

Основы программирования. Конспект лекций

Притчин И.С.

Июль 2021 – Февраль 2022

Оглавление

1 Алгоритмы. Структурное программирование	9
1.1 Алгоритм. Свойства алгоритмов	9
1.1.1 Свойства алгоритма	10
1.1.2 Способы записи алгоритмов	10
1.1.3 Словесно-формульная запись	10
1.1.4 Блок-схемы (графический способ)	11
1.1.5 Псевдокод	12
1.1.6 Программа	12
1.2 Структурное программирование	19
1.3 Жизненный цикл программного обеспечения	21
2 Типы данных. Переменные	26
2.1 Целочисленные типы данных	27
2.1.1 <i>char / unsigned char</i>	27
2.1.2 <i>int / unsigned int</i>	30
2.1.3 <i>short int / unsigned short int; long int / unsigned long int; long long int / unsigned long long int</i>	31
2.2 Вещественные типы данных: <i>float, double</i>	32
2.3 Пустой тип: <i>void</i>	33
2.4 Объявление и инициализация переменных	34
2.5 Массивы	35
2.6 Указатели	37
2.7 Строки	38
2.8 Константы	39
3 Линейные алгоритмы	44
3.1 Задача о сумме	44
3.2 Задача обмена значений	46
3.3 Задача о последней цифре числа	47
3.4 Задача о вычислении расстояния между двумя точками	48
3.5 Задача о генерации числа в заданном диапазоне	49
3.6 Задача о сумме арифметической прогрессии	51
3.7 Задача перевода температуры из градусов Фаренгейта в градусы Цельсия	52
4 Операции	55
4.1 Арифметические операции	55
4.1.1 Унарный минус и унарный плюс	55
4.1.2 Сложение и вычитание	55
4.1.3 Умножение	58
4.1.4 Деление и остаток от деления	59
4.1.5 Приоритеты арифметических операций	60

4.2	Побитовые операции	60
4.2.1	Побитовое И	61
4.2.2	Побитовое ИЛИ	62
4.2.3	Побитовое НЕ	62
4.2.4	Побитовое исключающее ИЛИ	63
4.2.5	Побитовый сдвиг вправо	63
4.2.6	Побитовый сдвиг влево	64
4.2.7	Приоритеты побитовых операций	66
4.3	Логические операции и операции сравнения	67
4.4	Приоритеты операций	68
4.5	Приведение типов	70
4.5.1	Неявное приведение типов	70
4.5.2	Явное приведение типов	73
4.5.3	Последствия при приведении типов	73
5	Разветвляющиеся алгоритмы	77
5.1	Условный оператор <i>if</i>	77
5.1.1	Поиск модуля числа	77
5.1.2	Поиск максимального значения из двух	79
5.1.3	Поиск максимального значения из трёх	82
5.1.4	Упорядочивание двух чисел	86
5.1.5	Решение линейного уравнения	87
5.2	Цепочки операторов <i>if-else-if</i>	87
5.2.1	Задача о социальной поддержке	87
5.2.2	Решение квадратного уравнения	90
5.3	Оператор <i>switch</i>	90
5.3.1	Задача о зимнем месяце	92
6	Циклические алгоритмы	95
6.1	Цикл <i>for</i>	95
6.1.1	Вычисление суммы последовательности	99
6.1.2	Поиск значений выражений	100
6.1.3	Поиск максимума вводимой последовательности	103
6.1.4	Поиск последнего нечетного элемента	104
6.1.5	Поиска максимального значения последовательности и их количества	105
6.2	Цикл <i>while</i>	107
6.2.1	Подсчёт количества цифр в числе	108
6.2.2	Подсчёт количества единиц в двоичном представлении числа x .	109
6.2.3	Изменение порядка цифр в числе на обратный	110
6.2.4	Подсчёт количества положительных и отрицательных чисел последовательности	111
6.2.5	Первый элемент рекуррентно заданной последовательности больше x	112
6.2.6	Поиск суммы последних m цифр числа n	113
6.2.7	Получение средней цифры числа, если количество цифр в числе нечетно	114
6.2.8	Проверка числа на простоту	115
6.2.9	Поиск второго максимального значения	118

6.2.10	Максимальное количество подряд идущих положительных значений	120
6.2.11	Максимальное количество знакочередующихся элементов последовательности	121
6.2.12	Подпоследовательность с максимальной суммой (Алгоритм Джая Кадана)	122
6.2.13	Поиск элемента находящегося перед первым четным	123
6.2.14	Поиск элемента, находящегося после последнего минимального .	124
6.3	Цикл <i>do-while</i>	126
6.3.1	Поиск количества цифр в числе	127
6.4	Выбор подходящего цикла	128
6.5	Операторы <i>break</i> , <i>goto</i> , <i>continue</i>	128
6.5.1	<i>break</i>	128
6.5.2	<i>continue</i>	129
6.5.3	<i>goto</i>	130
7	Функции	134
7.1	Функции в математике	135
7.2	Структурное программирование. Метод пошаговой детализации	136
7.2.1	Задача о кирпиче	136
7.2.2	Поиск максимального положительного значения из двух чисел .	140
7.2.3	Задача о треугольнике	141
7.3	Определение функции	142
7.4	Формальные и фактические параметры функции	143
7.5	Действия, производимые при вызове функции	144
7.6	Возвращаемое значение функции	146
7.7	Передача аргументов	149
7.8	Решения задач при помощи нерекурсивных функций	153
7.8.1	Алгоритмы возведения в степень	157
7.8.2	Алгоритм Евклида	158
7.8.3	Вывод числа в 16-ой системе счисления.	159
7.8.4	Задача о счастливом билете.	161
7.8.5	Задача о разбиении числа.	163
7.8.6	Задача о совершенных числах.	164
7.8.7	Быки и коровы.	165
7.9	Рекурсивные функции	171
7.9.1	Вычисление значения $n!$	173
7.9.2	Вывод двоичного представления числа	175
7.9.3	Вычисление суммы массива	176
7.9.4	Алгоритм однопроходного удаления	178
7.9.5	Проверка массива на палиндром	178
7.9.6	Сортировка выбором	179
7.9.7	Вычисление многочлена в точке	179
7.9.8	Бинарный поиск	180
8	Работа в CLion	185
8.1	Статический анализ кода	185
8.2	Отладчик	186
8.3	Автодополнение	194
8.4	Выделение программных объектов	196

8.5	Горячие клавиши	198
8.6	Шаблоны кода	200
8.7	Шаблон-окружение	202
8.8	Вывод	204
9	Алгоритмы обработки одномерных массивов	206
9.1	Алгоритмы, не зависящие от упорядоченности	206
9.1.1	Ввод массива	206
9.1.2	Вывод массива	209
9.1.3	Поиск позиции элемента, удовлетворяющему условию	210
9.1.4	Поиск количества элементов, удовлетворяющему условию	212
9.1.5	Поиск максимального количества подряд идущих элементов, удовлетворяющих условию	214
9.1.6	Однопроходный алгоритм удаления	214
9.1.7	Вставка элемента с сохранением порядка элементов	218
9.1.8	Удаление элемента с сохранением порядка элементов	219
9.1.9	Обращение элементов массива	220
9.1.10	Проверка упорядоченности массива по неубыванию	221
9.2	Неупорядоченные массивы	222
9.2.1	Добавление элемента в массив	222
9.2.2	Удаление элемента без сохранения порядка элементов	222
9.3	Одномерные массивы и работа с динамической памятью	223
9.3.1	Ввод и вывод массива (вариация 1)	225
9.3.2	Ввод и вывод массива (вариация 2)	227
9.4	* Передача функций, как параметров	229
9.5	* Префиксная сумма массива	234
9.5.1	Поиск количества подотрезков с нулевой суммой	235
9.6	* Разностные массивы	236
9.6.1	Добавление на отрезке	237
9.6.2	Добавление арифметической прогрессии на отрезке	238
9.7	Решения задач на одномерные массивы с использованием принципа пошаговой детализации	239
9.7.1	Сортировка подпоследовательности до первого вхождения нуля	239
9.7.2	Получение упорядоченной последовательности из неуникальных элементов	241
9.7.3	Сортировка с шагом	243
9.7.4	Подсчёт количества вхождений элементов	246
9.7.5	Сортировка точек по удалённости от другой точки	248
9.7.6	Вывод чисел, непринадлежащих последовательности	249
9.7.7	Сортировка элементов массива, выбранных по критерию	249
10	Поиск	254
10.1	Быстрый линейный поиск	254
10.2	Линейный поиск в отсортированном массиве	255
10.3	Блочный поиск. <i>Sqrt</i> -декомпозиция	255
10.4	Бинарный поиск (вариация №1)	259
10.5	Бинарный поиск (вариация №2)	261
10.6	Бинарный поиск по критерию	263
10.7	Решение задач на бинарный поиск	265
10.7.1	Поиск позиции первого значения равному x	265

10.7.2 Поиск позиции последнего значения равному x	265
10.7.3 Задача о верёвках	266
10.7.4 Задача о ксероксах	267
10.7.5 Задача об уравнении	268
10.7.6 Задача о сборе	268
10.7.7 Задача о коровах и стойлах	270
10.7.8 Поиск максимального среднего арифметического на отрезке длины не менее D	270
10.8 Тернарный поиск	274
11 Сортировки	276
11.1 <i>BogoSort</i>	276
11.2 Сортировка выбором	278
11.3 Сортировка вставками	279
11.4 Обменная сортировка	282
11.5 Сортировка расческой	283
11.6 Сортировка подсчётом	285
11.7 Цифровая сортировка	288
11.8 Сортировка слиянием	290
11.9 Куча и сортировка кучей	293
11.10 Быстрая сортировка	301
12 Система непересекающихся множеств	303
12.1 Наивный подход	304
12.2 Модификация №1 - Переход к спискам	305
12.3 Модификация №2 - Переход к деревьям	306
12.4 Задача про людей	309
12.5 Алгоритм Краскала	310
13 Структуры данных на одномерных массивах	314
13.1 Организация библиотек в языке программирования С	314
13.2 Представление числовых множеств и мульти множеств	320
13.2.1 Реализация множества на массиве битов	321
13.2.2 Реализация множества и мульти множества на массиве типа <i>char</i>	327
13.2.3 Множество на неупорядоченном массиве	328
13.2.4 Множество на упорядоченном массиве	330
14 Лабораторные работы	331
14.1 Лабораторная работа №1 «Стандартный ввод и вывод»	331
14.2 Лабораторная работа №2а «Алгоритмы разветвляющейся структуры»	336
14.3 Лабораторная работа №2б «Алгоритмы разветвляющейся структуры»	346
14.4 Лабораторная работа №3а «Циклы»	350
14.5 Лабораторная работа №3б «Циклы»	358
14.6 Лабораторная работа №4а «Введение в функции»	360
14.7 Лабораторная работа №4б «Обработка одномерных массивов с использованием функций»	367
14.8 Лабораторная работа №4с «Бинарный поиск»	373
14.9 Лабораторная работа №4д «Рекурсивные функции»	376
14.10 Лабораторная работа №4е «Структуры. Функции для работы со структурами»	378
14.11 Лабораторная работа №5а «Множества»	383

14.12Лабораторная работа №5b «Реализация структуры данных «Вектор»	390
14.13Лабораторная работа №5d «Работа с многомерными массивами»	407
14.14Лабораторная работа №5e «Работа со строками»	419

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 является 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 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 время¹.

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».

¹Если 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.

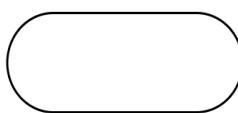
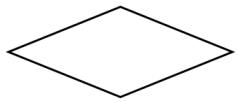
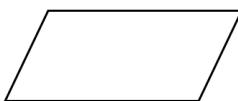
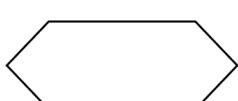
Блок-схема алгоритма представляет собой набор геометрических фигур (блоков), соединенных линиями или линиями со стрелками, для указания направления перехода от блока к блоку. Движение от блока к блоку сверху вниз или слева направо считается стандартным. В этом случае стрелки можно не указывать. Если же направление отлично от стандартного, то стрелки обязательны². 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 12 приведены основные блоки.

Пример

Блок-схема алгоритма решения уравнения вида $ax = b$ представлена на рисунке 1.1 (страница 13).

²В 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.

Пример

Программа для решения уравнения $ax = b$.

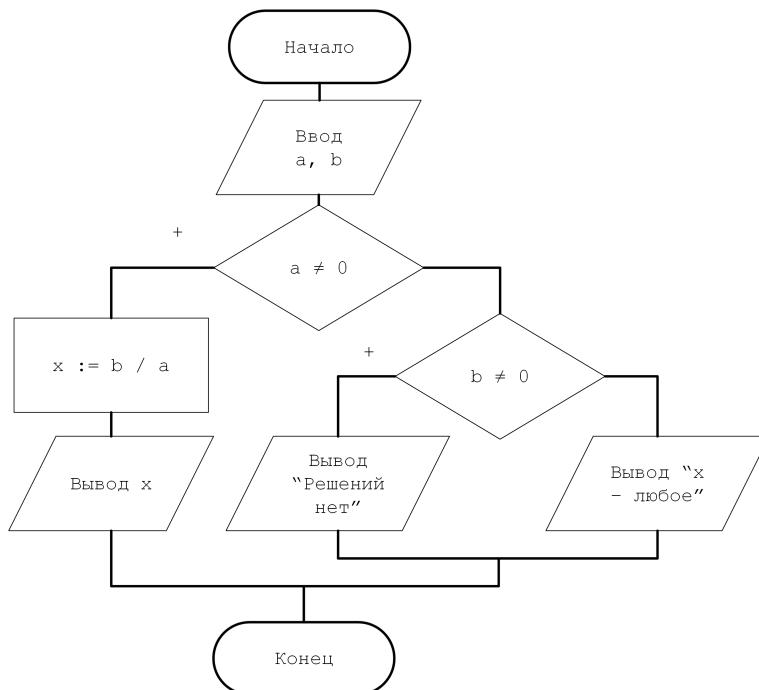


Рис. 1.1 – Блок-схема алгоритма решения уравнения вида $ax = b$

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6
7     float a, b;
8     scanf("%f %f", &a, &b);
9
10    if (a) {
11        float x = b / a;
12        printf("%f", x);
13    } else {
14        if (b)
15            printf("решений нет");
16        else
17            printf("x - любое");
18    }
19
20    return 0;
21 }
```

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 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 уровня³. 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 ЭВМ⁴. 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 пор используемых 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, С и др.).

Достоинства языков программирования высокого уровня:

- Набор операций, допустимых для использования, не зависит от набора машин-

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

⁴ Такие языки называются **машинонезависимыми**.

ных 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 поддерживали 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 Ритчи⁵ во время совместной работы с Кеном Томпсоном над операционной системой UNIX. Он был задуман 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.

Многие из ведущих в настоящее время языков являются производными от С, включая C++, C#, Java, JavaScript, Perl, PHP и Python⁶. 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.).

1983: C++. Бьёрн 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 обеспечении⁷.

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

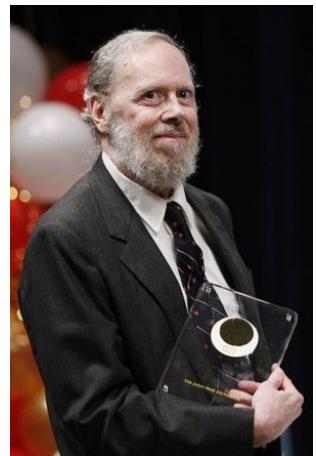


Рис. 1.2 – Деннис Ритчи

⁵На 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).

⁶Говорят, что C++ и Java растут быстрее, чем простой С, но я уверен, что С ещё поживёт. (Деннис Ритчи)

⁷При помощи С вы легко можете выстрелить себе в ногу. При помощи C++ это сделать сложнее, но если это произойдёт, вам оторвёт всю ногу целиком. (Бьёрн Страуструп)

1995: Java - это универсальный язык высокого уровня, созданный Джеймсом Гослингом для проекта интерактивного телевидения. Он обладает кросс - платформенной 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 стола⁸. Почти каждый крупный веб-сайт использует JavaScript. Gmail, Adobe Photoshop и Mozilla Firefox включают несколько хорошо известных примеров.

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 простой 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 языка⁹, 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.

Язык программирования С

В данном пособии будет описан язык С — компилируемый статически типизированный язык программирования общего назначения¹⁰. Первоначально был разработан для реализации операционной системы UNIX, но впоследствии 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 С¹¹:

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

⁸Java относится к JavaScript так же, как Сом к Сомали.

⁹Любой, кто приходит к вам и говорит, что у него есть совершенный язык — либо наивен, либо торговец (Страуструп).

¹⁰**Языки программирования общего назначения** - языки, которые предназначены для написания программного обеспечения в самых различных прикладных областях

¹¹Существует два вида языков программирования: одни — все ругают, другими не пользуются.

- 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.

Подробнее о функционировании языка ответ будет дан в последующих главах.
Простейшая программа на С¹²:

```
1 int main() {
2
3     return 0;
4 }
```

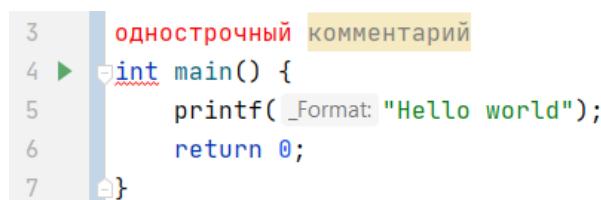
состоит из функции `main`. Данная программа ничего не получает на вход и ничего не выводит. Она завершается, возвращая значение `0` в операционную систему (ведь именно она производит запуск приложения). Код завершения `0` означает, что программа успешно закончила выполнение. Другие коды выхода могут указывать на проблему¹³ Наличие оператора возврата `return` не является обязательным; код выхода по умолчанию равен нулю:

```
1 // программа без явного возврата
2 // так делать можно, но не рекомендуется
3 int main() {
4
5 }
```

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

```
1 // односторочный комментарий
2
3 /* многострочный
   комментарий */
4
```

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



¹²Настоятельно рекомендую 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.

¹³Одной 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*¹⁴ на языке программирования С будет выглядеть следующим образом:

```

1 #include <stdio.h>
2
3 int main() {
4     printf("Hello world");
5     return 0;
6 }
```

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 — производится:
 - 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 древо разбора.
 - Семантический анализ. На этой фазе древо 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.
 - 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.

¹⁴Хотя 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.

- 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 1.3.

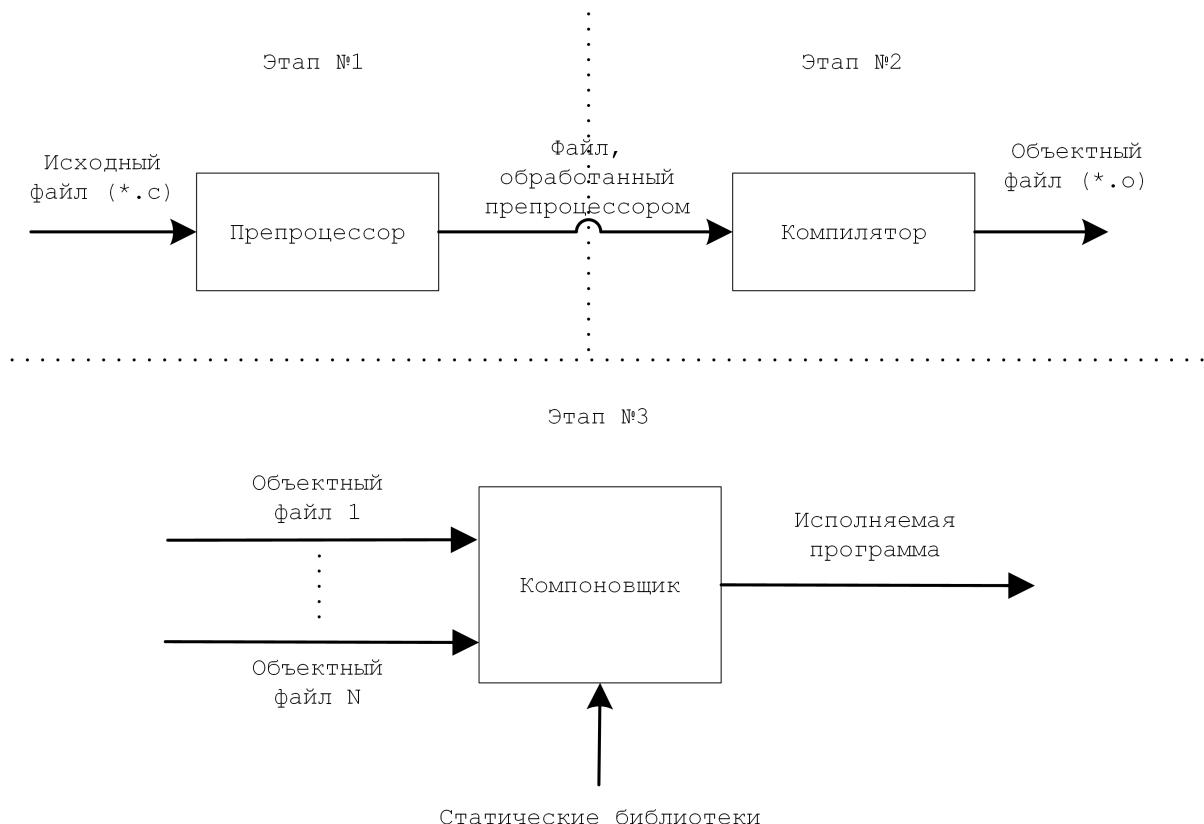


Рис. 1.3 – Этапы компиляции

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 самостоя-

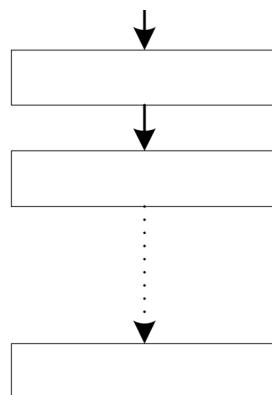
ятельную подзадачу. Размеры модулей должны быть небольшими, а инструкции, входящие в состав модуля, должны давать исчерпывающее представление о действиях, выполняемых модулем.

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

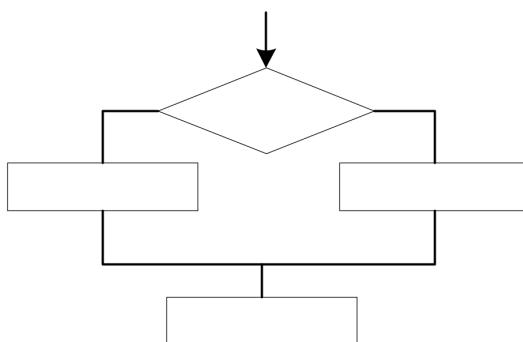
3. Каждый модуль должен иметь один вход и один выход. Это позволяет упростить стыковку модулей в сложной программе.

4. Логика алгоритма должна опираться на небольшое число достаточно простых базовых управляющих структур:

- Следование



- Развилка



- Цикл

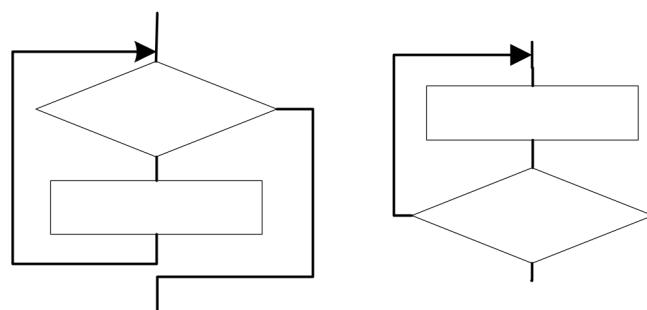
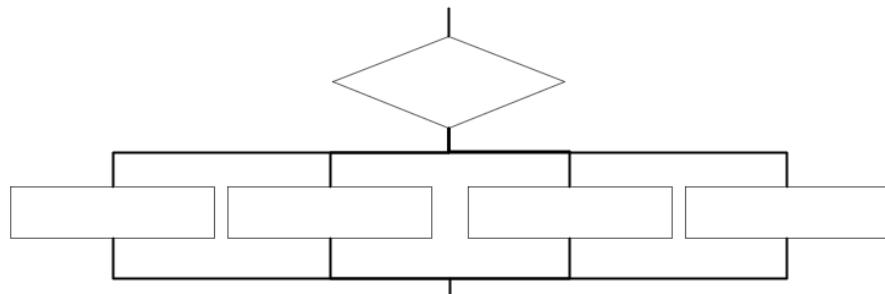


Рис. 1.4 – Цикл с предусловием и цикл с постусловием

- 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:

- Анализ требований¹⁵,
 - Проектирование¹⁶,
 - Кодирование (программирование),
 - Тестирование¹⁷ и отладка¹⁸,
 - Эксплуатация и сопровождение¹⁹.

¹⁵Требования – это перечень того, что необходимо продукту для достижения текущей проектной цели.

¹⁶Проектирование — процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или её части.

¹⁷ **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.

18 Отладка - процесс поиска, анализа и устранения причин отказов в программном обеспечении.

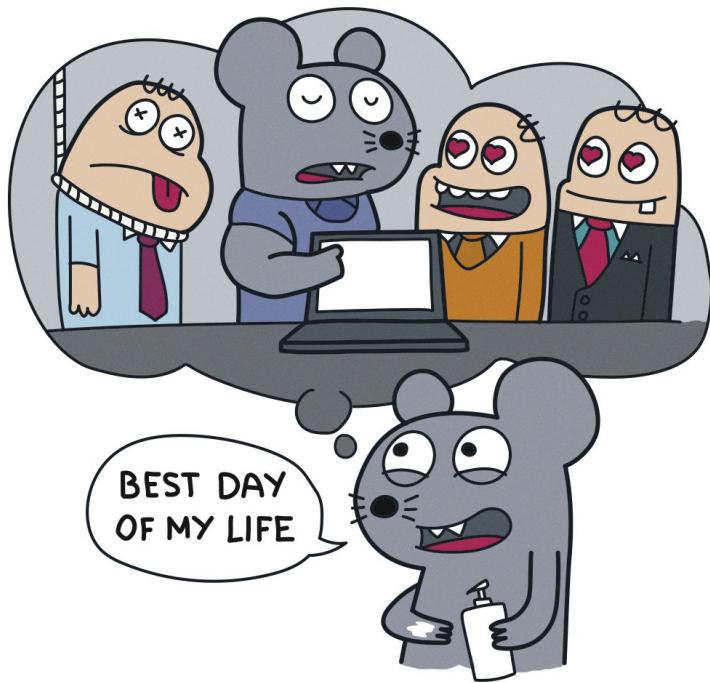
19 Сопровождение - модификация программного продукта после его поставки с целью исправления дефектов, улучшения производительности или других характеристик или для адаптации продукта к изменившемуся окружению.



Как и любая модель она обладает своими достоинствами и недостатками. Данный вопрос выходит за рамки данного пособия. Стоит отметить, что 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), 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, приближенных 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 управляемых 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. Что является входными и выходными данными?
3. Перечислите способы описания алгоритмов. Приведите примеры каждого способа. От чего зависит выбор способа записи алгоритма?
4. Перечислите основные достоинства и недостатки языка С.
5. Препроцессор, компилятор и компоновщик. Этапы компиляции.
6. При использовании директивы препроцессора `#include` в чем заключается разница между `< >` и `" "`.
7. Перечислите принципы структурного программирования.
8. 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 1¹. 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 константа², переменная³, результат вычисления выражения относится к определенному типу данных. Каждая операция требует операндов 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 типы⁴:

- `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 литерал⁵ соответствующего типа.

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

²**Константа** – это программный объект, не изменяющий своего значения.

³**Переменная** – программный объект, изменяющий своё значение.

⁴Именованные 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.

⁵**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*⁶-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 кавычками⁷. Для записи некоторых символов используются, так называемые, *escape*-последовательности. Приведём некоторые из них:

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

⁶ASCII (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.

⁷Имеется 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)⁸;
- `'\'` – апостроф;
- `'\"'` – кавычки;
- `'\\'` – обратный слеш;
- `'%%` – знак процента;
- `'\?'` – знак вопроса;
- `'\ooo'` – восьмеричное значение;
- `'\xhh'` – шестнадцатеричное значение.

Мы можем объявить переменные данного типа и присвоить им символьный литерал или целочисленный литерал⁹:

```

1 char a = 13;
2 char b = 'a'; // b = 97, так как код буквы 'a' в таблице ASCII равняется 97.
3 char c = ' '; // c = 32 (код пробела)
4 char d = 0x41; // d = 65 (целочисленный литерал в 16-ричной СС)
5 char e = 0101; // e = 65 (целочисленный литерал в 8-ричной СС)10
```

В листинге¹¹ выше мы впервые встречаемся с **операцией**¹² **присваивания** (`=`). Она работает следующим образом: вычисляется значение выражения справа от знака присваивания и результат сохраняется в объекте, находящемся слева¹³. 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:

```
1 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`:

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

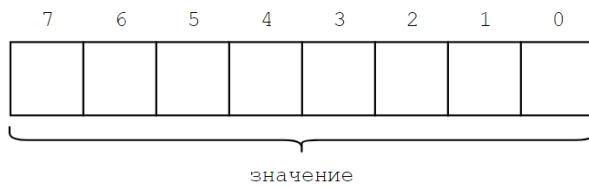
⁹Если вы явно хотите указать, что значение интерпретируется как код символа, присваивайте символьный литерал, вместо его кода

¹⁰Восьмеричный префикс является пережитком языка B, который использовали еще во времена компьютера PDP-8 и вездесущих восьмеричных литералов. Языки C/C++ продолжают сомнительную традицию.

¹¹Листинг – бумажная распечатка текста компьютерной программы или её части

¹²Операция – конструкция в языках программирования; специальный способ записи некоторых действий. Операнд – то, чем оперируют операции.

¹³Объект данных – область хранения данных, которая может применяться для удержания значений. Термин *l* – *value* в C 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.



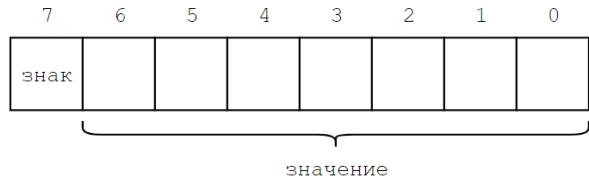
и при его помощи могут быть представлены любые числа из диапазона:

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

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

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

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

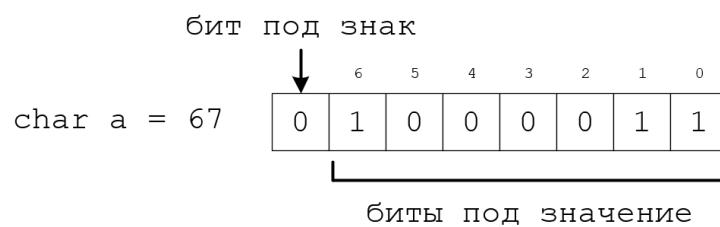


тогда под значение останется 7 бит. Можно представить 2^7 положительных значений и 2^7 отрицательных значений. 0 будет отдан положительным. Значит:

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

На самом деле количество представляемых значений $|E|$ зависит только от количества бит, отводимых для данного типа. Если отводится 8 бит, то $|E| = 2^8$; если 16 бит - $|E| = 2^{16}$ 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 $|E| = 2^3 = 8$ каких-то значений.

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



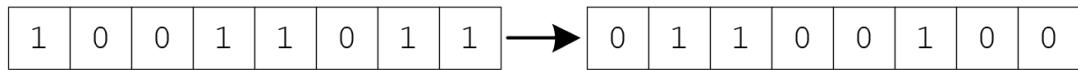
$$1000011_2 = 2^6 + 2^1 + 2^0 = 64 + 2 + 1 = 67_{10}$$

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 объявления¹⁴ переменных:

```
1 signed char a;
2 unsigned char b;
```

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

```
1 // данные записи эквивалентны
2 // при написании программ остановитесь на варианте без signed
3 signed char a;
4 char a;
```

Последний вопрос по данному типу - вопроса 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 значение¹⁵:

```
1 #include <stdio.h>
2
3 int main() {
4     char c = 65;
5     printf("%d\n", c); // 65
6     printf("%c\n", c); // A
7     return 0;
8 }
```

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).

¹⁴Объявление переменной говорит компилятору, что он должен связать определенное количество байт памяти с именем данной переменной.

¹⁵Для того, чтобы воспользоваться функцией `printf` необходимо подключить заголовочный файл `stdio.h`

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:

```
1 int a;
2 unsigned int b;
3 unsigned c;
```

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

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

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

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

```
1 unsigned a = 54;
2 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:

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

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

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

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

```
1 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*¹⁶. А тип *long long int* может использовать больше памяти чем *long int*:

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

¹⁶Ключевыми словами в прошлых предложениях выступает слово 'может'. Например, переменная типа *short* может быть меньше переменной *int*, но может и не быть.

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:

```

1 #include <stdio.h>
2
3 int main() {
4     printf("%d\n", sizeof(short));
5     printf("%d\n", sizeof(int));
6     printf("%d\n", sizeof(long));
7     printf("%d\n", sizeof(long long));
8     return 0;
9 }
```

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:

```
1 float a;
2 double b;
```

Проблему с точностью представления вещественных чисел можно заметить в листинге:

```
1 #include <stdio.h>
2
3 int main() {
4     float a = 123456789;
5     float b = 1;
6     float c = a + b;
7     printf("%f %f", a, c);
8     return 0;
9 }
```

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:

```
1 float a = 1.424;
2 double b = 5242e23;
3 printf("%f\n", a); // 1.424000
4 printf("%e\n", a); // 1.424000e+000
5
6 // в следующем выводе видна проблема представления вещественных чисел
7 printf("%f\n", b); // 52420000000000000000000000000000.000000
8 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:

```
1 // <тип> <переменная>
2 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):

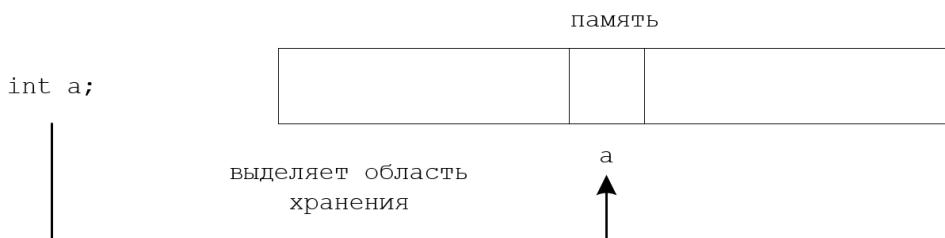


Рис. 2.2 – Объявление переменной

Одновременно можно объявить несколько переменных одного типа:

```
1 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).

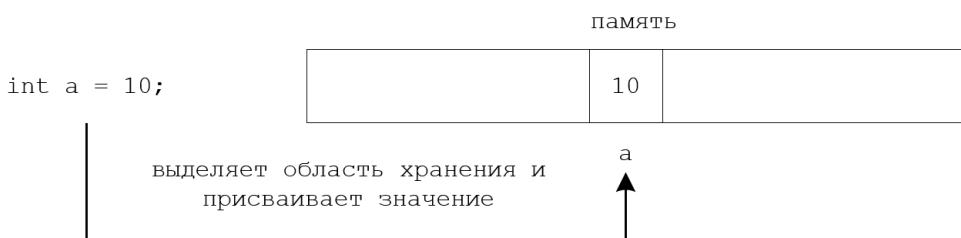


Рис. 2.3 – Объявление и инициализация

Значение неинициализированной переменной неопределено. Можно было бы предложить инициализировать целые значения нулём. 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):

```
1 int a = 10;
2 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*:

```
1 int a;
2 scanf("%d", &a);
3
4 printf("%d", a);
```

Переменные могут быть инициализированы результатом вычисленного выражения:

```
1 int a = 10;
2 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.

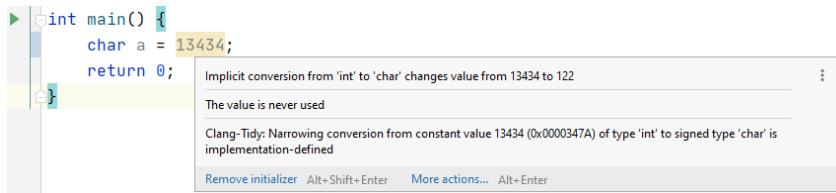


Рис. 2.4 – Предупреждение среды о неявном преобразовании значения 13434 в 122.

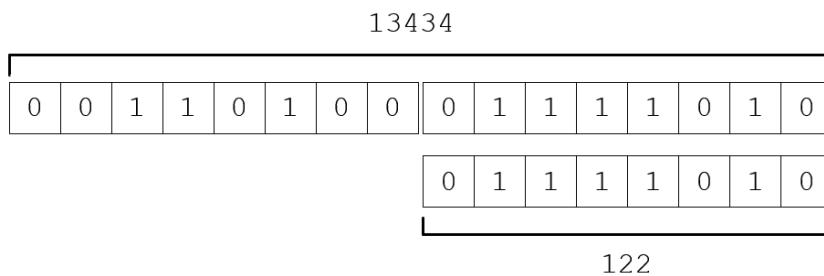


Рис. 2.5 – В процессе присваивания старший байт числа 13434 будет отброшен.
Остается значение 122.

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:

```
1 int a[10];      // массив из 10 элементов
2 int b[100];     // массив из 100 элементов
3 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 нуля¹⁷. Отображение на память представлено на рисунке 2.6.

```

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

```

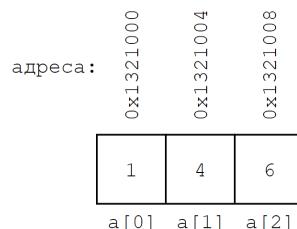


Рис. 2.6 – Расположение элементов массива в памяти

Массив может быть инициализирован при объявлении:

```

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

```

Приведём небольшой фрагмент, в котором покажем работу с массивом:

```

1 #include <stdio.h>
2
3 int main() {
4     // объявление и инициализация массива размера 3
5     // компилятор сам может вычислить размер массива, если
6     // имеется список инициализации
7     int a[] = {1, 4, 2};
8
9     // вывод a[0]
10    printf("%d\n", a[0]);
11
12    // присвоение a[1] значения 100
13    a[1] = 100;
14
15    // ввод значения a[2]
16    scanf("%d", &a[2]);
17
18    // вывод трёх элементов массива
19    printf("%d ", a[0]);
20    printf("%d ", a[1]);
21    printf("%d ", a[2]);
22
23    return 0;
24 }

```

¹⁷Нашел 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.

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 (*):

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

Для работы с указателями в Си определены две операции:

- 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**:

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

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

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

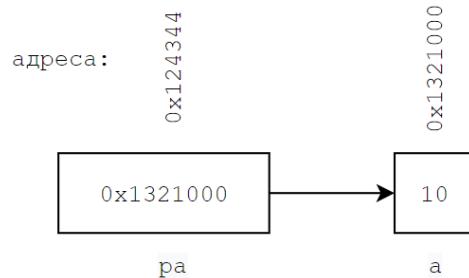


Рис. 2.7 – Переменная *pa* хранит адрес переменной *a*

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

```
1 *pa = 20;
```

Теперь значение переменной `a` равняется 20.

Ещё один пример программы:

```

1 #include <stdio.h>
2
3 int main() {
4     int a = 10;
5     int *pa = &a; // указатель хранит адрес переменной a
6     printf("%d %d\n", pa, &a);
7     printf("%d %d\n", *pa, a);
8
9     *pa = 4; // изменяем значение по адресу pa
10    printf("%d %d\n", *pa, a);
11
12    a = 1; // изменяем значение переменной a
13    printf("%d %d\n", *pa, a);
14
15    return 0;
16 }
```

Результат выполнения на рисунке 2.8 :

```

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:

```

1 int a[3] = {1, 4, 2};
2 *a = 3;
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

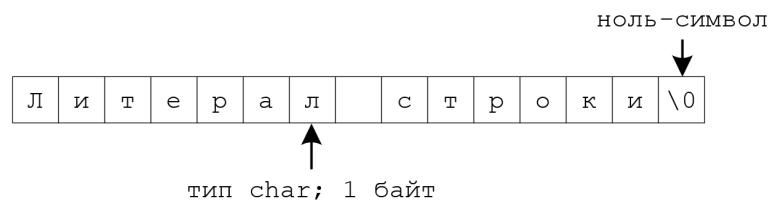
В языке программирования С отсутствует отдельный строковый тип. **Символьные строки** в нём являются последовательностью из одного или большего количества символов (массив символов). Строковые константы обрамляются в двойные кавычки¹⁸:

¹ "Литерал строки"

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

¹⁸Нужно понимать, что символ '`x`' и строка "`x`" - разные вещи. Первое – одиночный символ, второе – строка из двух символов: '`x`' и '`\0`'.

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:

```
1 char s[6] = "Hello";
2 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:

```
1 char s[] = "Hello\t!"; // размер массива s = 8 элемантам:
2 // 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`:

```
1 #include <stdio.h>
2
3 int main() {
4     char name[15];
5     gets(name);
6
7     printf("Hello %s", name); //вместо %s будет выведено значение name
8
9     return 0;
10 }
```

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

```
1 printf("It's a long string.
2         "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:

```
1 const int a = 10;
```

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.

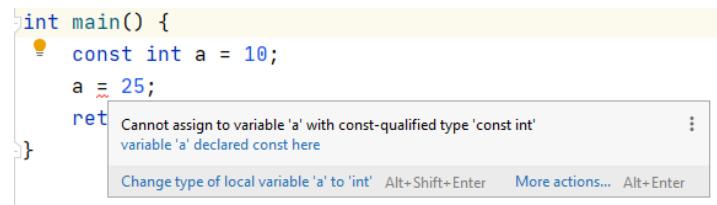


Рис. 2.9 – Попытка выполнить изменение константы

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

```

1 int main() {
2     int a = 10;
3
4     // p - может указывать на int переменные
5     // (а в С - ещё и на const int переменные, но не в C++)
6     // значение указателя может быть изменено
7     // значение по указателю может быть изменено
8     int *p;
9
10    // p1 - может указывать на int и const int переменные
11    // значение указателя может быть изменено
12    // значение по указателю не может быть изменено
13    const int *p1;
14
15    // p2 - может указывать на int и const int переменные
16    // (а в С - ещё и на const int переменные, но не в C++)
17    // значение указателя должно быть задано при инициализации и не может
18    // быть изменено
19    // значение по указателю может быть изменено
20    int * const p2 = &a;
21
22    // p3 - может указывать на int и const int переменные
23    // значение указателя должно быть задано при инициализации и не может
24    // быть изменено
25    // значение по указателю не может быть изменено
26    const int * const p3 = &a;
27
28    return 0;
29 }

```

В языке имеется способ изменения константы, объявленной как `const` через её указатель¹⁹:

```

1 #include <stdio.h>
2
3 int main() {
4     const int b = 10;
5     int *a = &b;
6     *a = 20;
7
8     printf("%d", b);
9
10    return 0;
11 }

```

При этом вы получите только предупреждение от компилятора.

¹⁹Такое поведение недопустимо в C++.

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

```

1 #include <stdio.h>
2
3 #define N 10
4
5 int main() {
6     int a = N;
7     printf("%d", a);
8     return 0;
9 }
```

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:

```

1 // содержимое <stdio.h>
2
3 int main() {
4     int a = 10; // выполнена замена N на 10
5     printf("%d", a);
6     return 0;
7 }
```

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 C отсутствует, вместо него используется массив символов.
- В C 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 C.
2. Целочисленный тип `char`. Вывод диапазона значений `char` / `unsigned char`.
3. Целочисленный тип `int`. Вывод диапазона значений.
4. Вещественный тип `float` и `double`. В чём разница между ними?
5. В каких системах счисления могут быть представлены целочисленные литералы? Приведите примеры.
6. Приведите примеры *escape*-последовательностей.
7. В каком случае инициализация переменной при ее объявлении будет излишней?

8. Массивы в С. Объявление. Инициализация.
9. Указатели в С. Операции взятия адреса и косвенного доступа. Макрос `NULL`.
10. В чем заключается разница между литералами `'x'` и `"x"`?
11. 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.

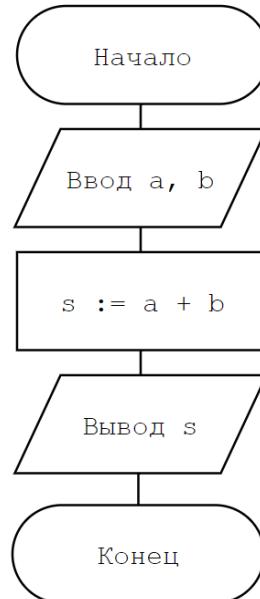


Рис. 3.1 – Блок-схема алгоритма задачи о сумме

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

```

1 #include <stdio.h>
2
3 int main() {
4
5     return 0;
6 }
```

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:

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5
6     return 0;
7 }
```

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):

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

`&a` – операция взятия адреса для переменной `a`¹. 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:

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

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

Выведем результат при помощи `printf`:

```
1 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:

```

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

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

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6
7     int a, b;
8     scanf ("%d %d", &a, &b);
```

¹`a` – переменная, `&a` – значение адреса, где хранится переменная `a`.

```

10 int s = a + b;
11
12 printf("Сумма = %d", s);
13
14 return 0;
15 }
```

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 ^a.

^aЭта задача может решаться как с использованием, так и без использования третьей переменной.

Блок-схема алгоритма представлена на рисунке 3.2 (страница 46).



Рис. 3.2 – Блок-схема алгоритма задачи обмена значений

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

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

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

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

Код программы:

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     scanf("%d %d", &a, &b);
6
7     int t = a;
8     a = b;
9     b = t;
10
11    printf("a = %d b = %d", a, b);
12
13    return 0;
14 }
```

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 47).

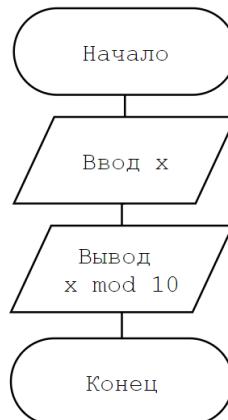


Рис. 3.3 – Блок-схема алгоритма задачи о последней цифре числа

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 и пример работы программы представлены ниже:

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     printf("last digit = %d", x % 10);
8
9     return 0;
10 }
```

```

243
last digit = 3
Process finished with exit code 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 49).

Для того чтобы воспользоваться функциями из математической библиотеки, необходимо их импортировать из `math.h`. Код и результат работы программы представлены на рисунке 3.5.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int x1, x2, y1, y2;
6     scanf("%d %d", &x1, &y1);
7     scanf("%d %d", &x2, &y2);
8
9     int dx = x2 - x1;
10    int dy = y2 - y1;
11    float distance = sqrt(dx*dx + dy*dy);
12
13    printf("distance = %f", distance);
14
15    return 0;
16 }
```

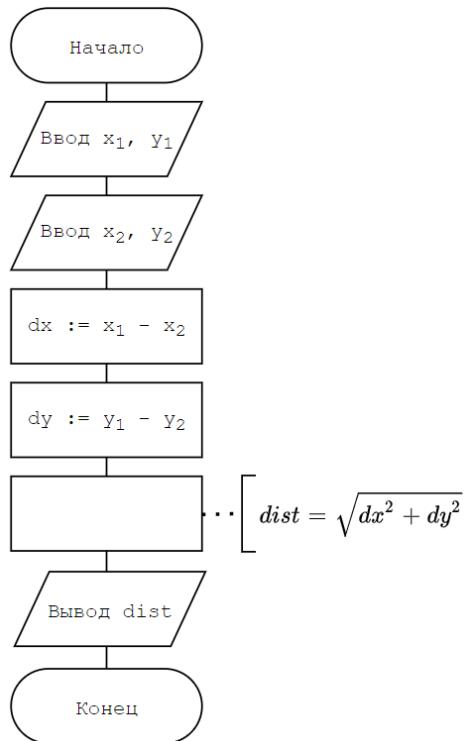


Рис. 3.4 – Блок-схема алгоритма задачи о вычислении расстояния между двумя точками

```

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 случайные².

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

- `rand()` 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.

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

- `rand()` генерирует следующее случайное число в последовательности. Оно будет находиться в диапазоне от 0 до `RAND_MAX` (константа и функции `srand`, `rand` определены в `<stdlib.h>`; значение `RAND_MAX` в моей системе она равна $7FFF_{16}$).

Рассмотрим на примерах³:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     int a = rand();                                // генерация чисел от 0 до RAND_MAX
6     int b = rand() % 10;                           // генерация чисел от 0 до 9
7     int c = rand() % 10 + 3;                      // генерация чисел от 3 до 12
8     printf("%d %d %d ", a, b, c);
9
10    a = 5;
11    b = 10;
12
13    int d = rand() % a;                          // генерация чисел от 0 до a - 1
14    int e = rand() + b;                          // генерация чисел от b до RAND_MAX + b
15    int f = rand() % b + a;                      // генерация чисел от a до a + b - 1
16    int g = a + rand() % (b - a + 1);           // генерация чисел от a до b
17    printf("%d %d %d %d", d, e, f, g);
18
19    return 0;
20 }
```

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:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     srand(42);
6     ...
7 }
```

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

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4
5 int main() {
6     srand(time(0));
7     ...
8 }
```

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

⁴Функция возвращает текущее календарное значение времени в секундах.

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:

```

1 #include <stdio.h>
2
3 int main() {
4     int a1, an, d;
5     scanf("%d %d %d", &a1, &an, &d);
6
7     int n = (an - a1) / d + 1;
8     int s = (a1 + an) / 2 * n;
9
10    printf("%d", s);
11
12    return 0;
13 }
```

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:

```
1 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:

```
1 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:

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

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

Таким образом, код функции `main`:

```
1 #include <stdio.h>
2
3 int main() {
4     int a1, an, d;
5     scanf("%d %d %d", &a1, &an, &d);
6
7     int n = (an - a1) / d + 1;
8     int s = (a1 + an) * n / 2;
9
10    printf("%d", s);
11
12    return 0;
13 }
```

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`:

```
1 #include <stdio.h>
2
3 int main() {
4     int F;
5     scanf("%d", &F);
6     // int C = (F - 32) / 9 * 5; Не сработает, так как деление целочисленное
7     // Предположим, будет введено F = 40. Тогда выполняются следующие расчеты:
8     // (40 - 32) / 9 * 5 -> 8 / 9 * 5 -> 0 * 5 -> 0
9
10    // Ситуацию можно решить одним из следующих способов:
11    // 1. Представить 9 как литерал вещественного типа
12    // (40 - 32) / 9.0 * 5 -> 8 / 9.0 * 5 -> 0.888889 * 5 -> 4.444445
13    float C = (F - 32) / 9.0 * 5;
14    // 2. Представить 32 как литерал вещественного типа:
15    // float C = (F - 32.0) / 9 * 5;
16    // (40 - 32.0) / 9 * 5 -> 8.0 / 9 * 5 -> 0.888889 * 5 -> 4.444445
17    // 3. Заменить / 9 * 5 на / 1.8
18    // float C = (F - 32) / 1.8;
19
20    printf("%f", C);
21    return 0;
22 }
```

Случай второй: F – переменная типа float:

```

1 #include <stdio.h>
2
3 int main() {
4     float F;
5     scanf("%f", &F);
6     float C = (F - 32) / 9 * 5;
7     // (40.0 - 32) / 9 * 5 -> 8.0 / 9 * 5 -> 0.888889 * 5 -> 4.444445
8     printf("%f", C);
9     return 0;
10 }
```

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 случайный 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:

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

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

```
1 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:

```
1 char a = 28;
2 char b = 10;
3 char c = a + b;
4 char d = a + 100;
5 printf("%d %d", c, d);
```

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

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

+	0	0	0	1	1	1	0	0	28
	0	0	0	0	1	0	1	0	10
	0	0	1	0	0	1	1	0	38

Первый пример Америку не открыл. А теперь второй:

+	0	0	0	1	1	1	0	0	28
	0	1	1	0	0	1	0	0	100
	1	0	0	0	0	0	0	0	0

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:

```

1 unsigned char a = 255;
2 unsigned char b = 1;
3 unsigned char c = a + b;
4 printf("%d", c);

```

Снова посчитаем столбиком:

+	1	1	1	1	1	1	1	1	255
	0	0	0	0	0	0	0	1	1
	1	0	0	0	0	0	0	0	0

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.

