7. Реализуйте функцию удаления (`void erase(unsigned x)`) элемента из множества.
8. Было бы удобно выполнять инициализацию множества так:

---

```
BitSet a = {1, 2, 4};
```

---

или так:

---

```
BitSet a({1, 2, 4});
```

---

Для этого нужно реализовать конструктор, принимающих переменную типа `const vector<int> &`.

9. Выполните перегрузку оператора сравнения на равенство и неравенство<sup>142</sup>:

---

```
bool operator==(BitSet &other);
bool operator!=(BitSet &other);
```

---

10. Реализуйте функцию `bool isSubset(const BitSet &set)`, которая возвращает значение истина, если текущее множество является подмножеством некоторого множества `set`. Например, если даны два множества, и необходимо проверить, является ли текущее множество подмножеством другого множества вызывается функция:

---

```
BitSet a = {1, 4};
BitSet b = {1, 2, 4};

cout << b.isSubset(a);
```

---

11. Выполним перегрузку функции `isSubset`:
- ```
static bool isSubset(const BitSet &subset, const BitSet &set)
```
- Здесь мы уже видим использование ключевого слова `static` в заголовке функции. К чему нас это обязывает:

- Функцию можно вызвать без создания объекта:

---

```
BitSet a = {1, 4};
BitSet b = {1, 2, 4};

cout << BitSet::isSubset(a, b);
```

---

- В теле функций не могут находиться обращения к нестатичным элементам класса (полям и функциям).

---

<sup>142</sup>Функции должны содержать одну строку.

Возможная реализация:

---

```
static bool isSubset(const BitSet &subset, const BitSet &set) {
    return (subset.values_ | set.values_) == set.values_;
}
```

---

Так как функция стала перегруженной, теперь мы можем делать как первый, так и второй вызов:

---

```
BitSet a = {1, 4};
BitSet b = {1, 2, 4};

cout << a.isSubset(b) << '\n';
cout << BitSet::isSubset(a, b) << '\n';
```

---

Зачем всё это нужно? Например, представим, что придётся реализовывать операции над множествами. Если реализовать функцию в таком виде:

---

```
BitSet a;
BitSet b;

a.union_(b);
```

---

вызов функции по моему субъективному мнению смотрится не особо. Было бы более приятным осуществлять вызов так:

---

```
BitSet a;
BitSet b;

BitSet::union_(a, b);
```

---

т. е. передавать две множества в функцию. Возможная реализация<sup>143</sup>:

---

```
static BitSet union_(const BitSet &lhs, const BitSet &rhs) {
    return BitSet(lhs.values_ | rhs.values_);
}
```

---

Вам остаётся создать приватный конструктор, который принимает число в двоичном представлении, в котором хранится множество, и инициализирует оставшиеся поля. Основная проблема заключается в расчете количества элементов в множестве. Поэтому вам будет полезна реализация приватной функции `getPower()`, возвращающей количество элементов в множестве. Реализуйте все оставшиеся операции над множествами:

- static BitSet intersection\_(const BitSet &lhs, const BitSet &rhs)
- static BitSet difference\_(const BitSet &lhs, const BitSet &rhs)
- static BitSet symmetricDifference\_(const BitSet &lhs,
 const BitSet &rhs)

---

<sup>143</sup>Нижнее подчёркивание в конце сделано из тех соображений, что слово `union` является зарезервированным

```
- static BitSet complement_(const BitSet &set)
```

12. Перегрузите операцию вывода в поток.
13. Проведите тестирование (без фанатизма). Особое внимание уделите тестированию функции вывода в поток.

- **PrefixSum**

1. Создайте структуру, при помощи которой удобно находить сумму массива на полуинтервале:

---

```
struct PrefixSum {
    PrefixSum(const int *a, size_t n);

    PrefixSum(const vector<int> &a);

    // возвращает сумму элементов от l до r не включая r.
    // например, для массива 1, 3, 5
    // запрос getSum(0, 2) должен возвращать 1 + 3 = 4;
    long long getSum(int l, int r);

    // данный запрос должен каким-то образом,
    // без хранения старого массива
    // обновлять префиксный массив так, как будто мы
    // заменили i-ый элемент старого массива на u
    void set(size_t i, int v);

private:
    vector<int> prefixSum_;
};
```

---

2. Опишем процесс, называемый "делегированием конструкторов". Иногда случается, что перед вызовом какого-то конструктора, мы бы хотели вызвать другой конструктор. Рассмотрим на примере:

---

```
PrefixSum(const int *a, size_t n);

PrefixSum(const vector<int> &a);
```

---

У вектора есть два метода:

---

```
a.data() // возвращает константный указатель на хранимые данные
a.size() // возвращает количество элементов в векторе
```

---

Но если у нас реализован конструктор `PrefixSum(const int *a, size_t n)`, то мы легко сможем реализовать второй конструктор через вызов первого:

---

```
PrefixSum(const vector<int> &a) : PrefixSum(a.data(), a.size())
{
    // действия, производимые после вызова конструктора
```

---

```
// PrefixSum(const int *a, size_t n)
}
```

---

После делегирования вы не сможете использовать списки инициализации.