¹Магия 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.

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`:

```
1 unsigned char a = 255;
2 unsigned char b = 1;
3 short c = a + b;
4 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:

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

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

лучше остановиться на более короткой форме:

```
1 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:

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

Выполнить увеличение переменной можно и посредством инкремента:

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

а вычесть единицу при помощи декремента:

```

1 a--; // постфиксный декремент
2 --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 выражений². Рассмотрим такой пример:

```

1 int a = 4;
2 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 нашем случае, сначала будет переменная, а потом операция. То есть сперва будет использована переменная `a` для выражения:

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

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

```

1 int a = 4;
2 int b = 1 + a;
3 a += 1;

```

Если бы инкремент был бы префиксным:

```

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

```

тогда поведение было бы эквивалентно следующему:

```

1 int a = 4;
2 a += 1;
3 int b = 1 + a;

```

Ситуация с декрементом аналогична³.

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

```

1 int a, b;
2 scanf("%d %d", &a, &b);
3
4 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,

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

³Теперь 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.

что были введены значения $a = 100000$ и $b = 100000$. Переведём значения в двоичную систему счисления (красным выделены 32 бита результата):

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

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

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 70).

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:

```

1 #include <stdio.h>
2
3 int main() {
4     int a = 10;
5     int b = 3;
6     printf("%d\n", a / b);    // 3
7     printf("%d\n", a / 4);    // 2
8     printf("%d\n", 100 / b); // 33
9     printf("%d\n", 100 / 9); // 11
10    return 0;
11 }
```

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

```

1 #include <stdio.h>
2
3 int main() {
4     float a = 10;
5     float b = 3;
6     printf("%f\n", a / b);    // 3.333333
7     printf("%f\n", a / 4);    // 2.500000
8     printf("%f\n", 100 / b); // 33.333333
9     printf("%f\n", 100. / 9); // 11.111111
10    return 0;
11 }
```

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

```

1 int a = 10;
2 int b = 6;
3 printf("%d\n", a % b);    // 4
4 printf("%d\n", a % 4);    // 2
5 printf("%d\n", 100 % b);  // 4
6 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. таблицу 4.1).⁴

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:

```

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

```

4.2 Побитовые операции

Побитовые операции⁵ – 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.

⁴Очерёдность 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.

⁵Рекомендуется использовать беззнаковые целые в качестве operandов для побитовых операций.

Существуют следующие операции побитового сдвига:

1. Побитовый сдвиг влево (\ll).
2. Побитовый сдвиг вправо (\gg).

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.

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

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.

&	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr><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><td></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><td></td></tr></table>										0	0	1	0	1	0	1	0		0	0	0	1	1	0	0	0		a = 42
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%;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></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><td></td></tr><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><td></td></tr></table>										0	0	0	1	1	0	0	0		0	0	0	0	1	0	0	0		b = 24
0	0	0	1	1	0	0	0																						
0	0	0	0	1	0	0	0																						
	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 10px; height: 10px;"></td><td style="width: 10px; height: 10px;"></td></tr><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><td></td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td></td></tr></table>										0	0	0	0	1	0	0	0		0	0	0	0	0	1	0	0		f = 8
0	0	0	0	1	0	0	0																						
0	0	0	0	0	1	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:

```

1 char a = 42;
2 char b = 24;
3 char f = a & b;
4 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

<code> </code>	0 0 1 0 1 0 1 0	<code>a = 42</code>
	0 0 0 1 1 0 0 0	<code>b = 24</code>
	<hr/>	
	0 0 1 1 1 0 1 0	<code>f = 58</code>

Рис. 4.2 – Побитовое ИЛИ двух чисел

Код примера:

```

1 char a = 42;
2 char b = 24;
3 char f = a | b;
4 printf("%d", f);

```

На экране отобразится значение 58⁶.

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)⁷ получим значение 213:

0 0 1 0 1 0 1 0	<code>a = 42</code>
<hr/>	
1 1 0 1 0 1 0 1	<code>f = 213</code>

Рис. 4.3 – Побитовое НЕ

⁶Работает? Не трогай. (Любой программист).

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

```

1 unsigned char a = 42;
2 unsigned char f = ~a;
3 printf("%d", f); // 213

```

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" style="border-collapse: collapse; width: 100%;"> <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%;"> <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	$f = 50$								
0	0	1	1	0	0	1	0											

Рис. 4.4 – Побитовое исключающее ИЛИ двух чисел

```

1 char a = 42;
2 char b = 24;
3 char f = a ^ b; // 50
4 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:

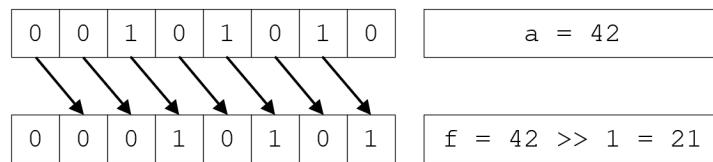


Рис. 4.5 – Побитовый сдвиг на 1 вправо

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

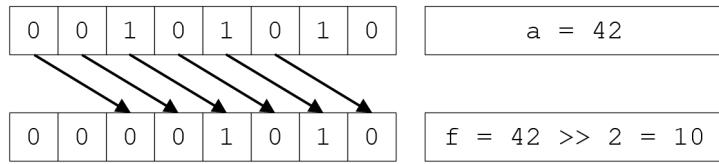


Рис. 4.6 – Побитовый сдвиг на 2 вправо

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:

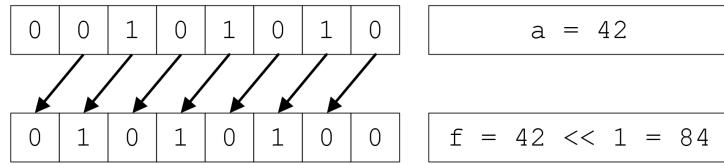


Рис. 4.7 – Побитовый сдвиг на 1 влево

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

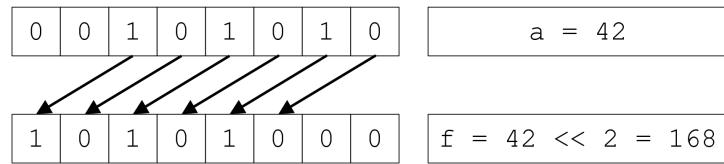


Рис. 4.8 – Побитовый сдвиг на 2 влево

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 единицу⁸:

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

⁸Скобки в примере не являются обязательными.

Пример:

			5 -H					
n				1 0 0 1 1 0 1 1				
1 << i				0 0 1 0 0 0 0 0				
n 1 << 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:

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

Пример

			5 -H					
n				1 0 1 0 1 0 1 1				
1 << 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)⁹:

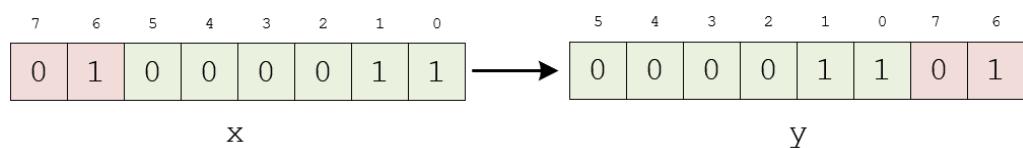


Рис. 4.11 – Циклический сдвиг влево на 2

⁹Если 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.

Вычисления можно организовать так:

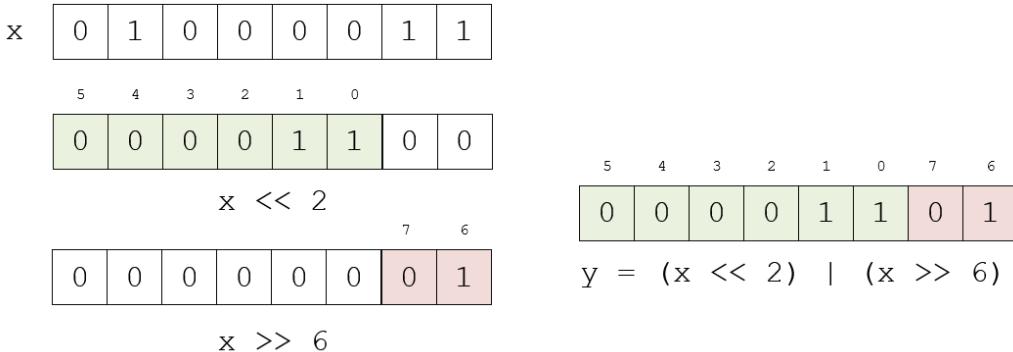


Рис. 4.12 – Процесс вычисления побитового сдвига влево на 2

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:

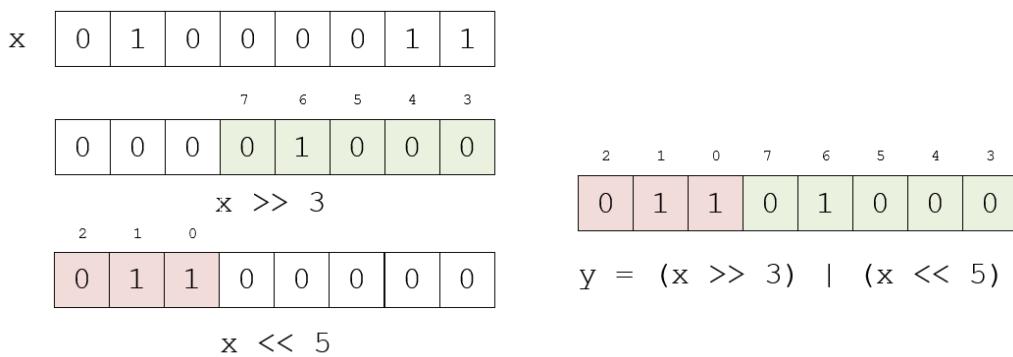


Рис. 4.13 – Циклический сдвиг вправо на 3

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><<</code> <code>>></code>	Слева направо
<code>&</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:

```

1 int a = 5;
2 if (a = 10)
3     printf("a = 10");
4 else
5     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 равенство¹⁰:

```

1 #include <stdio.h>
2
3 int main() {
4     if (0.1 + 0.1 + 0.1 == 0.3)
5         printf("=");
6     else
7         printf("!=");
8     return 0;
9 }

```

Программа выведет, что данные значения неравны¹¹. Как жить с этим? – находить модуль разности между левой и правой частью, и сравнивать с некой допустимой погрешностью:

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float a = 0.3;
6     float b = 0.1 + 0.1 + 0.1;
7     float eps = 0.00000001;
8
9     if (fabs(a - b) < eps)
10        printf("=");
11     else
12        printf("!=");
13
14     return 0;
15 }

```

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

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

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

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

- `!` - отрицание
- `&&` - 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 прекращаются¹². Пример:

```
1 int a = 10;
2 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:

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

Проверить год на високосность можно следующим образом:

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

Проверка на то является ли символ *c* цифрой:

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

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

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

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

```
1 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.

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

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

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

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

1. Ставить скобки для явного указания порядка вычисления.
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>-></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>&</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><<</code> <code>>></code>	Побитовый сдвиг влево Побитовый сдвиг вправо	Слева направо
6	<code><</code> <code><=</code> <code>></code> <code>>=</code>	Меньше Меньше или равно Больше Больше или равно	Слева направо
7	<code>==</code> <code>!=</code>	Равно Не равно	Слева направо
8	<code>&</code>	Побитовое И	Слева направо
9	<code>^</code>	Побитовое исключающее ИЛИ	Слева направо
10	<code> </code>	Побитовое ИЛИ	Слева направо
11	<code>&&</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

Так же не определена очерёдность вычисления аргументов функций:

```

1 int a = 5;
2 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 повышение¹³. Если `int` не способен представить значение операнда – происходит повышение до `unsigned int`¹⁴.

```

1 char c = '?';
2 unsigned short var = 100;
3
4 if (c < 'A')           // Символьная константа 'A' имеет тип int;
5                         // Значение переменной с продвигается до int
6                         // для выполнения сравнения.
7
8     var = var + 1;      // До сложения, значение переменной var будет продвинуто
                         // до int или unsigned int.
9

```

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.

¹³Объекты 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**.

¹⁴Если `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`.

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

Рассмотрим правила на примерах. Неустаревающая классика:

```

1 #include <stdio.h>
2
3 int main() {
4     // выведет ?!
5     unsigned a = 1;
6     if (a > -1)
7         printf("a > -1");
8     else
9         printf("?! ");
10    return 0;
11 }
```

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

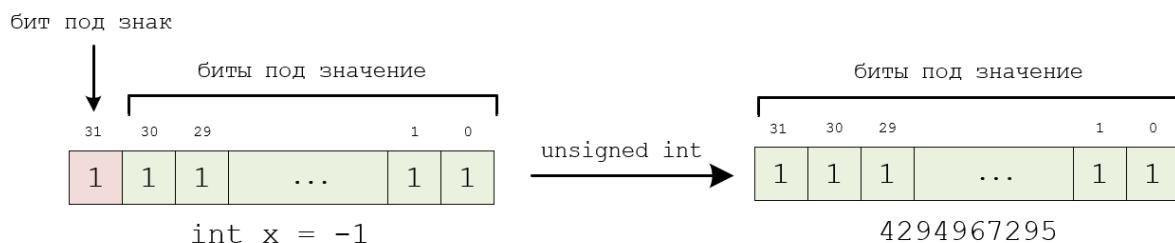


Рис. 4.14 – Преобразование из `int` в `unsigned int`

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

Второй случай:

```

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

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

Если следовать приоритету операций, сперва выполнится умножение. Умножаются две переменных различных типов. Тогда после приведения переменная `a` будет

¹⁵Существуют 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.

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 операнд 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:

```

1 int i = -1;
2 unsigned limit = 200U;
3 long n = 30L;
4
5 if (i > limit)      // (int > unsigned int) -> (unsigned int > unsigned int) ->
6                      // (-1 > 200)           -> (4294967295 > 200)           -> 1
7
8
9     x = limit * n; // вариант будет выбран в зависимости от архитектуры
10                  // вариант 1 - если long может уместить значения unsigned int
11                  // (unsigned int * long) -> (long * long)
12                  // вариант 2 - если long не может уместить значения unsigned int
13                  // (unsigned int * 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:

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int a, b, c;
6     scanf("%d %d %d", &a, &b, &c);
7
8     long long D = (long long)b*b - (long long)4*a*c;
9     if (D > 0) {
10         float sqrtD = sqrt(D);
11         float x1 = (-b + sqrtD) / (2 * a);
12         float x2 = (-b - sqrtD) / (2 * a);
13         printf("%f %f", x1, x2);
14     } else if (D == 0) {
15         float x = -b / (2.0 * a);
16         printf("%f", x);
17     } else
18         printf("No roots");
19
20     return 0;
21 }
```

Приведения:

```
1 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:

```

1 float x1 = (-b + sqrtD) / (2 * a);
2 float x1 = (-int + float) / (int * int);
3 float x1 = (-float + float) / int;
4 float x1 = float / int;
5 float x1 = float / float;
6 float x1 = float;

```

Проанализируем и другие строки таким образом:

```

1 float x = -b / (2.0 * a);
2 float x = -int / (double * int);
3 float x = -int / (double * double);
4 float x = -int / double;
5 float x = -double / double;
6 float x = double

```

Когда значение будет вычислено, будет выполнено приведение типов при присваивании.

Рассмотрим ещё один исключительно учебный пример:

```

1 char ch;
2 int i;
3 float f;
4 double d;
5
6 int result = (ch / i) + (f * d) - (f + i);
7 int result = (char / int) + (float * double) - (float + int);
8 int result = (int / int) + (double * double) - (float + float);
9 int result = int + double - float;
10 int result = double + double - float;
11 int result = double - float;
12 int result = double - double;
13 int result = double;

```

За счёт последнего приведения типов вещественная часть будет отброшена.

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

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 58 в одном из прошлых случаев можно так:

```

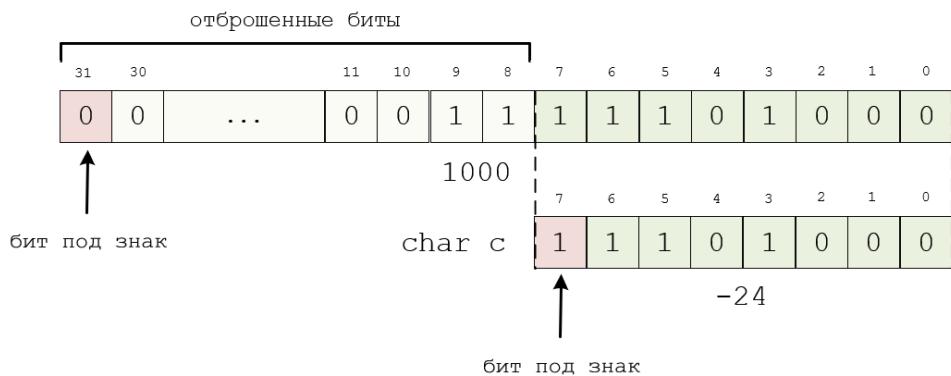
1 int a, b;
2 scanf("%d %d", &a, &b);
3
4 long long c = (long long)a * b;

```

4.5.3 Последствия при приведении типов

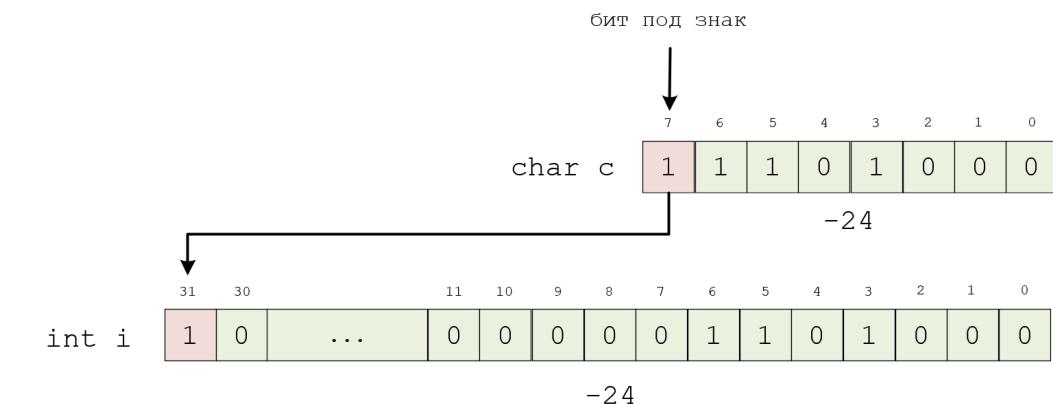
Если целое большего размера приводится к целому меньшего размера, то старшие байты отбрасываются:

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

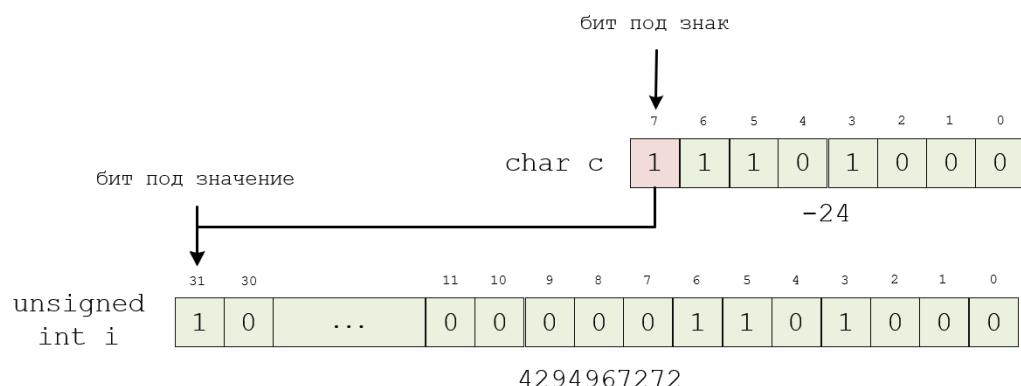


Целое меньшего размера преобразуется к целому типу большего размера, не изменяя своего значения с сохранением знака:

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



```
1 char c = -24;
2 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
1 int i1 = 100.25; // 100
2 int i2 = 1.99; // 1
3 int i3 = 300.00; // 300
```

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

```

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

```

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

```

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

```

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

```

1 float f = 1000.1234; // 1000.123413
2 float f2 = 1000.1234f; // 1000.123413
3 double lf = 1000.1234; // 1000.123400

```

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

```

1 float f = 1234.012f; // f = 1234.011963
2 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. Дайте определение терминам 'операция' и 'операнд'. Перечислите виды операций в зависимости от количества operandов. Приведите примеры.
2. Арифметические операции и их приоритеты.
3. Какие существуют способы в C увеличить или уменьшить значение переменной на единицу?
4. В чем заключается разница между постфиксной и префиксной записи инкремента и декремента?
5. Какая проблема может возникнуть при умножении чисел в C?
6. Каким образом работает деление в языке программирования C.
7. Побитовые операции и их приоритеты.
8. Перечислите какие бывают приведения типов в C. В каких случаях они выполняются?
9. Перечислите последствия при приведении типов. Приведите примеры.
10. Логические операции и их приоритеты. Опишите как работает 'ленивая' схема вычислений.

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.

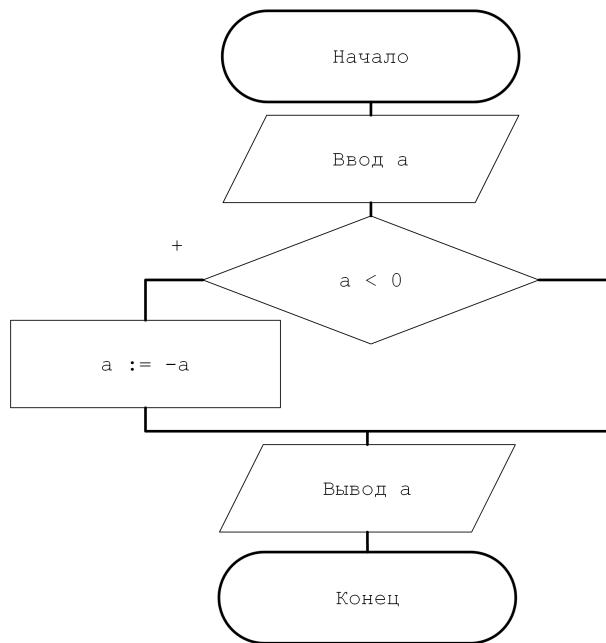


Рис. 5.1 – Блок-схема поиска значения модуля числа

2. Ставятся фигурные скобки.

3. В фигурных скобках одна за другой перечисляются операторы по ветке '+'.

Если вы умеете выполнять алгоритмы в словесном виде, получится следующее:

```

1 if (a < 0) {
2     a = -a;
3 }
```

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

```

1 #include <stdio.h>
2
3 int main() {
4     int a;
5     scanf("%d", &a);
6
7     if (a < 0)
8         a = -a;
9
10    printf("%d", a);
11
12    return 0;
13 }
```

Общая форма оператора `if`:

```

1 // if (<логическое выражение>)
2 //     <оператор>
```

Если `<логическое выражение>` является истинным, то `<оператор>` выполняется. `<Оператор>` может быть как одиночным, так и составным¹:

```

1 if (a < 0)
2     a = -a;
```

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

```

3 if (b > 0) {
4     a = b * 2;
5     b = a + 3;
6 }
```

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:

```

1 if (a < 0)
2     a = -a;
3     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:

```

1 if (a < 0)
2     a = -a;
3 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 отличии 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 `'+'`.
4. После второй закрывающей скобки пишется инструкция `else`, после которой ставятся фигурные скобки.
5. В фигурных скобках одна за другой перечисляются инструкции по ветке `'-'`.

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.

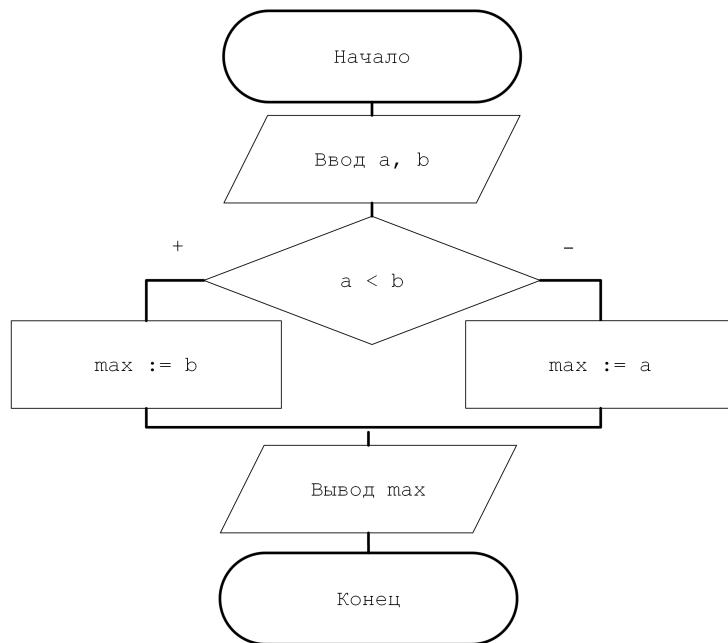


Рис. 5.2 – Блок-схема к задаче о максимальном значении из двух

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

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     scanf("%d %d", &a, &b);
6
7     // фигурные скобки необязательны
8     int max;
9     if (a < b)
10         max = b;
11     else
12         max = a;
13
14     printf("%d", max);
15
16     return 0;
17 }
  
```

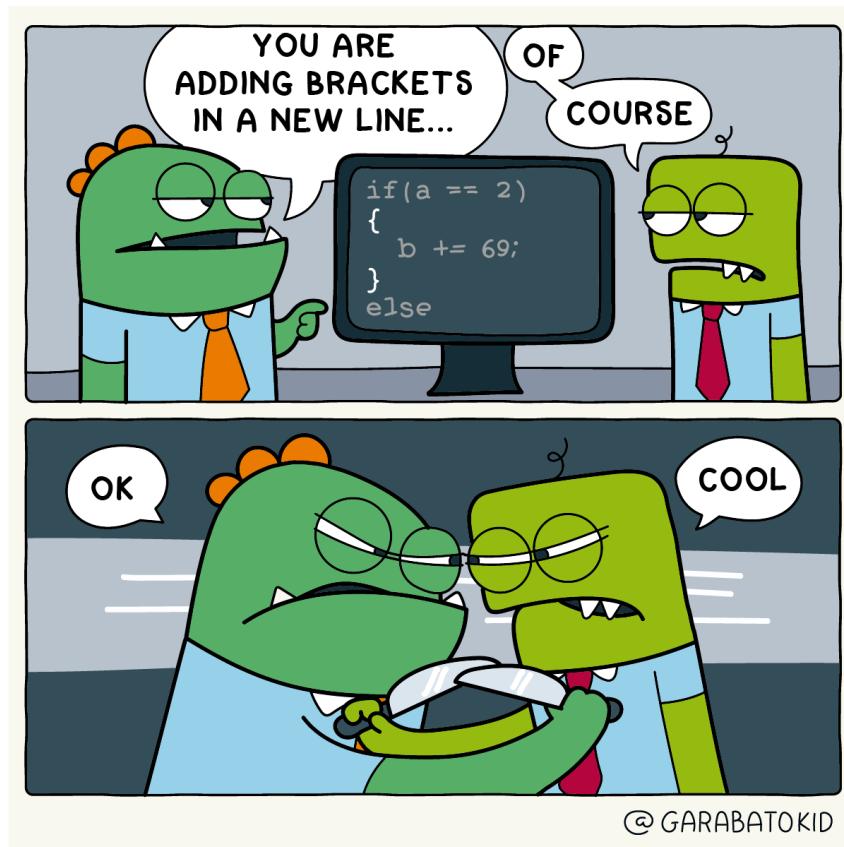
Общая форма оператора `if-else`:

```

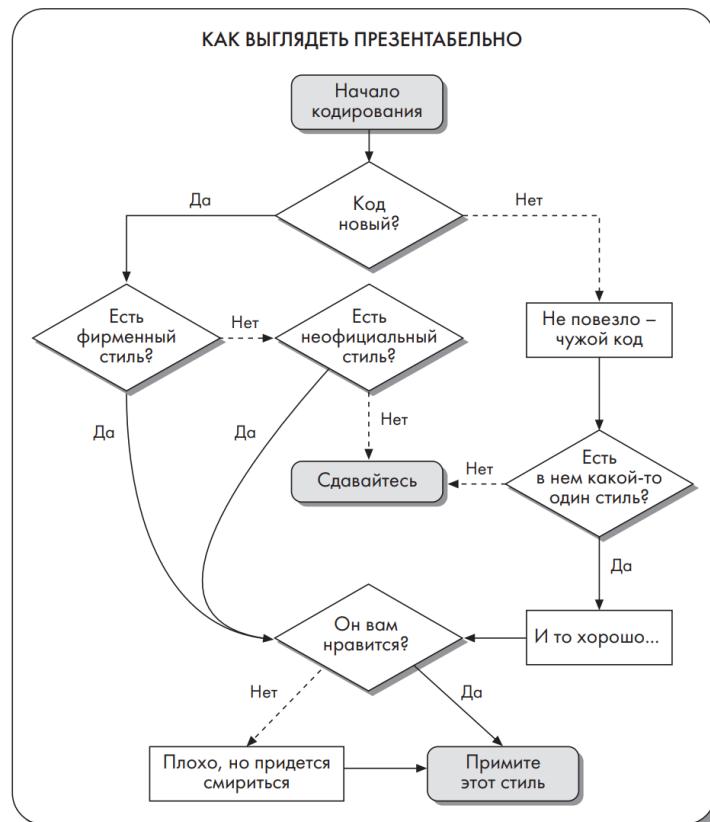
1 // if (<логическое выражение>)
2 //     <оператор1>
3 // else
4 //     <оператор2>
  
```

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):

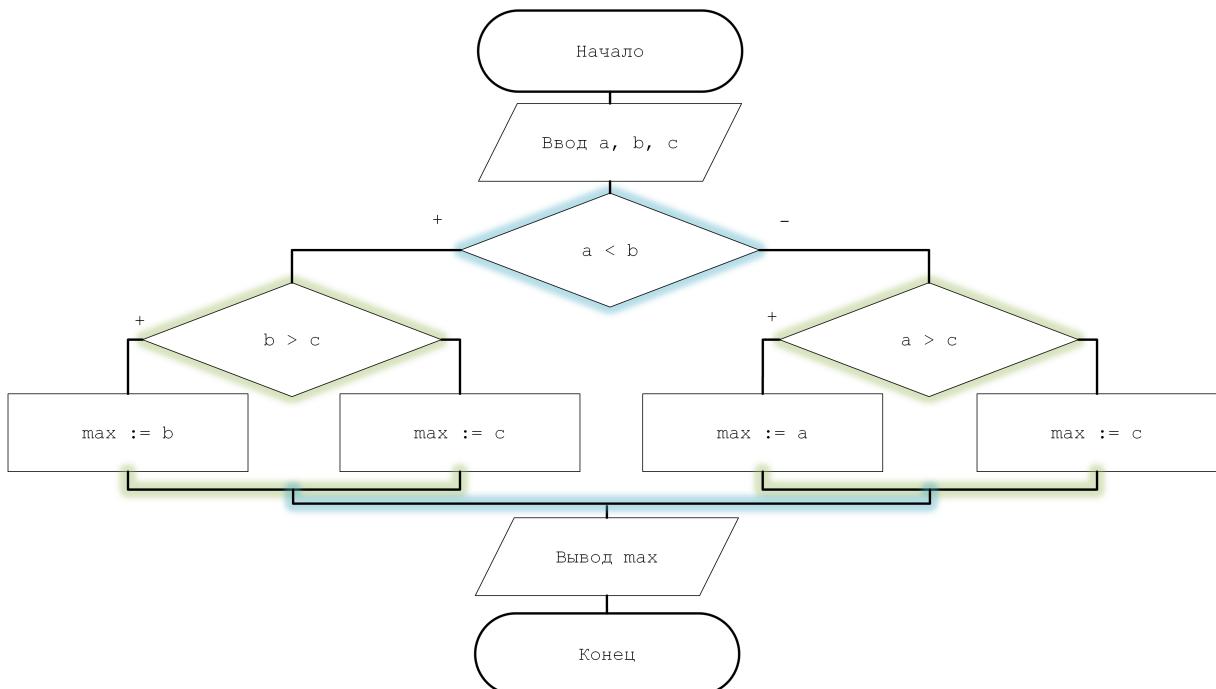


Рис. 5.3 – Блок-схема для решения задачи поиска максимума из двух

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 закроются 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.

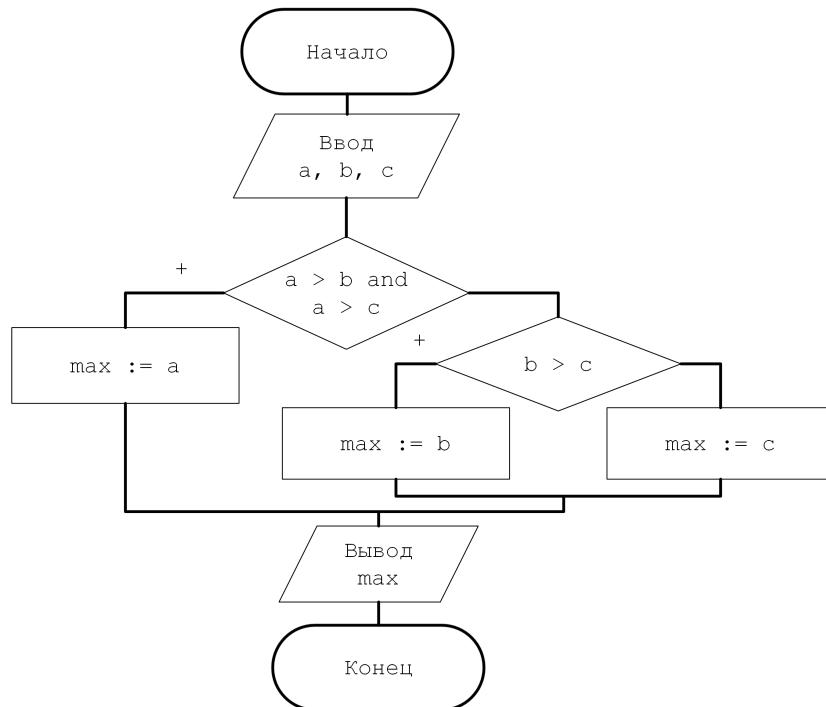
Код решения прошлой задачи представлен на листинге ниже:

```
1 #include <stdio.h>
2
3 int main() {
4     int a, b, c;
5     scanf ("%d %d %d", &a, &b, &c);
6
7     int max;
8     if (a < b) {
9         if (b > c)
10             max = b;
11
12     else
```

```

12         max = c;
13     } else {
14         if (a > c)
15             max = a;
16     else
17         max = c;
18 }
19
20 printf("%d", max);
21
22 return 0;
23 }
```

В следующем варианте условие из крайнего левого блока 'перешло' в первый блок 'решение', но проблема с расширением никуда не исчезнет:

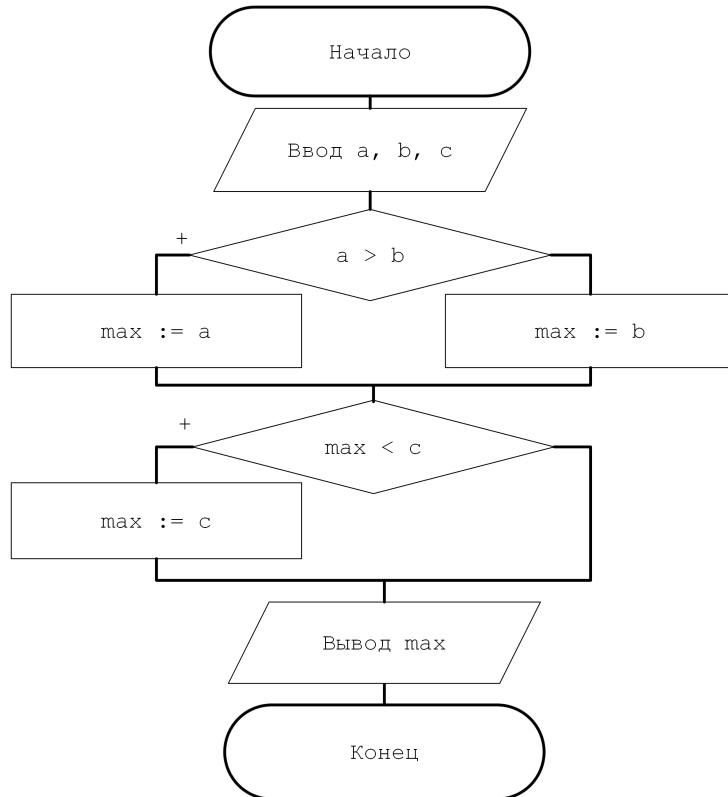


Код программы:

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b, c;
5     scanf("%d %d %d", &a, &b, &c);
6
7     int max;
8     if (a > b && a > c)
9         max = a;
10    else {
11        if (b > c)
12            max = b;
13        else
14            max = c;
15    }
16
17    printf("%d", max);
18
19    return 0;
20 }
```

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 времени².

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b, c;
5     scanf("%d %d %d", &a, &b, &c);
6
7     int max;
8     if (a > b)
9         max = a;
10    else
11        max = b;
12
13    if (max < c)
14        max = c;
15
16    printf("%d", max);
17
18    return 0;
19 }
```

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

```

1 if (a > b)
2     max = a;
```

²Есть 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)

```

3 else
4     max = b;

```

которые лучше закодировать при помощи тернарной операции³:

```

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

```

Используйте тернарную операцию для таких случаев.

Тернарная операция принимает три операнда:

```

1 // <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.

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b, c;
5     scanf("%d %d %d", &a, &b, &c);
6
7     int max = a;
8     if (max < b)
9         max = b;
10    if (max < c)
11        max = c;
12
13    printf("%d", max);
14
15    return 0;
16 }

```

³Было 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".

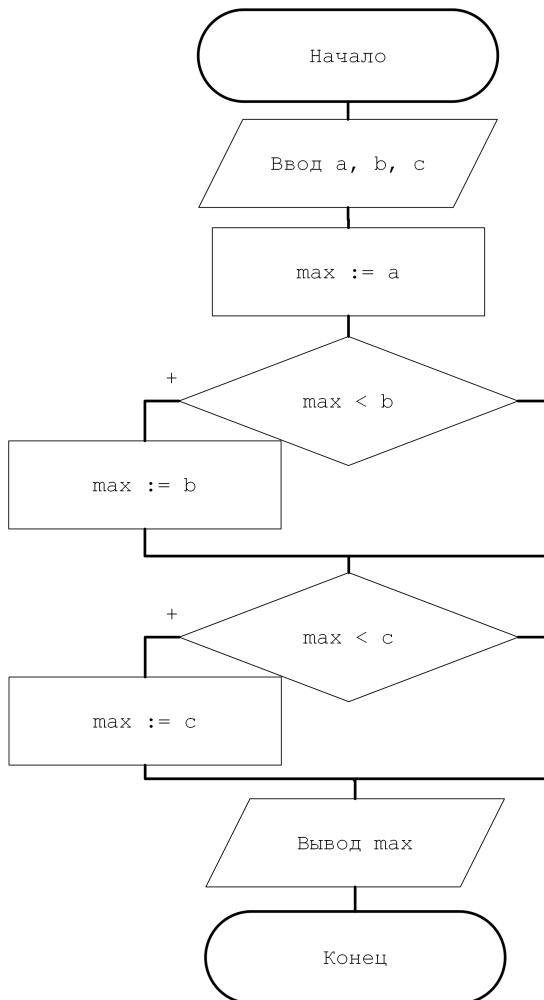


Рис. 5.4 – Блок схема алгоритма поиска максимального значения из трёх

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):

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     scanf("%d %d", &a, &b);
6
7     if (a > b) {
8         int t = a;
9         a = b;
10        b = t;
11    }
12
13    printf("%d %d", a, b);
14
15    return 0;
16 }
```

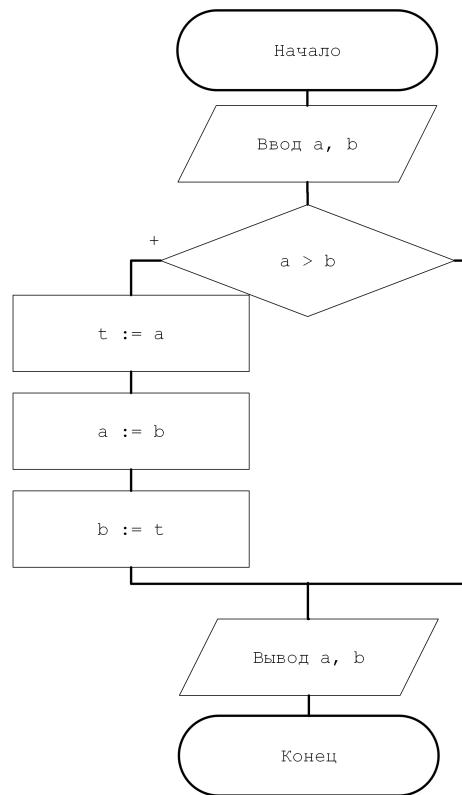


Рис. 5.5 – Блок-схема алгоритма упорядочивания двух чисел

5.1.5 Решение линейного уравнения

Решение линейного уравнения

Решить уравнение $ax + b = 0$.

Блок-схема представлена на странице 13, код – на странице 12.

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:

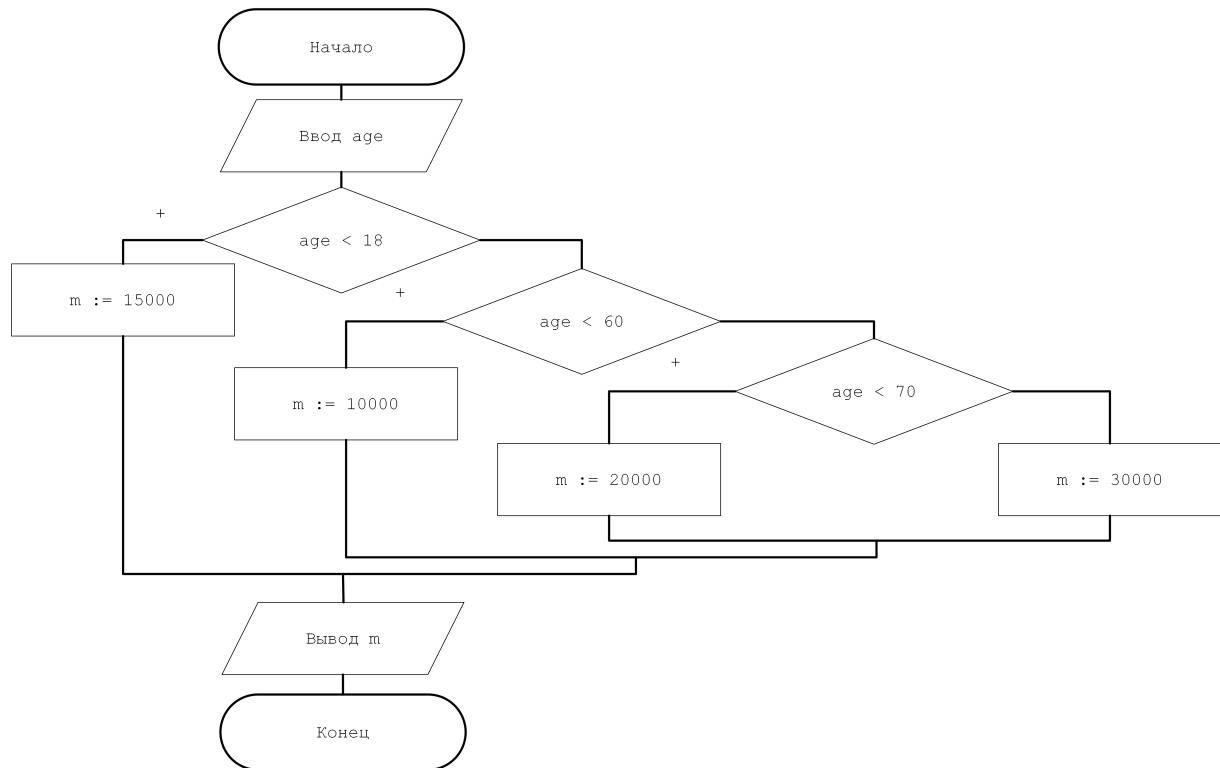


Рис. 5.6 – Блок-схема алгоритма решения задачи о социальной поддержке

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:

```

1 if (age < 18)
2     m = 15000;
3 else {
4     if (age < 60)
5         m = 10000;
6     else {
7         if (age < 70)
8             m = 20000;
9         else
10            m = 30000;
11    }
12 }
```

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

```

1 if (age < 18)
2     m = 15000;
3 else {
4     if (age < 60)
5         m = 10000;
6     else
7         if (age < 70)
8             m = 20000;
9         else
10            m = 30000;
11 }
```

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

```

1 if (age < 18)
2     m = 15000;
3 else
4     if (age < 60)
5         m = 10000;
6     else
7         if (age < 70)
8             m = 20000;
9     else
10        m = 30000;

```

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

```

1 if (age < 18)
2     m = 15000;
3 else if (age < 60)
4     m = 10000;
5 else if (age < 70)
6     m = 20000;
7 else
8     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:

```

1 if (points > 6)
2     if (points < 12)
3         printf("Количество очков > 6 и < 12")
4 else
5     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:

```

1 if (points > 6)
2     if (points < 12)
3         printf("Количество очков > 6 и < 12")
4     else
5         printf("Количество очков <= 6")

```

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

```

1 if (points > 6) {
2     if (points < 12)
3         printf("Количество очков > 6 и < 12")
4 } else
5     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:

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <windows.h>
4
5 int main() {
6     SetConsoleOutputCP(CP_UTF8);
7
8     long long a, b, c;
9     scanf("%lld %lld %lld", &a, &b, &c);
10
11    long long D = b*b - 4*a*c;
12    if (D > 0) {
13        float sqrtD = sqrtl(D);
14        printf("%f %f", (-b + sqrtD) / (2*a), (-b - sqrtD) / (2*a));
15    } else if (D == 0) {
16        printf("%f", -b / (2.0*a));
17    } else {
18        printf("Действительных корней нет");
19    }
20
21    return 0;
22 }
```

Данное решение не является единственным и не претендует быть таковым.

5.3 Оператор *switch*

Структура оператора *switch*:

```

1 // switch (<целочисленное выражение>) {
2 //     case <константное выражение1>:
3 //         <операторы>
4 //     case <константное выражение2>:
5 //         <операторы>
6 //     ...
7 //     case <константное выражениеN>:
8 //         <операторы>
9 //     default:
10 //         <операторы>
11 // }
```

Оператор *switch* работает следующим образом:

1. Вычисляется значение <целочисленного выражения>.
2. Среди <константных выражений⁴ ищется вычисленный результат.

⁴Выражение называется **константным**, если оперирует только константами.

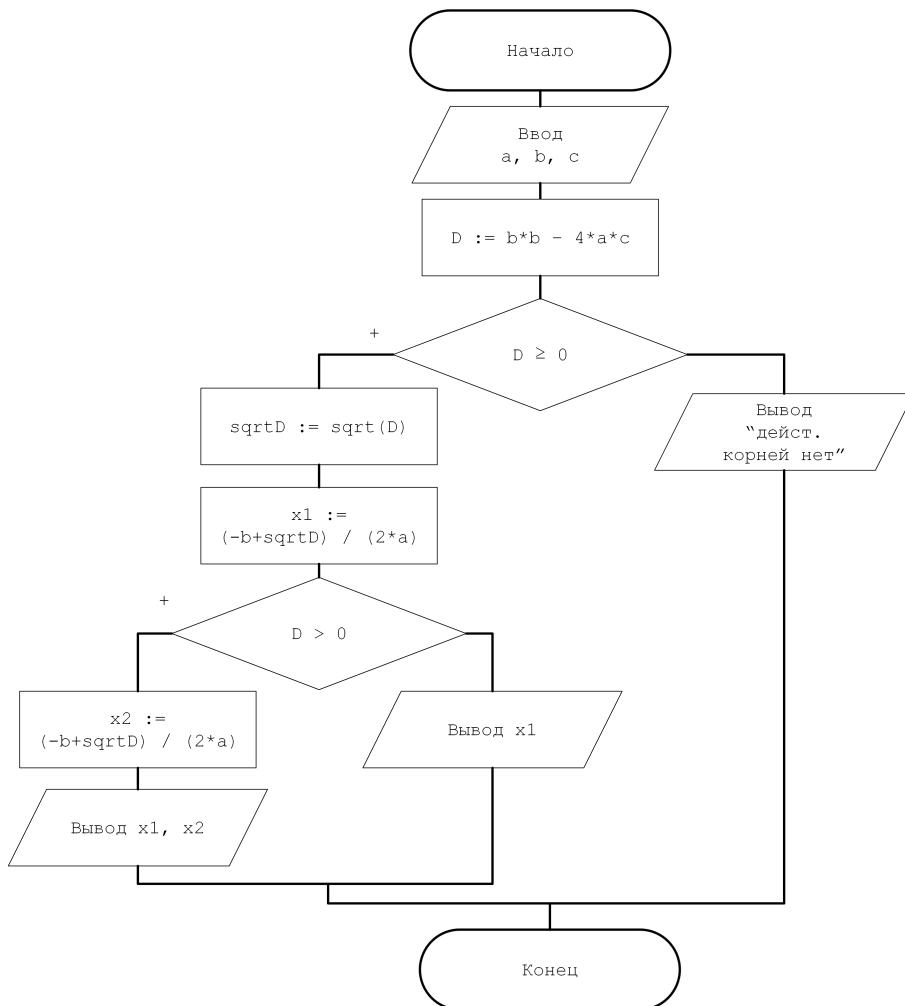


Рис. 5.7 – Блок-схема алгоритма решения квадратного уравнения

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 92) 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.

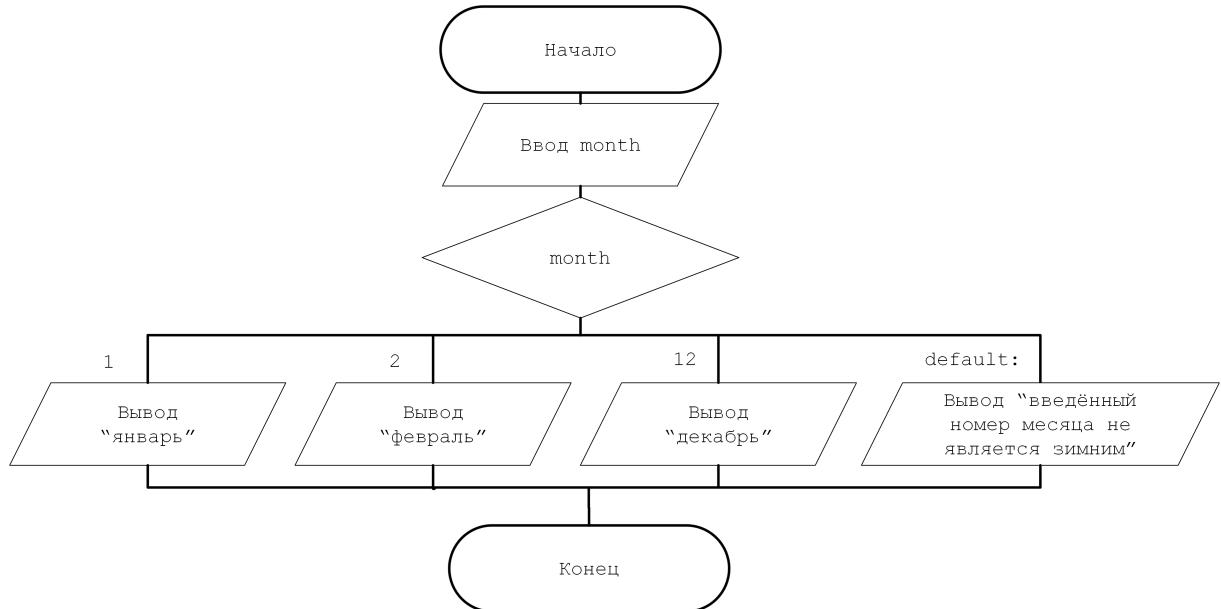


Рис. 5.8 – Блок-схема решения задачи о зимнем месяце

Код:

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6
7     int month;
8     scanf("%d", &month);
9
10    switch (month) {
11        case 1:
12            printf("Январь");
13            break;
14        case 2:
15            printf("Февраль");
16            break;
17        case 12:
18            printf("Декабрь");
19            break;
20        default:
21            printf("Введенный номер месяца не является зимним");
22    }
23
24    return 0;
25 }
  
```

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`:

```

1 // if (<логическое выражение>)
2 //   <оператор1>
3 // else
4 //   <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`:

```

1 // switch (<целочисленное выражение>) {
2 //   case <константное выражение1>:
3 //     <операторы>
4 //   case <константное выражение2>:
5 //     <операторы>
6 //   ...
7 //   case <константное выражениеN>:
8 //     <операторы>
9 //   default:
10 //     <операторы>
11 // }
```

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`:

```
1 // for (<выражение1>; <выражение2>; <выражение3>)
2 //      <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:

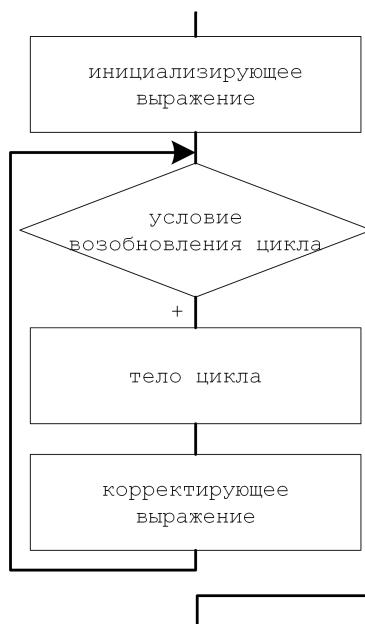


Рис. 6.1 – Цикл `for` с нефиксированным числом повторений

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):

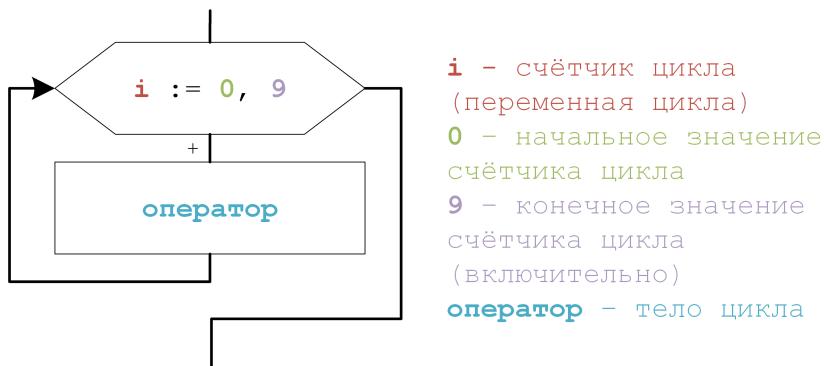


Рис. 6.2 – Обозначения цикла *for* (вариант: цикл с фиксированным числом повторений)

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

```

1 #include <stdio.h>
2
3 int main() {
4     for (int i = 0; i < 10; i++)
5         printf("%d\n", i);
6
7     return 0;
8 }
```

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:

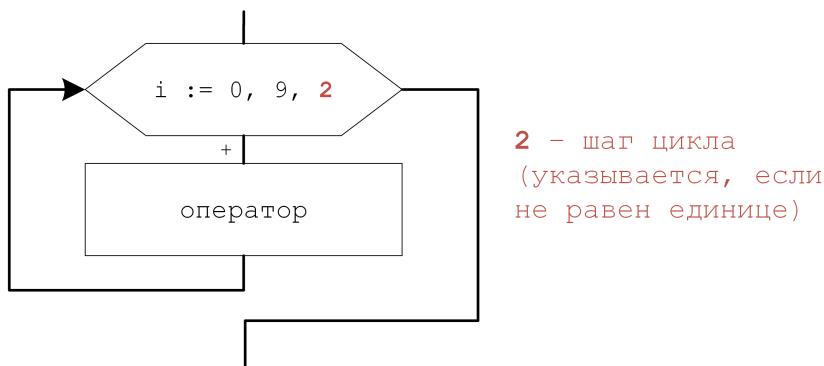


Рис. 6.3 – Обозначения цикла *for* (вариант: цикл с фиксированным числом повторений; шаг отличен от +1)

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 (,)¹. Пара выражений, разделенных запятой, вычисляется слева направо. Фрагмент

```
1 for (int i = 0, j = 10; i <= j; i++, j--)
2     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:

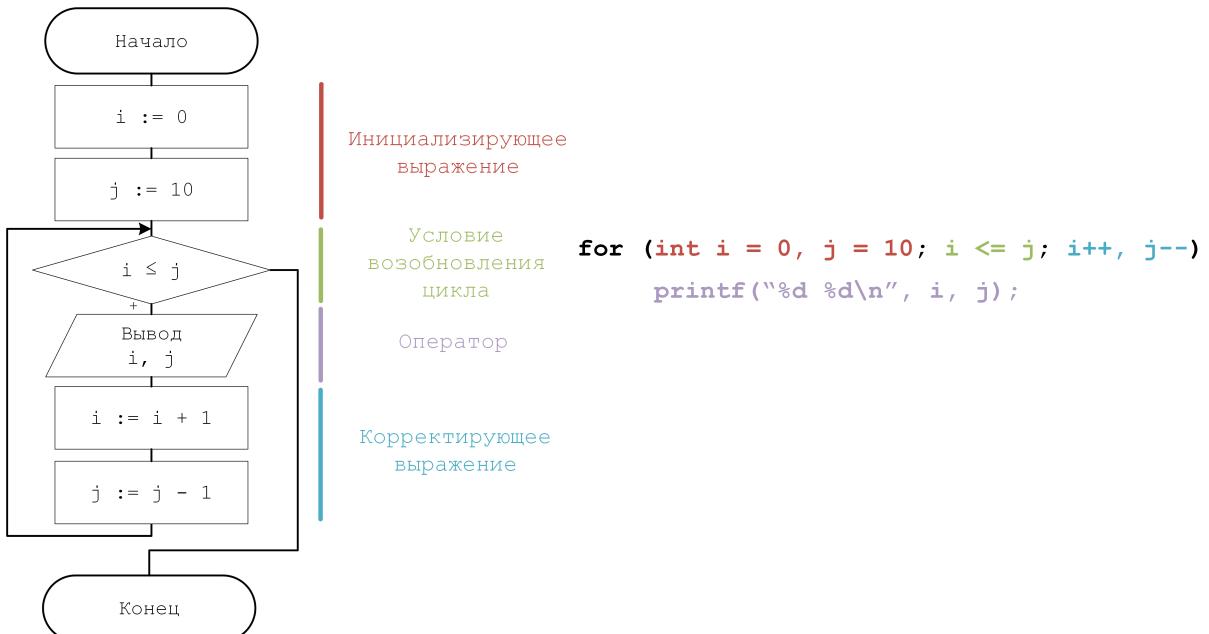


Рис. 6.4 – Цикл *for* с нефиксированным числом повторений

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

```
1 for (int i = 0; i < 10; i++) {
2     int x = 10;
3     // ...
4 }
5 // 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:

```
1 int x = 10;
2 int i = 20;
3 for (int i = 0; i < 2; i++) {
4     int x = i * 2;           // это уже другие x и i (не те, что были до цикла)
5     printf("%d %d\n", x, i); // 0 0
6                             // 2 1
7 }                         // x и i до цикла
8 printf("%d %d\n", x, i);   // 10 20
```

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

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;
        prin Declaration shadows a local variable
        previous declaration is here
        Rename local variable 'x' Alt+Shift+Enter More actions... Alt+Enter
        int x = i * 2
    }
    printf("format %d, %d, %d, %d, %d\n", x, i, x, i, x);
}
```

A screenshot of an IDE interface showing a code editor with C-like syntax. A tooltip is displayed over the line 'int x = i * 2;' inside the for loop. The tooltip contains the message 'Declaration shadows a local variable' and 'previous declaration is here'. It also includes options: 'Rename local variable 'x'' with keyboard shortcut 'Alt+Shift+Enter', 'More actions...', and 'Alt+Enter'. The code editor shows other parts of the program, including variable declarations and a printf statement.

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):

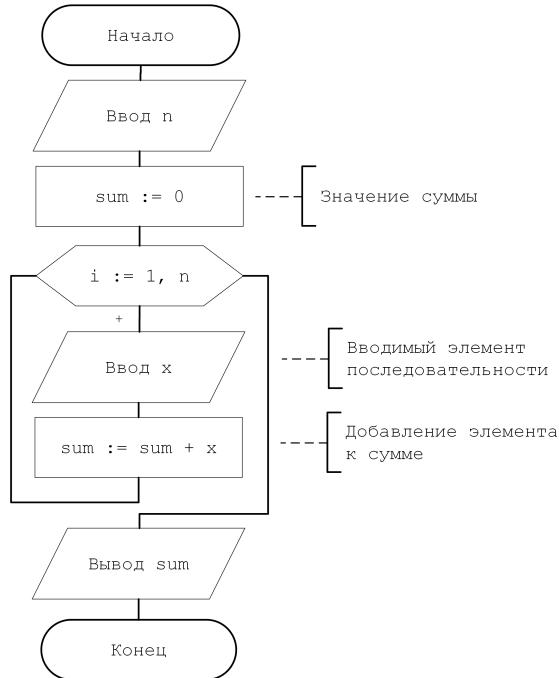


Рис. 6.5 – Блок-схема решения задачи вычисления суммы последовательности

Код функции `main`:

```

1 #include <stdio.h>
2
3 int main() {
4     int n;
5     scanf("%d", &n);
6
7     long long sum = 0;
8     for (int i = 1; i <= n; i++) {
9         int x;
10        scanf("%d", &x);
11
12        sum += x;
13    }
14
15    printf("%lld", sum);
16
17    return 0;
18 }
```

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

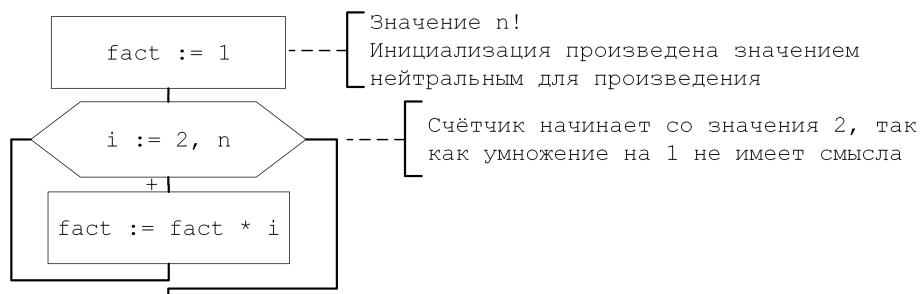
Поиск значений выражений

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.

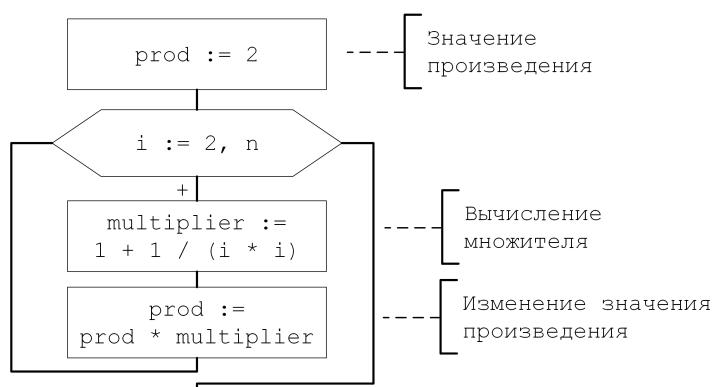
Вычисление $n!$:



```

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

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



```

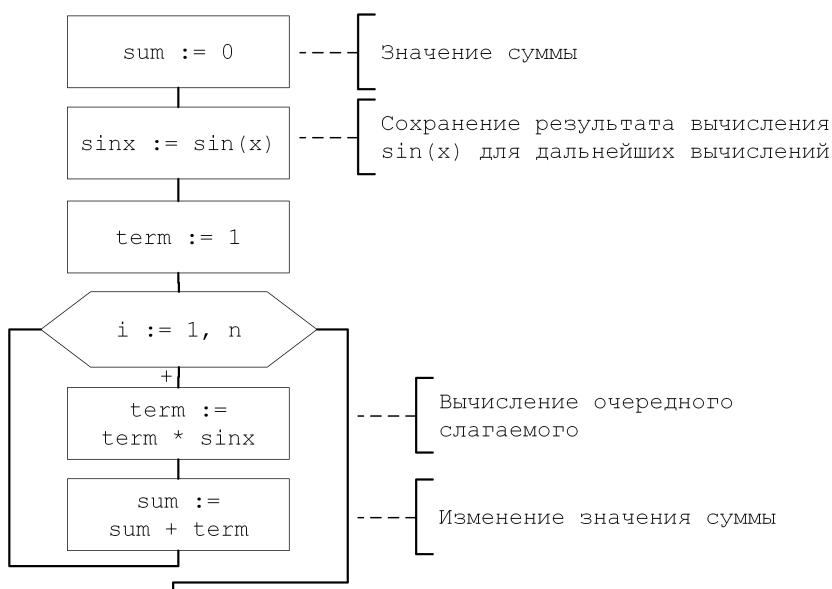
1 double prod = 2; // значение произведения будет вещественным
2 for (int i = 2; i <= n; i++) {
3     double multiplier = 1 + 1.0 / i / i;
4     // или double multiplier = 1 + 1.0 / ((long long) i * i)
5     // приведение к long long - борьба с переполнением
6     prod *= multiplier;
7 }

```

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 $\sin^i(x)$:



```

1 double sum = 0;
2 double sinx = sin(x);
3 double term = 1;
4 for (int i = 1; i <= n; i++) {
5     term *= sinx;
6     sum += term;
7 }

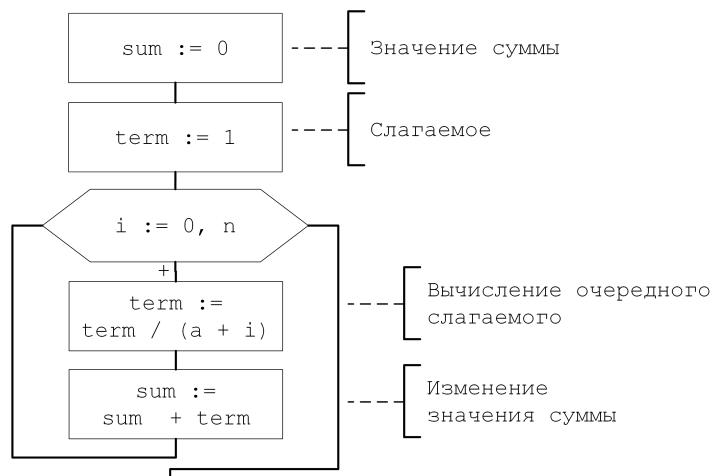
```

Вычисление значения выражения

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

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

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



```

1 double term = 1.0 / a; // значение произведения будет вещественным,
2 // деление должно быть вещественным
3 double sum = term;
4 for (int i = 1; i <= n; i++) {
5     term = term / (a + i);
6     sum += term;
7 }
  
```

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

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

```

1 double term = 1;
2 double r = 0;
3 for (int i = 1; i <= n; i++) {
4     term = term / i * x;
5     r += term;
6 }
  
```

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 86), 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`²:

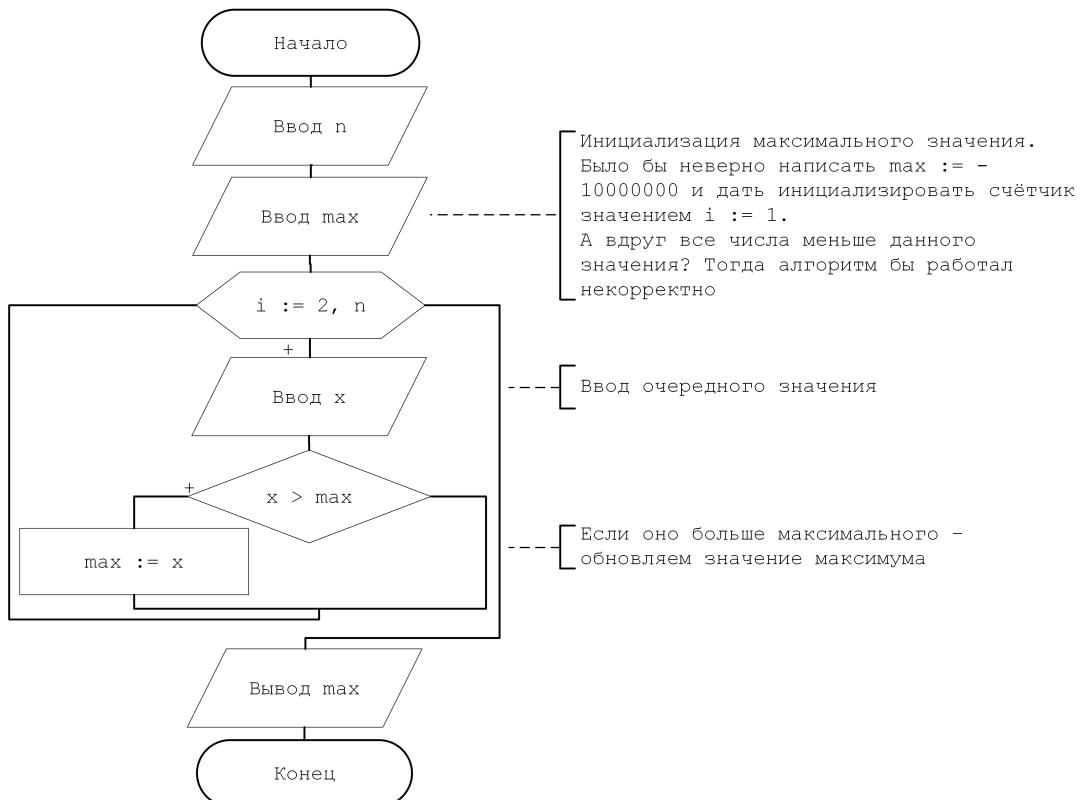


Рис. 6.6 – Блок-схема для решения задачи поиска максимума

```

1 #include <stdio.h>
2
3 int main() {
4     int n, max;
5     scanf("%d %d", &n, &max);
6
7     for (int i = 2; i <= n; i++) {
8         int x;
9         scanf("%d", &x);
10
11         if (x > max)
12             max = x;
13     }
14 }
```

²Существуют 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.

```

15     printf("%d", max);
16
17     return 0;
18 }
```

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.

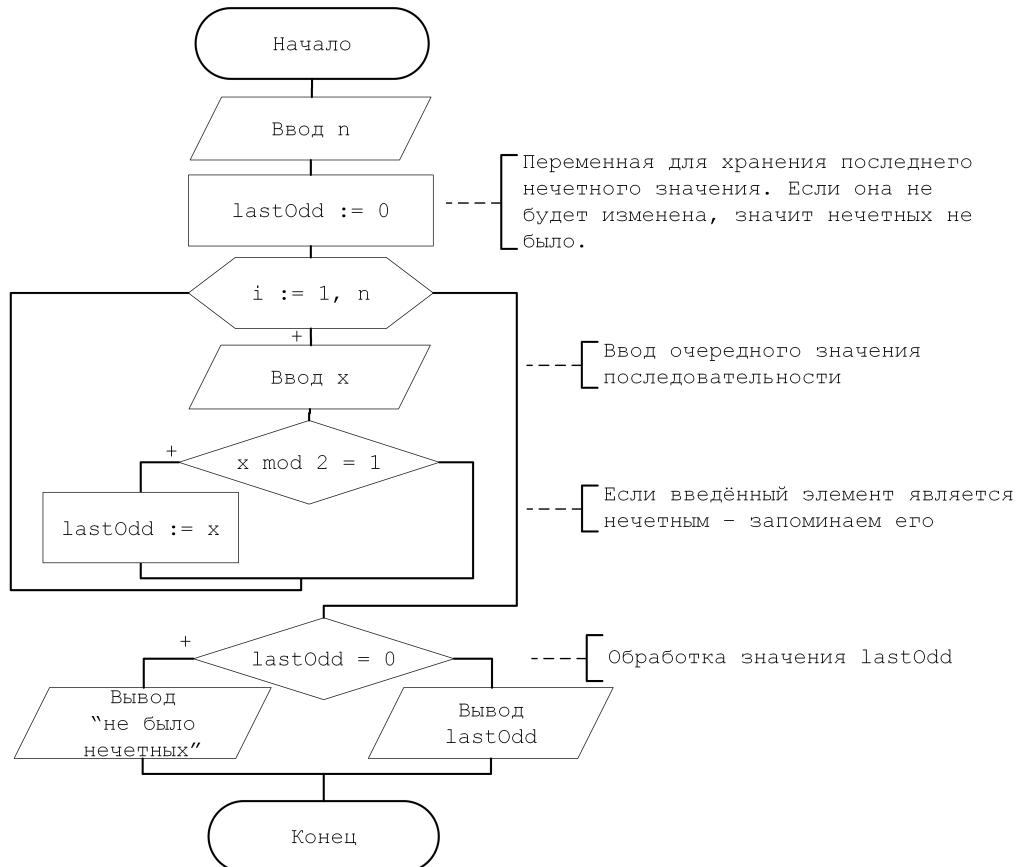


Рис. 6.7 – Блок-схема для решения задачи последнего нечетного элемента

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 быть³. 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.

Можно дополнить решение приёмом: чтобы значение 0 не воспринималось магической константой, опишем макрос

```
1 #define NO_ODD_VALUES 0
```

³Опытный разработчик всегда посмотрит направо и налево, даже если переходит улицу с односторонним движением.

который говорит о значении для того случая, когда нечетных элементов не будет.

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 #define NO_ODD_VALUES 0
5
6 int main() {
7     SetConsoleOutputCP(CP_UTF8);
8
9     int n;
10    scanf("%d", &n);
11
12    int lastOdd = NO_ODD_VALUES;
13    for (int i = 1; i <= n; i++) {
14        int x;
15        scanf("%d", &x);
16
17        if (x % 2 == 1)
18            lastOdd = x;
19    }
20
21    if (lastOdd == NO_ODD_VALUES)
22        printf("В последовательности не было четных элементов");
23    else
24        printf("%d", lastOdd);
25
26    return 0;
27 }
```

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.

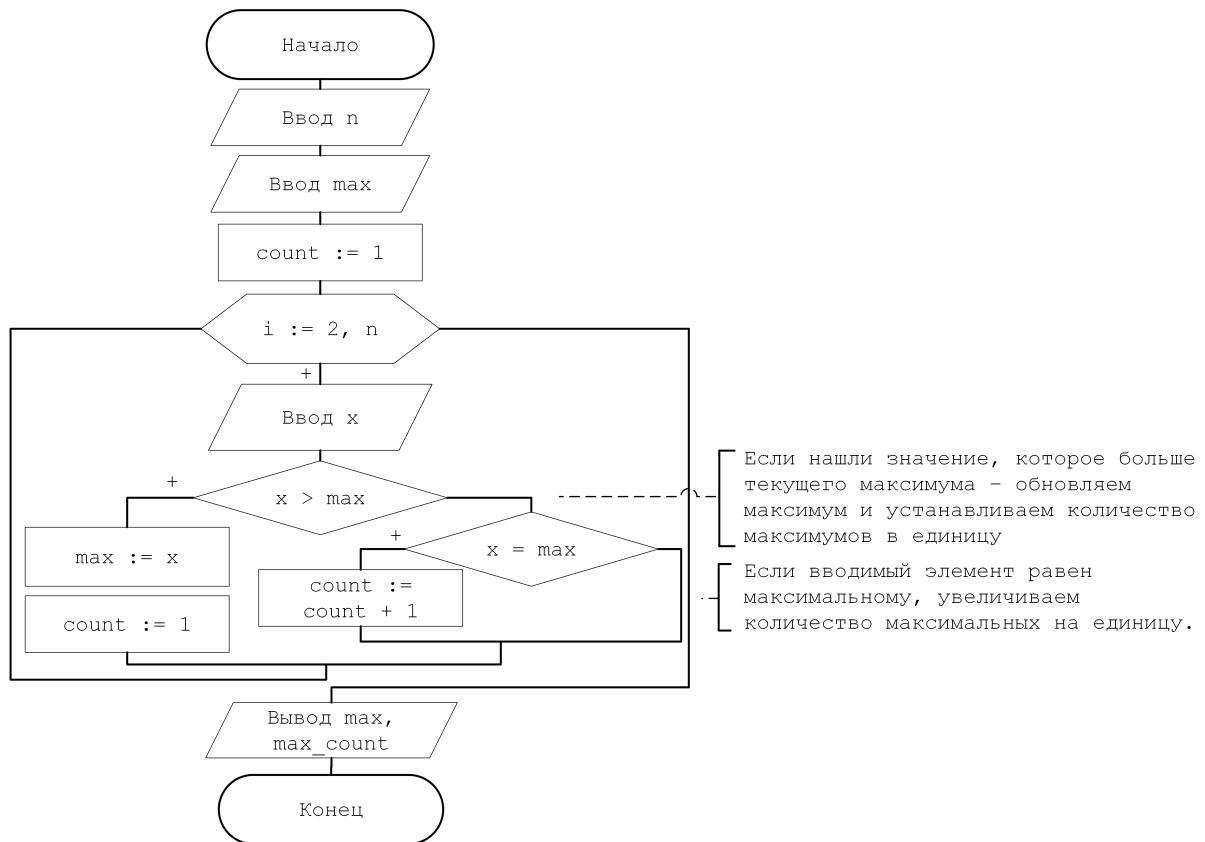


Рис. 6.8 – Блок-схема решения задачи поиска максимального значения последовательности и их количества

```

1 #include <stdio.h>
2
3 int main() {
4     int n, max;
5     scanf("%d %d", &n, &max);
6
7     int count = 1;
8     for (int i = 2; i <= n; i++) {
9         int x;
10        scanf("%d", &x);
11
12        if (x > max) {
13            count = 1;
14            max = x;
15        } else if (x == max)
16            count = count + 1;
17    }
18
19    printf("%d %d", max, count);
20
21    return 0;
22 }

```

6.2 Цикл *while*

Цикл `while` является циклом с предусловием:

```
1 // while (<логическое выражение>)
2 //     <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:

```
1 int i = 5;
2 while (i > 10)
3     i--;
```

Цикл `while` может стать бесконечным, если не влиять на переменные, которые указаны в логическом выражении:

```
1 int i = 15;
2 int a = 0;
3 while (i > 10)
4     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:

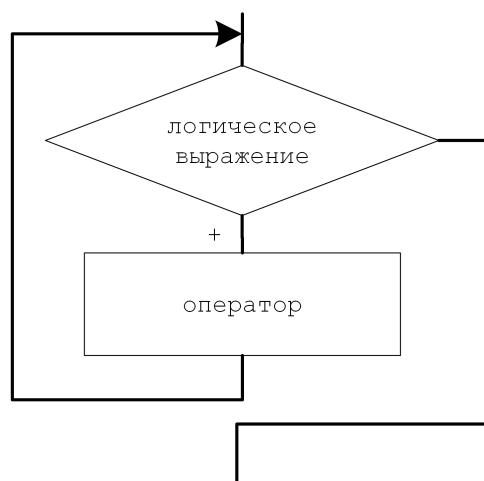


Рис. 6.9 – Обозначение цикла *while*

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:

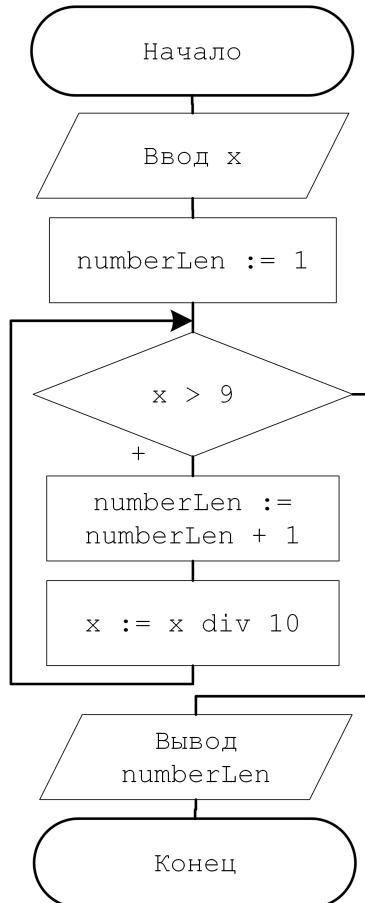


Рис. 6.10 – Блок-схема для решения задачи подсчёта количества цифр в числе

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int numberLen = 1;
8     while (x > 9) {
9         numberLen++;
10        x /= 10;
11    }
12
13    printf("%d", numberLen);
14
15    return 0;
16}

```

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:

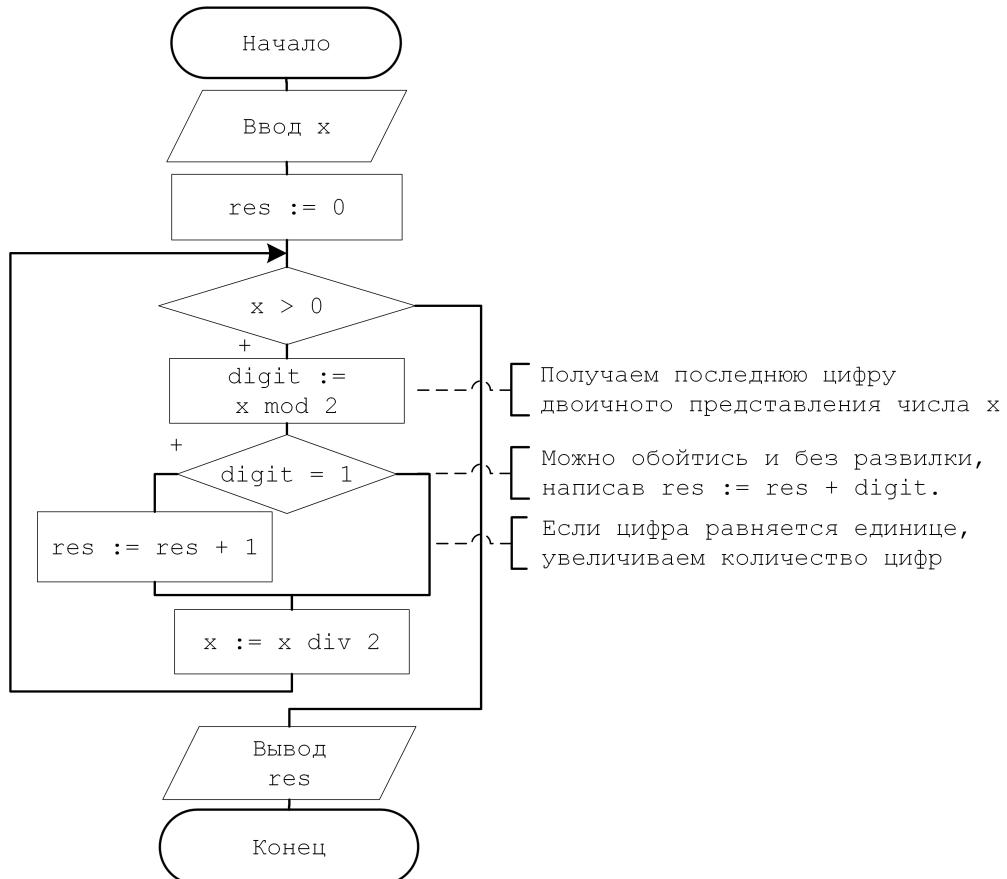


Рис. 6.11 – Блок-схема для решения задачи подсчёта количества единиц в двоичной записи числа x

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int res = 0;
8     while (x > 0) {
9         int isOne = x % 2 == 1;
10        res += isOne;
11        x /= 2;
12    }
13
14    printf("%d", res);
15
16    return 0;
17 }

```

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.

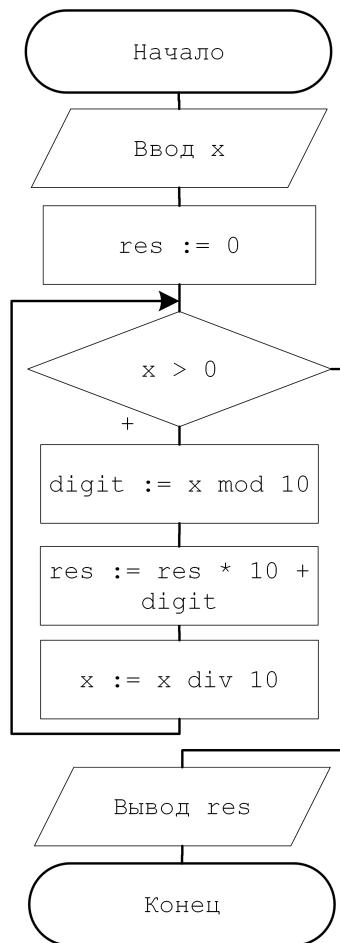


Рис. 6.12 – Блок схема решения задачи изменения порядка цифр

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int res = 0;
8     while (x > 0) {
9         int digit = x % 10;
10        res = res * 10 + digit;
11        x /= 10;
12    }
13
14    printf("%d", res);
15
16    return 0;
17 }
```

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.

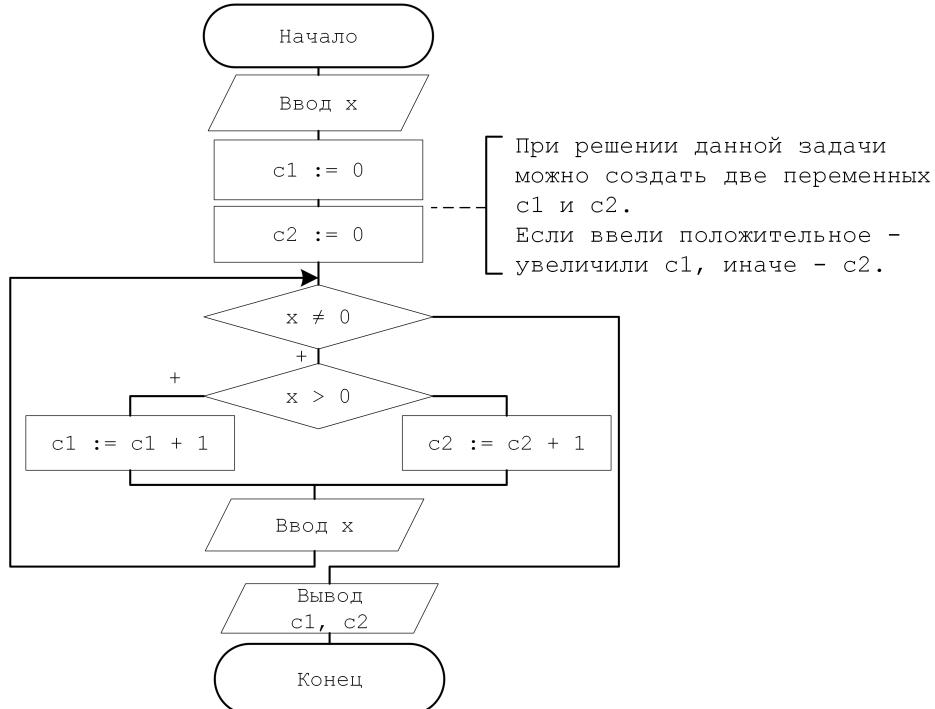


Рис. 6.13 – Блок-схема к задаче о подсчёте положительных и отрицательных чисел последовательности

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int c1 = 0;
8     int c2 = 0;
9     while (x != 0) {
10         if (x > 0)
11             c1++;
12         else
13             c2++;
14
15         scanf("%d", &x);
16     }
17
18     printf("c1 = %d, c2 = %d", c1, c2);
19
20     return 0;
21 }

```

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 .

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     long long a = 1;
8     int k = 1;
9     while (a <= x) {
10         a = a * k * k;
11         k++;
12     }
13
14     printf("%lld", a);
15
16     return 0;
17 }
```

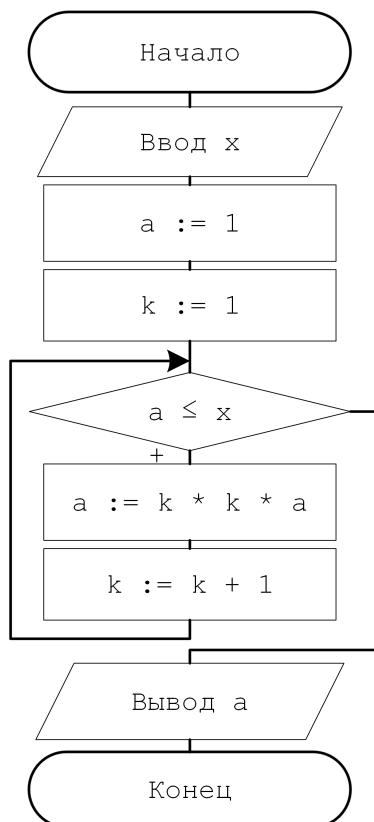


Рис. 6.14 – Блок схема для поиска первого элемента рекуррентно заданной последовательности больше x

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 .

```

1 #include <stdio.h>
2
3 int main() {
4     int n, m;
5     scanf("%d %d", &n, &m);
6
7     int sum = 0;
8     while (n != 0 && m != 0) {
9         sum += n % 10;
10        m--;
11        n /= 10;
12    }
13
14    printf("%d", sum);
15
16    return 0;
17 }
```

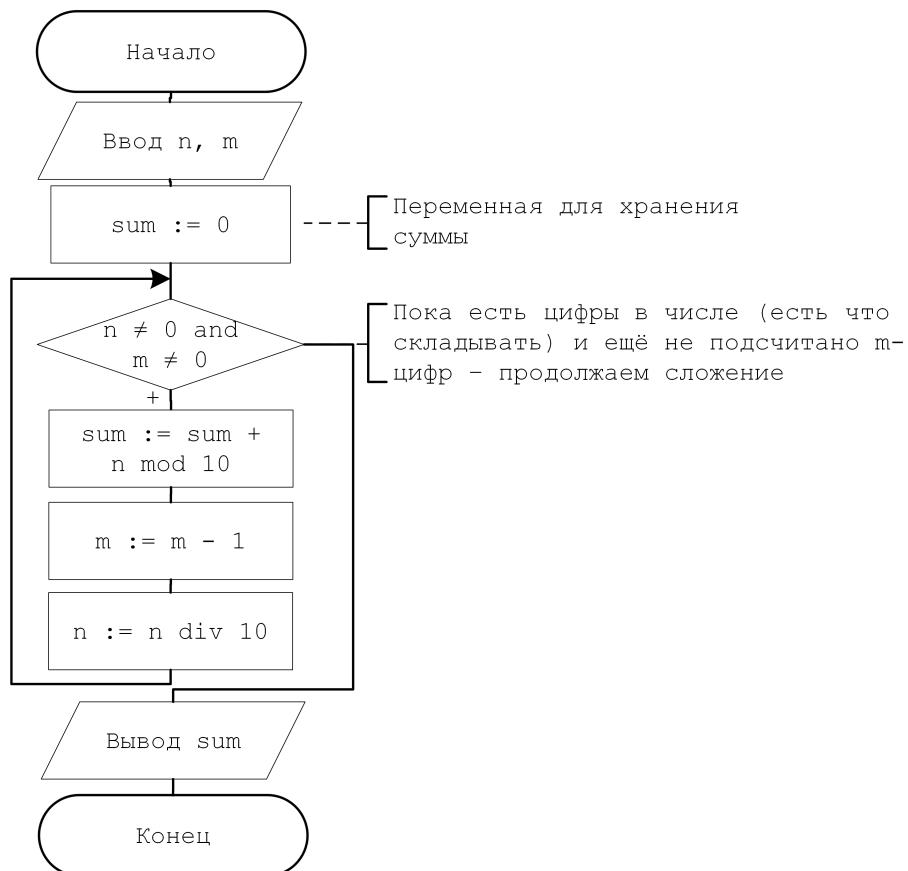


Рис. 6.15 – Блок схема решения задачи о поиске суммы последних цифр

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:

$$\text{digit} = x \text{ div } k \text{ mod } 10$$

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int copyX = x;
8     int nDigits = 1;
9     int k = 1;
10    while (copyX > 9) {
11        nDigits++;
12        if (nDigits % 2)
13            k *= 10;
14        copyX /= 10;
15    }
16
17    int digit = nDigits % 2 ? x / k % 10 : -1;
18    printf("%d", digit);
19
20    return 0;
21 }
```

6.2.8 Проверка числа на простоту

Проверка числа на простоту

С клавиатуры вводится число n . Необходимо проверить, является ли n простым^a.

^aЧисло является простым, если не имеет делителей кроме себя и единицы. Единица к простым числам не относится.

Решение можно осуществить так:

```

1 #include <stdio.h>
2
3 int main() {
4     int n;
5     scanf("%d", &n);
6
7     int d = 2;
8     while (d < n && n % d != 0)
9         d++;
10
11    if (d == n)
12        printf("Yes");
13    else
14        printf("No");
15
16    return 0;
17 }
```

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:

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int x;
6     scanf("%d", &x);
7
8     int d = 2;
9     int max_d = sqrt(x);
10    while (d <= max_d && x % d != 0)
11        d++;
12
13    if (d == max_d + 1 && x != 1)
14        printf("Yes");
15    else
```

```

16     printf("No");
17
18     return 0;
19 }
```

Можно выполнить ещё оптимизацию и сделать шаг, равный двум:

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int x;
6     scanf("%d", &x);
7
8     int d = 3;
9     int maxPotentialDivider = sqrt(x);
10    int isPrime = !(x == 1 || x % 2 == 0 && x != 2);
11    while (d <= maxPotentialDivider && isPrime) {
12        isPrime = x % d;
13        d += 2;
14    }
15
16    if (isPrime)
17        printf("Yes");
18    else
19        printf("No");
20
21    return 0;
22 }
```

Опишу немного момента оценки сложности алгоритмов. Сам процесс вычислений, который организован в компьютере, достаточно сложный. Мы абстрагируемся⁴ 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:

```

1 #include <stdio.h>
2
3 int main() {
4     int n, max;
5     scanf("%d %d", &n, &max); // 2 операции
6
7     int i = 2; // 1 операция
8     while (i <= n) { // 1 операция выполнится  $n$  раз, для значений от 2 до  $n+1$ 
9         int x;
10        scanf("%d", &x); // 1 операция  $n-1$  раз
11
12        if (x > max) // 1 операция  $n-1$  раз
13            max = x; // 1 операция, которая в худшем случае выполнится  $n-1$  раз
14
15        i++; // 1 операция,  $n-1$  раз
16    }
17
18    printf("%d", max); // 1 операция
19
20    return 0;
21 }
```

⁴Абстрагироваться – мысленно отвлекаться от тех или иных сторон, свойств или связей предметов и явлений с целью выделения существенных и закономерных их признаков

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 $\Omega(g(x))$ ⁵. Если порядки функций временной сложности для наихудшего и наилучшего случая совпадают $O(g(x)) = \Omega(g(x))$, говорят, что функция имеет асимптотическую сложность $\Theta(g(x))$ ⁶

Несколько жизненных примеров:

- «почистить ковёр пылесосом» требует времени, линейно зависящее от его площади $\Theta(S)$, то есть на ковёр, площадь которого больше в два раза, уйдет в два 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 имён 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

⁵ Ω - 'омега' (произношение).

⁶ Θ - 'тета' (произношение).

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.

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6
7     int max1;
8     scanf("%d", &max1);
9
10    if (max1 != 0) {
11        int max2;
12        scanf("%d", &max2);
13
14        if (max2 != 0) {
15            if (max1 < max2) {
16                int t = max1;
17                max1 = max2;
18                max2 = t;
19            }
20
21            int x;
22            scanf("%d", &x);
23
24            while (x != 0) {
25                if (x >= max1) {
26                    max2 = max1;
27                    max1 = x;
28                } else if (x > max2)

```

```

29         max2 = x;
30
31         scanf ("%d", &x);
32     }
33
34     printf ("max2 = %d", max2);
35 } else
36     printf ("Длина последовательности меньше двух");
37 } else
38     printf ("Пустая последовательность");
39
40 return 0;
41 }
```

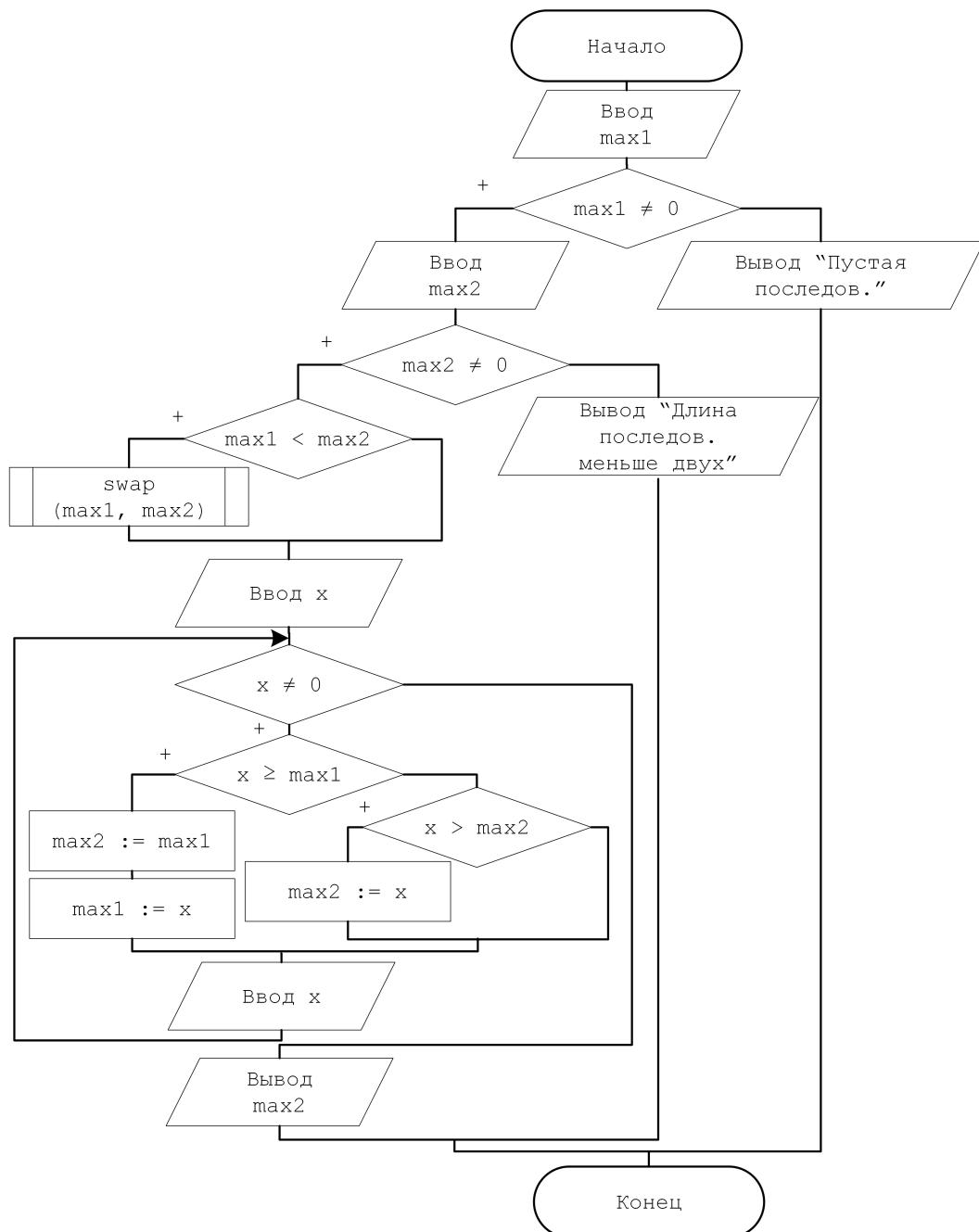


Рис. 6.16 – Блок-схема задачи о поиске второго максимума

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

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int maxLen = 0;
8     int currentLen = 0;
9     while (x != 0) {
10         if (x > 0) {
11             // если число положительное, увеличиваем текущую длину
12             currentLen++;
13             // проверяем, длиннее ли текущая цепочка максимальной
14             // если да - обновляем максимум
15             if (currentLen > maxLen)
16                 maxLen = currentLen;
17         } else
18             // иначе скидываем счётчик положительных значений.
19             currentLen = 0;
20
21         scanf("%d", &x);
22     }
23
24     printf("%d", maxLen);
25
26     return 0;
27 }
```

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:

$$a_{i-1} * a_i < 0$$

```

1 #include <stdio.h>
2
3 int main() {
4     int last;
5     scanf("%d", &last);
6
7     int maxLen = 0;
8     // инициализация выполнена нулём, элементов может не быть
9     int currentLen = 0;
10    if (last != 0) {
11        int cur;
12        scanf("%d", &cur);
13
14        // один элемент сам по себе образует знакочередующуюся
15        // последовательность длины 1; именно по этой причине, когда
16        // знакочередование заканчивается, сброс идёт в единицу
17        currentLen += 1;
18        while (cur != 0) {
19            if (last * cur < 0)
20                currentLen++;
21            else {
22                if (currentLen > maxLen)
23                    maxLen = currentLen;
24                currentLen = 1;
25            }
26            last = cur;
27
28            scanf("%d", &cur);
29        }
30    }
31
32    if (currentLen > maxLen)
33        maxLen = currentLen;
34
35    printf("%d", maxLen);
36
37    return 0;
38 }
```

6.2.12 Подпоследовательность с максимальной суммой (Алгоритм Джейя Кадана)

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.

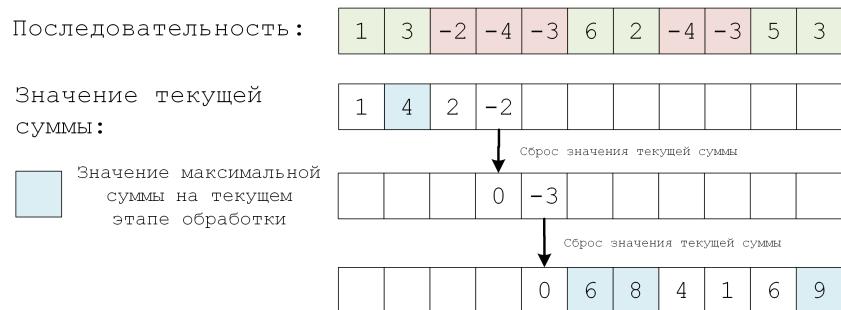


Рис. 6.17 – Изменения значений *curSum* и *maxSum*

Создаётся переменная, которая хранит максимальное значение суммы, а также текущей суммы. Если текущая сумма больше максимума, происходит обновление результата.

Код:

```

1 #include <stdio.h>
2
3 int main() {
4     long long curSum; // значение текущей суммы
5     scanf("%lld", &curSum);
6
7     long long maxSum = curSum; // максимальная сумма подпоследовательности
8     if (maxSum != 0) {
9         int x;
10        scanf("%d", &x);
11
12        while (x != 0) {
13            curSum += x; // добавляем элемент к последовательности
14            if (curSum > maxSum)
15                maxSum = curSum; // обновление максимума
16            if (curSum < 0)
17                curSum = 0; // если значение суммы стало отрицательным,
18                           // сбрасываем сумму в ноль, начинаем
19                           // новую последовательность
20
21            scanf("%d", &x);
22        }
23
24        printf("%lld", maxSum);
25
26        return 0;
27    }
}

```

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. Предложите 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 выводится число перед первым четным (а не перед последним).