3. Самостоятельно реализуйте оставшиеся функции класса. Автоматизировано протестируйте разработку.

# Глава 19

## Вопросы к экзамену

1. Определение «Алгоритм». Свойства алгоритмов.
2. Определения «Разработчик», «Исполнитель», «Пользователь».
3. Способы записи алгоритмов. На чём основывается выбор способа записи алгоритма?
4. Словесно-формульная запись. Пример алгоритма в словесно-формульной записи. Когда именно применяется данный способ?
5. Блок-схемы. Блоки, используемые при создании блок-схем, их названия и назначения.
6. Псевдокод. Когда применяется данный способ записи? Определение «Программа».
7. Определения «Язык программирования», «Синтаксис», «Семантика».
8. История развития ЭВМ и языков программирования.
9. История языка С. Преимущества и недостатки среди языков, поддерживающих структурное программирование. *Zero-overhead principle*.
10. Однострочные и многострочные комментарии. Рекомендации по использованию комментариев при разработке программного обеспечения.
11. Этапы компиляции. Определения «Препроцессор», «Компилятор», «Компиляция». Действия, производимые препроцессором над исходным кодом программы.
12. Определения «Язык программирования низкого уровня», «Язык программирования высокого уровня».
13. Структурное программирование. Принципы структурного программирования. Базовые управляющие конструкции.
14. Определение «Жизненный цикл программного обеспечения». Этапы жизненного цикла. Каскадная (водопадная) модель разработки. Преимущества и недостатки.
15. Определения «Тестирование», «Отладка», «Баг».

16. Что определяют типы данных? Типы данных стандарта С90. Модификаторы `signed` / `unsigned`, `short` / `long`.
17. Определения «Переменная», «Константа», «Литерал».
18. Целочисленные типы данных языка С. Выведение минимального и максимального значения переменных целочисленного типа.
19. Таблица *ASCII*. Перевод символов алфавита между регистрами, перевод символа-цифры в числовое значение. Определение «Символьный литерал». *Escape* (экранированные) последовательности.
20. Определения «Операция», «Операнд», «Объект данных», «*l-value*», «*r-value*».
21. Представление отрицательных целых чисел в языке программирования С. Получение значения по двоичному представлению. Приведите пример перевода.
22. Определение «Машинное слово». Унарная операция для определения размера типа. Связь между размерами целочисленных типов.
23. Спецификаторы с модификаторами для вывода переменных различных типов. Вывод символа и кода символа.
24. Вещественные типы данных. Число с плавающей запятой. Проблемы вещественной арифметики (приведите пример). Количество значений, представляемых переменными вещественного типа.
25. Определение «Инициализация». Почему переменные в языке программирования С не инициализируются по умолчанию? Объявление переменной. Рекомендации по объявлению и именованию переменных.
26. Определение «Массив». Объявление массива с инициализацией. Чем характеризуются каждый элемент массива? Вычисление объема памяти, занимаемым массивом. Чем является имя массива?
27. Нумерация элементов массива. Опишите причину, из-за которой нумерация элементов во многих языках программирования происходит с нуля.
28. Указатели. Операции над указателями. Значение `NULL`.
29. Строки в языке программирования С. Функции ввода и вывода строк.
30. Ключевое слово `const`. Использование `const` в контексте указателей. Способ изменения константной переменной в С.
31. Определение «Именованная константа». Три способа создания именованных констант в С. В чём заключается разница между `#define` и `const`.
32. Определения «Линейный алгоритм», «Разветвляющийся алгоритм», «Циклический алгоритм».
33. Генерация случайных чисел в С. Функции `rand()`, `strrand()`. Недостатки ГПСЧ в языке программирования С.
34. Определения «Операция», «Операнд». Классификация операций по количеству operandов. Определение «Выражение». Что определяет выражение?