Код решения задачи:

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6
7     int x;
8     scanf("%d", &x);
9
10    int r = -1;
11    int last = 0;
12    while (x != 0) {
13        if (r == -1 && x % 2 == 0)
14            r = last;
15        last = x;
16
17        scanf("%d", &x);
18    }
19
20    if (last == 0)
21        printf("Последовательность пуста");
22    else if (r == 0)
23        printf("Четное число - первое в последовательности");
24    else if (r == -1)
25        printf("Четных чисел не было");
26    else
27        printf("%d", r);
28
29    return 0;
30 }
```

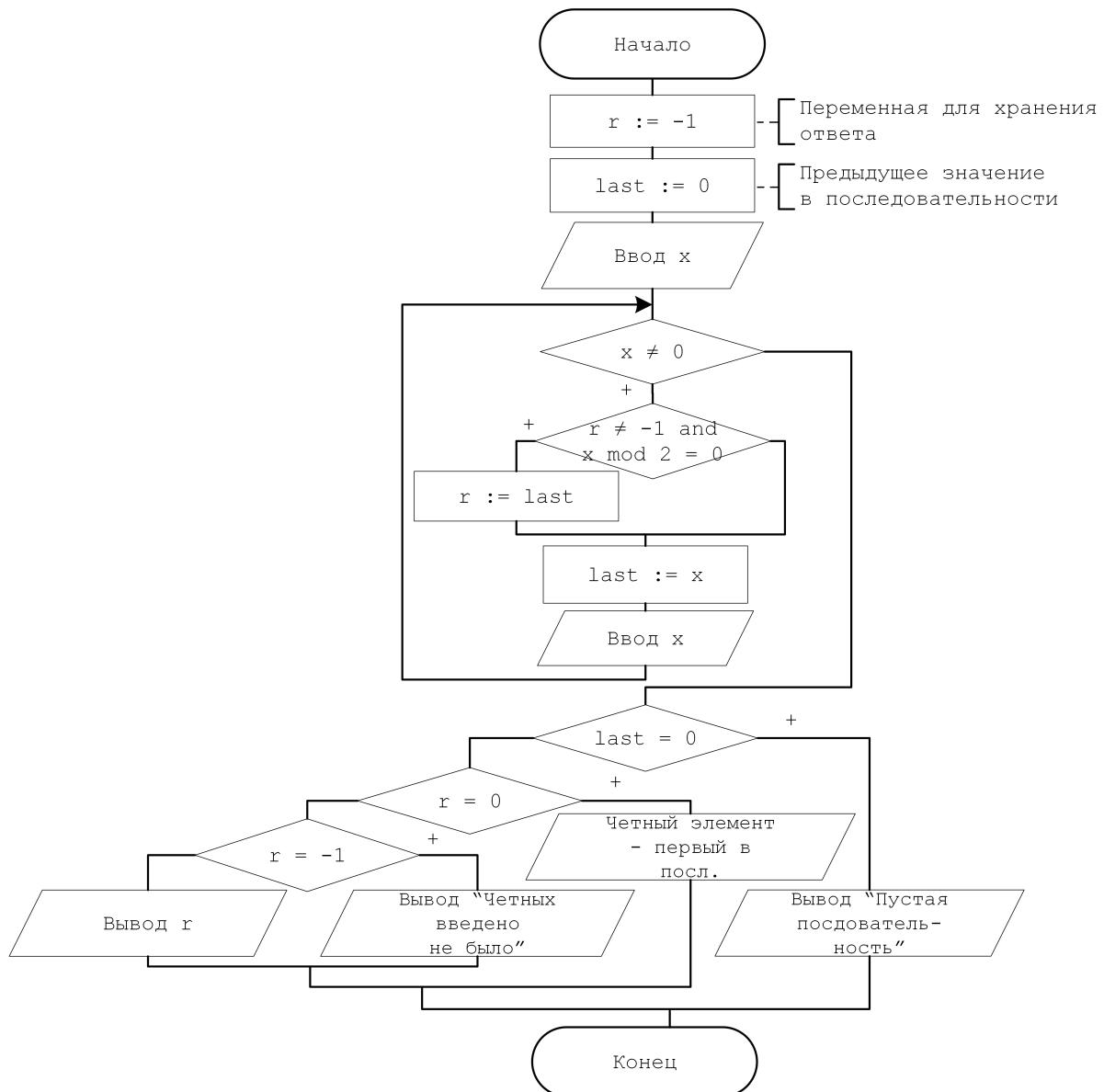


Рис. 6.18 – Блок-схема к задаче о поиске числа перед первым чётным

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.

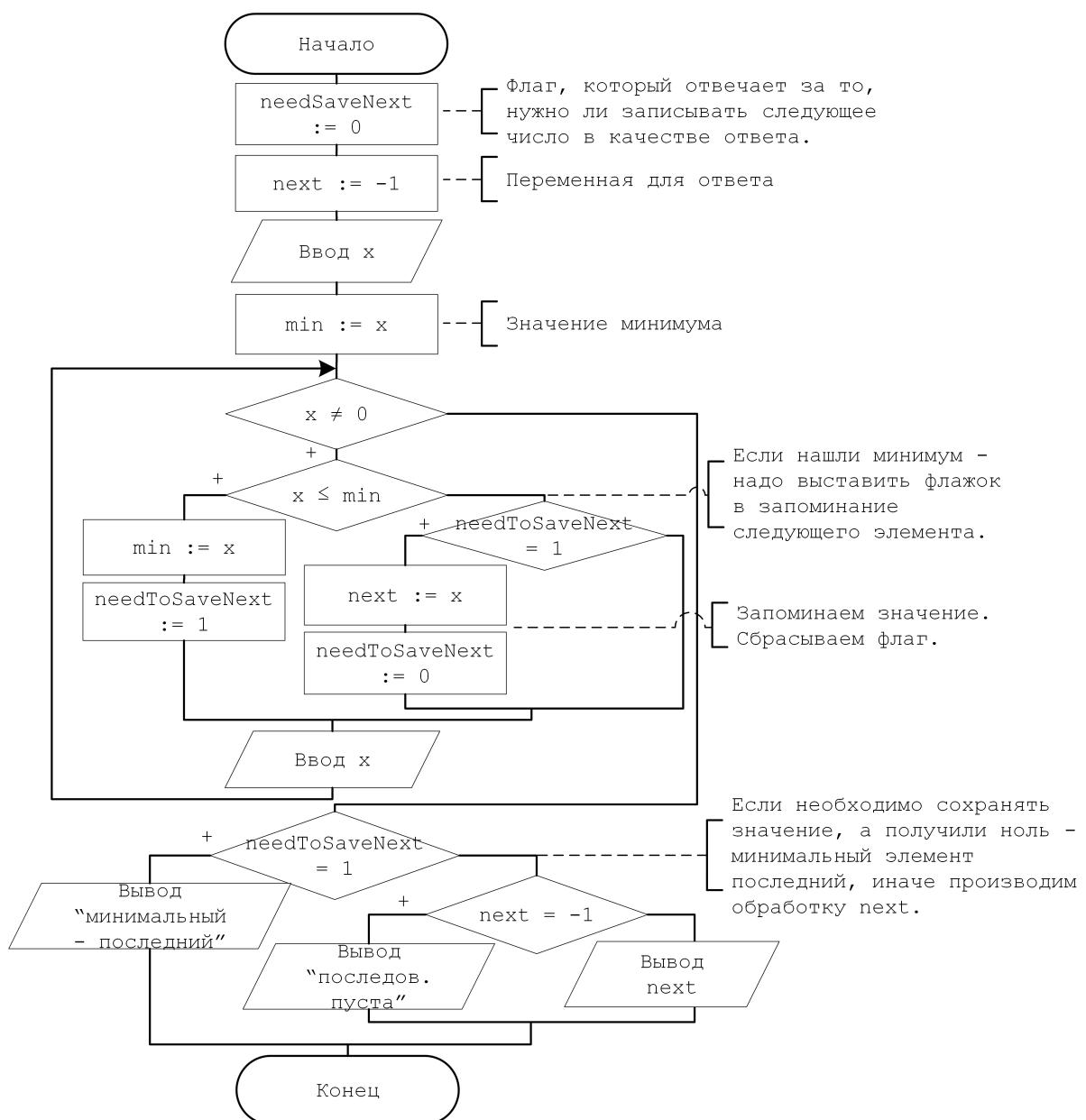


Рис. 6.19 – Блок-схема к задаче о поиске элемента после последнего минимального

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6
7     int x;
8     scanf("%d", &x);
9     int needToSaveNext = 0;
10
11    int min = x;
12    int next = -1;
13    while (x != 0) {
14        if (x <= min) {
15            min = x;
16            needToSaveNext = 1;
17        } else if (needToSaveNext) {
18            next = x;
19            needToSaveNext = 0;
20        }
21        scanf("%d", &x);
22    }
23
24    if (needToSaveNext == 1)
25        printf("Минимальный элемент - последний");
26    else if (next == -1)
27        printf("Последовательность пуста");
28    else
29        printf("%d", next);
30
31    return 0;
32 }
```

6.3 Цикл *do-while*

Цикл *do-while* описывается конструкцией:

```

1 // do
2 //     <оператор>
3 // 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.

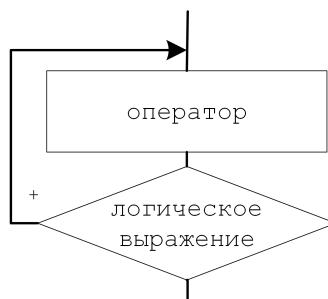


Рис. 6.20 – Обозначение цикла *do – while*

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:

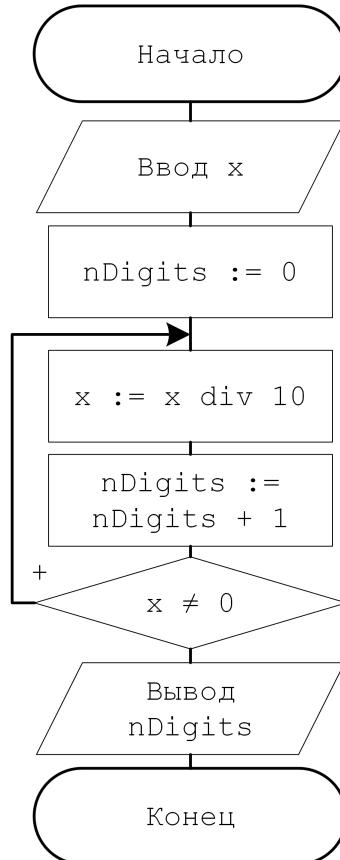


Рис. 6.21 – Блок-схема решения задачи о количестве цифр

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     scanf("%d", &x);
6
7     int nDigits = 0;
8     do {
9         x /= 10;
10        nDigits += 1;
11    } while (x != 0);
12
13    printf("%d", nDigits);
14
15    return 0;
16 }
  
```

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`

```
1 // for ( ; <условие возобновления цикла> ; )
```

который идентичен циклу `while`:

```
1 // while (<условие возобновления цикла>)
```

Или пример цикла `while`:

```
1 // <инициализирующее выражение>
2 // while (<условие возобновления цикла>) {
3 //     <тело цикла>
4 //     <корректирующее выражение>
5 // }
```

который работает почти так же, как цикл `for`⁷. 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:

```
1 int a = 10;
2 while (a > 0) {
3     ...
4
5     int x;
6     scanf("%d", &x);
```

⁷Существует 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 отличии от цикла `for`.

```

8     if (x % 2 == 0)
9         break;
10    ...
11    a--;
12 }
```

Если на какой-то итерации⁸ будет введено четное значение, выполнится выход из цикла. Обозначение:

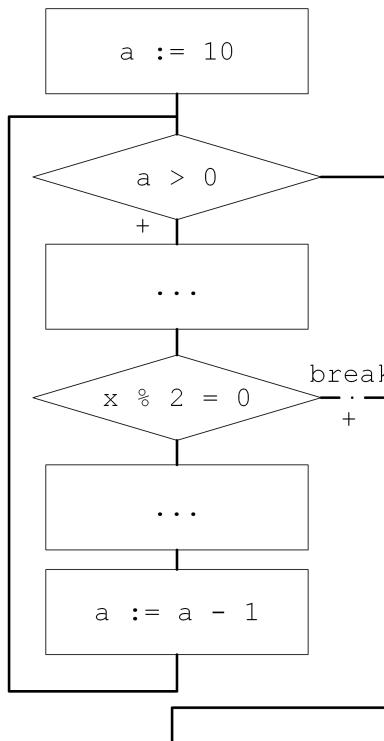


Рис. 6.22 – Обозначение оператора *break* на блок-схеме

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.

```

1 #include <stdio.h>
2
3 int main() {
4     for (int i = 0; i < 10; i++) {
5         int x;
6         scanf("%d", &x);
7
8         if (x % 2 == 0)
9             continue;
10        // ... обработка x
11    }
12
13    return 0;
14 }
```

⁸Итерация – в узком смысле – один шаг итерационного, циклического процесса

Однако конструкция выше может быть записана и без использования `continue`:

```

1 #include <stdio.h>
2
3 int main() {
4     for (int i = 0; i < 10; i++) {
5         int x;
6         scanf("%d", &x);
7
8         if (x % 2 != 0) {
9             // ... обработка x
10        }
11    }
12
13    return 0;
14 }
```

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:

```

1 for (i = 0 ; i < n; i++)
2     for (j = 0; j < m; j++)
3         if (a[i] == b[j])
4             goto found;
5 /* нет одинаковых элементов */
6
7 found:
8 /* 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:

```
1 found = 0;
2 for (i = 0; i < n && !found; i++)
3     for (j = 0; j < m && !found; j++)
4         if (a[i] == b[j])
5             found = 1;
6
7 if (found)
8     /* обнаружено совпадение */
9     ...
10 else
11     /* нет одинаковых элементов */
12     ...
```

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.

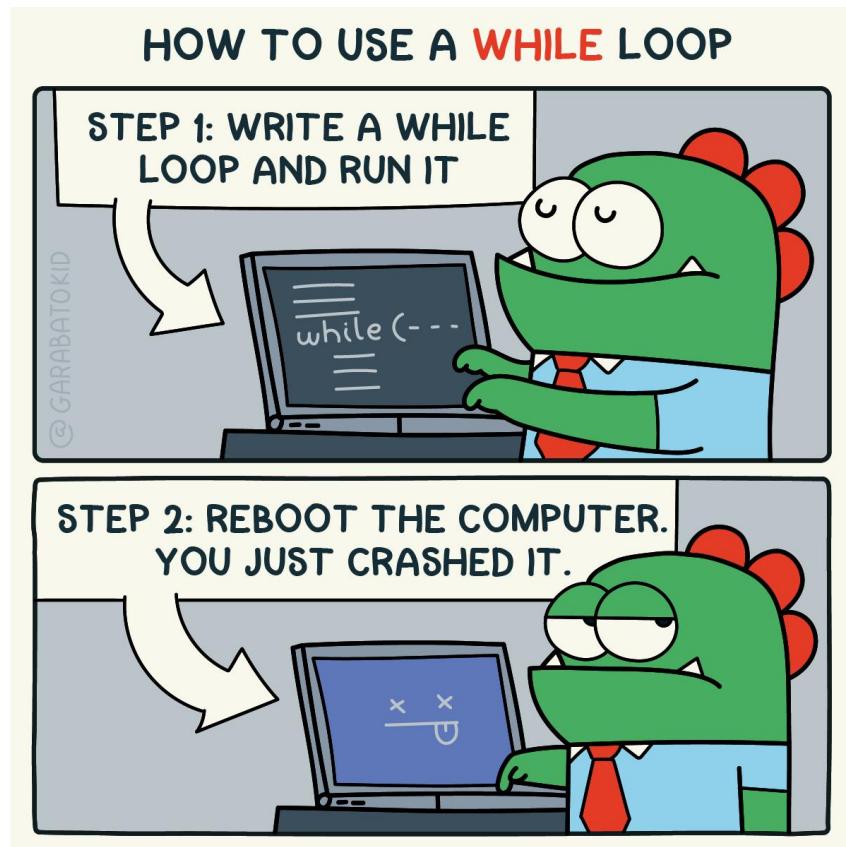
Термины и определения

- **Циклический алгоритм** – алгоритм, в которых некоторая часть операций выполняется многократно.

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*)¹. 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.

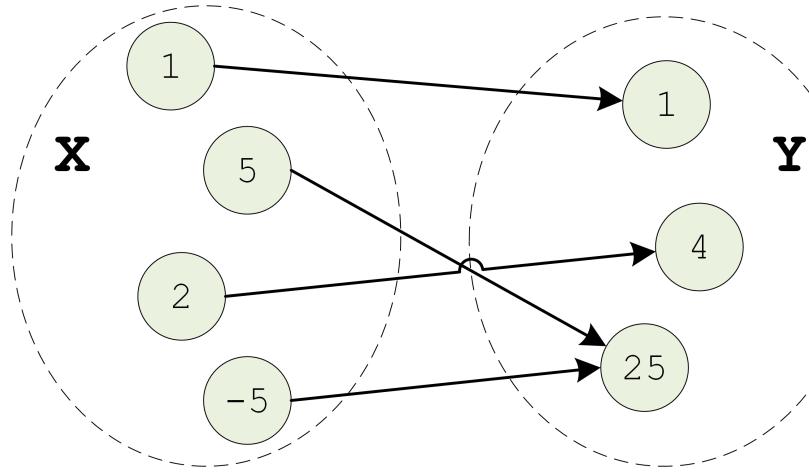
¹Нарушения принципа *DRY* называют *WET* — «*Write Everything Twice*» (рус. Пиши всё дважды) или «*We enjoy typing*» (рус. Нам нравится печатать). Это игра английских слов «*dry*» (рус. сухой) и «*wet*» (рус. влажный).

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).

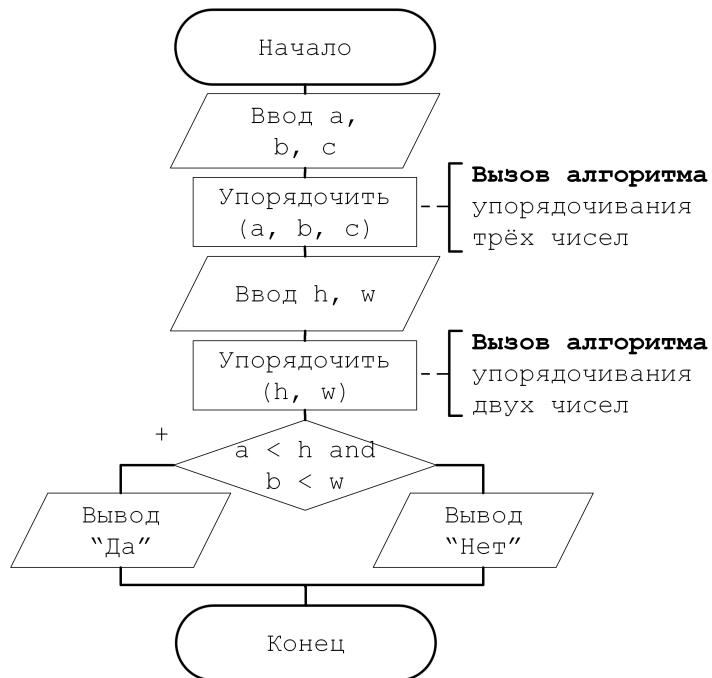


Рис. 7.1 – Блок-схема алгоритма в укрупненных блоках

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:

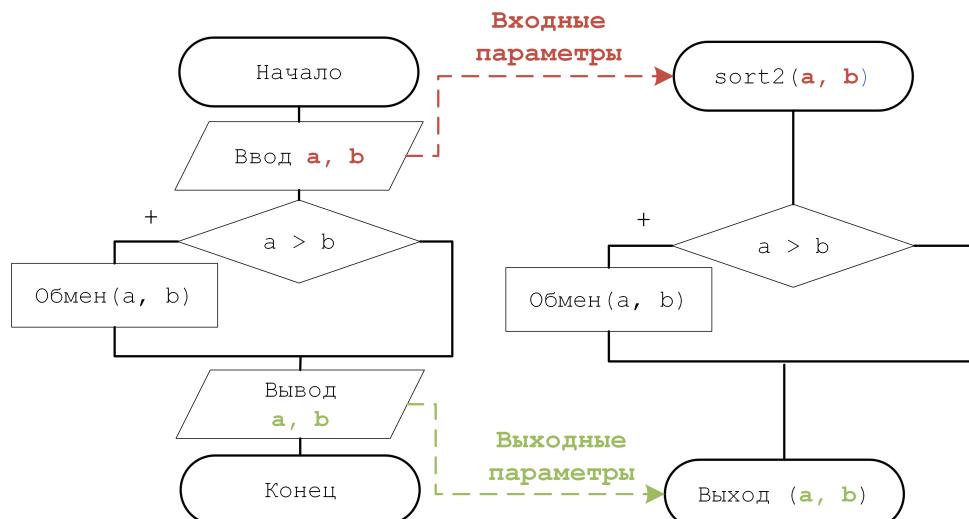


Рис. 7.2 – Переход от самостоятельной задачи к функции на блок-схеме

Но в блок-схеме для решения задачи упорядочивания содержится задача обмена значений двух переменных. Опишем её таким же образом:

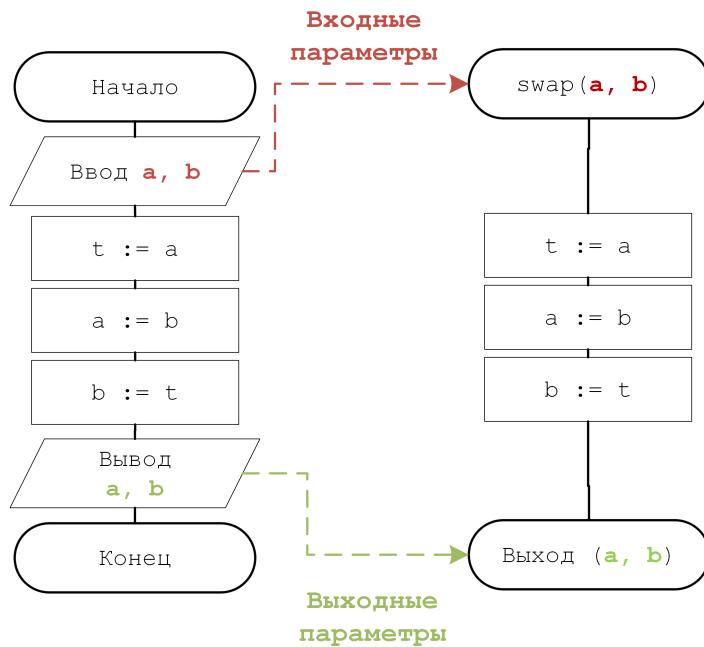


Рис. 7.3 – Переход от самостоятельной задачи к функции на блок-схеме

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:

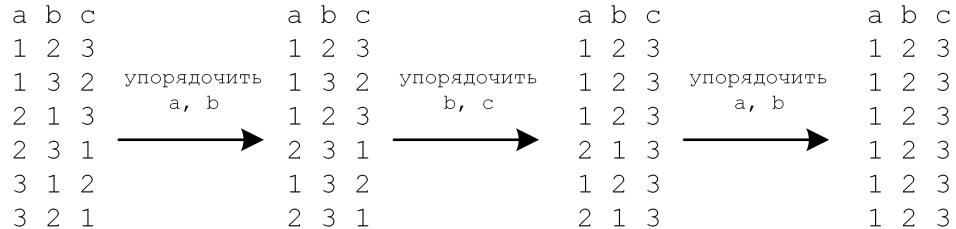


Рис. 7.4 – Переход от самостоятельной задачи к функции на блок-схеме

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.

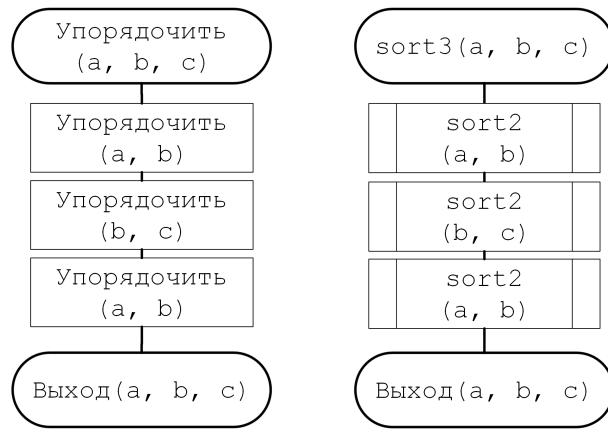


Рис. 7.5 – Блок-схемы в укрупненных блоках и при помощи блоков предопределенный процесс

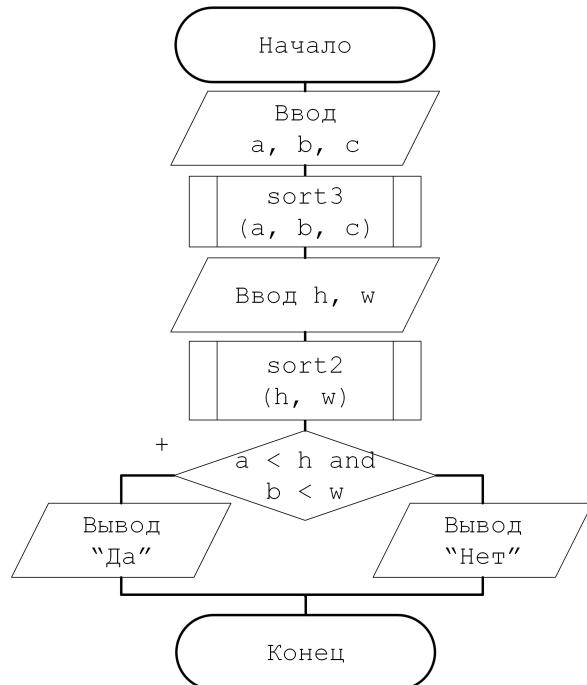


Рис. 7.6 – Блок-схема решения задачи при помощи блоков предопределенный процесс

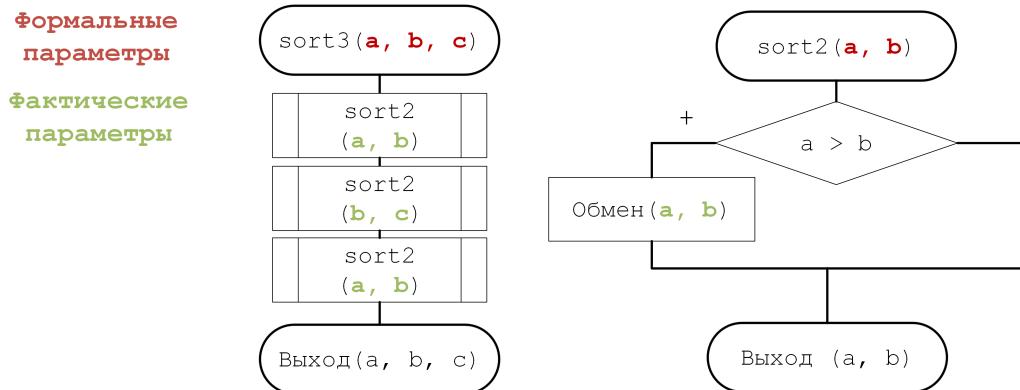


Рис. 7.7 – Формальные и фактические параметры

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.

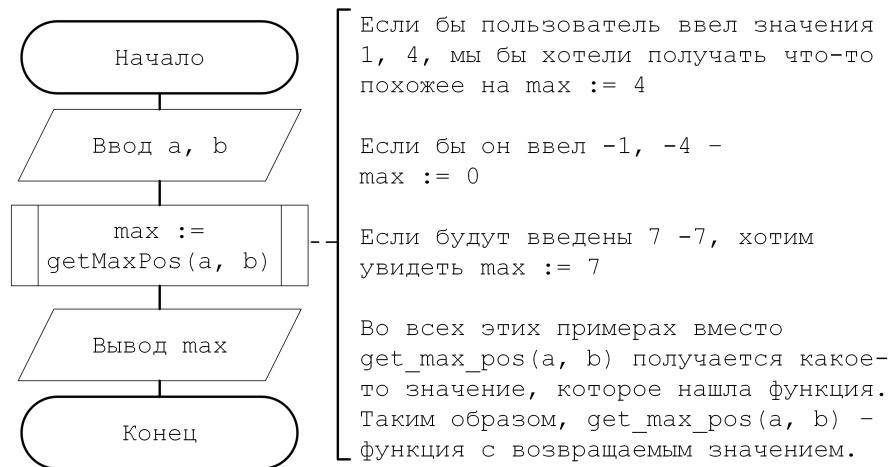


Рис. 7.8 – Блок-схема задачи о максимальном положительном значении из двух

Опишем блок-схему для функции `getMaxPos`. В ней нет ничего необычного, кроме слова возврат (этой идеи пока что достаточно):

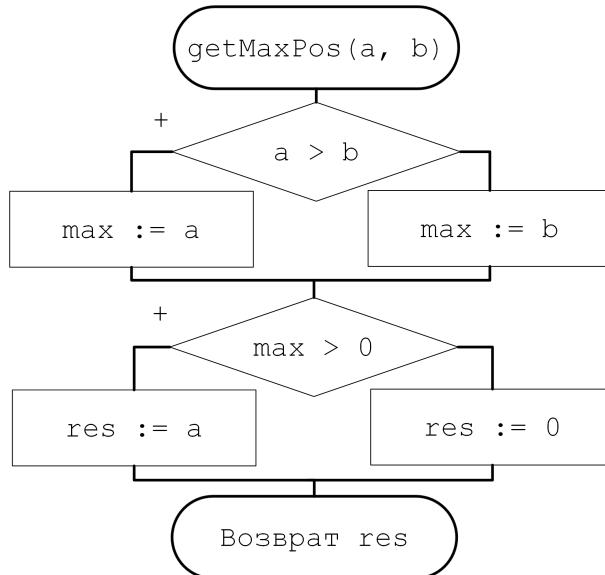


Рис. 7.9 – Пример блок-схемы функции с возвращаемым значением

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:

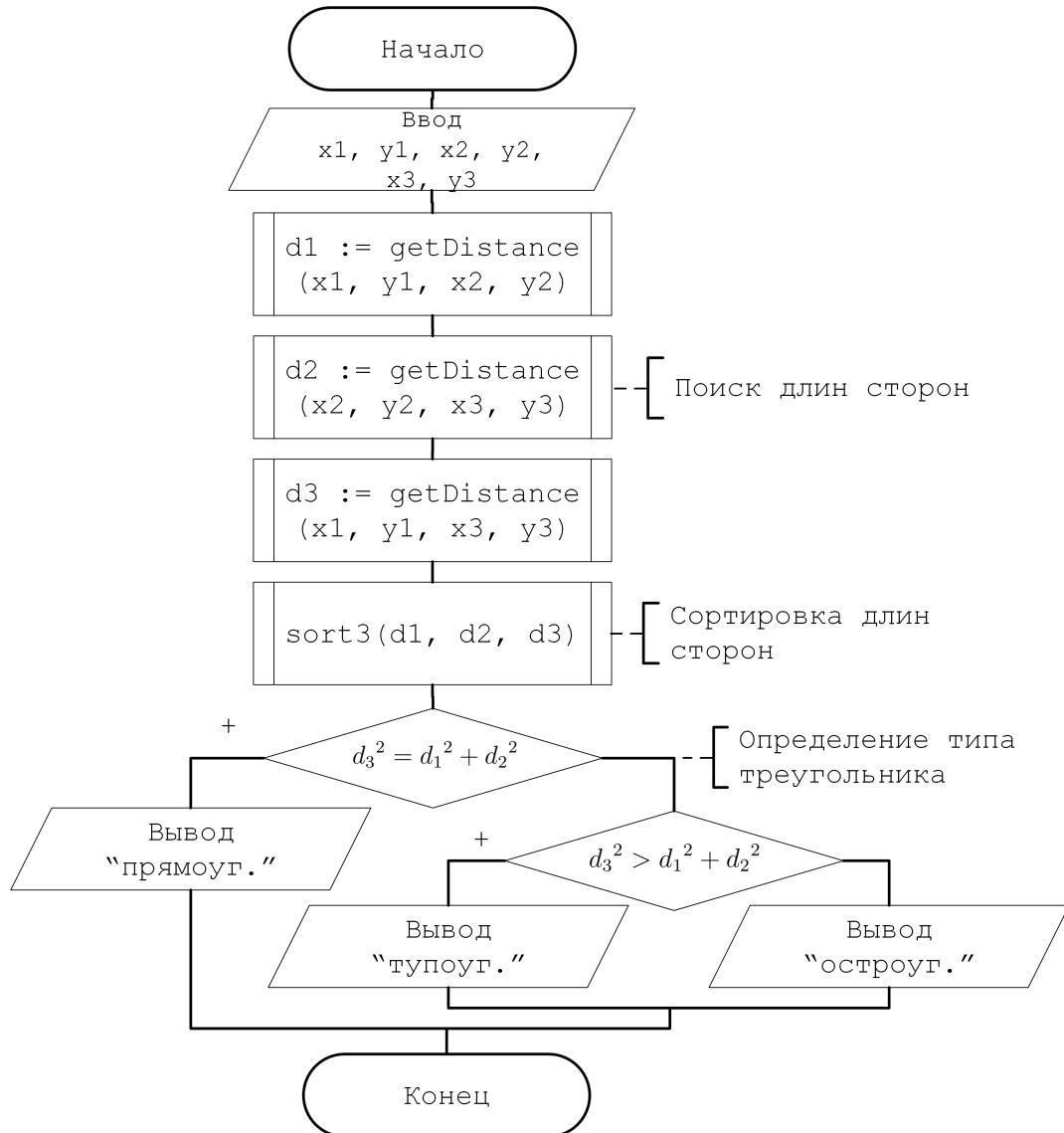


Рис. 7.10 – Блок-схема с использованием блоков предопределенный процесс

Опишем алгоритм для вычисления расстояния между двумя точками:

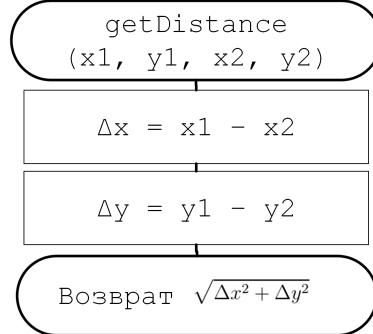


Рис. 7.11 – Блок-схема функции *distance*

7.3 Определение функции

Опишем при помощи блок-схемы алгоритм, который находил бы значение факториала. Рассмотрим функцию `getFactorial`:

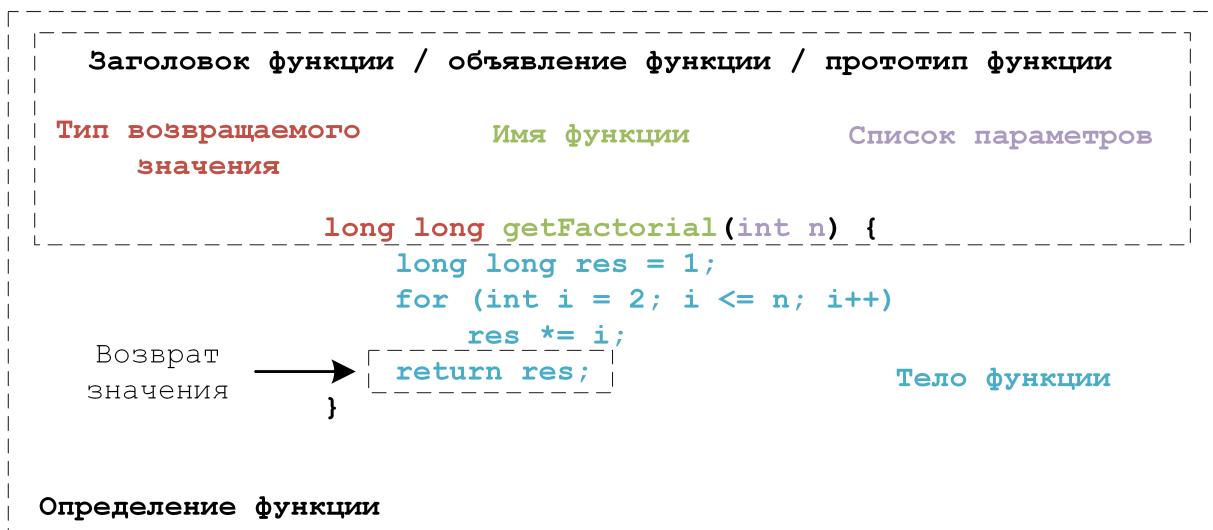
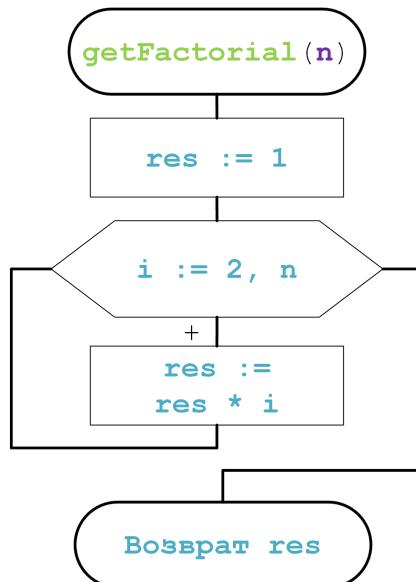


Рис. 7.12 – Компоненты функций

Определение функции состоит из следующих частей:

```
1 // <тип возвращаемого значения> <имя функции>([<список параметров>]) {
2 //     <тело функции>
3 // }
```

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:

```
1 long long getFactorial(int)
2 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:

```

1 // a, b - формальные параметры
2 void funcName(int a, int b);

```

Имена формальных параметров не могут повторяться.

Если функция не содержит параметров, круглые скобки не могут быть опущены:

```

1 int funcName();

```

Если список параметров содержит несколько переменных одного и того же типа, нельзя осуществлять такое объединение:

```

1 int funcName(int a, b);

```

каждый параметр должен быть описан отдельно:

```

1 int funcName(int a, int b);

```

Аргумент (фактический параметр) – это значение, которое передаётся в функцию при её вызове:

```

1 funcName(1, 3); // Аргументы: 1 и 3.
2 funcName(x, 7); // Аргументы: x и 7.

```

Аргументы функции должны быть перечислены через запятую.

Для того чтобы вызвать функцию, необходимо использовать оператор вызова ():

```

1 // <имя функции>(<список аргументов>)

```

Даже если функция не содержит формальных параметров, оператор вызова всё равно должен быть указан:

```

1 // <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):

```

1 #include <stdio.h>
2
3 long long getFactorial(int n) {
4     long long res = 1;
5     for (int i = 2; i <= n; ++i)
6         res *= i;
7     return res;
8 }
9
10 int main() {
11     int n;
12     scanf("%d", &n);
13
14     printf("%lld", getFactorial(n));
15
16     return 0;
17 }

```

в функции `getFactorial` создается переменная `n` и при выполнении тела вернёт в точку вызова `n!`. По окончанию работы `getFactorial` работа функции `main` возобновляется.

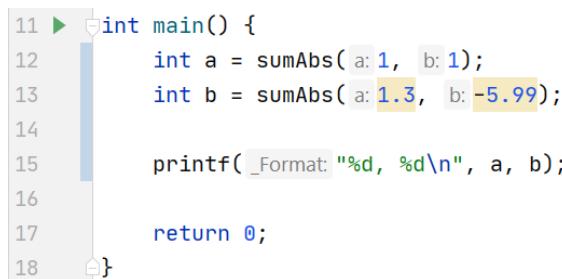
В процессе компиляции, компилятор проверяет возможность преобразования аргументов функции к типам формальных параметров. Рассмотрим на примере:

```

1 int sumAbs(int a, int b) {
2     if (a < 0)
3         a = -a;
4     if (b < 0)
5         b = -b;
6     return a + b;
7 }
8
9 int main() {
10    int a = sumAbs(1, 1);
11    int b = sumAbs(1.3, -5.99);
12
13    printf("%d, %d\n", a, b);
14
15    return 0;
16 }
```

При вычислении значения `a` (строка 10) никаких преобразований выполнено не будет, в отличии 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:



The screenshot shows a code editor with the following C code:

```

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 }
```

The IDE highlights the arguments of the `sumAbs` function calls with color-coded boxes: the first argument of each call is blue, and the second is orange. A tooltip "Format: %d, %d\n" appears over the `printf` call. The code editor interface includes a status bar at the bottom.

Если мы попробуем выполнить вызов:

```
1 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:

```
1 sum(1);
2 sum(1, 2, 3);
```

Ещё раз отметим, С/С++ 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:

```

1 int sum(int a, int b) {
2     return a + b;
3 }
4
5 int main() {
6     int a = sum(1, 1);
7     int b = sum(1.3, -5.99);
8
9     printf("%d, %d", a, b);
10
11    return 0;
12 }
```

или

```

1 int sum(int a, int b);
2 // или int sum(int, int);
3
4 int main() {
5     int a = sum(1, 1);
6     int b = sum(1.3, -5.99);
7
8     printf("%d, %d", a, b);
9
10    return 0;
11 }
12
13 int sum(int a, int b) {
14     return a + b;
15 }
```

7.6 Возвращаемое значение функции

Результатом работы функции может быть некоторое значение, которое называют **возвращаемым значением функции**:

```

1 int abs(int x) {
2     if (x < 0)
3         x = -x;
4     return x;
5 }
```

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:

```

1 int abs(int x) {
2     if (x < 0)
3         x = -x;
4     return x;
5 }
6
7 int main() {
8     int x = -3;
9     int ax = abs(x); // ax = 3
10    return 0;
11 }
```

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:

```
1 func_name(int a, int b);
```

Если функция не возвращает значение, то в качестве типа указывается `void`:

```
1 void print(int a, int b);
```

Массив не может быть типом возвращаемого значения. Ошибочно будет написать:

```
1 int[10] funcName();
```

Но можно вернуть указатель на необходимую область памяти:

```
1 int* funcName();
```

Возврат значения осуществляется одной из двух инструкций:

```
1 return;
2 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:

```
1 int hasOdd(const int *a, const int n) {
2     // a - указатель на массив из n элементов
3     for (int i = 0; i < n; i++)
4         if (a[i] % 2)
5             return 1;
6     return 0;
7 }
8
9 int main() {
10     int a[] = {1, 2, 3, 4, 5};
11     printf("%d", has_odd(a, 5));
12
13     return 0;
14 }
```

Если тип возвращаемого значения не соответствует указанному в объявлении, происходит неявное преобразование типов:

```
1 int trunc(double a) {
2     return a;
3 }
4
5 int main() {
6     printf("%d", trunc(4.24)); // 4
7     return 0;
8 }
```

Среды выдают предупреждения при неявных преобразованиях (рисунок 14.15). 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).

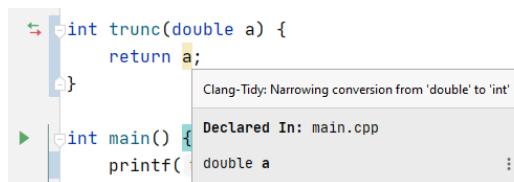


Рис. 7.13 – Предупреждение *IDE* о неявном преобразовании при возврате значения

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):

```

1 int* initializationA() {
2     int a[10];
3     for (int i = 0; i < 10; i++)
4         a[i] = i;
5     return a; // массив a создан внутри функции getA.
6             // нельзя вернуть ссылку на начальный элемент.
7 }
8
9 int main() {
10    int *a = initializationA();
11    return 0;
12 }
```

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:

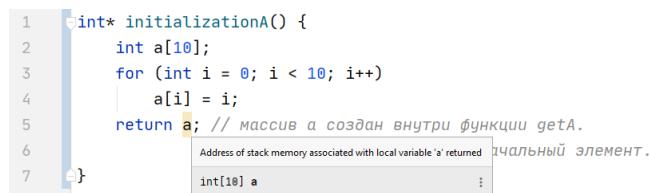


Рис. 7.14 – Предупреждение *IDE* для попытки вернуть локальную переменную *a*, которая является созданным в функции массивом

Возможное решение проблемы можно осуществить через динамические массивы (что будет описано позже) или следующим образом:

```

1 void initializationA(int *a, const int n) {
2     for (int i = 0; i < 10; i++)
3         a[i] = i;
4 }
5
6 int main() {
7     int a[10];
8     // чтобы передать массив в функцию,
9     // передаётся указатель на нулевой элемент массива
10    initializationA(a, 10);
11    return 0;
12 }
```

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):

```

1 int sum(int a, int b) {
2     return a + b;
3 }
4
5 int main() {
6     int v1 = 1;
7     int v2 = 7;
8     int s = sum(v1, v2);
9     printf("%d", s);
10
11    return 0;
12 }
```

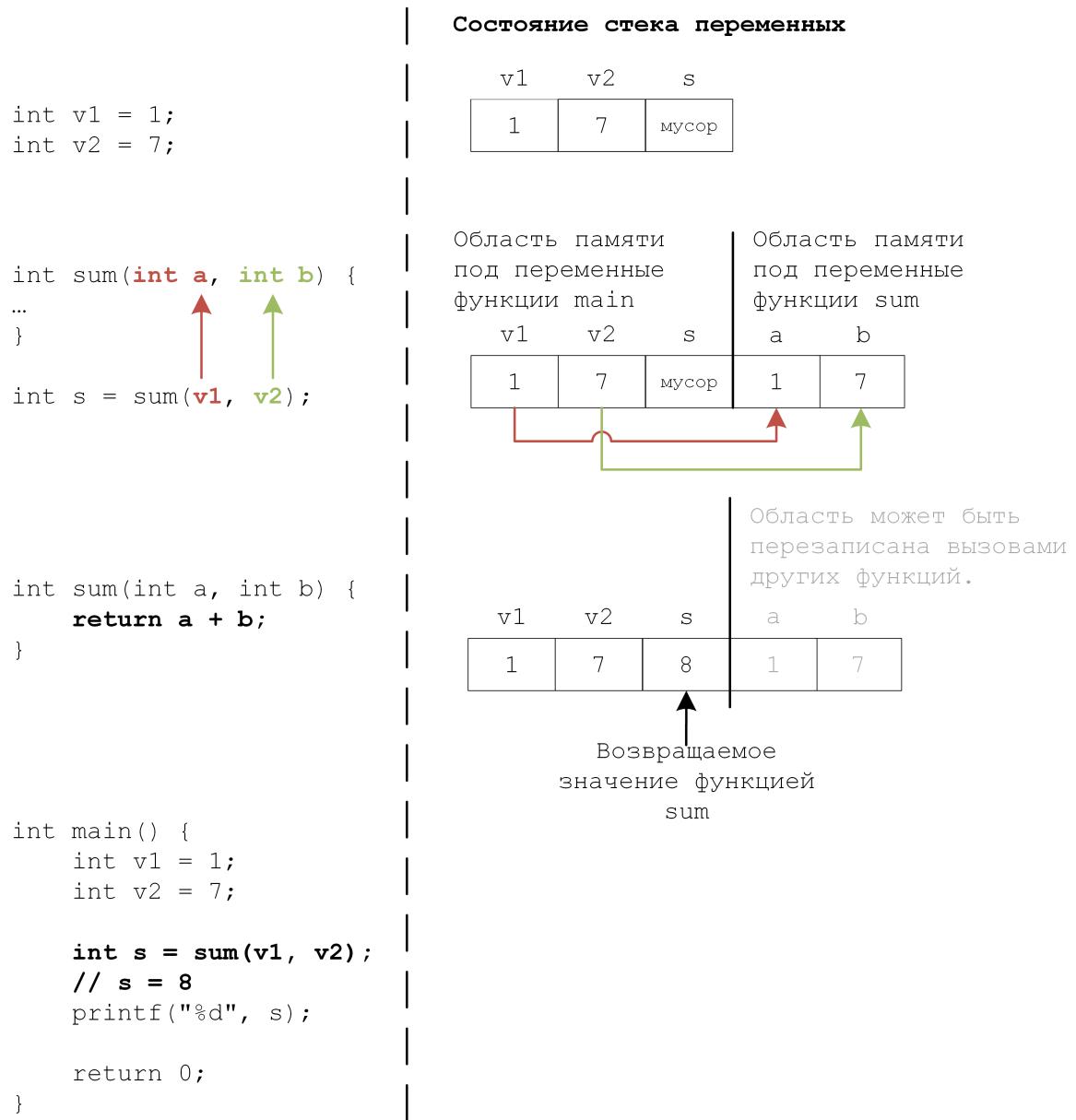


Рис. 7.15 – Передача аргументов в функцию и возврат значения из функции

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:

```

1 void reset(int a) {
2     a = 0;
3 }
4
5 int main() {
6     int a = 42;
7     reset(a);
8     printf("%d", a);
9
10    return 0;
11 }
```

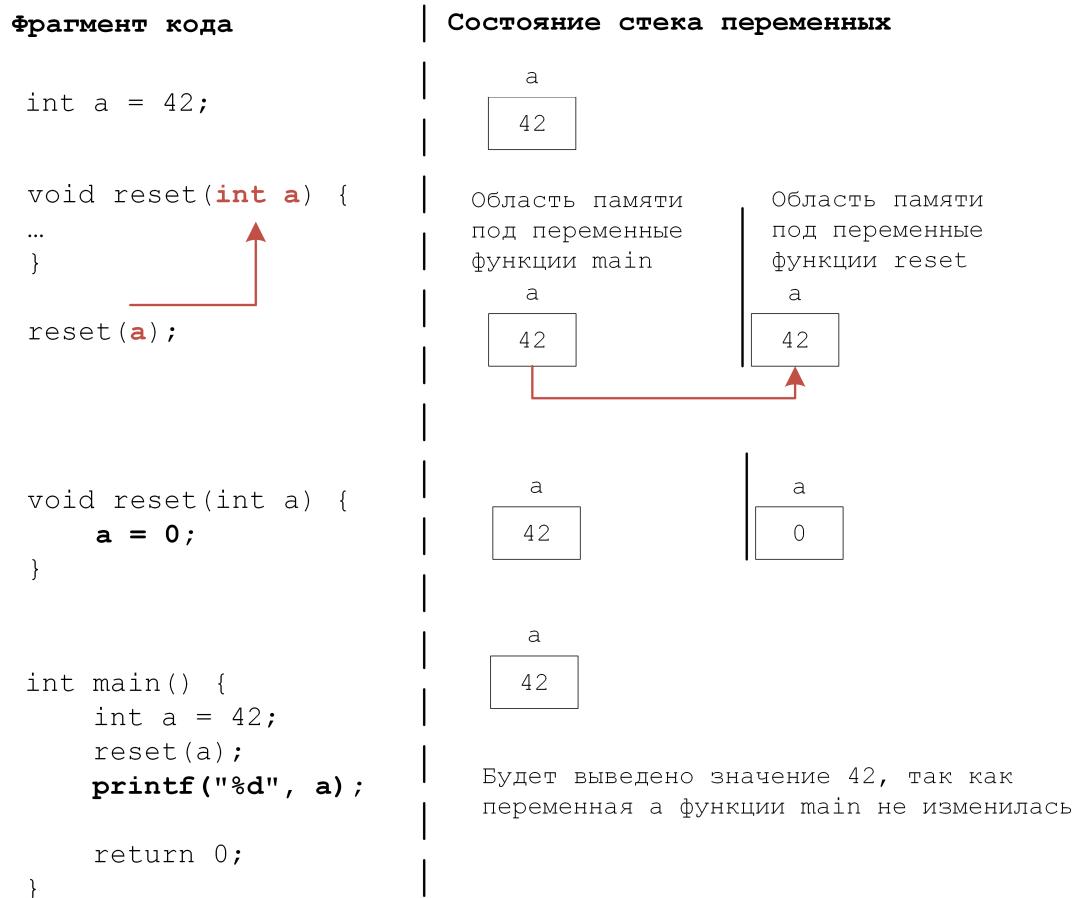


Рис. 7.16 – Передача значения в функцию

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:

```

1 void swap(int a, int b) {
2     int t = a;
3     a = b;
4     b = t;
5 }
6
7 int main() {
8     int a = 1;
9     int b = 2;
10    printf("Before: a = %d, b = %d\n", a, b);
11    swap(a, b);
12    printf("After: a = %d, b = %d\n", a, b);
13    return 0;
14 }
```

не работает (передача осуществляется по значению):

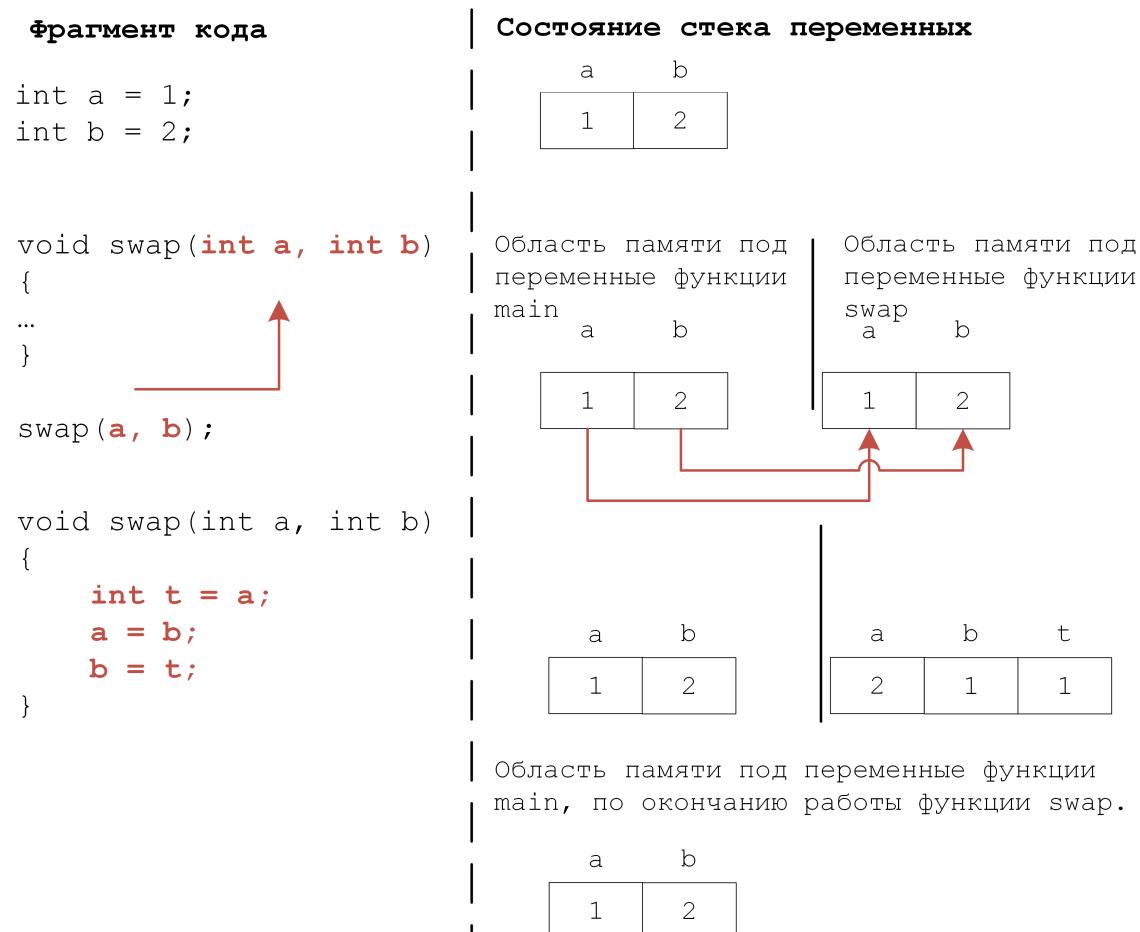


Рис. 7.17 – При передаче параметров таким образом исходные данные не будут поменяны

Для того, чтобы обмен всё-таки осуществился, необходимо передать в функцию адреса изменяемых данных (а в функции в качестве формальных параметров используются указатели):

```

1 void swap(int *a, int *b) {
2     int t = *a;
3     *a = *b;
4     *b = t;
5 }
```

```

7 int main() {
8     int a = 1;
9     int b = 2;
10    printf("Before: a = %d, b = %d\n", a, b);
11    swap(&a, &b);
12    printf("After: a = %d, b = %d\n", a, b);
13    return 0;
14 }

```

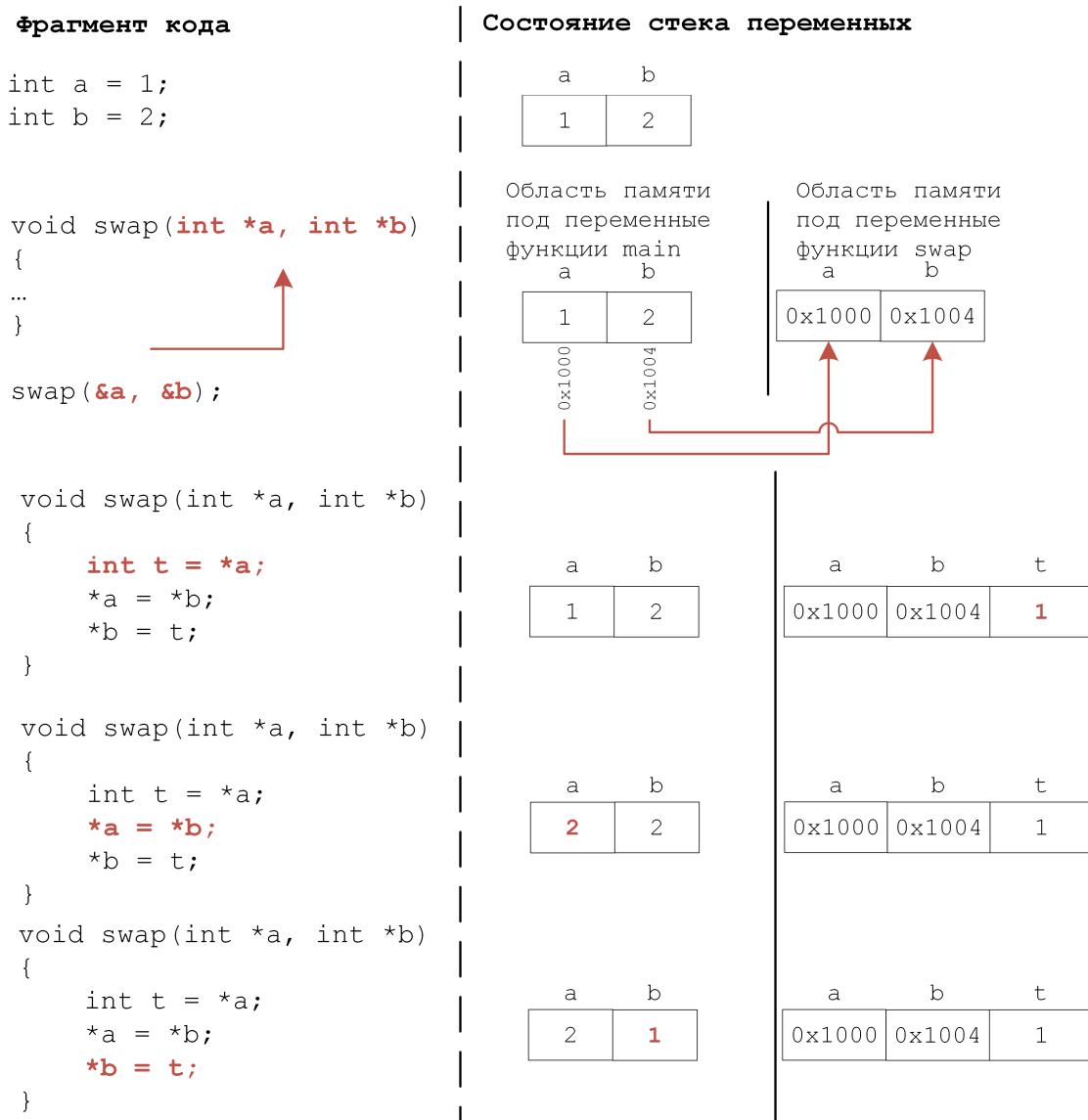


Рис. 7.18 – Передача аргументов в функцию при помощи указателей

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`:

```

1 void someFunc(const int *a) {
2     printf("%d", *a);
3     // *a = 4; - ошибка, а - указатель на константное целое
4     // значение по адресу а не может быть изменено
5 }

```

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.

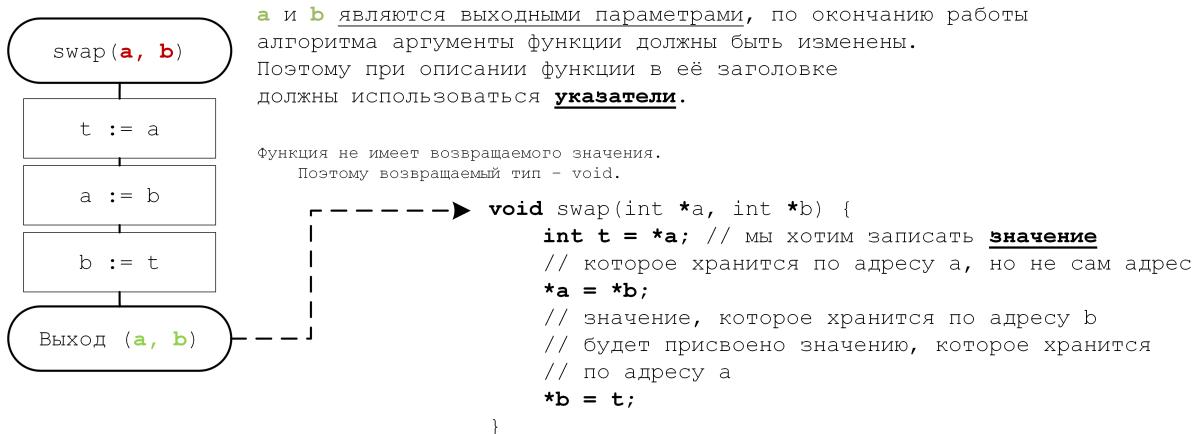


Рис. 7.19 – Реализация функции *swap*

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.

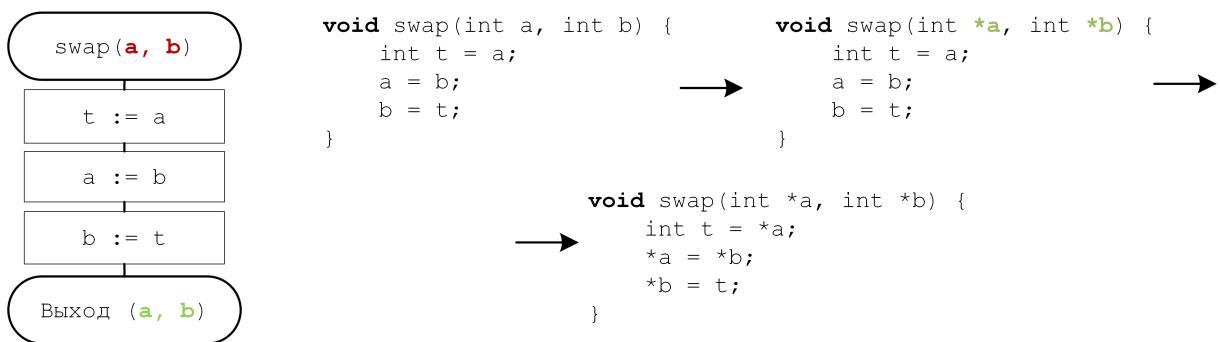


Рис. 7.20 – Процесс написания функции

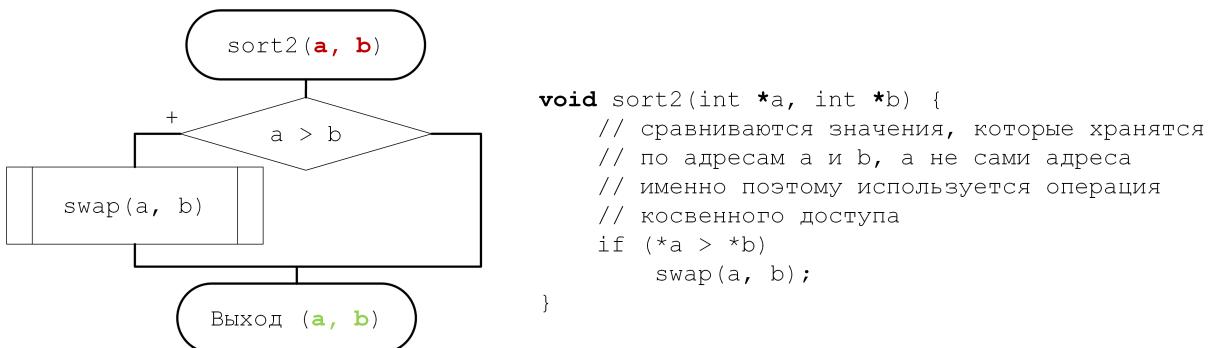


Рис. 7.21 – Реализация функции sort2. В блок-схеме видно, что вызывается функция swap

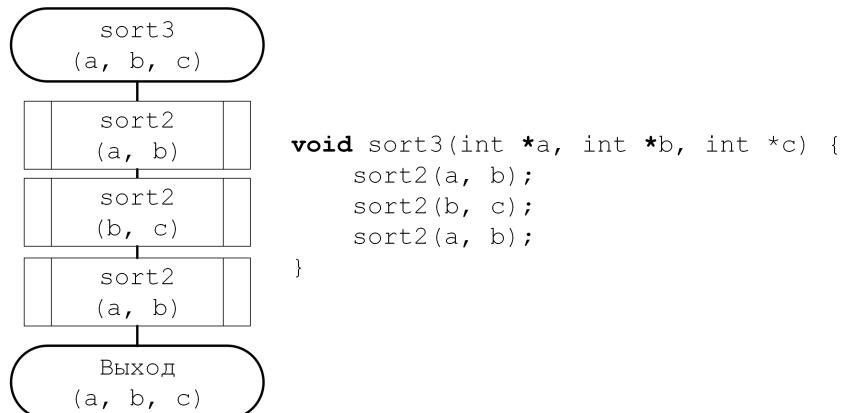
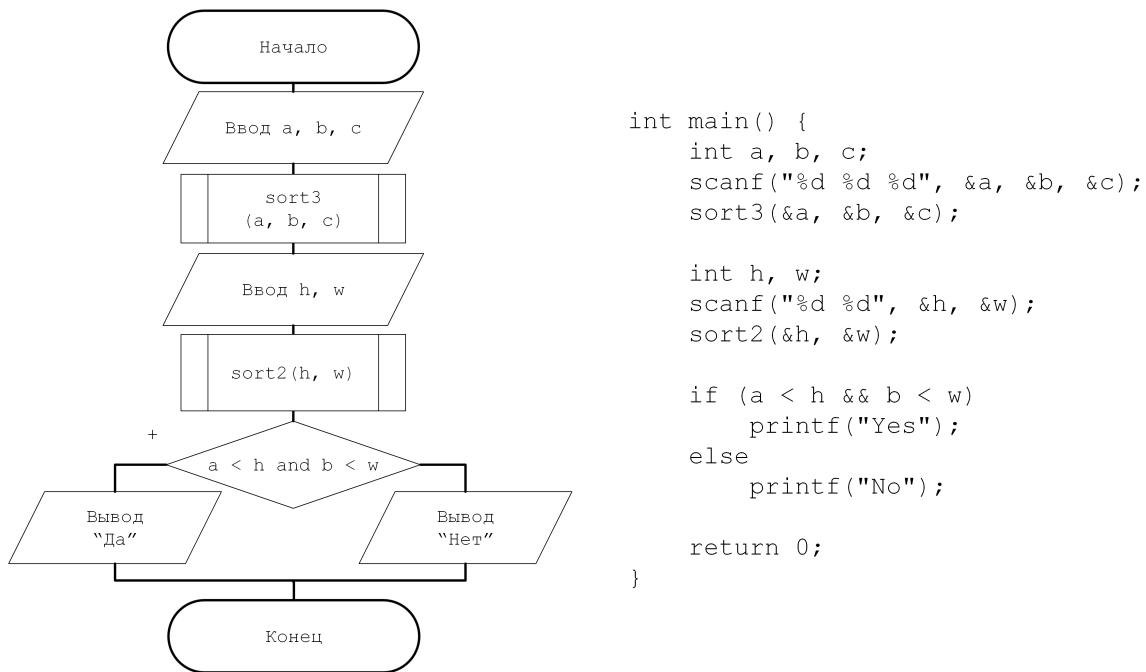


Рис. 7.22 – Реализация функции sort3

Рис. 7.23 – Реализация функции *main*

Исходный код:

```

1 #include <stdio.h>
2
3 void swap(int *a, int *b) {
4     int t = *a;
5     *a = *b;
6     *b = t;
7 }
8
9 void sort2(int *a, int *b) {
10    if (*a > *b)
11        swap(a, b);
12 }
13
14 void sort3(int *a, int *b, int *c) {
15    sort2(a, b);
16    sort2(b, c);
17    sort2(a, b);
18 }
19
20 int main() {
21    int a, b, c;
22    scanf("%d %d %d", &a, &b, &c);
23    sort3(&a, &b, &c);

25    int h, w;
26    scanf("%d %d", &h, &w);
27    sort2(&h, &w);

29    if (a < h && b < w)
30        printf("Yes");
31    else
32        printf("No");

33
34    return 0;
35 }
  
```

Код решения задачи о типе треугольника:

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <windows.h>
4
5 // обмен значений по адресам a и b
6 void swap(double *a, double *b) {
7     double t = *a;
8     *a = *b;
9     *b = t;
10}
11
12 // упорядочение значений по адресам a и b
13 void sort2(double *a, double *b) {
14     if (*a > *b)
15         swap(a, b);
16 }
17
18 // упорядочение значений по адресам a, b, c
19 void sort3(double *a, double *b, double *c) {
20     sort2(a, b);
21     sort2(b, c);
22     sort2(a, b);
23 }
24
25 // возвращает расстояние между точками (x1, y1), (x2, y2)
26 double getDistance(int x1, int y1, int x2, int y2) {
27     int deltaX = x1 - x2;
28     int deltaY = y1 - y2;
29     return sqrt(deltaX * deltaX + deltaY * deltaY);
30 }
31
32 int main() {
33     SetConsoleOutputCP(CP_UTF8);
34
35     // ввод исходных данных
36     int x1, y1, x2, y2, x3, y3;
37     scanf("%d %d %d %d %d %d", &x1, &y1, &x2, &y2, &x3, &y3);
38
39     // поиск длин сторон и упорядочение
40     double d1 = getDistance(x1, y1, x2, y2);
41     double d2 = getDistance(x1, y1, x3, y3);
42     double d3 = getDistance(x2, y2, x3, y3);
43     sort3(&d1, &d2, &d3);
44
45     // определение типа треугольника
46     double bigSideSquare = d3 * d3;
47     double sumOfSquaresOfSides = d1 * d1 + d2 * d2;
48
49     // специфика сравнения вещественных на равенство
50     if (fabs(bigSideSquare - sumOfSquaresOfSides) < 1e-12)
51         printf("Прямоугольный");
52     else if (bigSideSquare > sumOfSquaresOfSides)
53         printf("Тупоугольный");
54     else
55         printf("Остроугольный");
56
57     return 0;
58 }
```

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:

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     scanf("%d %d", &a, &b);
6
7     long long resA = 1;
8     for (int i = 1; i <= 3; i++)
9         resA *= a;
10
11    long long resB = 1;
12    for (int i = 1; i <= 5; i++)
13        resB *= b;
14
15    printf("%lld", resA + resB);
16
17    return 0;
18 }
```

Было бы удобно для данной задачи выделить функцию:

```

1 long long int power(int a, int n) {
2     // res инициализируется значением 1, чтобы корректно обрабатывать  $a^0$ 
3     long long res = 1;
4     for (int i = 1; i <= n; i++)
5         res *= a;
6     return res;
7 }
```

которая возводит число `a` в степень `n`. Тогда код `main` не содержит дублирования и сократится до

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     scanf("%d %d", &a, &b);
6
7     printf("%lld", power(a, 3) + power(b, 5));
8
9     return 0;
10 }
```

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:

```

1 long long int power(int a, int n) {
2     long long x = a;
3     long long res = 1;
4     while (n != 0) {
5         if (n & 1)      // или n % 2
6             res *= x;
7         n >>= 1;        // или n /= 2;
8         x = x * x;
9     }
10    return res;
11 }
```

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.

Реализация²:

```

1 int gcd(int a, int b) {
2     while (a != b)
3         if (a > b)
4             a = a - b;
5         else
6             b = b - a;
7     return a;
8 }
```

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.

```

1 int gcd(int a, int b) {
2     while (a != 0 && b != 0)
3         if (a > b)
4             a = a % b;
5         else
6             b = b % a;
7     return a + b;
8 }
```

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;"><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	0	1	1	0	0	1	1	1	1	1
0	0	0	1										
1	0	0	1										
1	1	1	1										
x >> 8 & 15:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table> 1	0	0	0	1								
0	0	0	1										
x >> 4 & 15:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table> 9	1	0	0	1								
1	0	0	1										
x & 15:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table> F	1	1	1	1								
1	1	1	1										

Рис. 7.24 – Последовательность получения числа в 16-ричном представлении

² Greatest common divisor - наибольший общий делитель (англ.)

Опишем функцию, возвращающую количество цифр в 16-ричном представлении числа:

```

1 #define BINARY_DIGITS_IN_ONE_HEX_DIGIT 4U // U - литерал беззнакового типа
2
3 // возвращает количество цифр в 16-ом представлении числа x
4 int countHexDigits(unsigned int x) {
5     int count = 0;
6     while (x != 0) {
7         x >>= BINARY_DIGITS_IN_ONE_HEX_DIGIT;
8         count++;
9     }
10    return count;
11 }
12
13 // вывод 16-ричного представления числа x
14 void printHex(unsigned int x) {
15     int nHexDigits = countHexDigits(x);
16     int shift = (nHexDigits - 1) * BINARY_DIGITS_IN_ONE_HEX_DIGIT;
17     while (shift >= 0) {
18         int hexDigit = x >> shift & 15;
19         if (hexDigit < 10)
20             printf("%d", hexDigit);
21         else
22             printf("%c", (hexDigit - 10 + 'A'));
23         shift -= BINARY_DIGITS_IN_ONE_HEX_DIGIT;
24     }
25 }
```

Можно было рассуждать и иначе: последовательно смотреть группы из 4 цифр в двоичном представлении и осуществлять вывод значащих цифр:

```

1 #define BINARY_DIGITS_IN_ONE_HEX_DIGIT 4U // U - литерал беззнакового типа
2 #define BITS_IN_BYTE 8
3
4 void printHex(unsigned int x) {
5     if (x == 0) {
6         printf("0");
7         return;
8     }
9
10    int needPrint = 0;
11    for (int shift = sizeof(unsigned) * BITS_IN_BYTE -
12        BINARY_DIGITS_IN_ONE_HEX_DIGIT;
13        shift >= 0;
14        shift -= BINARY_DIGITS_IN_ONE_HEX_DIGIT) {
15        int hexDigit = x >> shift & 15;
16        if (hexDigit != 0)
17            needPrint = 1;
18        if (needPrint) {
19            if (hexDigit < 10)
20                printf("%d", hexDigit);
21            else
22                printf("%c", (hexDigit - 10 + 'A'));
23        }
24    }
```

Особое внимание стоит уделить наличию `return` в 4 строке. Мы могли бы обойтись без него и переместить весь дальнейший код в `else`:

```

1 void printHex(unsigned int x) {
2     if (x == 0)
3         printf("0");
4     else {
5         int needPrint = 0;
6         // ...
7     }
8 }
```

но это бы и так увеличило вложенность тела функции, что нежелательно. Решите для себя какого варианта придерживаться.

Задача вывода числа в 16-ричном представлении может быть решена рекурсивно:

```

1 void _printHex(unsigned int x) {
2     int hexDigit = x % 16;
3     if (x != 0) {
4         _printHex(x >> 4U);
5         if (hexDigit < 10)
6             printf("%d", hexDigit);
7         else
8             printf("%c", (hexDigit - 10 + 'A'));
9     }
10 }
11
12 void printHex(unsigned int x) {
13     if (x == 0)
14         printf("0");
15     else
16         _printHex(x);
17 }
```

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, $14420083 \rightarrow 1442|0083 \rightarrow 1 + 4 + 4 + 2 = 0 + 0 + 8 + 3 \rightarrow \rightarrow 11 = 11$ (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.

```

1 #include <stdio.h>
2
3 // возвращает количество цифр в записи числа x
4 int countDigits(long long x) {
5     int count = 1;
6     while (x > 9) {
7         count++;
8         x /= 10;
9     }
10    return count;
11 }
12
13 // возвращает сумму последних n цифр числа x
14 // уменьшает значение x на 10 в степени n раз
15 int getSumLastNDigits(long long *x, int n) {
16     // так как мы хотим, чтобы переданное значение в функцию изменилось,
17     // в качестве фактического аргумента передаётся адрес переменной;
18     // указатель - переменная, которая может хранить адрес другой переменной
19     // поэтому в качестве фактического параметра функции использован указатель
20     int sum = 0;
21     for (int i = 0; i < n; i++) {
22         sum += *x % 10;
23         *x /= 10;
24     }
25     return sum;
26 }
27
28 // возвращает значение 'истина', если сумма первых
29 // n/2 цифр числа x равна сумме последних n/2 цифр числа x,
30 // где n - количество цифр в числе x
31 int isLuckyNumber(long long x) {
32     int halfNumberLen = countDigits(x) / 2;
33     // мы хотим, чтобы значение переменной x после выполнения функции
34     // getSumLastNDigits изменилось, поэтому мы передаём адрес переменной
35     int secondHalfSum = getSumLastNDigits(&x, halfNumberLen);
36     int firstHalfSum = getSumLastNDigits(&x, halfNumberLen);
37     return firstHalfSum == secondHalfSum;
38 }
39
40 int main() {
41     long long x;
42     scanf("%lld", &x);
43
44     printf("%d", isLuckyNumber(x));
45
46     return 0;
47 }
```

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:

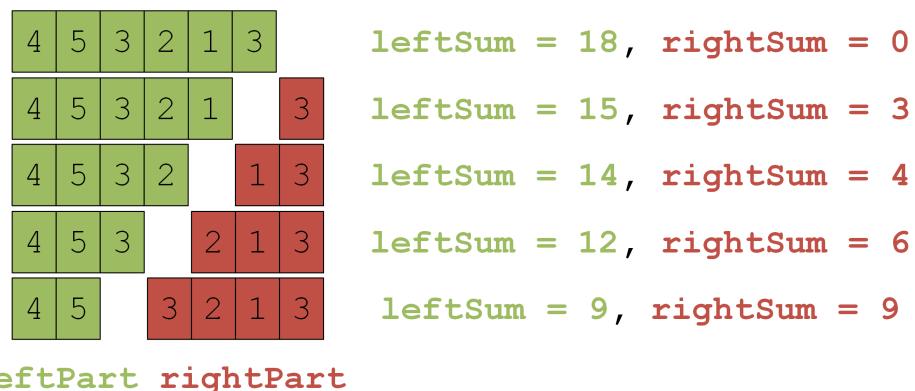


Рис. 7.25 – Последовательное вычисление сумм. Каждая последующая строка может быть вычислена через предыдущую

```

1 #include <stdio.h>
2
3 // возвращает сумму цифр числа x
4 int getSumOfDigits(long long x) {
5     int sum = 0;
6     while (x > 0) {
7         sum += x % 10;
8         x /= 10;
9     }
10    return sum;
11 }
12
13 // возвращает значение 'истина', если число x может быть разделено
14 // на две части, суммы которых равны. Записывает по адресу a
15 // первую часть числа, записывает по адресу b - вторую часть
16 // числа. Если разбиение невозможно, возвращает 'ложь'.
17 int getSumLastNDigits(long long x, long long *a, long long *b) {
18     int leftSum = getSumOfDigits(x);
19     int rightSum = 0;
20     long long y = 1;
21     long long leftPart = x;
22     long long secondPart = 0;
```

```

23     do {
24         int digit = leftPart % 10;
25         secondPart += y * digit;
26         y *= 10;
27         rightSum += digit;
28         leftSum -= digit;
29         leftPart /= 10;
30     } while (leftSum > rightSum);
31
32     if (leftSum == rightSum) {
33         *a = leftPart;
34         *b = secondPart;
35         return 1;
36     } else
37         return 0;
38 }
```

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.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 // возвращает значение 'истина' если число x является совершенным
5 // иначе - 'ложь'.
6 int isPerfectNumber(long long x) {
7     const int minPossibleDivider = 2;
8     const long long upperBound = sqrtl(x);
9     long long sumOfDividers = 1;
10    for (int possibleDivider = minPossibleDivider;
11        possibleDivider <= upperBound; possibleDivider++) {
12        if (x % possibleDivider == 0) {
13            sumOfDividers += possibleDivider;
14            sumOfDividers += x / possibleDivider;
15        }
16    }
17    if (upperBound * upperBound == x)
18        sumOfDividers -= upperBound;
19    return x == sumOfDividers;
20 }
21
22 void printPerfectNumbersReport(int lowerBound, int upperBound) {
23     int hasPerfectNumber = 0;
24     for (int number = lowerBound; number <= upperBound; number++) {
25         if (isPerfectNumber(number)) {
26             if (hasPerfectNumber == 0) {
27                 printf("perfect numbers: ");
28                 hasPerfectNumber = 1;
29             }
30             printf("%d ", number);
31         }
32     }
}
```

```

33     if (!hasPerfectNumber)
34         printf("No perfect numbers");
35 }
36
37
38 int main() {
39     int lowerBound, upperBound;
40     scanf("%d %d", &lowerBound, &upperBound);
41
42     printPerfectNumbersReport(lowerBound, upperBound);
43
44     return 0;
45 }
```

7.8.7 Быки и коровы.

Быки и коровы

Опишем правила игры "Быки и коровы"^a. 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.

^aДля 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 нулем).
 - 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):

```

1 #include <stdio.h> // ввод / вывод
2 #include <stdlib.h> // генерация случайных значений
3 #include <time.h> // из библиотеки потребуется функция time
4 // которая позволит задать случайное зерно генерации
5
6 // требуемое количество 'быков' в игре
7 #define NEED_BULLS_TOTAL 4
8
9
10 int main() {
11     srand(time(0)); // устанавливаем зерно генерации
12
13 }
```

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

```
1 int generateStartValue()
```

будет функцией с возвращаемым значением типа `int`. В точку вызова

```
1 int answer = generateStartValue();
```

в процессе выполнения программы подставится вычисленное значение. Например, 3476:

```
1 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 $x = 10^{NEED_BULLS_TOTAL}$. Например, если генератор выдаст значение 123456 и требуемое количество быков равно 4, получим 3456.

```

1 // возвращает случайное значение, удовлетворяющее условию игры
2 int generateStartValue() {
3     // переменная имеет модификатор const, так как не изменяется
4     // в процессе работы функции
```

```

5   const long long maxValueBound = intPow(10, NEED_BULLS_TOTAL);
6   int generatedValue;
7   do {
8       generatedValue = rand() % maxValueBound;
9   } while (!isCorrectNumber(generatedValue));
10
11   return generatedValue;
12 }
```

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 возвведения недоступна и придётся её реализовывать³.

Подловим себя на мысли о том, как ведётся разработка приложения. В процессе написания кода последовательно выделяются функций, которых нет в языке. Продолжаем процесс до тех пор, пока приложение не будет реализовано.

Опишем функцию возвведения в степень. Самую простую⁴:

```

1 // возвращает значение а в степени n
2 long long intPow(const int a, const int n) {
3     long long res = 1;
4     for (int i = 1; i <= n; i++)
5         res *= a;
6
7     return res;
8 }
```

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.

```

1 // возвращает значение 'истина', если число x является допустимым
2 // согласно правилам игры, иначе - ложь
3 int isCorrectNumber(const int x) {
4     // если число больше или равно  $10^{NEED_BULLS\_TOTAL}$ 
5     // будет возвращено значение 0
6     const long long maxValueBound = intPow(10, NEED_BULLS_TOTAL);
7     if (x / maxValueBound)
8         return 0;
9
10    // получаем множество цифр
11    int digits[NEED_BULLS_TOTAL];
12    getDigitSet(x, digits, NEED_BULLS_TOTAL);
13    // проверяем уникальность
14    return isUniqueArray(digits, NEED_BULLS_TOTAL);
15 }
```

³В работе над реальными проектами не изобретайте велосипед.

⁴Не старайтесь оптимизировать сразу. Лучше в готовом решении найти критичные по производительности места и оптимизировать только их.

Функция для получения последовательности цифр из числа:

```

1 // записывает по адресу digitSet цифры числа nDigits цифр числа x.
2 // если nDigits больше количества цифр в x - записывает в оставшиеся
3 // элементы массива значения 0
4 void getDigitSet(long long x, int *digitSet, const size_t nDigits) {
5     for (int i = nDigits - 1; i >= 0; i--) {
6         int digit = x % 10;
7         digitSet[i] = digit;
8         x /= 10;
9     }
10 }
```

Функция принимает указатель на массив длины *nDigits*. В процессе обработки числа она производит его заполнение:

	0	1	2	3		0	1	2	3
x = 4136					x = 12				
x = 413				6	x = 1				2
x = 41			3	6	x = 0			1	2
x = 4		1	3	6	x = 0		0	1	2
x = 0	4	1	3	6	x = 0	0	0	1	2

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:

```

1 int isCorrectNumber(const int x) {
2     int a[10] = {0}
3     for (int i = 0; i < NEED_BULLS_TOTAL; i++) {
4         int digit = x % 10;
5         if (a[digit]++)
6             return 0;
7     }
8     return 1;
9 }
```

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.

```

1 // возвращает значение 'истина', если все элементы массива a размера n
2 // являются уникальными иначе - 'ложь'
3 int isUniqueArray(const int *a, const size_t n) {
4     for (int i = 0; i < n - 1; i++)
5         for (int j = i + 1; j < n; j++)
6             if (a[i] == a[j])
7                 return 0;
8     return 1;
9 }
```

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:

```

1 // записывает по адресу nBulls количество быков,
2 // а по адресу nCows - количество коров при условии, что загадано
3 // число answer и выполнено предположение в виде числа hypothesis
4 void getNAnimals(const int answer, const int hypothesis,
5                  int *nBulls, int *nCows) {
6     int answerDigitSet[NEED_BULLS_TOTAL];
7     getDigitSet(answer, answerDigitSet, NEED_BULLS_TOTAL);
8
9     int hypothesisDigitSet[NEED_BULLS_TOTAL];
10    getDigitSet(hypothesis, hypothesisDigitSet, NEED_BULLS_TOTAL);
11
12    *nBulls = 0;
13    *nCows = 0;
14    for (int i = 0; i < NEED_BULLS_TOTAL; i++)
15        for (int j = 0; j < NEED_BULLS_TOTAL; ++j) {
16            if (hypothesisDigitSet[i] == answerDigitSet[j]) {
17                if (i == j)
18                    *nBulls += 1;
19                else
20                    *nCows += 1;
21            }
22        }
23 }
```

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:

```

1 int main() {
2     srand(time(0));
3
4     int answer = generateStartValue();
5     // при необходимости скрыть сгенерированное значение
6     // следующую строку нужно закомментировать;
7     // будет выведено NEED_BULLS_TOTAL-цифр, если в числе answer цифр меньше
8     // выведутся ведущие нули до длины NEED_BULLS_TOTAL
9     printf("generated value: %0*.d\n", NEED_BULLS_TOTAL, answer);
10
11    int hypothesis;
12    while (1) {
13        scanf("%d", &hypothesis);
14        if (!isCorrectNumber(hypothesis)) {
15            printf("No correct number\n");
16        } else {
17            int nBulls, nCows;
18            getNAnimals(answer, hypothesis, &nBulls, &nCows);
19            if (nBulls == NEED_BULLS_TOTAL) {
20                printf("win!");
21                break;
22            } else
23                printf("nCows = %d, nBulls = %d\n", nCows, nBulls);
24        }
25    }
26
27    return 0;
28 }
```

Разместим функции в файле в следующем порядке:

```

1 #include <stdio.h> // ввод / вывод
2 #include <stdlib.h> // генерация случайных значений
3 #include <time.h> // из библиотеки потребуется функция time
4 // которая позволит задать случайное зерно генерации
5
6 // требуемое количество 'быков' в игре
7 #define NEED_BULLS_TOTAL 4
8
9 int generateStartValue();
10 int isCorrectNumber(const int x);
11 int isUniqueArray(const int *a, const size_t n);
12 void getDigitSet(long long x, int *digitSet, const size_t nDigits);
13 long long intPow(const int a, const int n);
14 void getNAnimals(const int answer, const int hypothesis, int *nBulls,
15                  int *nCows);
16
17 int main() {
18     //
19 }
20 void getDigitSet(long long x, int *digitSet, const size_t nDigits) {
21     //
22 }
23
24 void getNAnimals(const int answer, const int hypothesis,
25                  int *nBulls, int *nCows) {
26     //
27 }
28 // ...
```

7.9 Рекурсивные функции

В программировании **рекурсия**⁵ – 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.

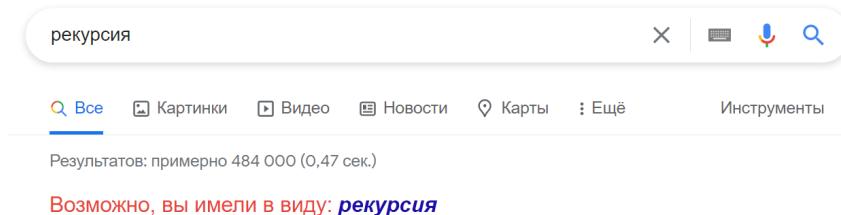


Рис. 7.28 – Разработчики поисковых сервисов не могли не пошутить об этом

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:

```

1 // function f(x):
2 //     if <условие прекращения рекурсии>:
3 //         //...
4 //     [else if //...
5 //         //...
6 //     ]
7 //     else:
8 //         //...

```

Так как в условии прекращения рекурсии имеется инструкция возврата в случае бинарного ветвления можно заметить два равнозначных варианта:

```

1 // function f(x):
2 //     if <условие прекращения рекурсии>:
3 //         // ...
4 //         return [<возвращаемое значение>];
5 //     else:
6 //         <оператор рекурсивной ветви>

```

или

```

1 // function f(x):
2 //     if <условие прекращения рекурсии >:
3 //         // ...
4 //         return [< возвращаемое значение >];
5 //     <оператор рекурсивной ветви>

```

⁵Перед 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...

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.

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 функций⁶:

$$isEven(n) = \begin{cases} true & \text{если } n = 0, \\ isOdd(n - 1) & \text{если } n > 0. \end{cases}$$

$$isOdd(n) = \begin{cases} false & \text{если } n = 0, \\ isEven(n - 1) & \text{если } n > 0. \end{cases}$$

⁶В качестве взаиморекурсивных определений приведены следующие: операция – действие, выполняемое над операндом; операнд – объект, которым оперируют операции.

- **Вложенная рекурсия** – вид рекурсии, при которой аргумент рекурсивной функции определяется как результат другого рекурсивного вызова:

$$F(s, n) = \begin{cases} 1 + s & \text{если } n = 1 \text{ или } n = 2, \\ F(n - 1, s, F(n - 2, 0)) & \text{если } n > 2. \end{cases}$$

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.

Большинство рассматриваемых в литературе задач на тему рекурсии часто являются чисто учебными, однако их решение помогает лучше адаптироваться к реальным условиям.

7.9.1 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:

```
1 long long getFactorial(int n) {
2     long long res = 1;
3     for (int i = 2; i <= n; i++)
4         res *= i;
5     return res;
6 }
```

Рекурсия – это когда Ипполит приходит к своей Наде Шевелевой, а там по телевизору показывают не "Соломенную шляпку", а "Иронию судьбы".

Но следуя из определения факториала можно получить рекурсивное решение:

```

1 long long getFactorialR(int n) {
2     if (n <= 1) // 
3         return 1;
4     else
5         return n * getFactorialR(n - 1);
6 }
7
8 int main() {
9     printf("%lld", getFactorialR(4))
10}

```

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**.

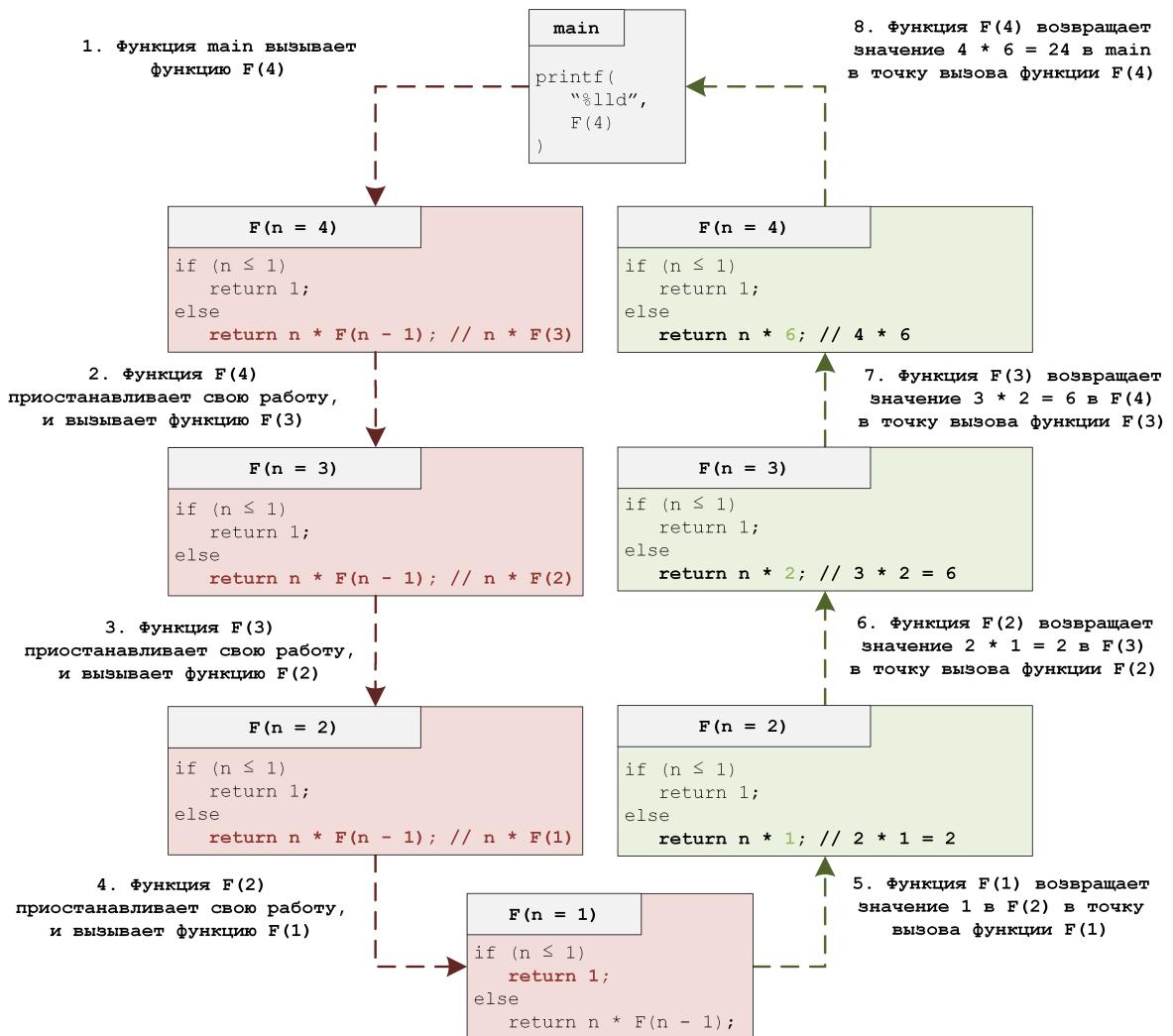


Рис. 7.29 – Процесс вычисления значения $4!$

На этапе рекурсивного спуска постепенно определялись выражения для вычисления $F(i)$, начиная с $F(4)$ до $F(1)$. На этапе рекурсивного подъёма последовательно вычисляются $F(2)$, $F(3)$ и $F(4)$ и получается значение 24.

7.9.2 Вывод двоичного представления числа

Вывод 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:

```

1 void printBin(int x) {
2     if (x == 0)
3         return;
4     else {
5         int digit = x & 1;
6         printBin(x >> 1);
7         printf("%d", digit);
8     }
9 }
```

```

1 void printBin(int x) {
2     if (x == 0)
3         return;
4     int digit = x & 1;
5     printBin(x >> 1);
6     printf("%d", digit);
7 }
```

Строки 2-4 относятся к рекурсивному спуску, строка 6 – к рекурсивному подъему. Отразим то, что происходит на схеме:

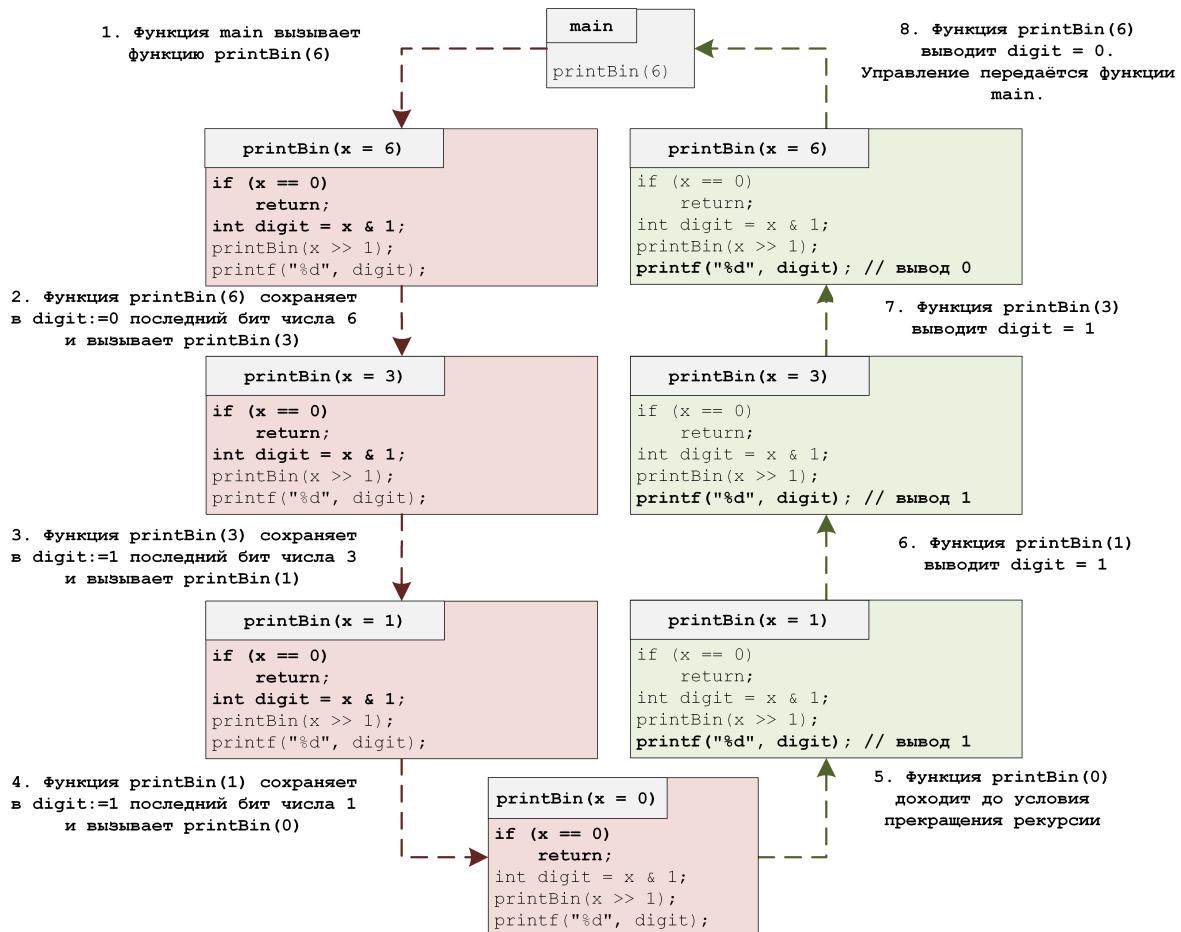


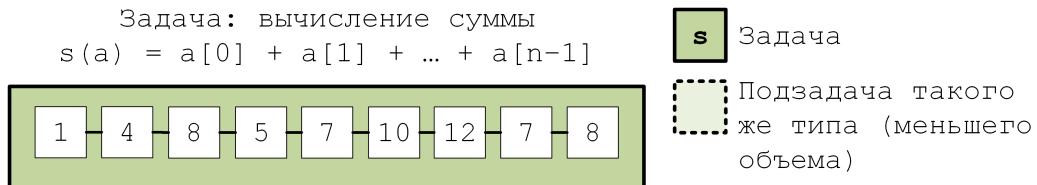
Рис. 7.30 – Процесс вывода двоичного представления числа 6

7.9.3 Вычисление суммы массива

Вычисление суммы массива

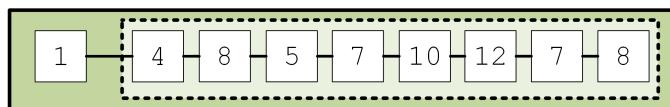
Дан массив a размера n . Необходимо вычислить сумму элементов массива.

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 ??).

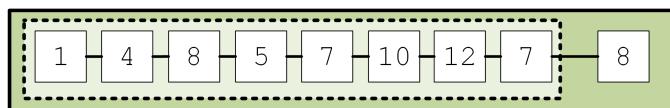


Способы разбиения:

$$s(a) = a[0] + s(a[1..n-1])$$



$$s(a) = s(a[0..n-2]) + a[n-1]$$



$$s(a) = s(a[0..n \text{ div } 2 - 1]) + s(a[n \text{ div } 2..n-1])$$

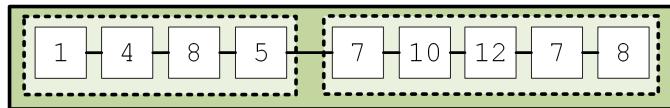


Рис. 7.31 – Способы вычисления суммы последовательности

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):

```

1 int getSum(const int *a, int left, int right) {
2     int n = right - left;
3     if (n == 0)
4         return 0;
5     else
6         return a[left] + getSum(a, left + 1, right);
7 }
8
9 int a[4] = {1, 2, 3, 4};
10 printf("%d", getSum(a, 0, 4));

```

С конца последовательности:

```

1 int getSum(const int *a, int left, int right) {
2     int n = right - left;
3     if (n == 0)
4         return 0;
5     else {
6         right--;
7         return getSum(a, left, right) + a[right];
8     }
9 }

```

Разбиение на две подзадачи:

```

1 int getSum(const int *a, int left, int right) {
2     int n = right - left;
3     if (n == 0)
4         return 0;
5     else if (n == 1)
6         return a[left];
7     else {
8         int middle = (left + right) / 2;
9         return getSum(a, left, middle) + getSum(a, middle, right);
10    }
11 }

```

7.9.4 Алгоритм однопроходного удаления

Алгоритм однопроходного удаления

Дан массив a размера n . Реализовать алгоритм однопроходного удаления.

Оставлен на самостоятельное изучение.

```

1 void deleteIf_(int *a, int (*deleteCondition)(int),
2                 int *size, int iRead, int iWrite) {
3     if (iRead == *size) {
4         *size = iWrite;
5         return;
6     } else if (!deleteCondition(a[iRead])) {
7         a[iWrite] = a[iRead];
8         iWrite++;
9     }
10    deleteIf_(a, deleteCondition, size, iRead + 1, iWrite);
11}
12
13 void deleteIf(int *a, int *n, int (*deleteCondition)(int)) {
14     deleteIf_(a, deleteCondition, n, 0, 0);
15 }
```

Обратите внимание, что все переменные, которые участвовали в итеративном алгоритме вычисления 'оказались' в качестве формальных параметров функции:

7.9.5 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.

```

1 static int isPalindrome_(const int *a, int left, int right) {
2     if (left < right)
3         return a[left] == a[right] && isPalindrome_(a, left+1, right-1);
4     else
5         return 1;
6 }
```

Основной недостаток данной функции - крайне неудобный интерфейс. Чтобы проверить, является ли массив a размера 10 палиндромом, необходимо выполнить вызов:

```
1 isPalindrome_(a, 0, 10);
```

Чтобы предоставить более удобный интерфейс, опишем функцию обёртку:

```

1 int isPalindrome(const int *a, int n) {
2     return isPalindrome_(a, 0, n - 1);
3 }
```

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.

7.9.6 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 отсортированного фрагмента. Это выполняется до тех пор, пока не будут отсортированы все элементы:

```

1 static void selectionSort(int *unsortedPart, int nUnsorted) {
2     if (nUnsorted == 1) // массив из одного элемента является упорядоченным и так
3         return;
4     else {
5         int minElementIndex = getMinIndex(unsortedPart, nUnsorted);
6         swap(&unsortedPart[0], &unsortedPart[minElementIndex]);
7         selectionSort(unsortedPart + 1, nUnsorted - 1);
8     }
9 }
```

Преимущество данного подхода – хорошая читаемость. Если посмотреть на строчки алгоритма, там описано ровно то же самое, что и текстом до листинга.

7.9.7 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:

```

1 static long long getPolyValue_(long long *a, int x, int i) {
2     if (i == 0)
3         return a[0];
4     else
5         return a[i] + x * getPolyValue_(a, x, i - 1);
6 }
7
8 long long getPolyValue(long long *a, int n, int x) {
9     return getPolyValue_(a, x, n);
10 }
```

7.9.8 Бинарный поиск

Бинарный поиск

Дан отсортированный массив a размера n . Требуется выполнить поиск элемента x и вернуть его позицию. Если элемент не найден — -1 .

```

1 static int binarySearch_(const int *a, int x, int right, int left) {
2     if (left > right)
3         return -1;
4     int middle = (left + right) / 2;
5     if (a[middle] == x)
6         return middle;
7     else if (a[middle] < x)
8         return binarySearch_(a, x, right, middle + 1);
9     else
10        return binarySearch_(a, x, middle - 1, left);
11 }
12
13 int binarySearch(const int *a, int n, int x) {
14     return binarySearch_(a, x, n - 1, 0);
15 }
```

Оставлю решения нескольких задач, которые помогут чуть лучше разобраться в рекурсии:

- С клавиатуры вводятся целые числа. Признак конца ввода — ноль. Найти сумму введенных четных чисел.

```

1 long long getSum() {
2     int x;
3     scanf("%d", &x);
4
5     if (x == 0)
6         return 0;
7     else if (x % 2 == 0)
8         return x + getSum();
9     else
10        return getSum();
11 }
```

- С клавиатуры вводится последовательность неотрицательных целых чисел. Вывести сначала все четные, а затем все нечетные числа в обратном порядке. Последовательность заканчивается нулем.

```

1 void getNumbers() {
2     int number;
3     scanf("%d", &number);
4
5     if (number == 0)
6         return;
7     else if (number % 2 == 0) {
8         printf("%d ", number);
9         getNumbers();
10    } else {
11        getNumbers();
12        printf("%d ", number);
13    }
14 }
```

3. Найти номер позиции последнего вхождения элемента со значением x .

```

1 static int linearSearchLast_(const int *a, const int n,
2                               const int x, const int i) {
3     if (i == -1)
4         return -1;
5     else if (a[i] == x)
6         return i;
7     else
8         return linearSearchLast_(a, n, x, i - 1);
9 }
10
11 int linearSearchLast(const int *a, const int n, const int x) {
12     return linearSearchLast_(a, n, x, n - 1);
13 }
```

4. Найти номер первого вхождения минимального значения в последовательность длины n .

```

1 static int getFirstMinPos_(const int *a, int n,
2                           int i, int minPos) {
3     if (i >= n)
4         return minPos;
5     else {
6         if (a[i] < a[minPos])
7             minPos = i
8         getFirstMinPos_(a, n, i + 1, minPos);
9     }
10
11
12 int getFirstMinPos(const int *a, int n) {
13     return getFirstMinPos_(a, n, 1, 0);
14 }
```

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.

```

1 void printDigits() {
2     int c = getchar();
3     if (c == '\n')
4         return;
5     else {
6         if (isdigit(c))
7             printf("%c", c);
8         printDigits();
9         if (isdigit(c))
10            printf("%c", c);
11     }
12 }
```

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.

```

1 static void printXY_(double x, double y, int n, int i) {
2     if (i == n + 1)
```

```

3     printf("%lf %lf", x, y);
4 else {
5     double lastX = x;
6     x += y / 2;
7     y += lastX / 3;
8     printXY_(x, y, n, i + 1);
9 }
10}
11
12 void printXY(int n) {
13     printXY_(1, 1, n, 2);
14 }

```

7. Дано натуральное число s . Определить, может ли число s быть суммой некоторого числа первых членов последовательности Фибоначчи. Последовательность Фибоначчи задается следующим образом:

$$f_1 = f_2 = 1 \quad f_i = f_{i-1} + f_{i-2} \text{ для } i > 2$$

```

1 static int isFibSum_(int s, int accSum,
2                     int lastFib2, int lastFib1) {
3     accSum += lastFib1;
4     if (accSum >= s)
5         return accSum == s;
6     else
7         return isFibSum_(s, accSum, lastFib1, lastFib1 + lastFib2);
8 }
9
10 int isFibSum(int s) {
11     int accSum = 0;
12     int lastFib1 = 1;
13     int lastFib2 = 0;
14     return isFibSum_(s, accSum, lastFib2, lastFib1);
15 }

```

8. Дан первый член арифметической прогрессии и ее разность. Вычислить n -й член прогрессии без использования операции умножения.

```

1 static int multiplyRecursion(int i, int n, int d) {
2     if (i == n)
3         return d;
4     else
5         return d + multiplyRecursion(i + 1, n, d);
6 }
7
8 int getLastElementOfAP(int a0, int n, int d) {
9     return a0 + multiplyRecursion(1, n, d);
10 }

```

9. Определить количество букв в тексте, вводимом с клавиатуры. Текст заканчивается символом переноса на новую строку `\n`.

```

1 int countLetters() {
2     int c = getchar();
3     if (c == '\n')
4         return 0;
5     else
6         return ('a' <= c && c <= 'z' || 'A' <= c && c <= 'Z') +
7             countLetters();
8 }

```

10. Найти наибольший общий делитель натуральных чисел n и m .

```

1 int gcd(int a, int b) {
2     if (b == 0)
3         return a;
4     else
5         return gcd(b, a % b);
6 }
```

Резюме

- Функция — это самодостаточная единица кода программы, спроектированная для выполнения отдельной задачи.
- В каждой программе на С/С++ 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 С 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 чем заключается побочный эффект функции?
9. Как изменить аргументы, передаваемые в функцию?

Глава 8

Работа в CLion

*IDE*¹ – 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.

8.1 Статический анализ кода

Clion выполняет анализ² 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:

```
1 #include <stdio.h>
2
3 int main() {
4     SetConsoleOutputCP(CP_UTF8);
5     char s[255];
6     printf("Введите ваше имя\n");
7     scanf("%s", s, s);
8     printf("Привет %s", s)
```

¹Глава 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.

²Разделяют 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.

```

9     return 0;
10 }
```

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 8.1):

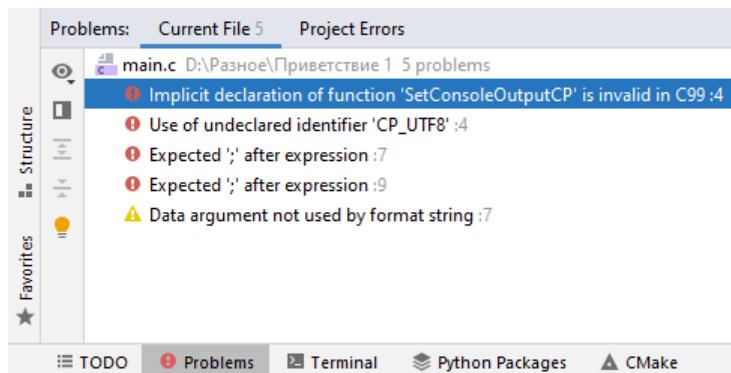


Рис. 8.1 – Ошибки и предупреждения среды

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 вспомнил, подсунув *IDE* следующий код (представляющий собой предыдущий с исправлением ошибок и удалением открывающей фигурной скобки блока `main`):

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 int main()
5     SetConsoleOutputCP(CP_UTF8);
6     char s[255];
7     printf("Введите ваше имя\n");
8     scanf("%s", s);
9     printf("Привет %s", s);
10    return 0;
11 }
```

Здесь среда не поправит ошибку Васи, поставив фигурную скобку, из-за чего впоследствии возникнет масса 'мнимых' ошибок:

На рисунке 8.2 первая ошибка с открывающей скобкой отмечена верно, однако все следующие малость не корректны. Если Вася начнет исправлять программу с них, то не уверен, что он своего добьется. Поэтому можно вывести рекомендацию начинать исправление с самой ранней ошибки.

8.2 Отладчик

Представим ситуацию: уже вы написали программу, она успешно исполняется, но выдает неверный ответ, а анализатор молчит. Как решить эту проблему?

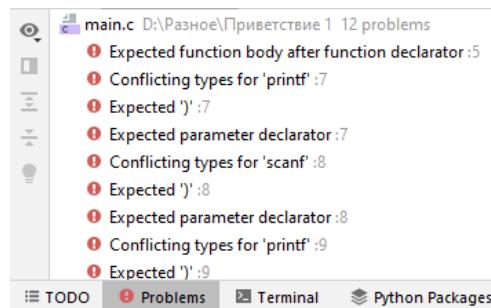


Рис. 8.2 – Мнимые ошибки

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 – отладчика³⁴. Последний вариант (после овладения им) самый продуктивный.

Допустим, мы писали программу для вывода считанной строки (в которой записано число) в обратном порядке:

```

1 #include <stdio.h>
2 #include <windows.h>
3 #include <string.h>
4
5 int main() {
6     SetConsoleOutputCP(CP_UTF8);
7
8     char s[255];
9     gets(s);
10
11    printf("Ответ: \n");
12    for (size_t i = strlen(s); i >= 0; --i) {
13        printf("%c", s[i]);
14    }
15    printf("\n");
16
17    return 0;
18 }
```

Запустили ее, и увидели что, что-то пошло не так (рисунок 8.3):

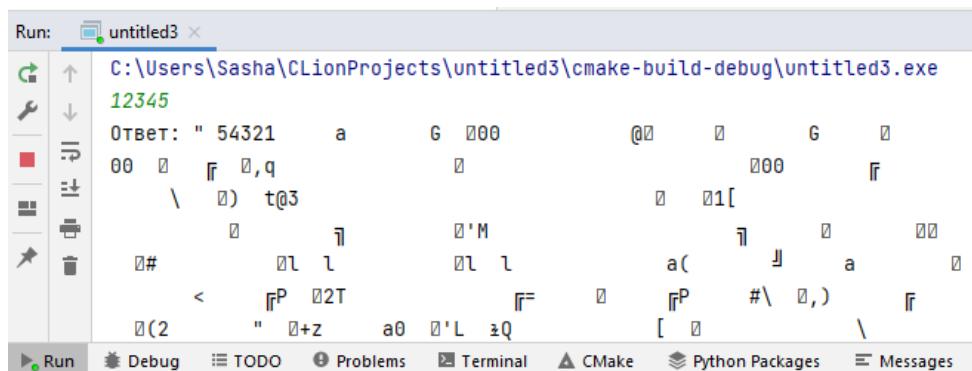
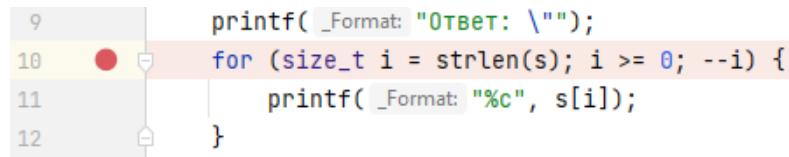


Рис. 8.3 – Бесконечный цикл вывода символов

³Отладчик – программа позволяющая программисту контролировать выполнение кода. В современных средах она обычно встроена.

⁴Вариант отвлечься в пособии не рассматривается.

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 8.4.



```

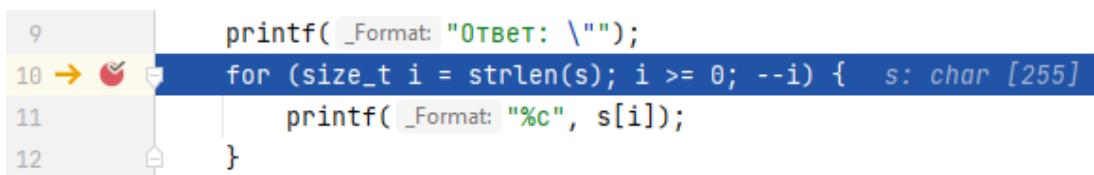
9     printf(_Format: "Ответ: \n");
10    ●   for (size_t i = strlen(s); i >= 0; --i) {
11        printf(_Format: "%c", s[i]);
12    }

```

Рис. 8.4 – Точка останова на 10 строке

Говорят, что в помеченной строке находится **точка останова**⁵.

Запустим программу в режиме отладки, нажав на зеленого жучка (Либо комбинацию *Shift + F9*)⁶. *IDE* покажет остановку следующим образом.



```

9     printf(_Format: "Ответ: \n");
10   ➔ ●   for (size_t i = strlen(s); i >= 0; --i) { s: char [255]
11        printf(_Format: "%c", s[i]);
12    }

```

Рис. 8.5 – Остановка исполнения программы на 10 строке

Порою можно ввестись в заблуждение, что строка из рисунка 8.5, помеченная желтой стрелочкой, уже исполнена. Это не совсем так, исполнен весь код до нее.

Помимо появления стрелок, на интерфейсе *IDE* появилось окно отладчика:

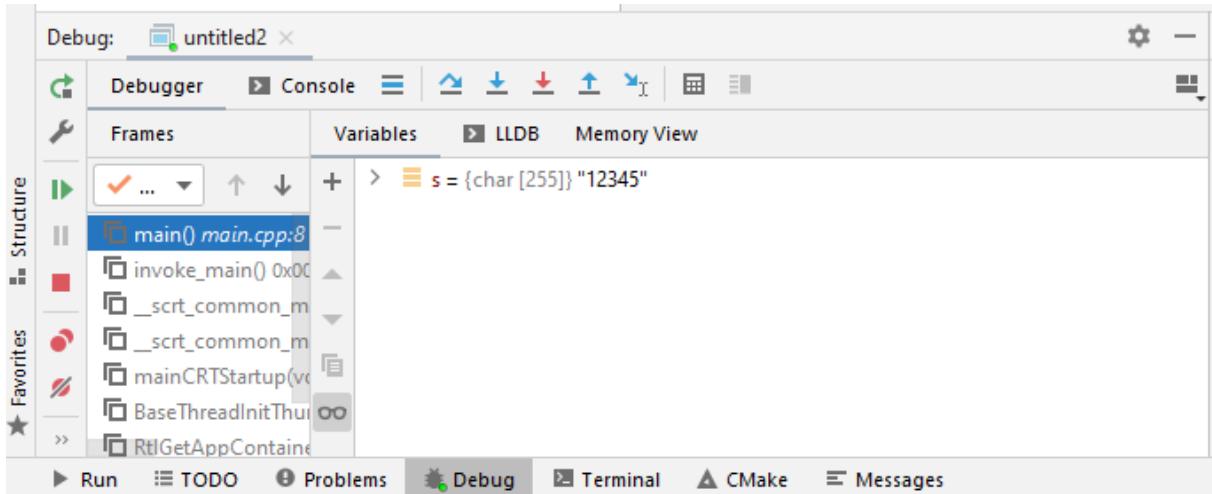


Рис. 8.6 – Встроенный отладчик

На рисунке 8.6 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 помеченной желтой стрелкой строки. Продолжим искать ошибку, войдя в цикл при помощи крайней левой

⁵Эти 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.

⁶Стоит отметить, что элементы интерфейса могут отличаться. Однако комбинации клавиш остаются схожими. Поэтому сфокусируйтесь на комбинациях клавиш для того или иного действия.

стрелочки панели (Вместо этого нажать *F8* на клавиатуре). Смысл данной стрелки – исполнение строки. С другими стрелками мы познакомимся чуть-чуть позже.

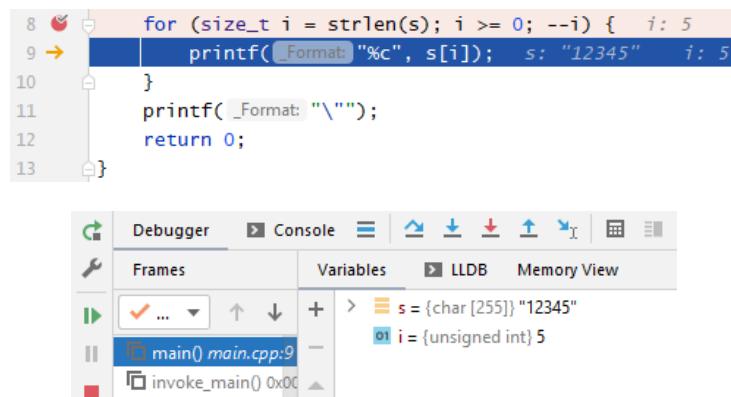


Рис. 8.7 – Изменения интерфейса после шага отладки

Заметим, что на рис. 8.7 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 8.8):

```

s = {char [255]} "12345"
i = {unsigned int} 4294967295

```

Рис. 8.8 – 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 8.9:

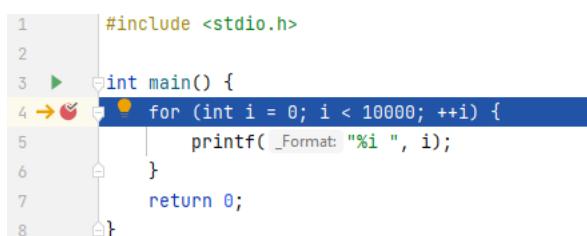


Рис. 8.9 – Вход в цикл

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. 8.10:

```

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. 8.10 – Остановка на выходе из цикла

Обратим внимание на то, что на данный момент времени в консоли выведены не все числа рис. 8.11:

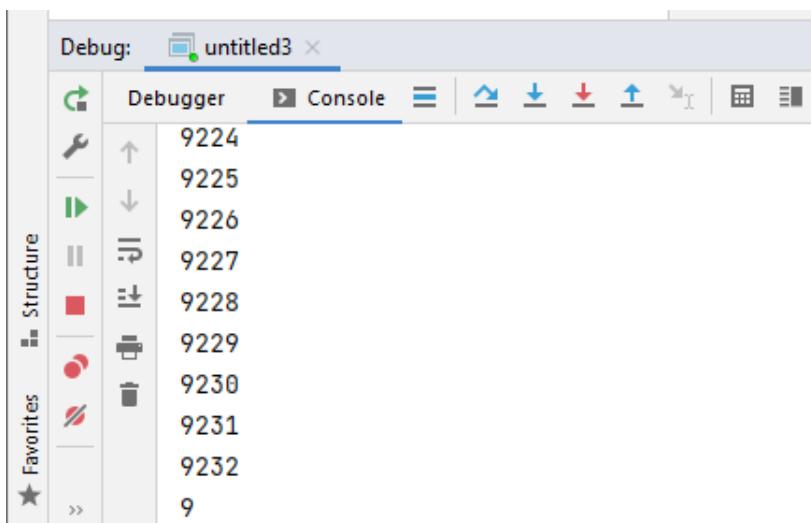


Рис. 8.11 – Не полный вывод

Эта странность связана с буферизацией вывода в С и особенностями встроенной консоли *Clion*: перед тем как попасть на консоль, символы скапливаются в буфере⁷. 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. 8.12.

⁷Буфер – это область памяти, используемая для временного хранения данных при вводе или выводе

```

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. 8.12 – 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 (*F11*). 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*). Смысль 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 появятся 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. 8.13:

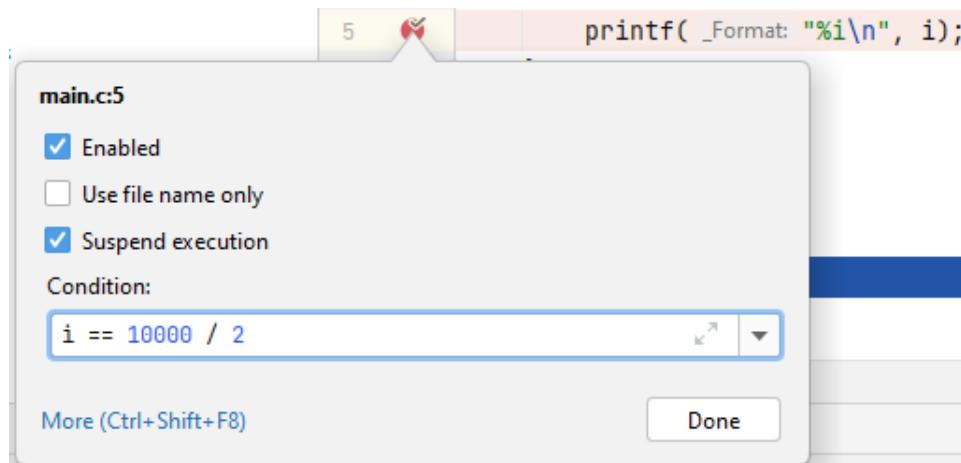


Рис. 8.13 – Точка останова с условием

Запустим исполнение и увидим что-то вроде рис. 8.14:

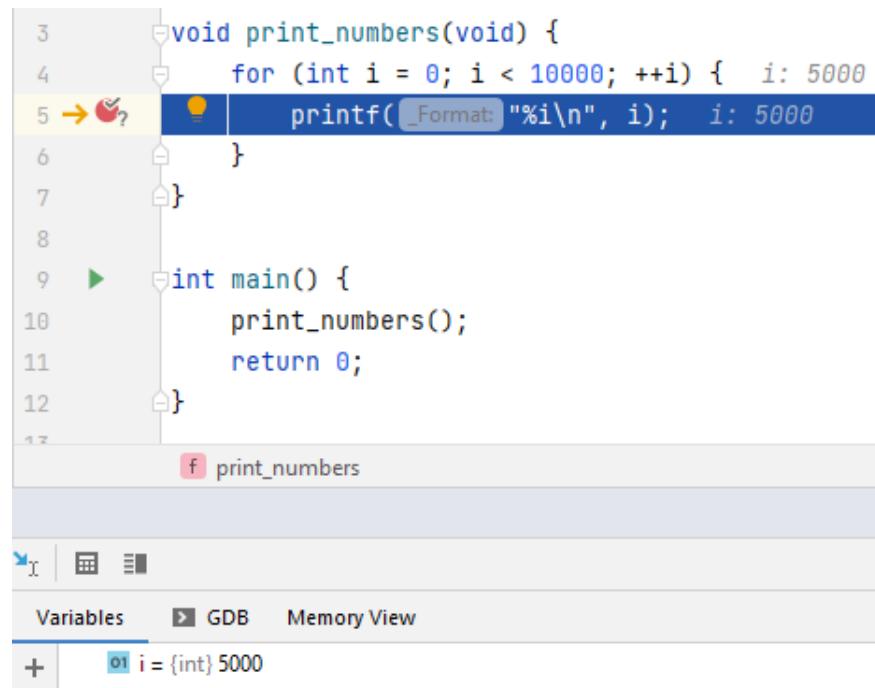


Рис. 8.14 – Остановка программы при заданном значении счетчика

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:

```

1 #include <stdio.h>
2 #include <windows.h>
3
4 long long body_fib(long long i1, long long i2, long long n) {
5     if (n <= 0)
6         return(i2);
7     else
8         return(body_fib(i2, i1 + i2, n - 1));
9 }
10
11 long long fibonacci (long long n) {
12     return(body_fib(1, 1, n - 2));
13 }
14
15 int main() {
16     SetConsoleOutputCP(CP_UTF8);
17
18     long long n;
19     printf("Введите номер числа Фибоначчи\n");
20     scanf("%lli", &n);
21
22     printf("%lli", fibonacci(n));
23
24     return 0;
25 }
```

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 оконке есть стек вызовов рис. 8.15, 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.

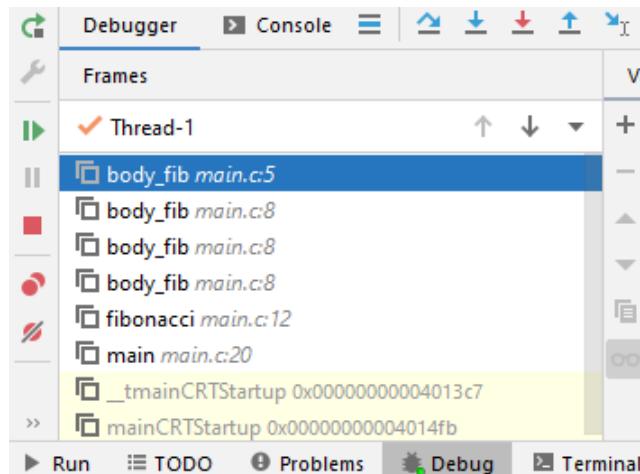


Рис. 8.15 – Стек вызовов

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 8.16

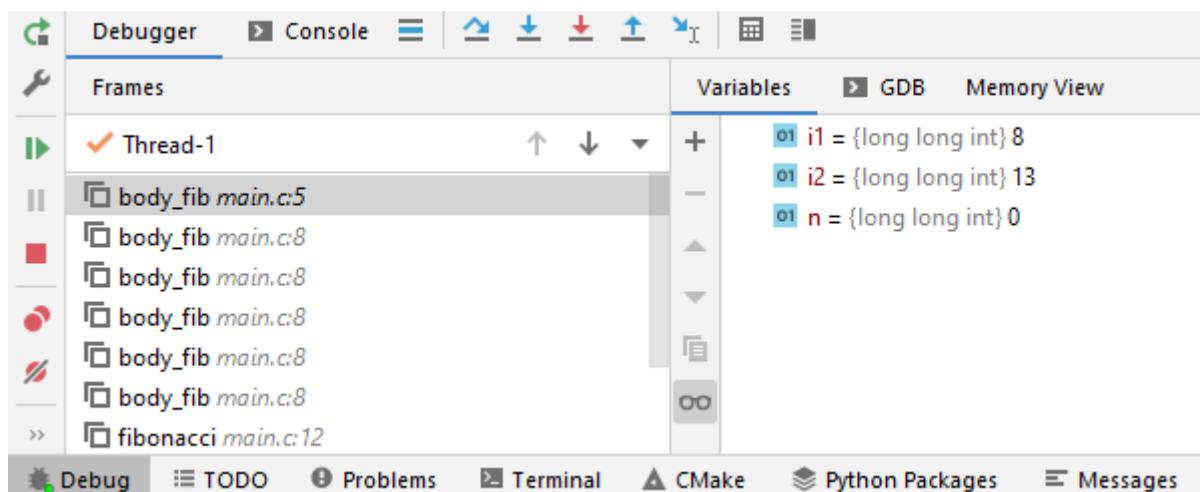


Рис. 8.16 – Окно отладчика

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.



8.3 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. 8.17:

```

1 > int main() {
2     printf("Hello world");
3     return 0;
4 }
```

Рис. 8.17 – Некорректная программа

Наведем курсор на красную функцию и нажмем на правую кнопку мыши, появится окно, представленное на рис. 8.18:

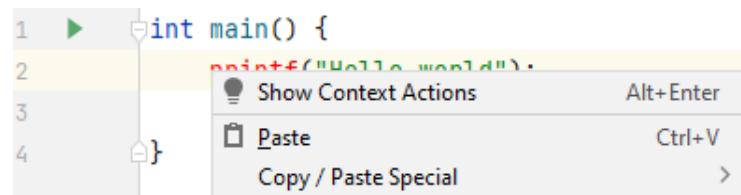


Рис. 8.18 – Контекстное меню

Выберем **Show Context Actions**, и увидим предложения среды по исправлению ошибки. Остановимся на варианте **Import symbol 'printf'** и импортируем функцию `printf` из

`stdio.h` **f** `function 'printf'` **<stdio.h>**. Программа за нас напечатает знакомую директиву.

Если у нас имеется следующая программа:

```

1 #include <stdio.h>
2
3 int many_letters_sum(int a, int b) {
4     return a + b;
5 }
6
7 int main() {
8     int many_letters_1;
9     int many_letters_2;
10
11     return 0;
12 }
```

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. 8.19, нажмем клавишу *Tab*.

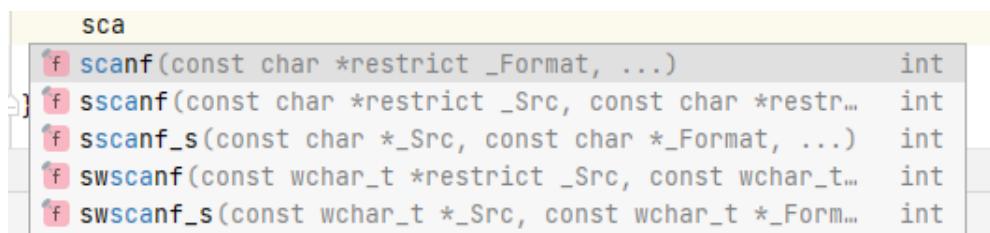


Рис. 8.19 – Применение автодополнения

Учитывая, что в имени функции 5 символов. Смысла в этом видится мало. Теперь воспользуемся автодополнением для обращения к длинным переменным рис. 8.20. Когда увидим 2-ую переменную нажмем стрелку вниз и *Tab* или *Enter*:

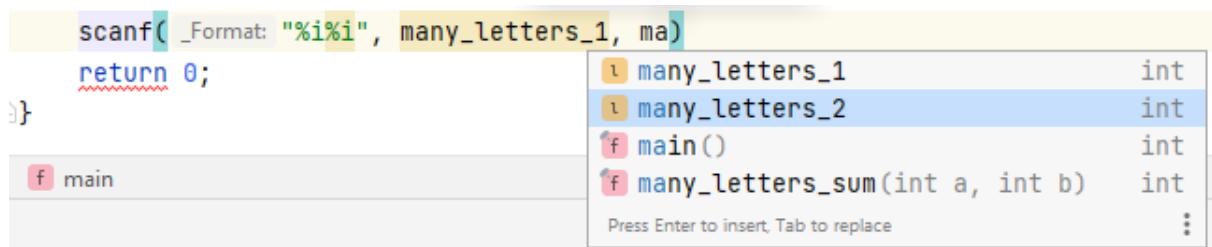


Рис. 8.20 – Применение автодополнения

Далее напишем аналогично `printf`, вызвав длинную функцию рис. 8.21:

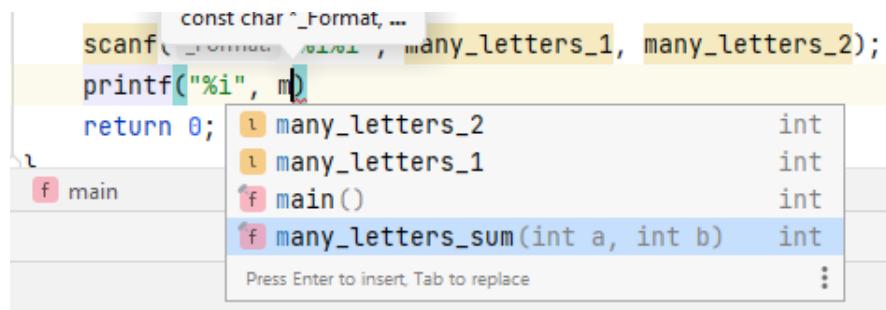


Рис. 8.21 – Применение автодополнения

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. 8.22 – Полученные строки

8.4 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

$$f(x, y) = 2 * x + y + |2 * x + y|$$

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int x, y;
6     scanf("%i%i", &x, &y);
7     printf("%i", 2 * x + y + abs(2 * x + y));
8     return 0;
9 }
```

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. 8.23:

```
#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. 8.23 – 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 быстрее⁸. Вот и сейчас у него получилось:

```

1 #include <stdio.h>
2
3 int main () {
4     int a[2500];
5     int n;
6     scanf ("%i", &n);
7     for (int i = 0; i < n; ++i)
8         scanf ("%i", &a[i]);
9     int imax, t;
10    for (int i = 0; i < n - 1; ++i) {
11        imax = i;
12        for (int j = i + 1; j < n; ++j)
13            if (a[j] > a[imax])
14                imax = j;
15        t = a[imax];
16        a[imax] = a[i];
17        a[i] = t;
18    }
19    for (int i = 0; i < n; ++i)
20        printf ("%i ", a[i]);
21 }
22

```

Однако преподаватель принимает лабораторные, только если выделены функции. Поэтому Петя выделяет цикл ввода массива (строки 7-8) и после нажатия комбинации *Ctrl+Alt+M* появляется окошко, в которой он указывает имя функции `read_arr` и ее расположение на *above* рис. 8.24:

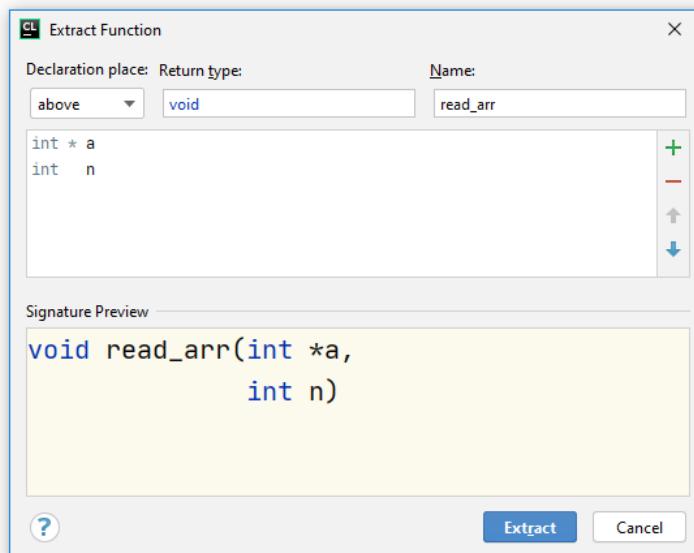


Рис. 8.24 – Окно выделение функции

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:

```

1 #include <stdio.h>

```

⁸Выделять функции стоит по ходу написания проекта, но только не в самом конце.

```

2
3 void read_arr(int *a, int n) {
4     for (int i = 0; i < n; ++i)
5         scanf("%i", &a[i]);
6 }
7
8 void sort(int *a, int n) {
9     int imax, t;
10    for (int i = 0; i < n - 1; ++i) {
11        imax = i;
12        for (int j = i + 1; j < n; ++j)
13            if (a[j] > a[imax])
14                imax = j;
15        t = a[imax];
16        a[imax] = a[i];
17        a[i] = t;
18    }
19 }
20
21 void write_arr(const int *a, int n) {
22     for (int i = 0; i < n; ++i)
23         printf("%i ", a[i]);
24 }
25
26 int main () {
27     int a[2500];
28
29     int n;
30     scanf("%i", &n);
31
32     read_arr(a, n);
33     sort(a, n);
34     write_arr(a, n);
35 }
36

```

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.

8.5 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. 8.25: переименование программный объектов в проекте.

```
#include <stdio.h>
int main() {
    int zxsd;
    for (int zxsd = 0; zxsd < 100; ++zxsd) {
        printf(_Format: "%i\n", zxsd);
        zxsd += 2;
    }
    return 0;
}
```

Рис. 8.25 – 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. 8.26: вызов контекстного меню без мыши.

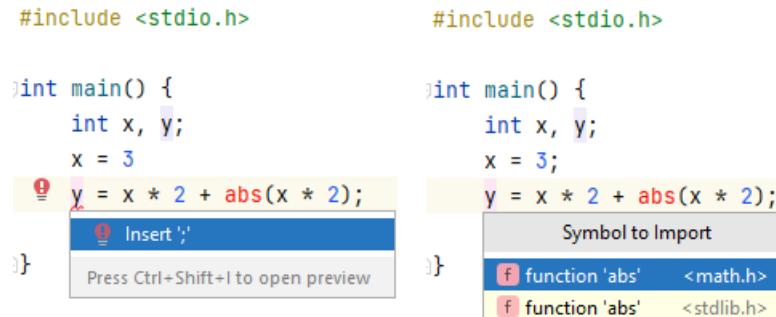
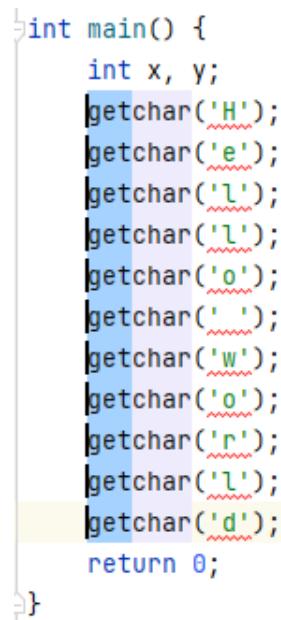


Рис. 8.26 – Контекстное меню

10. *Ctrl + Shift + <Стрелка вверх или вниз>* перемещение строки.

11. *Alt + <Выделение колонки текста>* создание мульти-курсора.

Пример применения представлен на рис. 8.27: 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. 8.27 – 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.

8.6 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 < ; i2++) {
}

```

Рис. 8.28 – Шаблон *for*

Вы воспользовались вставкой шаблона, называемого *Live Template* в *Clion*⁹.

⁹ В *Microsoft Visual Studio Community* шаблоны кода имеют название *Snippets*. Реализованы они там посложнее, чем в *Clion*

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 окошке рис. 8.29 нажмем на "+" и *Live Template*.

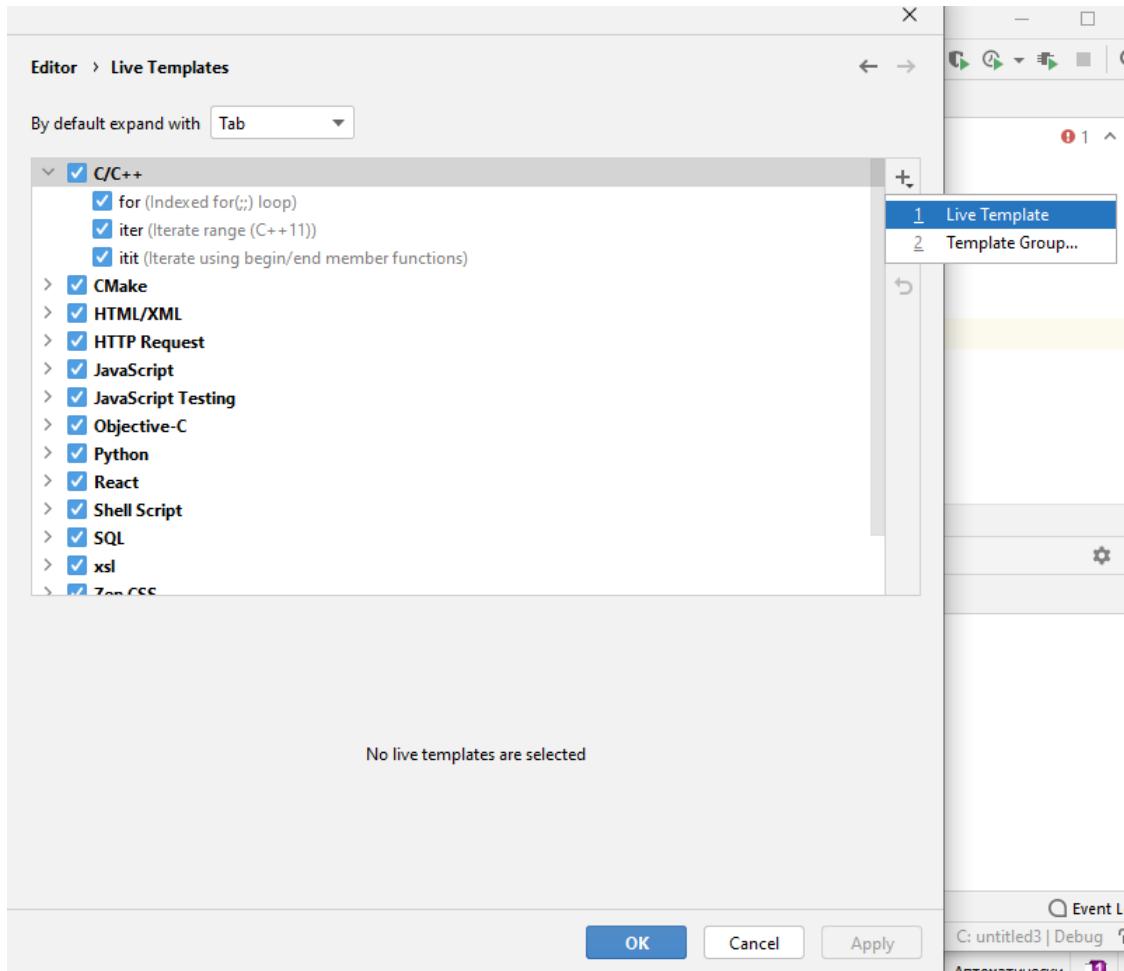


Рис. 8.29 – Список шаблонов

Введем данные в появившееся окошко рис. 8.30:

- *Abbreviation* – символы, нажав после которых *Tab* мы можем вызвать шаблон;
- *Description* – описание (подсказка, которая будет выведена в процессе просмотра *Abbreviation*);
- *Template text* – непосредственно шаблон.

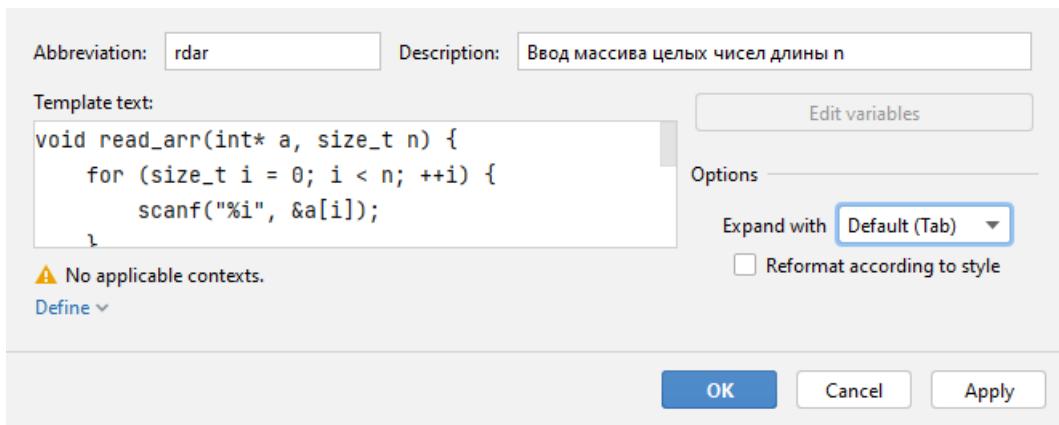


Рис. 8.30 – Подготовка шаблона

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 *rdar* и нажмем *Tab* или *Enter*. Должна произойти вот такая вставка рис. 8.31:

```
#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. 8.31 – 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*):

```
1 void read_arr($TYPE** a, size_t n) {
2     for (size_t i = 0; i < n; ++i) {
3         scanf("%i", &a[i]);
4     }
5 }
6
7 $END$
```

Теперь после вставки будет предложено ввести тип, а после ввода и нажатия *Tab* курсор будет перемещен в точку *END*.

8.7 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:

```

1 #include <stdio.h>
2
3 int main () {
4     int n;
5     scanf("%i", &n);
6     printf("%i\n", n + 1);
7 }
8

```

В такой:

```

1 #include <stdio.h>
2
3 int main () {
4     int t;
5     scanf("%i", &t);
6     while (t--) {
7         int n;
8         scanf("%i", &n);
9         printf("%i\n", n + 1);
10    }
11 }

```

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. 8.32 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. 8.32

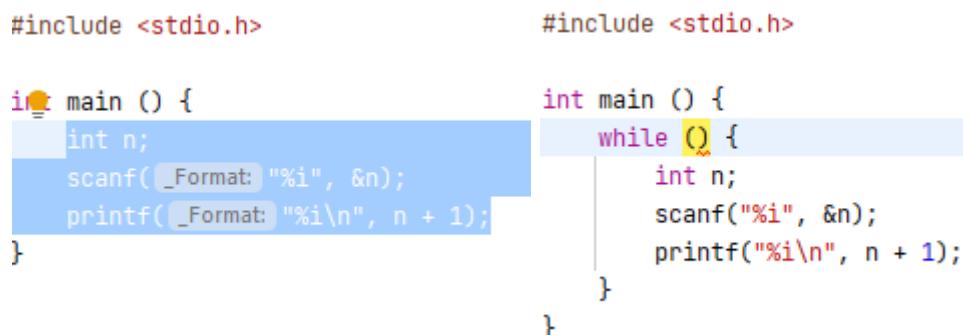


Рис. 8.32 – Применение шаблона-окружения

Далее, когда мы знаем как шаблоны-обертки работают, попробуем повторить приём Пети. Создадим *Live Template* следующего вида рис. 8.33:

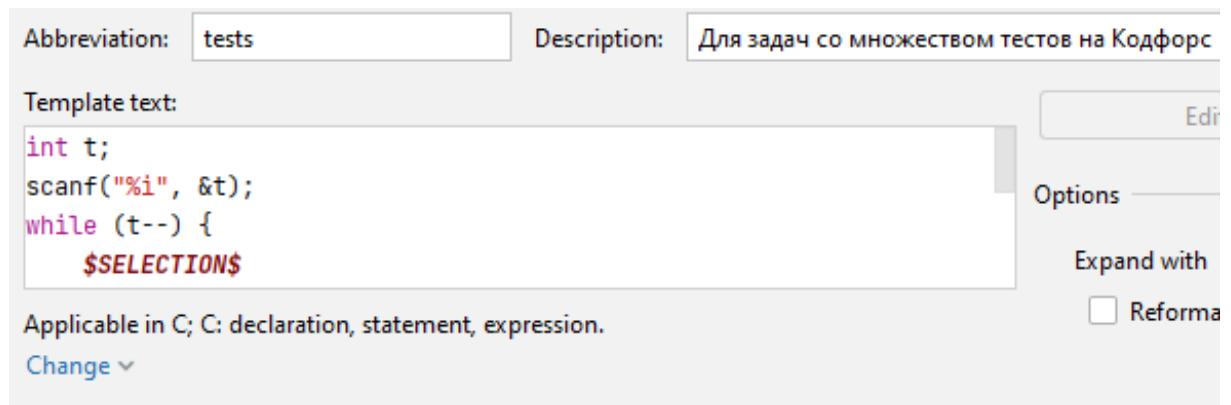


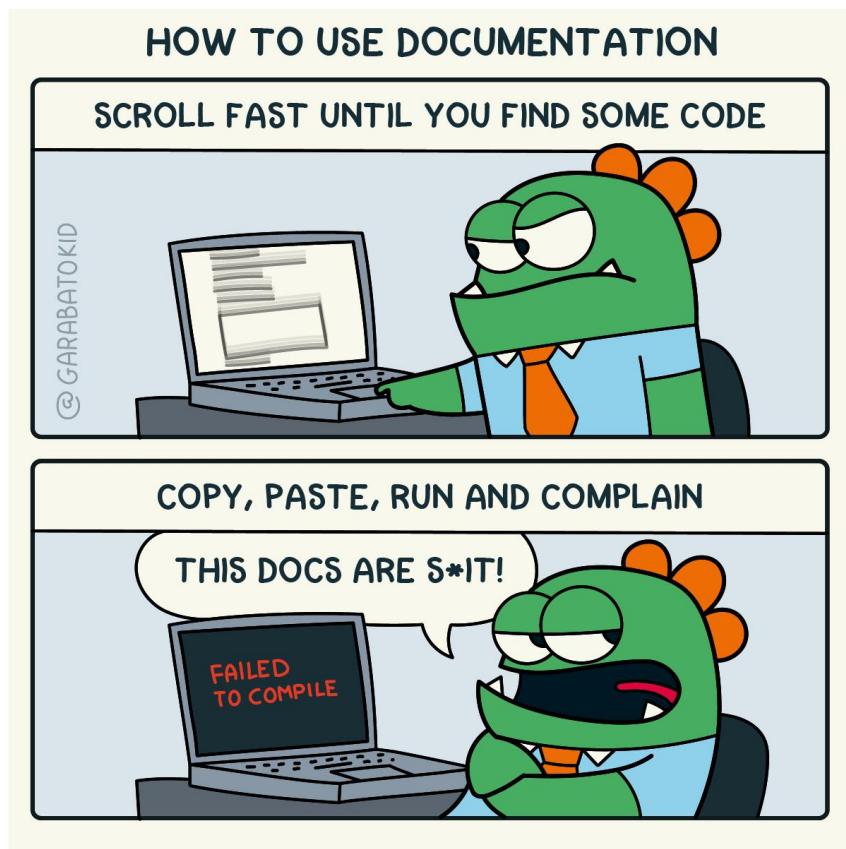
Рис. 8.33 – Применение шаблона-окружения

На рис. 8.33 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`.

8.8 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 интернет¹⁰.

¹⁰Сайт *Clion* на английском. Будьте готовы к *google translate* и к переводу.

Глава 9

Алгоритмы обработки одномерных массивов

9.1 Алгоритмы, не зависящие от упорядоченности

9.1.1 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

```
1 int main() {
2     int a[5];
3     //...
4 }
```

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. 9.1 – Объявление массива и состояние памяти

Ввести элементы массива можно было и так:

```
1 int main() {
2     int a[5];
3     scanf("%d %d %d %d %d", &a[0], &a[1], &a[2], &a[3], &a[4]);
4     //...
5 }
```

но довольно очевидно, что если их количество будет велико, следует воспользоваться циклом:

```

1 int main() {
2     int a[5];
3     for (int i = 0; i < 5; i++)
4         scanf("%d", &a[i]);
5     // ...
6 }
```

Можно выделить функцию, в которой бы происходил ввод 5 элементов:

```

1 void inputArray5(int *a) {
2     for (int i = 0; i < 5; i++)
3         scanf("%d", &a[i]);
4 }
5
6 int main() {
7     int a[5];
8     inputArray5(a);
9     // ...
10 }
```

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:

```

1 void inputArray(int *a, const int n) {
2     for (int i = 0; i < n; i++)
3         scanf("%d", &a[i]);
4 }
```

Тогда для ввода двух массивов подошел бы код:

```

1 int main() {
2     int a[5];
3     inputArray(a, 5);
4
5     int b[20];
6     inputArray(b, 20);
7     // ...
8 }
```

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`¹:

1. Заголовок: `void inputArray(int *a, const size_t n)`.
2. Назначение: ввод массива по адресу `a` размера `n`.

¹ В первом блоке 'терминатор' указываются входные параметры. `a` – указатель на массив не является входным параметром.

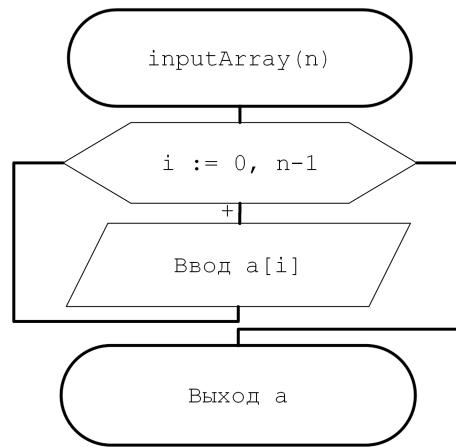


Рис. 9.2 – Блок схема функции ввода массива

Код функции:

```
1 void inputArray( int *a, const size_t n ) {  
2     for ( size_t i = 0; i < n; i++ )  
3         scanf( "%d", &a[i] );  
4 }
```

Она может быть использована для ввода массива и подмассивов:

```
1 int main() {
2     int a[10];
3     inputArray(a, 10); // ввод элементов с 0 по 10
4
5     int b[20];
6     inputArray(b, 20); // ввод элементов с 0 по 20
7
8     inputArray(&a[2], 5) // ввод элементов с индексами с 2 по 7
9     inputArray(a + 2, 5) // ввод элементов с индексами с 2 по 7
10
11    return 0;
12 }
```

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:

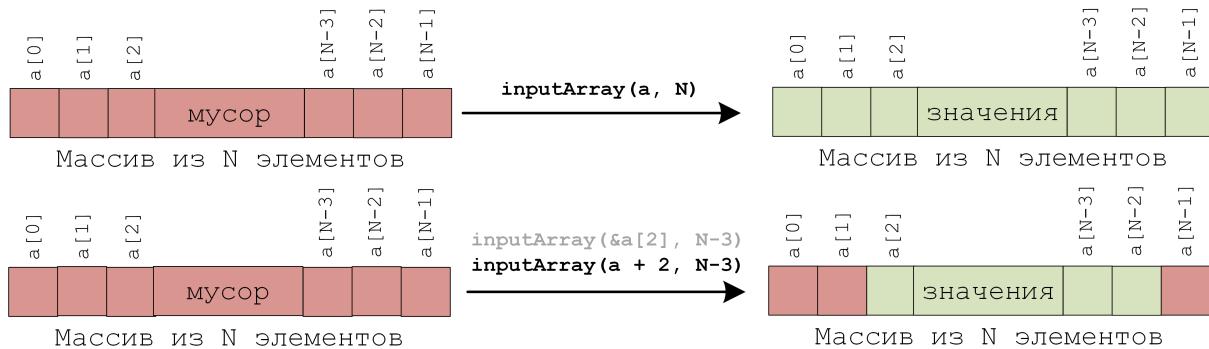


Рис. 9.3 – Вызовы функции *inputArray*

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):

```

1 void inputArrayRange(int *a, const size_t startIndex, const size_t
2   endIndex) {
3   size_t nElements = endIndex - startIndex + 1;
4   inputArr(a + startIndex, nElements);
5 }
```

9.1.2 Вывод массива

Вывод массива

Необходимо вывести на экран массив *a* размера *n*.

Сложность по времени: $\mathbf{O(n)}$.

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*²:

1. Заголовок: `void outputArray(const int *a, const size_t n);`
2. Назначение: вывод массива по адресу *a* размера *n*.

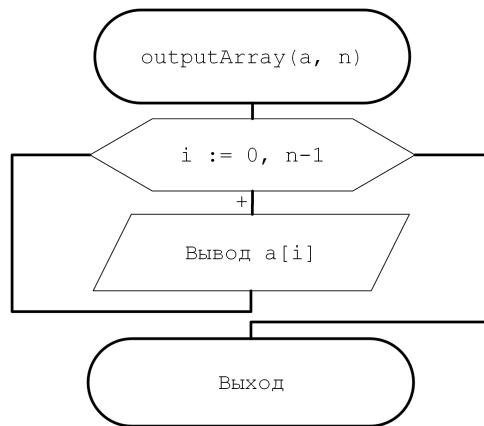


Рис. 9.4 – Блок схема функции вывода массива

Дополнительно выполним перенос на следующую строку³:

```

1 void outputArray(const int *a, const size_t n) {
2     for (size_t i = 0; i < n; i++)
3         printf("%d ", a[i]);
4     printf("\n");
5 }
```

²Обратите внимание на оформление последнего блока блок-схемы для функции без выходных параметров.

³Момент, улучшающий чтение выводимых данных на блок-схеме отображаться не должен.

9.1.3 Поиск позиции элемента, удовлетворяющему условию

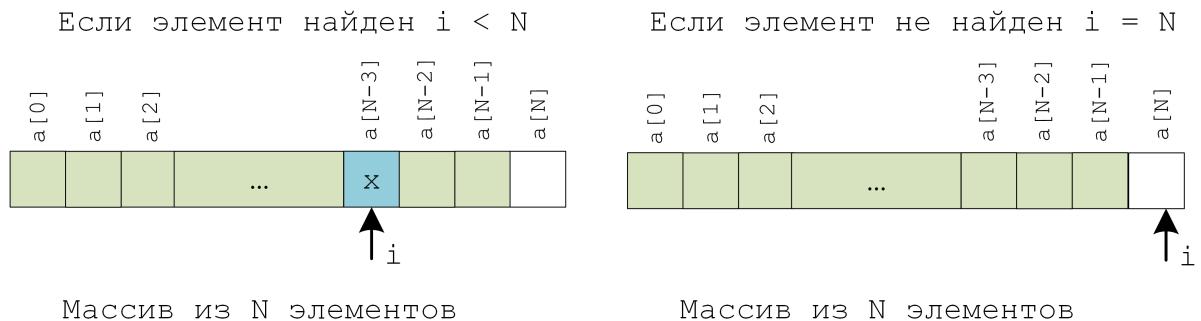
Поиск позиции элемента, удовлетворяющему условию

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 9.5, 9.6). 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

```
1 while (a[i] >= 0 && i < n)
```

вместо

```
1 while (i < n && a[i] >= 0)
```

является ошибочным.

Более того, 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.