35. Арифметические операции. Приоритет арифметических операций. Переполнение целочисленных типов при выполнении арифметических операций. Операция деления и остатка от деления в языке программирования С.
36. Префиксные и постфиксные инкременты и декременты. Приведите примеры, в которых использование префиксной и постфиксной формы дают различные результаты.
37. Определение «Побитовые операции». Классификация побитовых операций и их приоритеты.
38. Побитовое И. Проверка числа  $x$  на чётность / степень двойки с использованием  $\&$ . Побитовое ИЛИ, побитовое исключающее ИЛИ, побитовое НЕ.
39. Побитовые сдвиги влево и вправо. Реализация циклического сдвига влево и вправо.
40. Установка  $i$ -го бита в значения 0 и 1. Инвертирование  $i$ -го бита.
41. Логические операции. Приоритеты логических операций. Определение «Логическое выражение». Ленивая схема вычислений. Рекомендации по использованию ленивой схемы.
42. Операции сравнения. Приоритеты операций сравнения. Сравнение вещественных operandов.
43. Приоритеты операций. Определение «ассоциативность».
44. Приведения типов. Явные и неявные приведения типов. В каких случаях происходит неявное приведение типов? Последствия при приведении типов.
45. Определение «Разветвляющиеся алгоритмы» Способы организации бинарного и множественного ветвления в языке программирования С.
46. Условный оператор `if-else`. Определения «Логическое выражение», «Составной оператор».
47. Оператор `switch`. Определение «Константное выражение». Основные ошибки, допускаемые при использовании `if-else` / `switch`.
48. Определения «Циклический алгоритм», «Итерация», «Тело цикла»
49. Конструкция цикла `for`. Секции цикла. Описание работы цикла `for`. Обозначение цикла в блок-схемах в варианте с фиксированным и нефиксированным количеством повторений.
50. Цикл `while`. Определение «Логическое выражение». Обозначение цикла на блок-схеме. В каких случаях стоит использовать цикл `for`, а в каких `while`?
51. Алгоритм проверки числа на простоту со сложностями по времени  $O(N)$ ,  $O(\sqrt{N})$ .
52. Решето Эратосфена.

53. Оценка сложности алгоритма по времени. Переход от точного времени к порядку функции временной сложности: почему алгоритмы не оцениваются только временем выполнения. Приведите пример произвольного алгоритма и обоснуйте его сложность.
54. Оценка сложности алгоритма по памяти. Связь решений задач между сложностью по времени и сложностью по памяти.
55. Особенности тестирования разветвляющихся и циклических алгоритмов.
56. Алгоритм Джая Кадана (поиск подпоследовательности с максимальной суммой).
57. Цикл `do-while`. Определение «Логическое выражение». Обозначение цикла на блок-схеме. В каких случаях следует использовать данный цикл.
58. Оператор `break`. Случаи, при которых оправдано использование `break`. Способы записи данных случаев без `break`.
59. Оператор `goto`. Случаи, при которых оправдано использование `goto`.
60. Оператор `continue`. Особенности работы `continue` в циклах `for` и `while`. Случаи, при которых оправдано использование `continue`. Способы записи данных случаев без `continue`.
61. Определение «Функция». Причины использовать функции. Принципы *DRY/WET*.
62. Определение «Функция». Определения «Формальные параметры», «Фактические параметры». Составляющие объявления и определения функций.
63. Процесс вызова функции. Состояние стека переменных. Побочный эффект функций.
64. Алгоритм быстрого возведения в степень.
65. Алгоритм Евклида. Вычисление наибольшего общего кратного.
66. Алгоритм перевода числа в произвольную систему счисления в рекурсивной форме.
67. Определения «Рекурсия», «Глубина рекурсии». Примеры простой и косвенной рекурсии.
68. Виды рекурсии: линейная, хвостовая, параллельная, взаимная, вложенная. Плюсы и минусы рекурсивных функций.
69. Подходы к решению задач на последовательности с использованием рекурсии на примере вычисления суммы массива.
70. Алгоритм однопроходного удаления в итеративной и рекурсивной формах.
71. Алгоритм проверки массива на палиндром в итеративной и рекурсивной формах.
72. Алгоритм сортировки выбором в итеративной и рекурсивной формах.

73. Вычисление значения многочлена в точке в итеративной и рекурсивных формах.
74. Бинарный поиск в рекурсивной и итеративной формах.
75. Алгоритм поиска элемента, удовлетворяющего условию. Поиск количества элементов в массиве, удовлетворяющего условию.
76. Поиск максимального количества подряд идущих элементов, удовлетворяющих условию. Поиск максимального количества подряд идущих пар, удовлетворяющих условию.
77. Вставка элемента в массив с сохранением / без сохранения порядка элементов. Удаление элемента из массива с сохранением / без сохранения порядка элементов.
78. Определения «стек», «куча», «оперативное запоминающее устройство».
79. Функции для работы с динамической памятью: `malloc`, `calloc`, `realloc`, `free`. Возможные исходы при вызове функций.
80. Передача функций как параметров. Пример задачи с применением приёма.
81. Массив префиксных сумм.
82. Разностные массивы. Задача добавления на отрезке.
83. Линейный поиск. Быстрый линейный поиск. Линейный поиск в отсортированном массиве.
84. Блочный поиск. Выведение оптимального размера блока.  $\text{Sqrt}$ -декомпозиция.
85. Бинарный поиск для решения задачи поиска.
86. Бинарный поиск по критерию. Бинарный поиск при решении задач оптимизации.
87. Тернарный поиск.
88. Сортировка пузырьком: идея, пример, реализация.
89. Сортировка вставками: идея, пример, реализация.
90. Сортировка подсчётом: идея, пример, реализация.
91. Сортировка слиянием: идея, пример, реализация.
92. Идеи усовершенствованных сортировок: расческой, Шелла, цифровая.
93. Куча: определение кучи, операция вставки и удаления элементов в / из кучи. Идея сортировки кучей.
94. Система непересекающихся множеств: определение, наивный подход.
95. Система непересекающихся множеств: переход от наивного подхода к деревьям.
96. Система непересекающихся множеств: эвристика сжатия пути.

97. Система непересекающихся множеств: ранговая оптимизация.
98. Применение системы непересекающихся множеств при решении задачи Краскала.
99. Представление числовых множеств в памяти ЭВМ на неупорядоченном массиве. Сложность алгоритмов операций над множествами.
100. Представление числовых множеств в памяти ЭВМ на упорядоченном массиве. Сложность операций над множествами в данном представлении.
101. Представление числовых множеств в памяти ЭВМ на массиве битов. Сложность операций над множествами в данном представлении.
102. Сравнительный анализ представлений множеств на неупорядоченном, упорядоченном массивах и массиве битов. Преимущества и недостатки. Сложность операций.
103. Представление мульти множеств с элементами целочисленного типа на целочисленных массивах.
104. Предпосылки к появлению языка C++.
105. Объекты, связанные со стандартными потоками. Буферизация потоков. Ввод и вывод в языке программирования C++. Связь потоков ввода / вывода. Случай, при которых очищается стандартный поток вывода.
106. Изменения в управляющих конструкциях языка C++ в сравнении с языком программирования С.
107. Типы данных языка C++. Коллекции *STL*.
108. Коллекции `std::array`, `std::vector`. Сложность операций над коллекциями.
109. Коллекция `std::deque`. Определение «дек». Интерфейс дека. Сложность операций. Представление дека на физическом уровне.
110. Коллекция `std::stack`. Определение «стек». Интерфейс стека. Сложность операций. Применение стека в задаче о скобочной последовательности.
111. Коллекции `std::queue`, `std::priority_queue`. Определение «очередь», «приоритетная очередь». Интерфейс очереди, приоритетной очереди. Сложность операций.
112. Коллекция `std::map`. Определение «отображение». Интерфейс отображения. Сложность операций над отображением.
113. Коллекция `std::set`. Определение «отображение». Интерфейс отображения. Сложность операций над множеством.
114. Способы создания именованных констант в C++: `const`, `constexpr`, `enum class`.
115. Ссылки и указатели: задачи, решаемые при помощи указателей в языке программирования С. Какие из них были возложены на ссылки. Приведите примеры. Разница между ссылками и указателями.

116. Структурное связывание. Пары и кортежи: `std::pair`, `std::tuple`, `std::tie`.
117. Определения «перегрузка», «расширение перегрузки». Пример перегрузки функций. Параметры функций по умолчанию.
118. Лямбда-функции. Приведите пример лямбда функций с пустым и непустым списком захвата.
119. Причина появления шаблонов в языке программирования C++. Определение «инстанцирование шаблона», «объект шаблона». Процесс компиляции шаблона. Преобразования типов, в процессе вывода аргумента шаблона. Перегрузка шаблонов функций.
120. Итераторы: причина появления. Виды итераторов. Объявление итераторов. Алгоритмы на паре итераторов: поиск элемента в коллекции. Алгоритмы на паре итераторов: поиск максимального значения в коллекции. Функции для работы с итераторами.