```
1 int getFirstNegativeIndex(const int *a, const size_t n) {
2     int i = 0;
3     while (i < n && a[i] >= 0)
4         i++;
5     return i == n ? -1 : i;
6 }
```

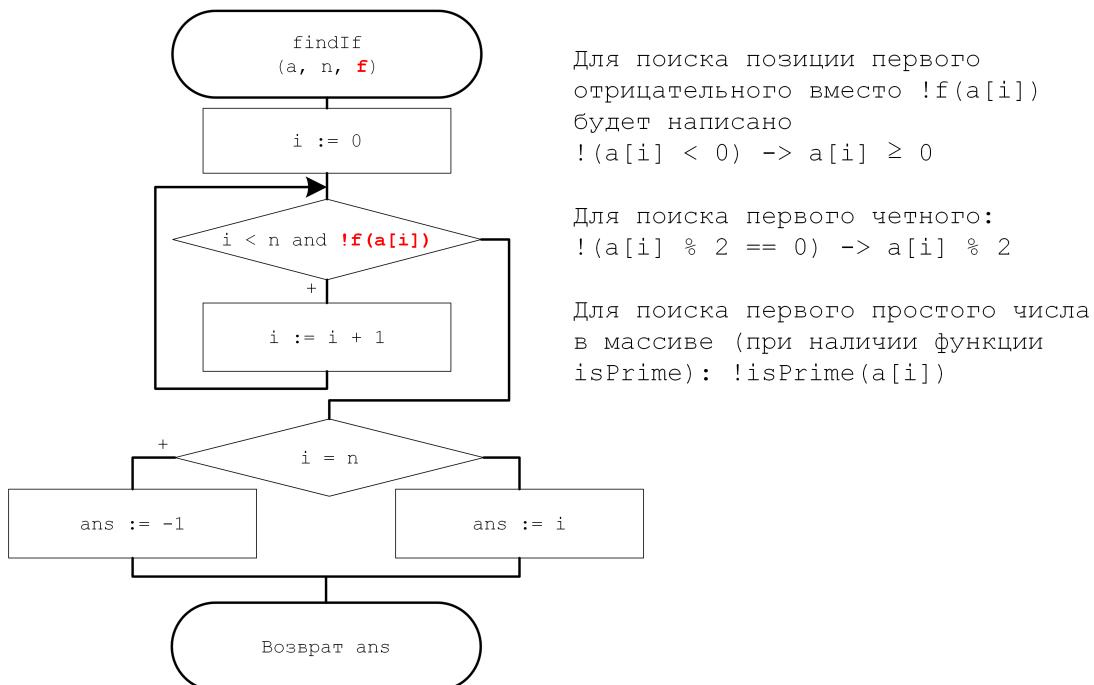


Рис. 9.5 – Первый способ решения задачи в общем случае. Вместо $!f(a[i])$ будет подставлена требуемая операция сравнения

Во втором способе используется две инструкции возврата:

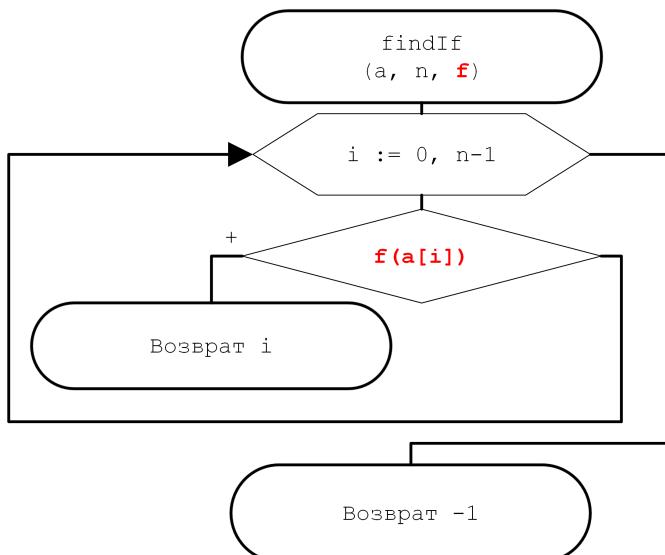


Рис. 9.6 – Второй способ решения задачи. На блок-схеме он выглядит нетривиально, а вот код считается легче

```

1 int getFirstNegativeIndex(const int *a, const size_t n) {
2     for (size_t i = 0; i < n; i++)
3         if (a[i] < 0)
4             return i;
5     return -1;
6 }
```

Спецификация функции `getFirstNegativeIndex`:

1. Заголовок: `int getFirstNegativeIndex(const int *a, const size_t n);`
2. Назначение: возвращает⁴ индекс первого вхождения отрицательного значения в массиве `a` размера `n`, если элемент найден, иначе – `-1`.

9.1.4 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: $\mathbf{O(n)}^a$.

^aБез 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`.

```

1 int getSumOfDigits(long long x) {
2     int sum = 0;
3     while (x > 0) {
4         sum += x % 10;
5         x /= 10;
6     }
7     return sum;
8 }
9
10 int countSumOfDigitsX(const int *a, const size_t n,
11                         const int digitsSum) {
12     int count = 0;
13     for (size_t i = 0; i < n; ++i)
14         count += getSumOfDigits(a[i]) == digitsSum;
15     return count;
16 }
```

⁴Если основной задачей функции является возврат значения, рекомендуется начинать назначение со слова 'возвращает'.

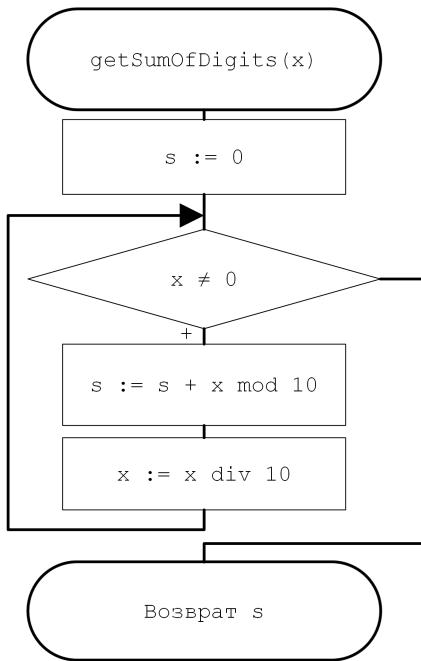


Рис. 9.7 – Блок схема функции подсчёта суммы цифр

Спецификация функции `countSumOfDigitsX`:

1. Заголовок:

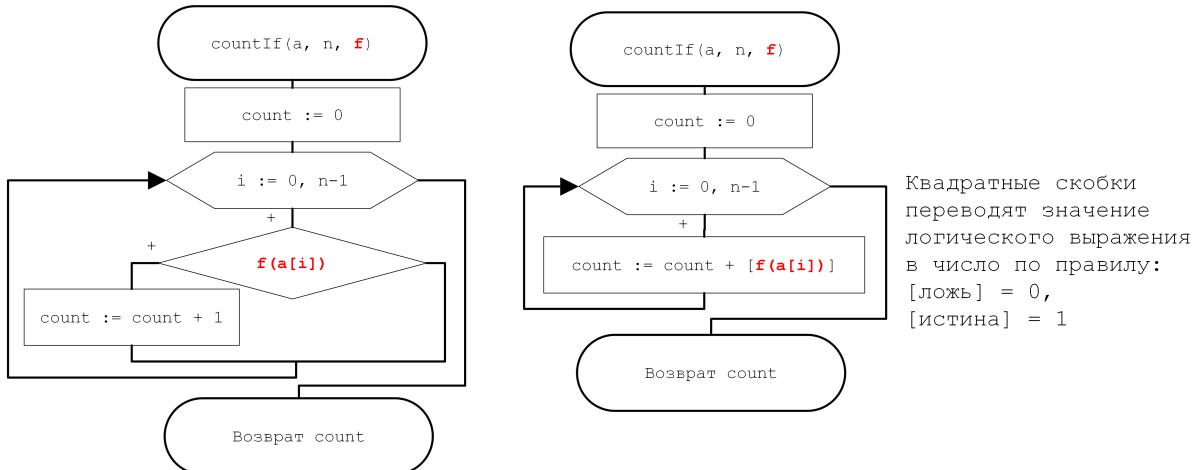
`int countSumOfDigitsX(const int *a, const int n, const size_t digitsSum);`2. Назначение: возвращает количество элементов массива a размера n сумма цифр которых равна `digitsSum`.

Рис. 9.8 – Блок схема функции подсчёта количества элементов, удовлетворяющих условию

9.1.5 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: $\mathbf{O(n)}$.

В решении данной задачи надо подумать, каким образом действовать. Если встретится 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.

```

1 int getMaxEvenChainLen(const int *a, const size_t n) {
2     int maxLen = 0;
3     int currentLen = 0;
4     for (int i = 0; i < n; i++) {
5         if (a[i] % 2 == 0)
6             currentLen++;
7         else {
8             maxLen = max(maxLen, currentLen);
9             current = 0;
10        }
11    }
12    maxLen = max(maxLen, currentLen);
13    return maxLen;
14 }
```

1. Заголовок: `int getMaxEvenChainLen(const int *a, const size_t n);`
2. Назначение: возвращает максимальное количество подряд идущих четных элементов массива `a` размера `n`.

9.1.6 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: $\mathbf{O(n)}$.

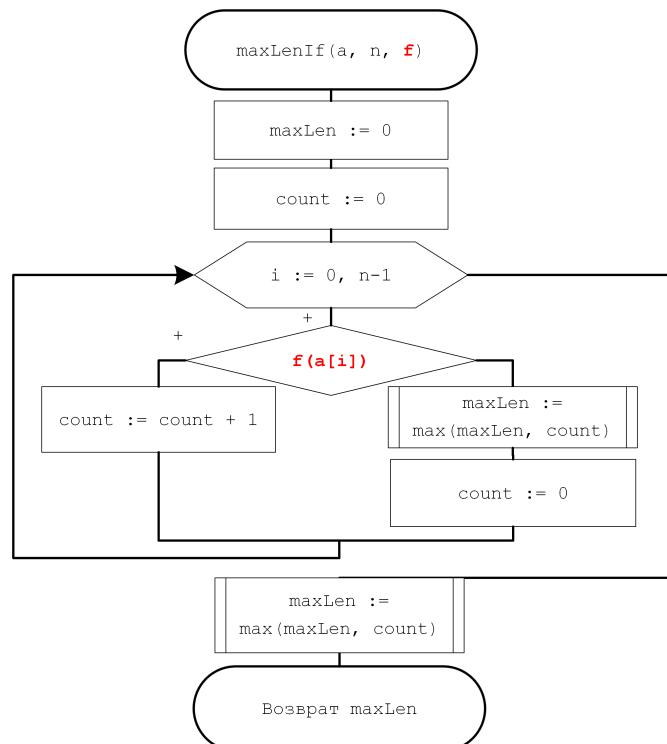


Рис. 9.9 – Блок-схема алгоритма для нахождения максимального количества подряд идущих элементов, удовлетворяющих условию

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.

Спецификация функции *deleteNegative*:

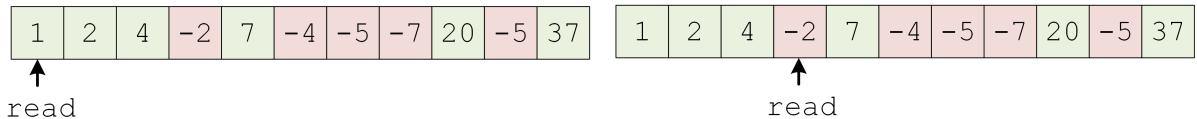
1. Заголовок: `void deleteNegative(int *a, size_t *n);`
2. Назначение: удаляет из массива `a` размера `n` отрицательные элементы. Сохраняет в `n` количество оставшихся элементов.

Вариант реализации:

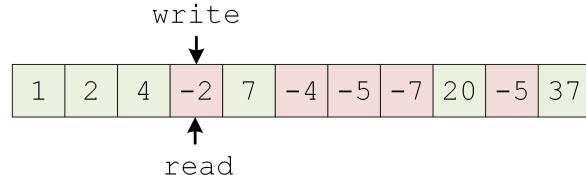
```

1 void deleteNegative(int *a, size_t *n) {
2     size_t iRead = 0;
3     while (iRead < *n && a[iRead] >= 0)
4         iRead++;
5     size_t iWrite = iRead;
6     while (iRead < *n) {
7         if (a[iRead] >= 0) {
8             a[iWrite] = a[iRead];
9             iWrite++;
10        }
11        iRead++;
12    }
13    *n = iWrite;
14 }
```

1. Пропускаем элементы, которые не будут удалены:

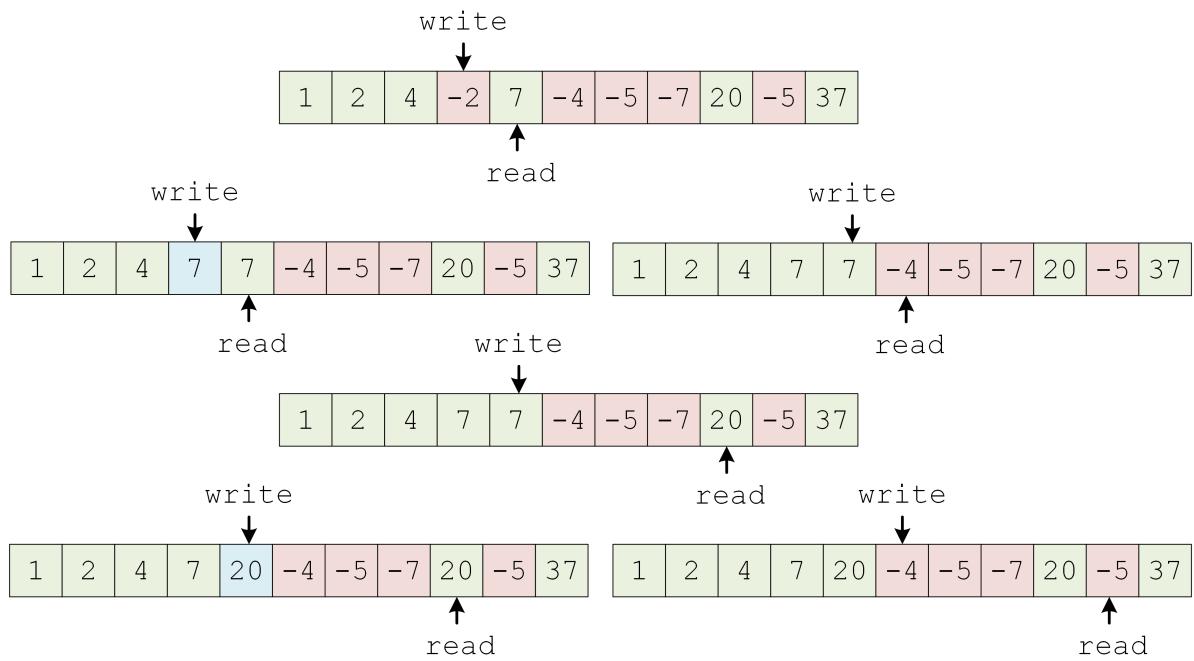


2. Если последовательность не закончилась, создаём индекс для записи:



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:

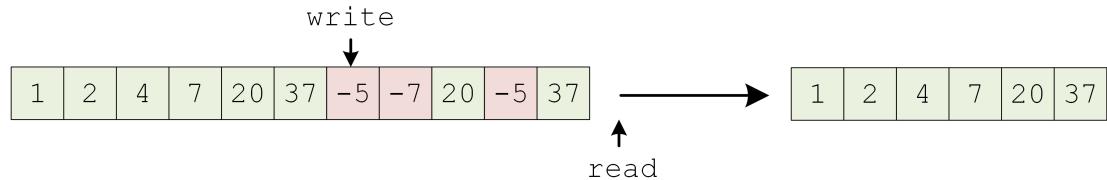


Рис. 9.10 – Однопроходный алгоритм удаления

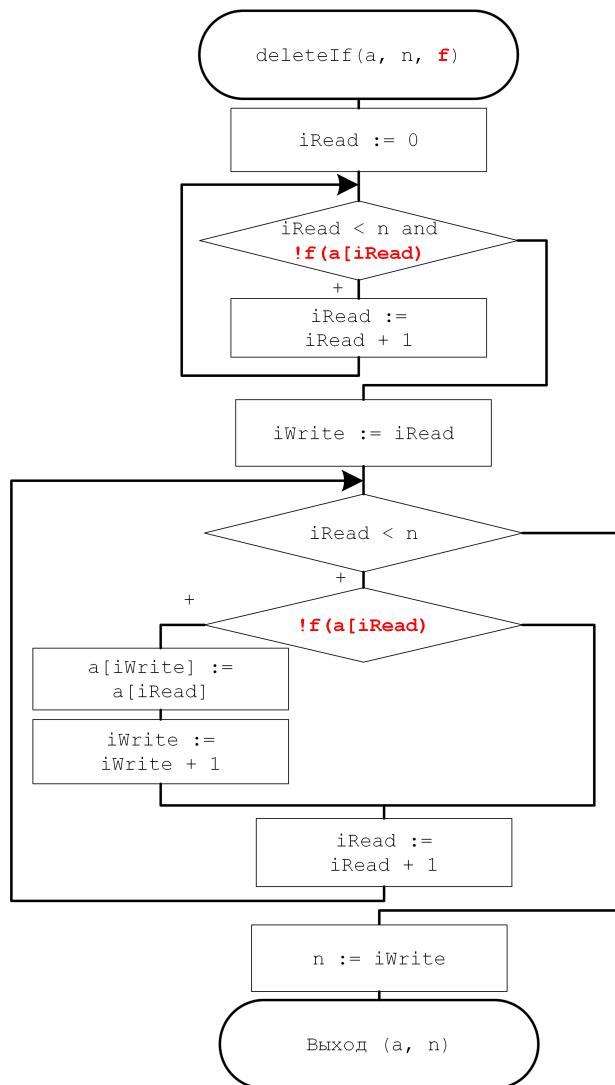


Рис. 9.11 – Блок-схема однопроходного алгоритма удаления

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 примеру⁵:

```

1 int main() {
2     int a[5] = {1, 2, -4, -3, -4};
3     size_t n = sizeof(a) / sizeof(int);
4
5     deleteNegative(a, &n);
6     printf("%d", n); // n = 2
7     // Функция изменила значение n, так как бы передан её адрес
8     // в deleteNegative.
9
10    // Если при помощи операции косвенного доступа происходят
11    // изменения переменной n - меняется непосредственно
12    // n, которая находится в функции main
13
14    return 0;
15 }
  
```

⁵Фрагмент `size_t n = sizeof(a) / sizeof(int)` работает только для случая, когда массив объявлен в той же функции, где и используется данный приём.

9.1.7 Вставка элемента с сохранением порядка элементов

Вставка элемента с сохранением порядка 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:



Спецификация функции *insert*⁶:

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.

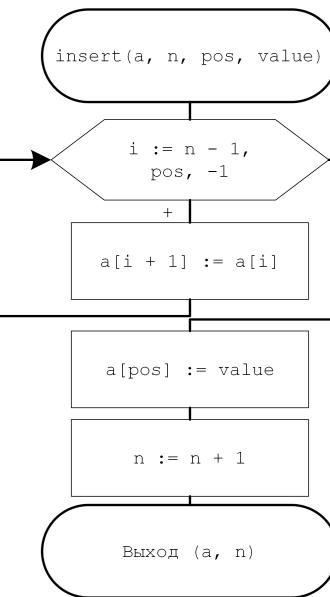


Рис. 9.12 – Блок-схема алгоритма вставки элемента

⁶Корректная реализация `insert` несколько более сложная, если использовать тип `size_t`.

```

1 void insert(int *a, size_t *n, const size_t pos, const int value) {
2     for (size_t i = *n - 1; i >= pos; i--)
3         a[i + 1] = a[i];
4     a[pos] = value;
5     (*n)++;
6 }
```

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:

```

1 int main() {
2     int a[10];
3
4     size_t size;
5     scanf("%u", &size);
6
7     inputArray(a, size);           // вводим массив размера size
8
9     insert(a, &size, 5, 100);    // осуществляем вставку элемента
10                           // количество заполненных массива увеличивается
11                           // на один
12
13 }
```

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.

9.1.8 Удаление элемента с сохранением порядка элементов

Удаление элемента с сохранением порядка 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): $\mathbf{O(n)}$.

При удалении элемента, размер массива должен уменьшиться на единицу. Все элементы с `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`:

- Заголовок: `void deleteByPosSaveOrder(int *a, size_t *n, const size_t pos);`
- Назначение: удаляет элемент по индексу `pos` из массива `a` размера `n`. Уменьшает количество элементов `n` на единицу.

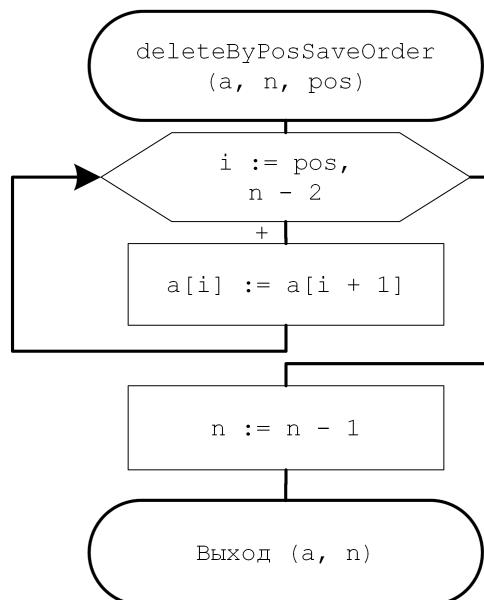


Рис. 9.13 – Блок-схема удаления элемента из массива

```

1 void deleteByPosSaveOrder(int *a, size_t *n, const size_t pos) {
2     for (size_t i = pos; i < *n - 1; i++)
3         a[i] = a[i + 1];
4     (*n)--;
5 }
```

9.1.9 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 9.14.

Приведём два решения: первый вариант будет использовать индексы, второй – через указатели:

```

1 void reverse(int *a, const size_t n) {
2     for (size_t i = 0, j = n - 1; i < j; i++, j--)
3         swap(&a[i], &a[j]);
4 }
5
6 void reverse(int *a, const size_t n) {
7     for (int *pLeft = a, *pRight = a + n - 1;
8          pLeft < pRight;
9          pLeft++, pRight--)
10        swap(pLeft, pRight);
11 }
```

Спецификация функции *reverse*:

1. Заголовок: `void reverse(int *a, const size_t n);`
2. Назначение: обращение элементов массива a размера n .

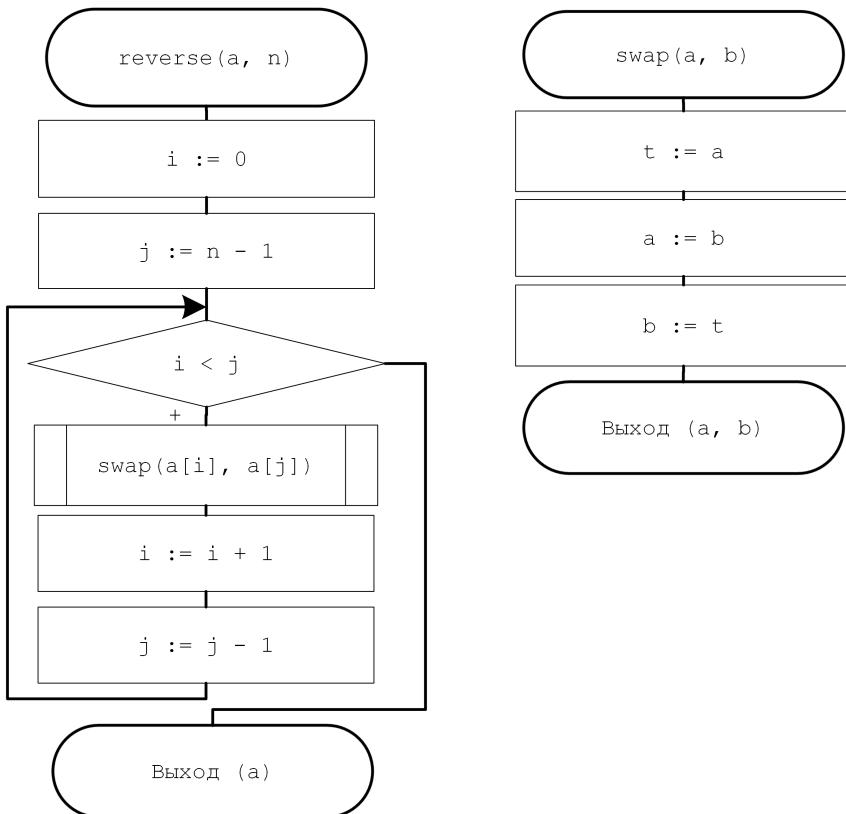


Рис. 9.14 – Блок-схема обращения элементов массива и обмена двух элементов

9.1.10 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:

```

1 int isNonDecreasing(const int *a, const int n) {
2     int i = 1;
3     while (i < n && a[i - 1] <= a[i])
4         i++;
5     return i == n;
6 }

1 int isNonDecreasing(const int *a, const int n) {
2     for (int i = 1; i < n && a[i - 1] <= a[i]; i++)
3         if (a[i - 1] > a[i])
4             return 0;
5     return 1;
6 }
  
```

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'.

9.2 Неупорядоченные массивы

9.2.1 Добавление элемента в массив

Добавление элемента в массив

Дан массив a размера n . Необходимо добавить элемент x в конец массива.

Сложность по времени: **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:

```

1 void append(int *a, size_t *n, const int value) {
2     a[*n] = value;
3     (*n)++;
4 }
```

9.2.2 Удаление элемента без сохранения порядка элементов

Удаление элемента без сохранения порядка элементов

Дан массив a размера n . Необходимо удалить элемент на позиции i . Конечный порядок элементов неважен.

Сложность по времени: **O(1)**.

Для удаления элемента из неупорядоченного массива достаточно переместить последний элемент массива на место удаляемого, и уменьшить размер массива на единицу:

```

1 void deleteByPosUnsaveOrder(int *a, size_t *n, size_t pos) {
2     a[pos] = a[*n - 1];
3     (*n)--;
4 }
```

9.3 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 ОЗУ⁷, 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.

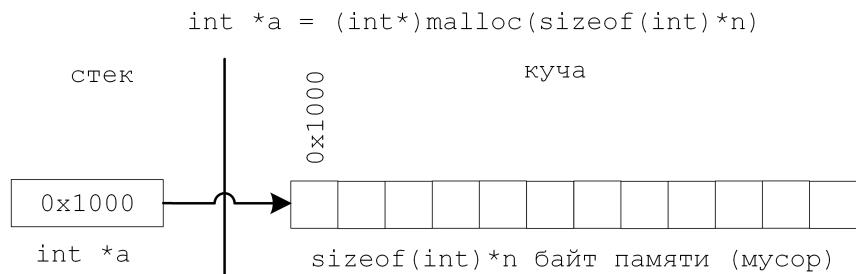
Первое, что приходит в голову – выделить памяти с запасом. С запасом это сколько? Сто, тысяча, десятки тысяч? Пользователь может вообще ввести 10 элементов, а наша избыточная память будет простаивать. С другой стороны, там может оказаться значение большее, чем наше предположение, тогда памяти будет не хватать.

⁷**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.

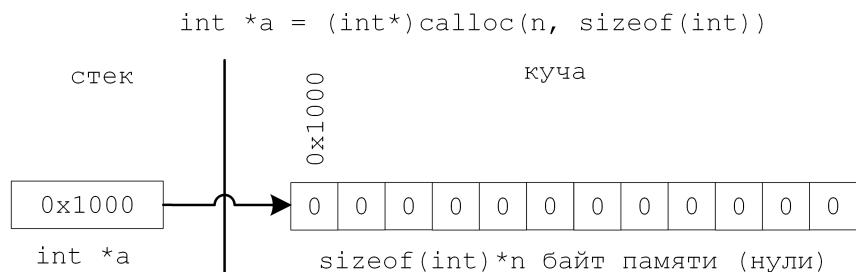
За работу с динамической памятью в языке С отвечают следующие функции, определенные в заголовочном файле `malloc.h`:

- `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:

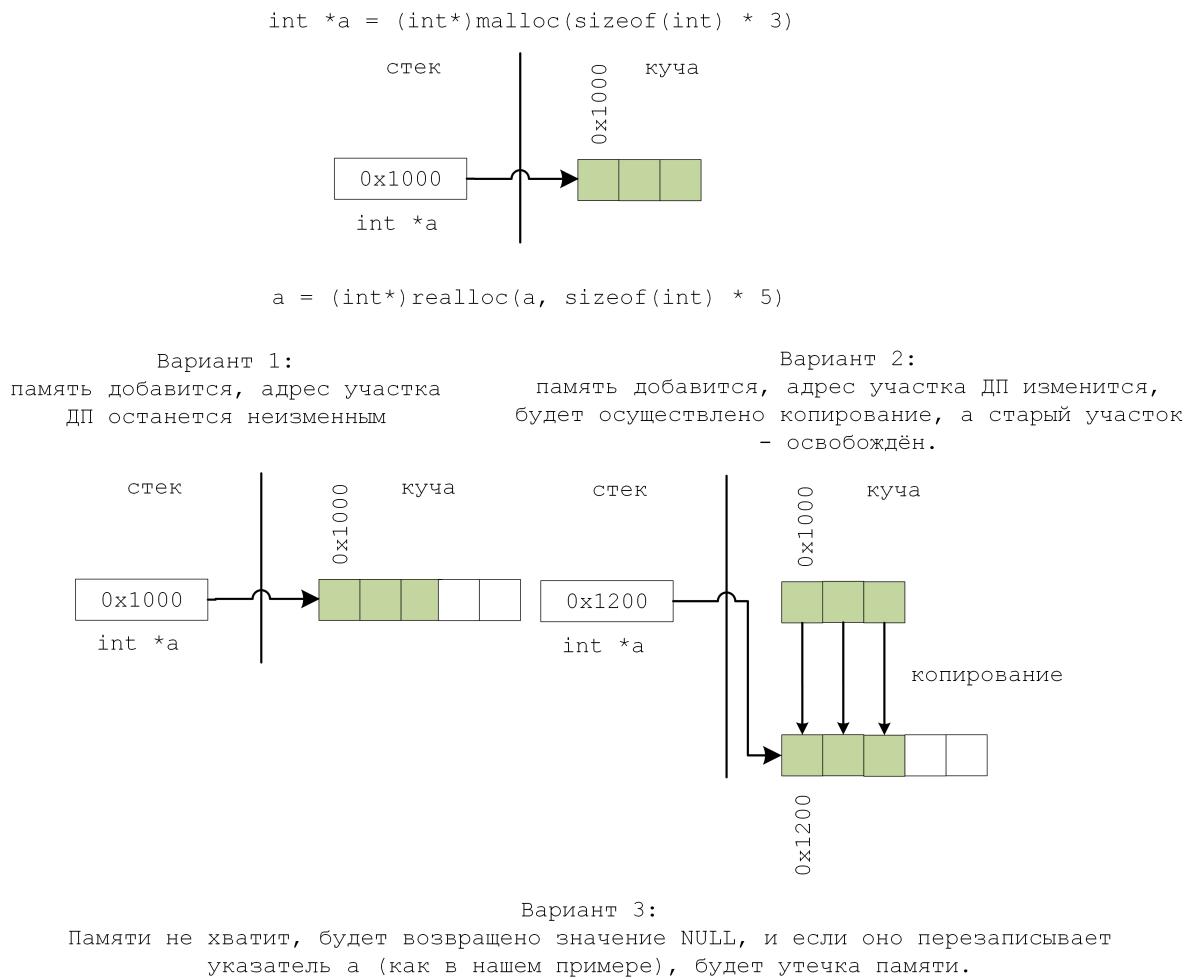


Рис. 9.15 – Варианты работы `realloc`

- `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.

9.3.1 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:

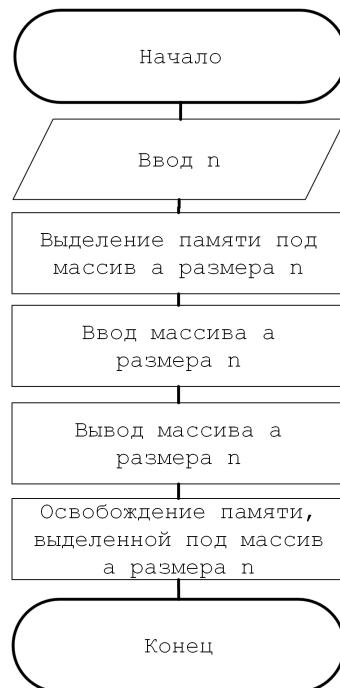


Рис. 9.16 – Алгоритм в укрупненных блоках

Код для решения задачи:

```

1 #include <stdio.h>
2 #include <malloc.h>
3
4 void inputArray(int *a, const size_t size) {
5     for (size_t i = 0; i < size; ++i)
6         scanf("%d", &a[i]);
7 }
8
9 void outputArray(int *a, const size_t size) {
10    for (size_t i = 0; i < size; ++i)
11        printf("%d ", a[i]);
12    printf("\n");
13 }
14
15 int main() {
16     int n;
17     scanf("%d", &n);
18
19     // выделение памяти под n элементов типа int
20     int *a = malloc(sizeof(int) * n);
21     inputArray(a, n);
22
23     outputArray(a, n);
24
25     // освобождение памяти
26     free(a);
27
28     return 0;
29 }
```

9.3.2 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:

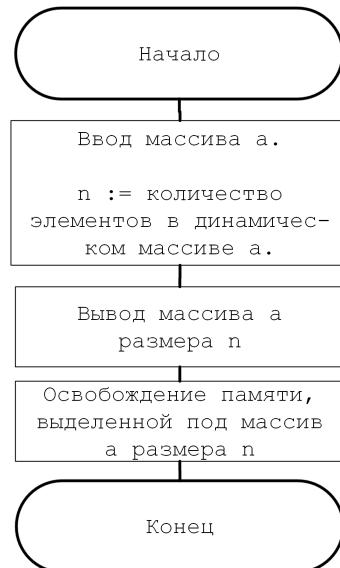


Рис. 9.17 – Алгоритм в укрупненных блоках

Больший интерес для нас будет представлять функция ввода и механизм выделения памяти. Неправильным будет вызывать `realloc` при вводе элемента, например так:

```

1 void inputArray(int **a, size_t *size) {
2     *a = malloc(sizeof(int));
3
4     int i = 0;
5     while (1) {
6         int value;
7         scanf("%d", &value);
8
9         if (value == 0)
10            break;
11
12         *a = realloc(*a, sizeof(int) * (i + 1));
13         (*a)[i++] = value;
14     }
15
16     *size = i;
17 }
```

Каждый раз будет происходить перевыделение памяти. Корректнее поступить так:

```

1 #include <stdio.h>
2 #include <malloc.h>
3
4 // функция принимает указатель на указатель, так как указатель в функции main
5 // должен быть изменен после работы функции. По окончанию работы там должен
6 // храниться адрес выделенного участка памяти.
7 void inputArray(int **a, size_t *size) {
8
9     // выделяем память под элемент
10    int maxSize = 1;
11    *a = malloc(sizeof(int) * maxSize);
12
13    int i = 0;
14    while (1) {
15        // считываем значение
16        int value;
17        scanf("%d", &value);
18
19        if (value == 0)
20            break;
21
22        // если выделенная память закончилась, выполняем перераспределение
23        if (i == maxSize) {
24            maxSize *= 2;
25            *a = realloc(*a, sizeof(int) * maxSize);
26        }
27
28        // сохраняем значение в массиве
29        (*a)[i++] = value;
30    }
31
32    // уменьшаем количество выделенной памяти
33    *a = realloc(*a, sizeof(int) * i);
34    *size = i;
35}
36
37 void outputArray(int *a, const size_t size) {
38     for (int i = 0; i < size; ++i)
39         printf("%d ", a[i]);
40     printf("\n");
41 }
42
43 int main() {
44     size_t n;
45     int *a;
46     inputArray(&a, &n);
47
48     outputArray(a, n);
49     free(a);
50
51     return 0;
52 }
```

9.4 * 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:

```

1 int isEven(int x) {
2     return x % 2 == 0;
3 }
4
5 int isNegative(int x) {
6     return x < 0;
7 }
```

Идентификаторы `isEven` и `isNegative` - имена функций, которые подобно переменным имеют свой адрес в памяти:

```

1 printf("%p\n", isEven);      // %p - спецификатор для вывода адреса
2 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:

```

1 int (*f)(int);           // указатель на функцию, которая принимает один параметр
2                                // типа int и возвращает значение типа int
3 float (*g)(float, int); // указатель на функцию, которая принимает два параметра
4                                // и возвращает значение типа float
```

Переменным-указателям можно присваивать адреса функций:

```

1 #include <stdio.h>
2
3
4 int isEven(int x) {
5     return x % 2 == 0;
6 }
7
8 int isNegative(int x) {
9     return x < 0;
10 }
11
12 int main() {
13     int (*f)(int);
14
15     int x = 10;
16     f = isEven;
17     printf("%d\n", f(x));          // выведет 1 в обоих случаях
18     printf("%d\n", isEven(x));    // варианты работают одинаково
19
20     f = isNegative;
21     printf("%d\n", f(x));        // выведет 0 в обоих случаях
22     printf("%d\n", isNegative(x)); // варианты работают одинаково
23
24     return 0;
25 }
```

Опишем возможную проблему при разработке приложения. Рассмотрим задачу поиска количества отрицательных чисел. Её можно было реализовать посредством функции `countNegative`:

```

1 int countNegative(const int *a, const size_t size) {
2     int negativeCount = 0;
3     for (size_t i = 0; i < size; i++)
4         negativeCount += a[i] < 0;
5     return negativeCount;
6 }
```

Позже была поставлена задача считать четные:

```

1 int countEven(const int *a, const size_t size) {
2     int evenCount = 0;
3     for (size_t i = 0; i < size; i++)
4         negativeCount += a[i] % 2 == 0;
5     return evenCount;
6 }
```

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:

```

1 int isEven(int x) {
2     return x % 2 == 0;
3 }
4
5 int isNegative(int x) {
6     return x < 0;
7 }
8
9 // последний параметр функции - указатель на функцию
10 int countIf(const int *a, const size_t size, int (*f)(int)) {
11     int counter = 0;
12     for (size_t i = 0; i < size; i++)
13         if (f(a[i]))
14             counter++;
15     return counter;
16 }
17
18 int main() {
19     int a[5] = {1, 2, 3, 4, -5};
20
21     // передаём адреса функций в функцию countIf
22     printf("evenCount = %d\n", countIf(a, 5, isEven));
23     printf("negativeCount = %d\n", countIf(a, 5, isNegative));
24
25     return 0;
26 }
```

Можно писать и другие алгоритмы обобщенно. Например, однопроходный алгоритм удаления:

```

1 #include <stdio.h>
2
3 // функции-предикаты
4 int isEven(int x) {
5     return x % 2 == 0;
6 }
7
8 int isNegative(int x) {
9     return x < 0;
10 }
```

```

12 // однопроходный алгоритм удаления
13 void deleteIf(int *a, size_t *n, int (*deletePredicate)(int)) {
14     size_t iRead = 0;
15     while (iRead < *n && !deletePredicate(a[iRead]))
16         iRead++;
17     size_t iWrite = iRead;
18     while (iRead < *n) {
19         if (!deletePredicate(a[iRead])) {
20             a[iWrite] = a[iRead];
21             iWrite++;
22         }
23         iRead++;
24     }
25     *n = iWrite;
26 }
27
28 void outputArray(const int *a, const size_t size) {
29     for (size_t i = 0; i < size; i++)
30         printf("%d ", a[i]);
31     printf("\n");
32 }
33
34 int main() {
35     int a[5] = {1, 2, 3, 4, -5};
36     printf("Before: ");
37     outputArray(a, 5);
38     size_t n = 5;
39     deleteIf(a, &n, isNegative);
40     printf("After: ");
41     outputArray(a, n); // 1, 2, 3, 4
42
43     int b[5] = {1, 2, 3, 4, -5};
44     printf("Before: ");
45     outputArray(b, 5);
46     size_t m = 5;
47     deleteIf(b, &m, isEven);
48     printf("After: ");
49     outputArray(b, m); // 1 3 -5
50 }

```

Раз можно создать указатель на функцию, то можно создать и массив таких указателей:

```

1 int (*fs[2])(int) = {
2     isEven,
3     isNegative,
4 };
5
6 for (int i = 0; i < 2; i++)
7     printf("%d\n", fs[i](10));

```

В качестве примера задачи, в которой передаётся функция в функцию рассмотрим вычисление значения определенного интеграла:

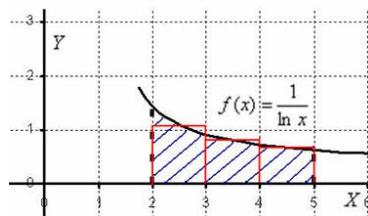
Вычисление значения определенного интеграла

Необходимо вычислить значение определенного интеграла:

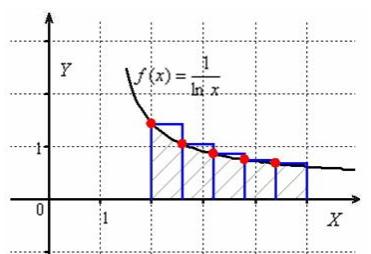
$$s = \int_a^b f(x)dx$$

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 .

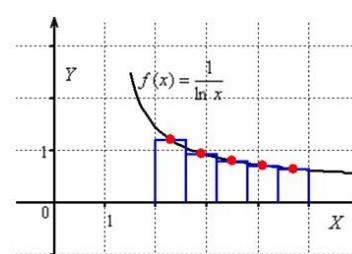
Одним из способов приближенного вычисления площади под графиком является метод прямоугольников. Отрезок интегрирования разбивается на несколько частей и строится ступенчатая фигура, которая по площади близка к искомой площади:



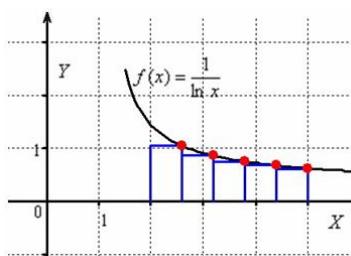
Выделяют метод левых средних и правых прямоугольников суть которых интуитивно понятна из рисунков:



(a) Левых прямоугольников



(b) Средних прямоугольников



(c) Правых прямоугольников

Рис. 9.18 – 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:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // enum - перечисление - позволяет задать набор
5 // связанных между собой именованных констант
6 // объявление перечисления:
7 enum typeOfMethods {
8     LEFT = 0,
9     CENTRAL = 1,
10    RIGHT = 2,
11};
```

```

12 // f - указатель на функцию, которая возвращает значение функции f(x) в точке x
13 // где x имеет тип double и функция возвращает значение типа double
14 // n - количество прямоугольников
15
16 double getS(double a, double b, int n, double (*f)(double), enum
17     typeOfMethods method) {
18     double delta = (b - a) / n;
19     double heightPoint;
20     switch (method) {
21         case LEFT:
22             heightPoint = a;
23             break;
24         case CENTRAL:
25             heightPoint = (a + delta) / 2;
26             break;
27         case RIGHT:
28             heightPoint = a + delta;
29             break;
30         default:
31             // выводим в поток ошибок сообщение
32             // об ошибке и заканчиваем работу программы
33             fprintf(stderr, "incorrect typeOfMethod");
34             exit(1);
35     }
36
37     double S = 0;
38     for (int i = 0; i < n; i++) {
39         S += f(heightPoint)*delta;
40         heightPoint += delta;
41     }
42     return S;
}

```

Для теста методов возьмём функцию $f(x) = x^2$:

```

1 double getX2(double x) {
2     return x*x;
3 }

```

И при необходимости вычислить приближенное значение определенного интеграла тем или иным способом достаточно осуществить любой из вызовов:

```

1 printf("%f\n", getS(0, 2, 1000, getX2, LEFT));      // 2.662668
2 printf("%f\n", getS(0, 2, 1000, getX2, CENTRAL)); // 2.666666
3 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.

9.5 * Префиксная сумма массива

Префиксной суммой массива 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. 9.19 – 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:

$$\sum_{i=l}^{r-1} a[i] = \left[a[0] + \dots + a[r - 1] \right] - \left[a[0] + \dots + a[l - 1] \right] = b[r] - b[l] \quad (9.1)$$

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 9.1 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:

$$\bigoplus_{i=l}^{r-1} a[i] = \left[a[0] \oplus \dots \oplus a[r-1] \right] \oplus \left[a[0] \oplus \dots \oplus a[l-1] \right] = b[r] \oplus b[l]$$

9.5.1 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(nlogn)**.

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:

```

1 #include <stdio.h>
2
3 int main() {
4     int a[] = {-1, 1, -1, 0, 0, 1};
5     int n = sizeof(a) / sizeof(int);
6
7     int count = 0;
8     for (int i = 0; i < n; i++) {
9         long long s = 0;
10        for (int j = i; j < n; j++) {
11            s += a[j];
12            count += s == 0;
13        }
14    }
15
16    printf("%d", count);
17
18    return 0;
19 }
```

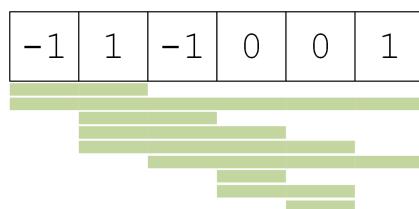


Рис. 9.20 – Все нулевые подмассивы в порядке их нахождения алгоритмом.

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. 9.21 – 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:

$$n = \sum_{x \in X} \frac{n_x(n_x - 1)}{2}$$

9.6 * 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. 9.22 – 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. 9.23 – 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.

9.6.1 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. 9.24 – 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. 9.25 – Изменение разностного массива при добавлении значения для случая когда правая граница – конец массива.

9.6.2 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. 9.26 – 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 $-(r - l + 1)x$.

9.7 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.

9.7.1 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:⁸

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:⁹

1. Ввод массива.
2. Поиск позиции первого вхождения нуля.
3. Сортировка элементов подмассива.
 - (а) Обмен двух значений.
4. Вывод массива.

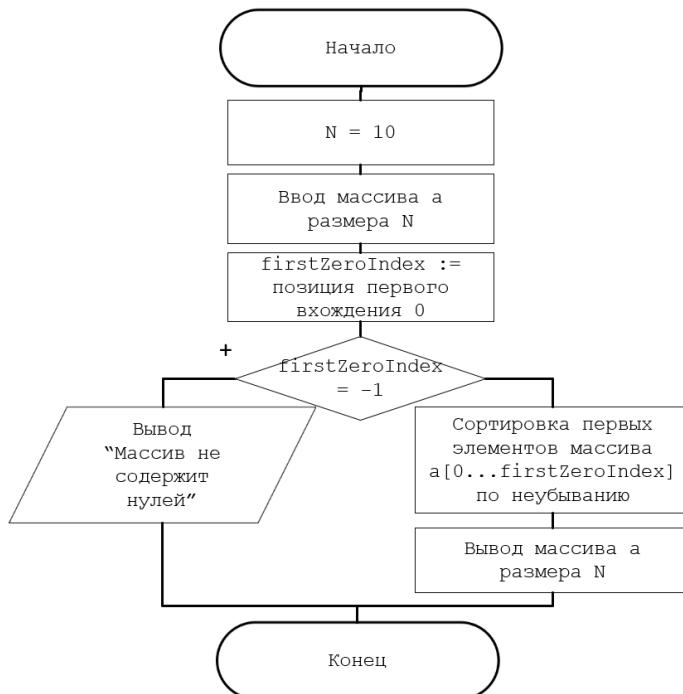
⁸Относительно оформления была принята следующая мера: описание спецификаций функций были упразднены из требований к решениям лабораторных работ. Ранее они содержали информацию о заголовке функции и её назначении. Прикладывалась блок-схема. Но

- 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.

⁹Подзадачи описываются без привязки к переменным. Просто подумайте, какие действия вам придётся осуществить при решении задачи.

3. Блок-схема в укрупненных блоках в терминах выделенных подзадач¹⁰:



4. Текст программы:

```

1 #include <stdio.h>
2
3 // определяем именованную константу, которая отвечает за размер массива
4 #define N 10
5
6 // ввод элементов массива а размера n
7 void inputArray(int *a, const size_t size) {
8     for (size_t i = 0; i < size; i++)
9         scanf("%d", &a[i]);
10 }
11
12 // вывод элементов массива а размера n
13 void outputArray(const int *a, const size_t size) {
14     for (size_t i = 0; i < size; i++)
15         printf("%d ", a[i]);
16     printf("\n");
17 }
18
19 // обмен значений двух переменных по адресам а и b
20 void swap(int *a, int *b) {
21     int t = *a;
22     *a = *b;
23     *b = t;
24 }
25
26 // сортировка элементов массива а размера n
27 void bubbleSort(int *a, const size_t size) {
28     for (int i = 0; i < size - 1; i++) {
  
```

¹⁰ 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.

```

29         for (int j = size - 1; j > i; j--) {
30             if (a[j - 1] > a[j]) {
31                 swap(a[j - 1], a[j]);
32             }
33         }
34     }
35 }
36
37 // возвращает позицию первого элемента со значением value в массиве a размера n,
38 // если таковой имеется, иначе - -1.
39 int linearFind(const int *a, const size_t n, const int value) {
40     size_t i = 0;
41     while (i < n && a[i] != value)
42         i++;
43     if (i == n)
44         i = -1;
45     return i;
46 }
47
48 int main() {
49     int a[N];
50     inputArray(a, N);
51
52     int firstZeroIndex = linearFind(a, N, 0);
53     if (firstZeroIndex == -1)
54         printf("No zeros");
55     else {
56         bubbleSort(a, firstZeroIndex);
57         outputArray(a, N);
58     }
59
60     return 0;
61 }
```

9.7.2 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.

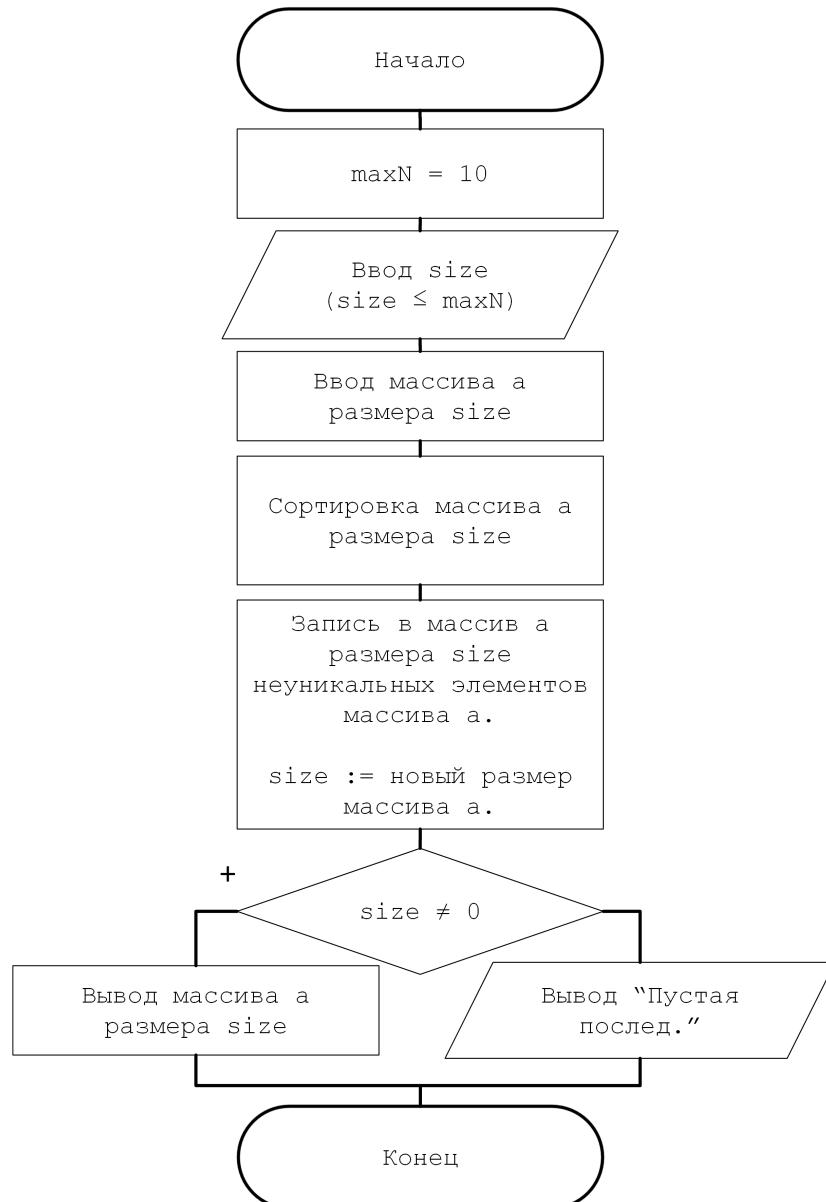
1. Опишем тестовые данные:

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:

```

1 #include <stdio.h>
2
3 // определяем именованную константу
4 // которая определяет максимальный размер массива
5 #define N 10
  
```

```

6 // записывает по адресу a неуникальные элементы массива по адресу
7 // a размера size. Записывает в size количество неуникальных элементов.
8 void saveDuplicate(int *a, size_t *size) {
9     int lastSavedValue = a[0] - 1;
10    int iRec = 0;
11    for (size_t i = 1; i < *size; i++) {
12        if (a[i] == a[i - 1] && a[i] != lastSavedValue) {
13            a[iRec++] = a[i];
14            lastSavedValue = a[i];
15        }
16    }
17    *size = iRec;
18 }
19
20
21 int main() {
22     int a[N];
23     size_t size;
24     scanf("%u", &size);
25
26     inputArray(a, size);
27
28     selectionSort(a, size);
29     saveDuplicate(a, &size);
30
31     if (size != 0)
32         outputArray(a, size);
33     else
34         printf("Empty sequence");
35
36     return 0;
37 }
```

9.7.3 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 2 3 4 1
5 4 3 2 1	
$N = 10$	3 0 3 0 3 2 1 2 1 2
3 2 1 0 3 2 1 0 3 2	

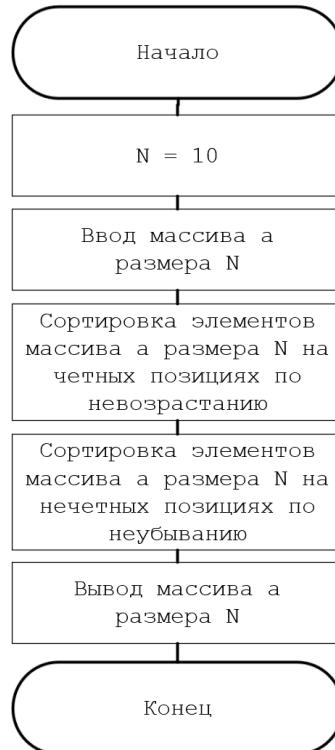
2. Выполним выделение подзадач:

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:

```

1 #include <stdio.h>
2
3 // определяем именованную константу, которая отвечает за размер массива
4 #define N 10
5
6 // сортировка элементов массива а размера size с шагом step по неубыванию
7 void selectionSortInc(int *a, const size_t size, int step) {
8     for (size_t i = 0; i < size - 1; i += step) {
9         int iMin = i;
10        int minValue = a[iMin];
11        for (size_t j = i + step; j < size; j += step) {
12            if (a[j] < minValue) {
13                iMin = j;
14                minValue = a[iMin];
15            }
16        }
17        swap(&a[i], &a[iMin]);
18    }
19 }
20
21 // сортировка элементов массива а размера size с шагом step по невозрастанию
22 void selectionSortDec(int *a, const size_t size, int step) {
23     for (size_t i = 0; i < size - 1; i += step) {
24         int iMax = i;
25         int maxValue = a[iMax];
26         for (size_t j = i + step; j < size; j += step) {
27             if (a[j] > maxValue) {
28                 iMax = j;

```

```

29         maxValue = a[iMax];
30     }
31 }
32 swap(&a[i], &a[iMax]);
33 }
34 }
35
36 int main() {
37     int a[N];
38     inputArray(a, N);
39
40     selectionSortDec(a, N, 2);
41     // +1 к указателю - возвращает указатель, который указывает не на
42     // нулевую ячейку массива, а на первую
43     selectionSortInc(a + 1, N - 1, 2);
44
45     outputArray(a, N);
46
47     return 0;
48 }
```

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 9.27):

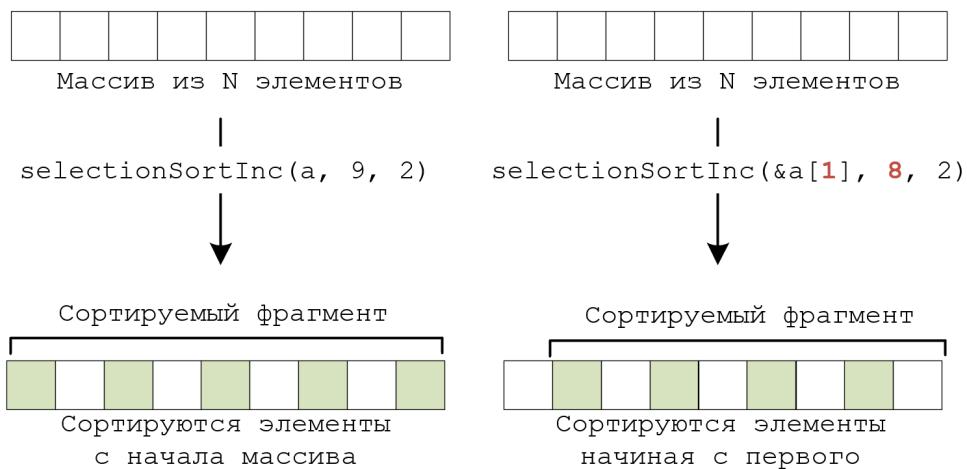


Рис. 9.27 – Подбор аргументов для функции `selectionSortInc`

Вызовы

- `selectionSortInc(&a[1], N - 1, 2)`
- `selectionSortInc(a + 1, N - 1, 2)`

работают одинаково.

9.7.4 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:

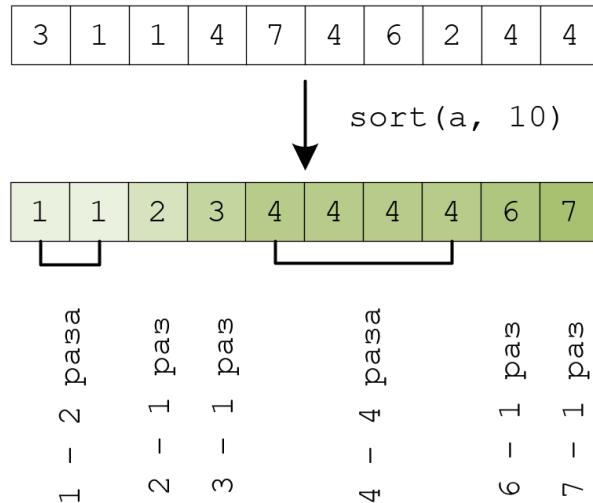


Рис. 9.28 – После сортировки исходного массива довольно легко выполнить подсчёт.

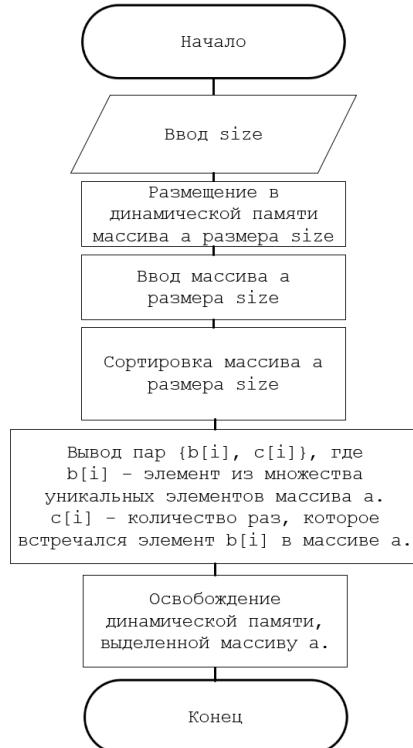
1. Опишем тестовые данные:

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:

```

1 #include <stdio.h>
2 #include <malloc.h>
3
4 // для каждого значения из множества элементов отсортированного массива a размера size
5 // выводит количество раз, сколько встречалось данное значение в массиве a
6 void outputCount(const int *a, const size_t size) {
7     int i = 0;
8     while (i < size) {
9         int countingValue = a[i];
10        int count = 0;
11        while (countingValue == a[i] && i < size) {
12            count++;
13            i++;
14        }
15        printf("%d: %d\n", countingValue, count);
16    }
17 }
18
19 int main() {
20     size_t size;
21     scanf("%u", &size);
22
23     int *a = (int*)malloc(sizeof(int) * size);
24     inputArray(a, size);
25
26     selectionSort(a, size);
27     outputCount(a, size);
28
29     free(a);
30
31     return 0;
32 }
  
```

9.7.5 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 .

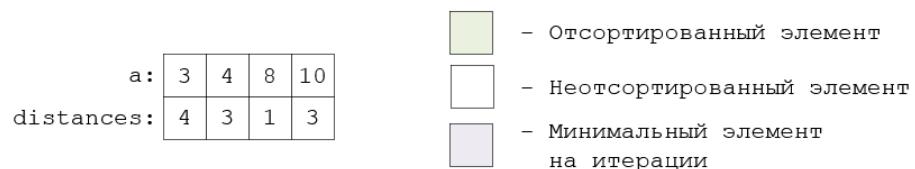
Предположим, имеются 4 точки: $a = \{3, 4, 8, 10\}$. В качестве x выступает значение 7. Вычислив массив расстояний получим:

$$distances = \{4, 3, 1, 3\}$$

Будем выполнять сортировку массива $distances$, но процесс обмена будем производить и на массиве a параллельно. Предположим, что используется сортировка выбором:

```

1 void selectionSort(int *values, int *keys, const size_t size) {
2     for (int i = 0; i < size - 1; i++) {
3         int minPos = i;
4         for (int j = i + 1; j < size; j++)
5             if (keys[j] < keys[minPos])
6                 minPos = j;
7         swap(&keys[i], &keys[minPos]);
8         swap(&values[i], &values[minPos]);
9     }
10 }
```



Ищем минимальный элемент среди неотсортированных

a:	3	4	8	10
distances:	4	3	1	3

a:	8	4	3	10
distances:	1	3	4	3

a:	8	4	3	10
distances:	1	3	4	3

Производим обмен

a:	3	4	8	10
distances:	4	3	1	3



a:	8	4	3	10
distances:	1	3	4	3



a:	8	4	10	3
distances:	1	3	3	4

Рис. 9.29 – Процесс сортировки.

9.7.6 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 $\max - 1$:

```

1 // выводит множество элементов, заключенных между min(a) и max(a) массива a
2 // размера n и непринадлежащих данному массиву
3 void outputSet(const int *a, const int n) {
4     int min = a[0];
5     int max = a[n - 1];
6     for (int v = min + 1, i = 1; v <= max; v++)
7         if (v == a[i])
8             i++;
9         else
10            printf("%d ", v);
11 }
```

9.7.7 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:

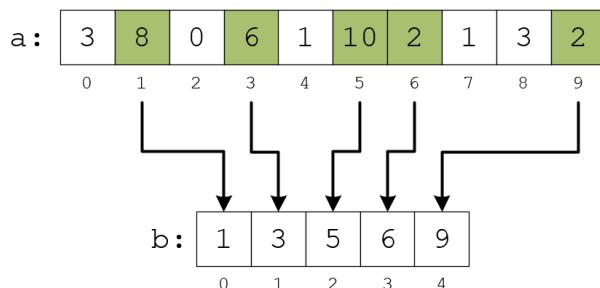
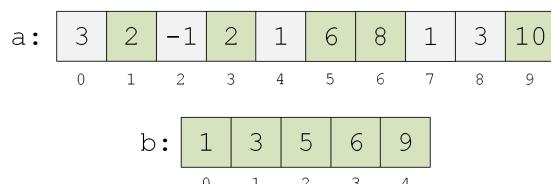
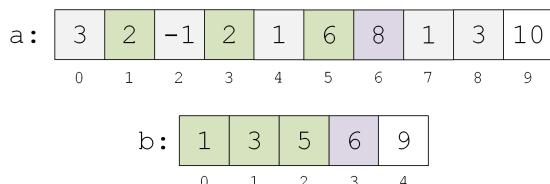
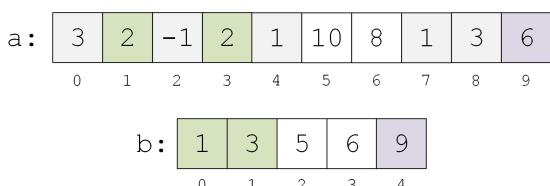
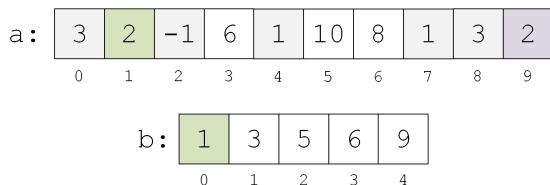
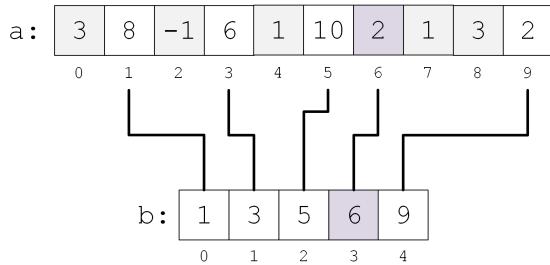


Рис. 9.30 – Создание вспомогательного массива.

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 $a[b[j]]$

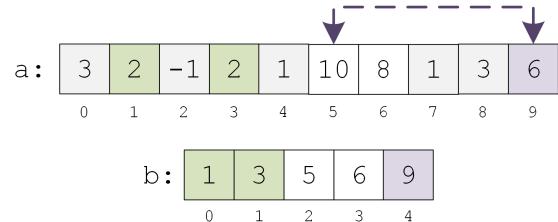
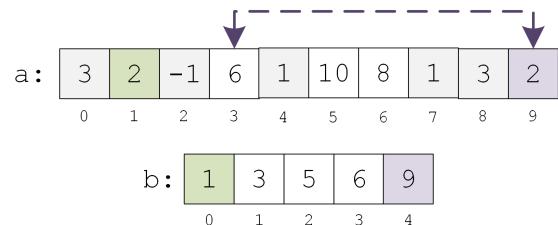
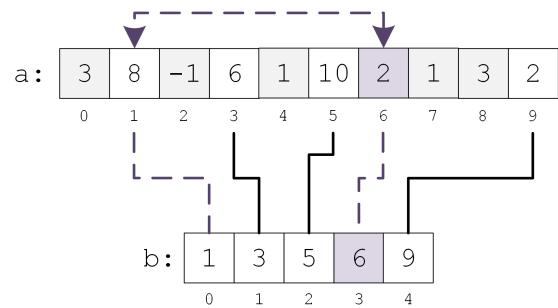


Рис. 9.31 – Сортировка массива.

```

1 #include <stdio.h>
2
3 #define N 10
4
5 // ввод массива a размера size
6 void inputArray(int *a, size_t size) {
7     for (size_t i = 0; i < size; i++)
8         scanf("%d", &a[i]);
9 }
10
11 // обмен значений двух переменных по адресам a и b

```

```

12 void swap(int *a, int *b) {
13     int t = *a;
14     *a = *b;
15     *b = t;
16 }
17
18 // сортировка выбором элементов массива a[indexes[i]] где i = [0...indexSize)
19 // по неубыванию
20 void selectionSort(int *a, const int *indexes, const size_t indexesSize)
21 {
22     for (int i = 0; i < indexesSize - 1; i++) {
23         int minPos = indexes[i];
24         for (int j = i + 1; j < indexesSize; j++)
25             if (a[indexes[j]] < a[minPos])
26                 minPos = indexes[j];
27         swap(&a[indexes[i]], &a[minPos]);
28     }
29 }
30
31 // запись по адресу indexes массива индексов четных элементов массива a размера size;
32 // indexesSize := количество четных элементов массива a размера size
33 void getArrayOfIndexesOfEven(const int *a, size_t size, int *indexes,
34                             size_t *indexesSize) {
35     size_t iWrite = 0;
36     for (int i = 0; i < size; i++)
37         if (a[i] % 2 == 0)
38             indexes[iWrite++] = i;
39     *indexesSize = iWrite;
40 }
41
42 // вывод массива a размера size
43 void outputArray(const int *a, size_t size) {
44     for (size_t i = 0; i < size; i++)
45         printf("%d ", a[i]);
46 }
47
48 int main() {
49     int a[N];
50     inputArray(a, N);
51
52     int indexes[N];
53     size_t indexesSize;
54     getArrayOfIndexesOfEven(a, N, indexes, &indexesSize);
55     selectionSort(a, indexes, indexesSize);
56
57     outputArray(a, N);
58 }
```

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 10

Поиск

10.1 Быстрый линейный поиск

Версия линейного поиска в неотсортированном массиве описана на странице 210. 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.

```
1 #include <stdio.h>
2
3 int linearFindFast(int *a, const size_t n, const int x) {
4     // на позицию n записываем искомое значение
5     a[n] = x;
6
7     int i = 0;
8     while (a[i] != x)
9         i++;
10
11    return i == n ? -1 : i;
12 }
13
14 int main() {
15     int a[10];
16     // для возможности осуществить быстрый линейный поиск
17     // должно иметься место для служебного элемента
18     inputArray(a, 9);
19     linearFindFast(a, 9, 5);
20
21     return 0;
22 }
```

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.

10.2 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:

```

1 int linearSearch(const int *a, const int n, int x) {
2     int i = 0;
3     while (i < n && a[i] < x) {
4         i++;
5     }
6     return i < n && a[i] == x ? i : -1;
7 }
```

Выполним поиск значения $x = 6$ в массиве a :

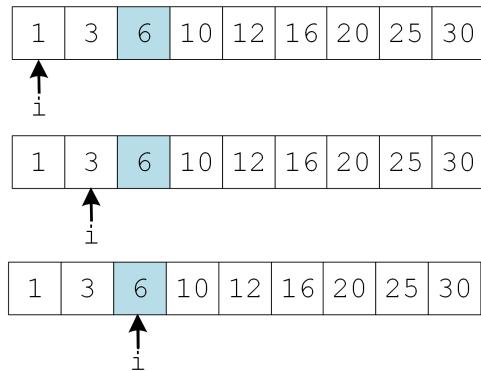


Рис. 10.1 – Вариация линейного поиска для случая, когда элемент в массиве есть

Как только находим значение, которое больше или равно x или значение i становится равным n , поиск прекращается. В его окончании проверяется условие

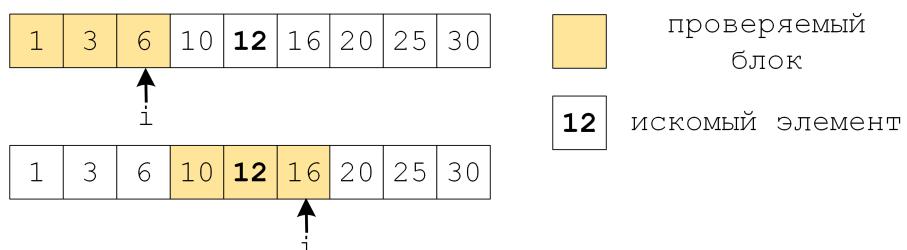
```
1 i < n && a[i] == x
```

на основании которого можно понять, имеется ли элемент x в массиве.

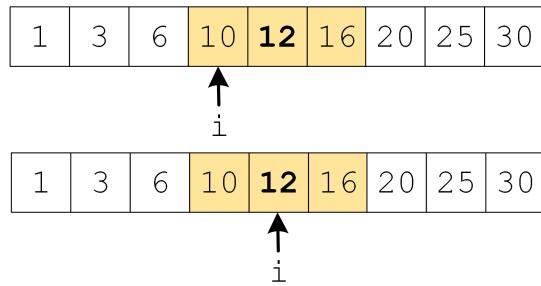
10.3 Блочный поиск. $Sqrt$ -декомпозиция

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:

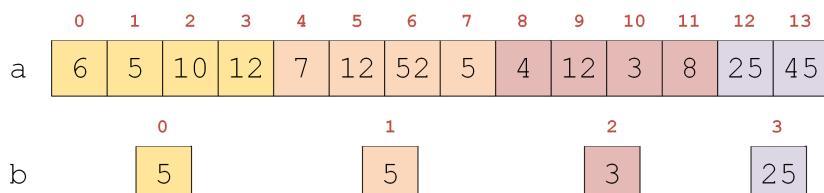
$$n'_{\text{compares}}(block_size) = -\frac{n}{block_size^2} + 1 = 0 \Rightarrow block_size = \sqrt{n}$$

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, xor и любых других ассоциативных операций). Существуют более эффективные методики для решения таких задач (например, деревья Фенвика, деревья отрезков, Segment Tree Beats и др., работающие за $O(\log n)$).

Рассмотрим произвольный массив размера $n = 14$. Разделим его на блоки размера $\lceil s = \sqrt{n} \rceil = 4$. Для каждого блока вычислим значение функции:

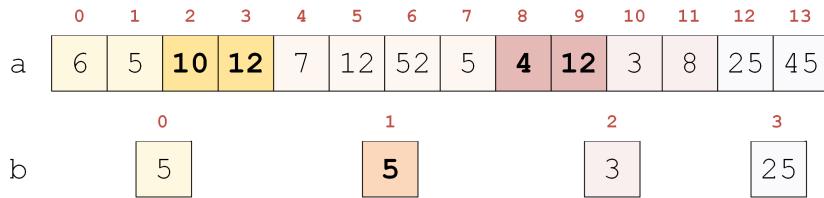
$$b[k] = \sum_{i=k*s}^{\min(n-1, (k+1)*s-1)} a[i]$$



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:



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.

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4
5 int min(const int a, const int b) {
6     return a < b ? a : b;
7 }
8
9 int getMin(const int *a, int left, int right) {
10    int _min = INT_MAX;
11    for (int i = left; i < right; i++)
12        _min = min(_min, a[i]);
13    return _min;
14 }
15
16 void getSqrtDecomposition(const int *a, const int n, int *b) {
17    const int blockSize = ceil(sqrt(n));
18    const int nBlocks = ceil(n / blockSize);
19    int startBlockIndex = 0;
20    for (int blockIndex = 0; blockIndex < nBlocks; blockIndex++) {
21        const int nElementsInBlock = min(blockSize, n - startBlockIndex);
22        b[blockIndex] = getMin(a, startBlockIndex,
23                               startBlockIndex + nElementsInBlock);
24        startBlockIndex += blockSize;
25    }
26 }
27
28 int getMinRequest(const int *a, int n, const int *b,
29                   const int left, const int right) {
30    const int blockSize = ceil(sqrt(n));
31    int _min = INT_MAX;
32    const int leftBlockIndex = left / blockSize;

```

```

33     const int rightBlockIndex = (right + 1) / blockSize;
34     if (leftBlockIndex == rightBlockIndex)
35         // элементы лежат в рамках одного блока (но не занимают весь блок)
36         _min = min(_min, getMin(a, left, right + 1));
37     else {
38         // минимум до целых блоков
39         _min = min(_min, getMin(a, left, leftBlockIndex * blockSize));
40         // минимум в целых блоках
41         _min = min(_min, getMin(b, leftBlockIndex, rightBlockIndex));
42         // минимум после целых блоков
43         _min = min(_min, getMin(a, rightBlockIndex * blockSize, right +
44             1));
44     }
45     return _min;
46 }
```

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 :

```

1 void setValue(int *a, int n, int *b, int i, int x) {
2     const int blockSize = ceil(sqrt(n));
3     const int blockNumber = i / blockSize;
4     if (x > a[i] && a[i] == b[blockNumber]) { // если изменяет потенциально
5                                         // минимальный элемент блока
6         a[i] = x;
7         const int jStart = i / blockSize * blockSize;
8         const int jEnd = min(jStart + blockSize, n);
9         b[blockNumber] = getMin(a, jStart, jEnd);
10    } else
11        a[i] = x;
12
13 }
```

10.4 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 индекс¹

$$middle = \left\lfloor \frac{left + right}{2} \right\rfloor = left + \left\lfloor \frac{right - left}{2} \right\rfloor$$

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 10.2, 10.3. Реализация поиска на C²:

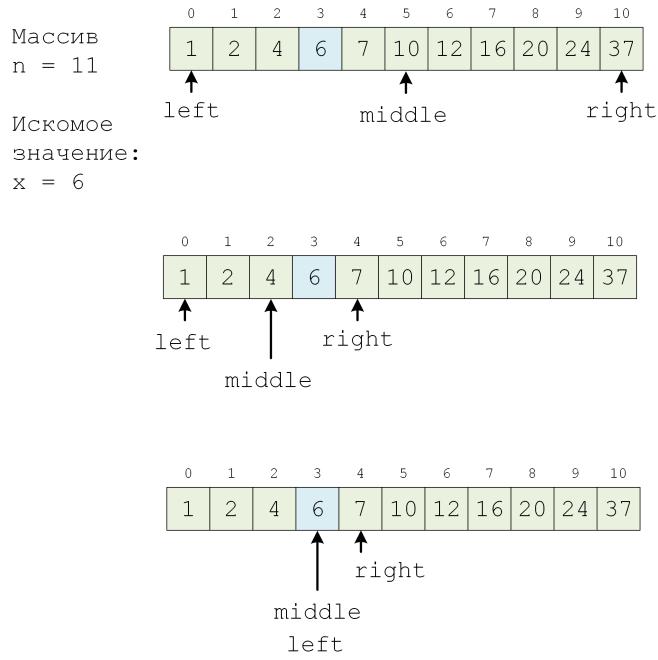
```

1 int binarySearch(const int *a, const int n, const int x) {
2     int left = 0;
3     int right = n - 1;
4     while (left <= right) {
5         int middle = (left + right) / 2;
6         if (a[middle] < x)           // если 'истина', искомый элемент лежит правее
7             left = middle + 1;
8         else if (a[middle] > x)    // если 'истина', искомый элемент лежит левее
9             right = middle - 1;
10        else
11            return middle;
12    }
13    return -1;
14 }
```

Недостаток такого варианта: сложно искать более сложные случаи, речь о которых пойдёт далее.

¹ Второй способ вычисления индекса предостерегает от потенциального переполнения.

² 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

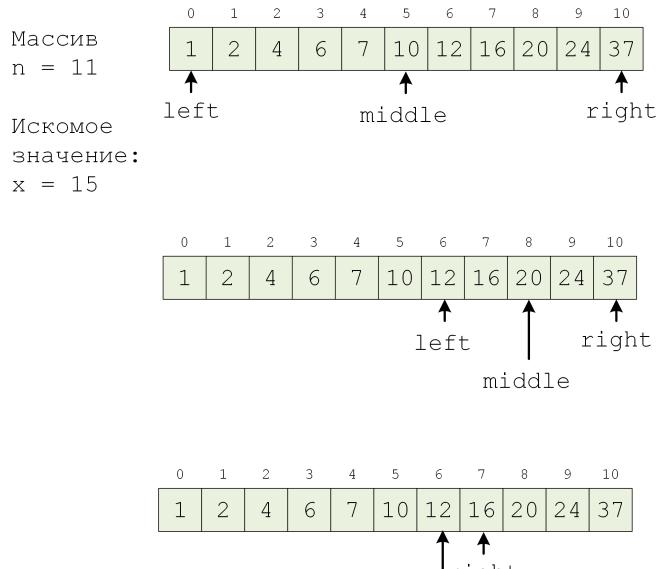
a[middle] < x -> left = middle + 1
      = 3

left = 3
right = 4
middle = (left + right) div 2
      = 3

a[middle] = x -> элемент найден

```

Рис. 10.2 – 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. 10.3 – Бинарный поиск (вариация №1) для случая, когда элемента нет в массиве

10.5 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 – больше или равны x . Если искомое значение найдено, то $a[right] = x$, если не найдено – $a[right] \neq x$.

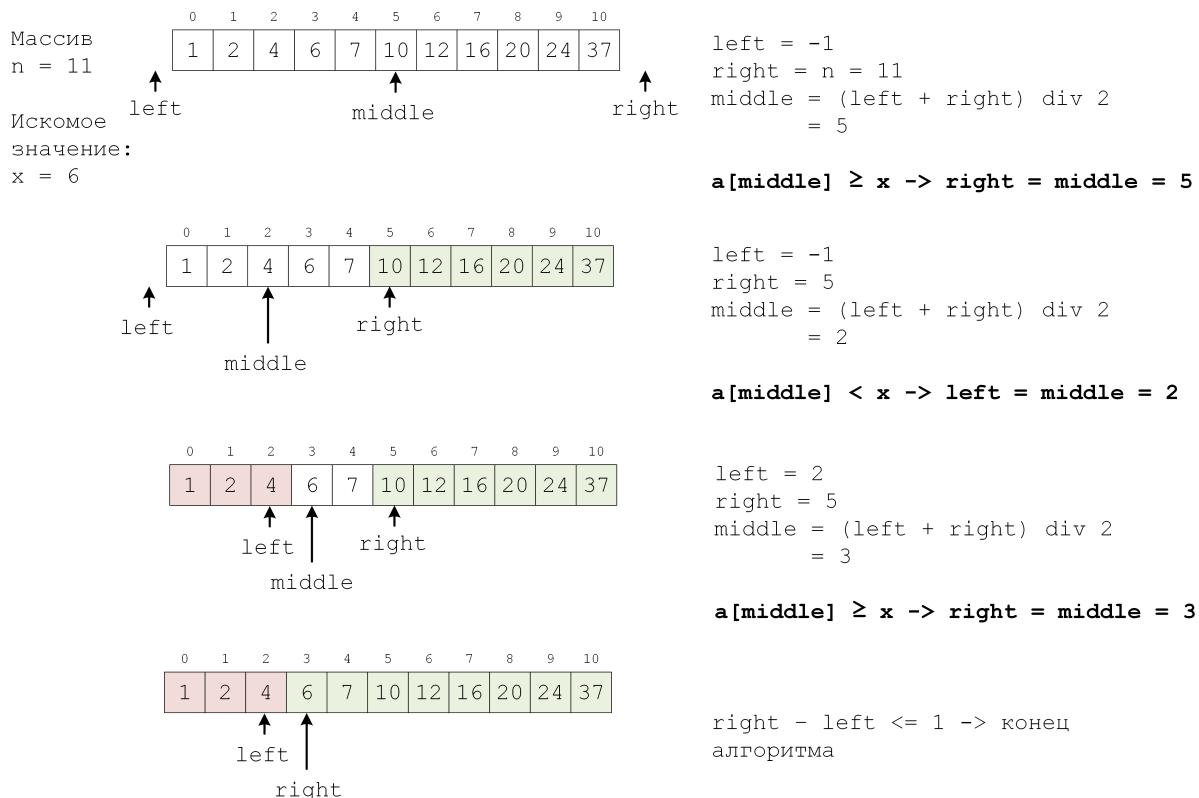
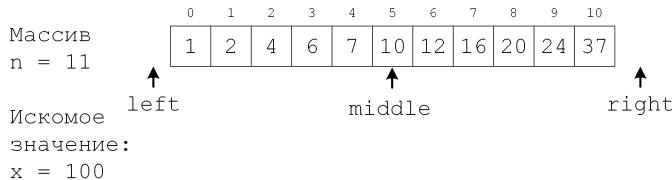


Рис. 10.4 – Бинарный поиск (вариация №2) для случая, когда элемент имеется в массиве

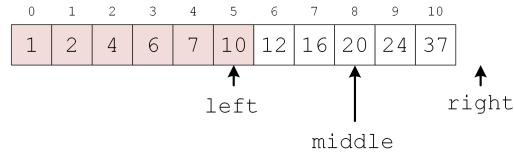
Если бы выполнялся поиск значения 5 – конечные значения *left* и *right* совпадали бы с примером выше.

Рассмотрим случай, когда искомый элемент больше всех, которые имеются в массиве:

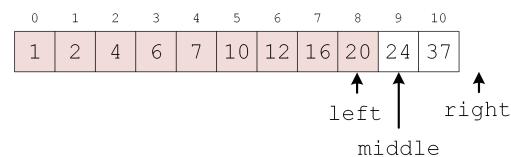


```
left = -1
right = n = 11
middle = (left + right) div 2
= 5

a[middle] < x -> left = middle = 5
```

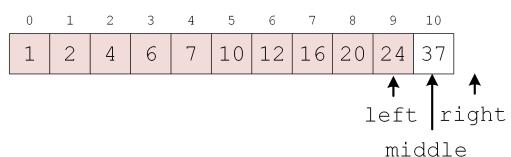


```
left = 5
right = 11
middle = (left + right) div 2
= 8
```



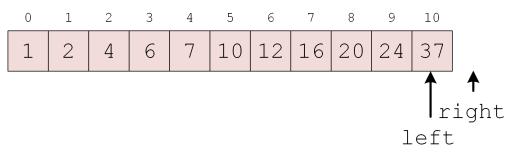
```
a[middle] < x -> left = middle = 8

left = 8
right = 11
middle = (left + right) div 2
= 9
```



```
a[middle] < x -> left = middle = 9

left = 9
right = 11
middle = (left + right) div 2
= 10
```



```
a[middle] < x -> left = middle = 10
right - left <= 1 -> конец алгоритма
```

Рис. 10.5 – Бинарный поиск (вариация №2) для случая, когда все элементы больше того, который ищем

Реализация поиска на C:

```
1 // возвращает позицию первого элемента большего или равного x
2 // (вернёт значение n если все элементы массива меньше x)
3 int binarySearchEqualOrMore(const int *a, const int n, const int x) {
4     int left = -1;
5     int right = n;
6     while (right - left > 1) {
7         int middle = (left + right) / 2;
8         if (a[middle] < x)
9             left = middle;
10        else
11            right = middle;
12    }
13    return right;
14 }
```

Для поиска

$$\max i : a[i] \leq x$$

можно воспользоваться следующим фрагментом:

```

1 // возвращает позицию последнего элемента меньшего или равного x
2 // (вернёт -1, если все числа больше x)
3 int binarySearchLessOrEqual(const int *a, const int n, const int x) {
4     int left = -1;
5     int right = n;
6     while (right - left > 1) {
7         int middle = (left + right) / 2;
8         if (a[middle] <= x)
9             left = middle;
10        else
11            right = middle;
12    }
13    return left;
14 }
```

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 $a[i] = x$.

10.6 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.

Пример:

$$a = \{2, 5, 8, 13, 21, 27, 35\}$$

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:

```
1 #include <stdio.h>
2
3 int isMore20(int x) {
4     return x > 20;
5 }
6
7 // вернёт значение n, если все элементы не удовлетворяют критерию
8 int binarySearchCriteria(const int *a, const int n,
9                          int (*fcriteria)(int)) {
10    int left = -1;
11    int right = n;
12    while (right - left > 1) {
13        int middle = (left + right) / 2;
14        if (fcriteria(a[middle]))
15            right = middle;
16        else
17            left = middle;
18    }
19    return right;
20 }
21
22
23
24 int main() {
25     int a[] = {2, 5, 8, 13, 21, 27, 35};
26     int n = 7;
27
28     printf("%d", a[binarySearchCriteria(a, n, isMore20)]);
29
30     return 0;
31 }
```

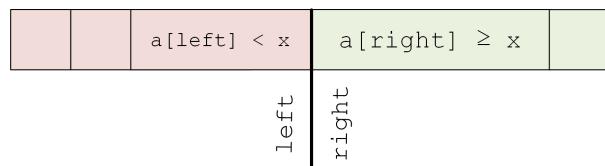
10.7 Решение задач на бинарный поиск

10.7.1 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 -1 , 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 . В конце добавим проверку:

```
1 right != n && a[right] == x
```

по которой сможем понять, имеется ли искомый элемент в массиве. Реализация:

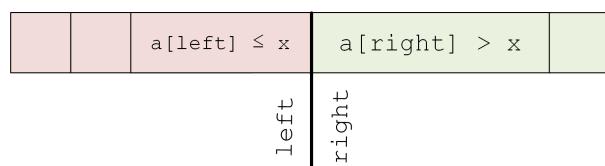
```
1 int binarySearchFirstEqual(const int * const a, const size_t n,
2                                const int x) {
3     int left = -1;
4     int right = n;
5     while (right - left > 1) {
6         int middle = left + (right - left) / 2;
7         if (a[middle] < x)
8             left = middle;
9         else
10            right = middle;
11    }
12    return (right != n && a[right] == x) ? right : -1;
13 }
```

10.7.2 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:

```

1 int binarySearchLastEqual(const int * const a, const size_t n,
2                             const int x) {
3     int left = -1;
4     int right = n;
5     while (right - left > 1) {
6         int middle = left + (right - left) / 2;
7         if (a[middle] <= x)
8             left = middle;
9         else
10            right = middle;
11    }
12    return (left != -1 && a[left] == x) ? left : -1;
13 }
```

10.7.3 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:

```

1 #include <stdio.h>
2 #include "libs/algorithms/array/array.h"
3
4 long long countRopes(int *ropeLengths, size_t n, double oneRopeLength) {
5     long long count = 0;
6     for (size_t i = 0; i < n; i++)
7         count += ropeLengths[i] / oneRopeLength;
8     return count;
9 }
10
11 int main() {
12     int n, k;
13     scanf("%d %d", &n, &k);
14
15     int ropeLengths[10000];
16     inputArray(ropeLengths, n);
17
18     double left = 0;
19     double right = 1e18;
20     double eps = 1e-7;
21     while (right - left > eps) {
22         double middle = left + (right - left) / 2;
23         if (countRopes(ropeLengths, n, middle) >= k)
24             left = middle;
25         else
26             right = middle;
```

```

27     }
28     printf("%.6f", left);
29
30     return 0;
31 }
```

10.7.4 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:

```

1 #include <stdio.h>
2
3 void order2(long long *a, long long *b) {
4     if (*a > *b) {
5         long long t = *a;
6         *a = *b;
7         *b = t;
8     }
9 }
10
11 long long countCopies(long long firstXeroxTime,
12                         long long secondXeroxTime,
13                         long long sumTime) {
14     order2(&firstXeroxTime, &secondXeroxTime);
15     return sumTime / firstXeroxTime +
16            (sumTime - firstXeroxTime) / secondXeroxTime;
17 }
18
19 int main() {
20     long long n, x, y;
21     scanf("%lld %lld %lld", &n, &x, &y);
22
23     long long left = 0;
24     long long right = 1e18;
25     while (right - left > 1) {
26         long long middle = left + (right - left) / 2;
27         if (countCopies(x, y, middle) >= n)
28             right = middle;
29         else
30             left = middle;
31     }
32
33     printf("%lld", right);
34
35     return 0;
36 }
```

10.7.5 Задача об уравнении

Задача об уравнении

Найдите такое число x , что $x^2 + \sqrt{x} = C$ ($C \geq 1$).

Функция является монотонно возрастающей и все её значения положительны.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 double f(double x) {
5     return x*x + sqrt(x);
6 }
7
8 int main() {
9     double C;
10    scanf("%lf", &C);
11
12    double left = 0;
13    double right = 1e16;
14    while (right - left > 1e-7) {
15        double middle = left + (right - left) / 2;
16        if (f(middle) > C)
17            right = middle;
18        else
19            left = middle;
20    }
21
22    printf("%f", right);
23
24    return 0;
25 }
```

10.7.6 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 время 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 10.6 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.

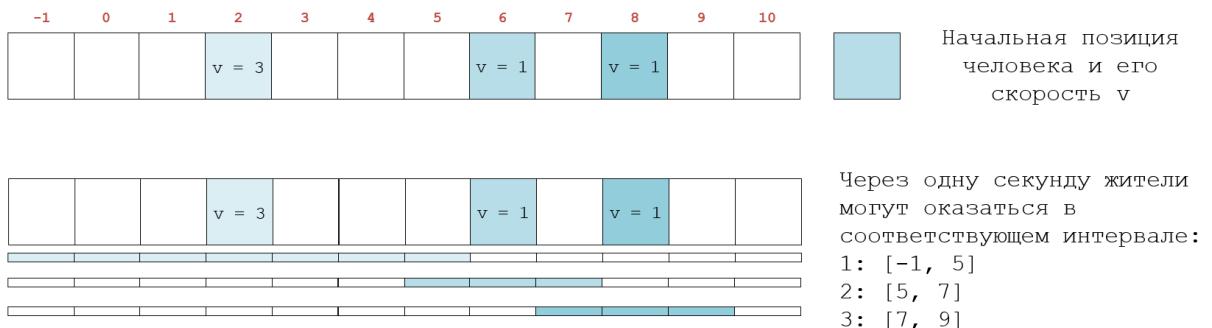


Рис. 10.6 – Начальное расположение жителей и их возможное положение через секунду

```

1 #include <stdio.h>
2 #include <float.h>
3
4 double max(double a, double b) { return a > b ? a : b; }
5
6 double min(double a, double b) { return a < b ? a : b; }
7
8 int isPossible(int *speeds, int *positions, int nPeople, double time) {
9     double left = -1e20;
10    double right = 1e20;
11    for (int peopleIndex = 0; peopleIndex < nPeople; peopleIndex++) {
12        double possibleLeft = positions[peopleIndex]
13                           - speeds[peopleIndex]*time;
14        double possibleRight = positions[peopleIndex]
15                           + speeds[peopleIndex]*time;
16        left = max(left, possibleLeft);
17        right = min(right, possibleRight);
18    }
19    return left <= right;
20 }
21
22 int main() {
23     int n;
24     scanf("%d", &n);
25
26     int coordinates[n], speeds[n];
27     for (int i = 0; i < n; i++)
28         scanf("%d %d", &coordinates[i], &speeds[i]);
29
30     double badTime = 0;
31     double goodTime = 1e10;
32     const double eps = 0.000001;
33     while (goodTime - badTime > eps) {
34         double middleTime = (badTime + goodTime) / 2;
35         if (isPossible(speeds, coordinates, n, middleTime))
36             goodTime = middleTime;
37         else
38             badTime = middleTime;
39     }
40
41     printf("%f", goodTime);
42
43     return 0;
44 }
```

10.7.7 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`.

```

1 int isPossible(const int *coordinates, int nCoordinates, int distance,
2               int nCows) {
3     // загоняем корову в первое стойло
4     int lastPosIndex = 0;
5     int lastCoordinateWithCow = coordinates[lastPosIndex];
6     nCows--;
7     for (int coordinateIndex = 1;
8          coordinateIndex < nCoordinates;
9          coordinateIndex++) {
10        int curCoordinate = coordinates[coordinateIndex];
11        int currentDistance = curCoordinate - lastCoordinateWithCow;
12        // если расстояние между текущим стойлом и стойлом с прошлой коровой
13        // стала не меньше проверяемого расстояния - загоняем корову в стойло
14        if (currentDistance >= distance) {
15            lastPosIndex = coordinateIndex;
16            lastCoordinateWithCow = coordinates[lastPosIndex];
17            nCows--;
18        }
19    }
20    return nCows <= 0;
}

```

Возможно, что для проверяемого расстояния `distance` можно будет разместить большее количество коров, поэтому возвращаемое значение

```
19     return nCows <= 0;
```

10.7.8 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 :

a	0	1	2	3	4	5	6	7	8
	6	7	5	3	2	6	4	1	9

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

$$\begin{cases} r - l \geq D \\ p[r] - p[l] \geq 0 \end{cases} \Leftrightarrow \begin{cases} l \leq r - D \\ p[l] \leq p[r] \end{cases} \quad (10.1)$$

Зафиксируем произвольное r и попытаемся определить, имеется ли хоть какое-нибудь l удовлетворяющее ограничениям выше. Пусть $r = 9$. Рассмотрим все $a_i : l \leq r - D$:

	0	1	2	3	4	5	6	7	8	9
p	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 $x' \geq 5$:

	0	1	2	3	4	5	6	7	8
a	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 10.1. 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
m	0	0	0	0	0	-2	-2	-2	-6	-6

Имея такой массив, можно утверждать, что имеется пара (l, r) , удовлетворяющая ограничениям 10.1 если

$$m_{r-D} \leq p_r$$

```

1 // код представлен для случая, когда нумерация элементов массива ведётся с 1
2 // в lBorder и rBorder записываются границы искомого интервала включительно.
3
4 double min(const double a, const double b) { return a < b ? a : b; }
5
6 int getBound(const int *a, const size_t n, double x, const int d,
7           int *lBound, int *rBound) {
8     double b[n];
9     for (int i = 0; i < n; i++)
10        b[i] = a[i] - x;
11
12    double p[n + 1];
13    p[0] = 0;
14    for (int i = 1; i <= n; i++)
15        p[i] = p[i - 1] + b[i - 1];
16
17    double m[n + 1];
18    m[0] = p[0];

```

```

19     for (int i = 1; i <= n; i++)
20         m[i] = min(m[i - 1], p[i]);
21
22     for (int r = d; r <= n; r++)
23         if (m[r - d] <= p[r]) {
24             *rBound = r;
25             // поиск левой границы
26             int l = r - d;
27             while (l >= 0 && m[l] <= m[r - d])
28                 l--;
29             *lBound = l + 1;
30             return 1;
31         }
32     return 0;
33 }
34
35 void getBorders(const int *a, const size_t n, const int d,
36                  int *lBorder, int *rBorder) {
37     double minX = 0;
38     double maxX = 1e9;
39     while (maxX - minX > 0.000001) {
40         double x = (minX + maxX) / 2;
41         int r = -1;
42         int l = -1;
43         if (getBound(a, n, x, d, &l, &r))
44             minX = x;
45         else
46             maxX = x;
47         if (r != -1) {
48             *rBorder = r;
49             *lBorder = l;
50         }
51     }
52 }
```

10.8 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:

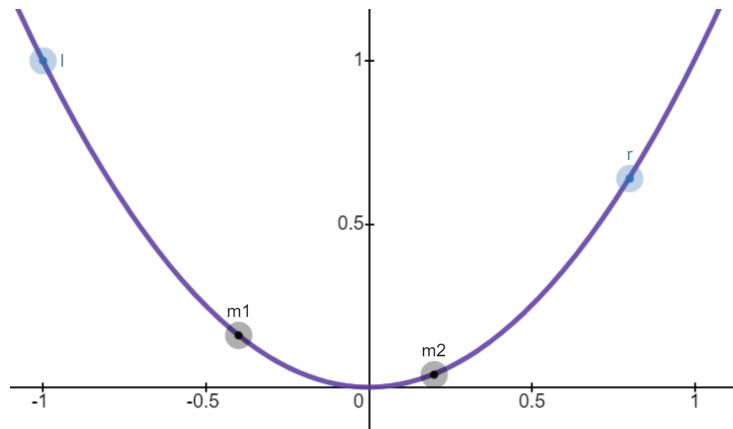


Рис. 10.7 – Так как $m_1 > m_2$, искомая точка находится между m_1 и r , поэтому на данной итерации $l := m_1$

На следующей итерации для данного примера получим другую картину:

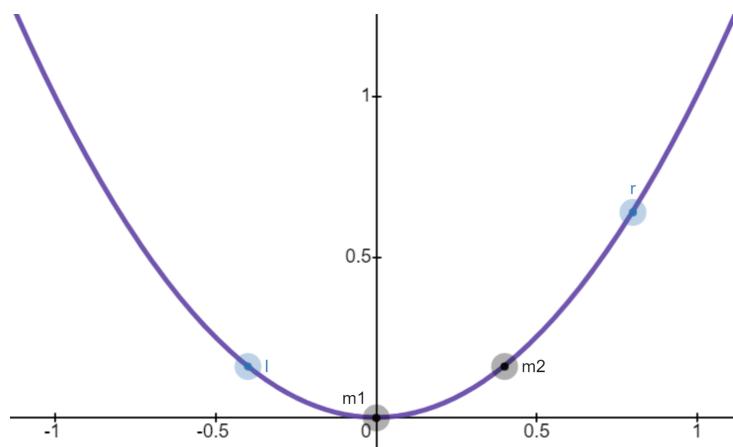


Рис. 10.8 – Так как $m_1 \leq m_2$, искомая точка находится между l и m_2 , поэтому на данной итерации $r := m_2$

Опишем алгоритм тернарного поиска для нахождения минимума на отрезке:

```

1 double l = ...;           // левая граница интервала
2 double r = ...;           // правая граница интервала
3 double eps = 0.0000001    // погрешность измерения

```

```
4 while (r - l > eps) {  
5     double m1 = l + (r - l) / 3;  
6     double m2 = l + 2*(r - l) / 3;  
7     if (m1 > m2)  
8         l = m1;           // ситуация на рисунке 10.7  
9     else  
10        r = m2;          // ситуация на рисунке 10.8  
11 }
```

Для ситуации, когда выполняется поиск максимума меняется лишь знак в 7 строке:

```
7     if (m1 < m2)  
8         l = m1;  
9     else  
10        r = m2;
```

Глава 11

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.

11.1 *BogoSort*

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. 11.1 – 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.

Вы можете проверить эту теорию посредством кода:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // обмен элементов по адресу a и b
5 void swap(int *a, int *b) {
6     int t = *a;
7     *a = *b;
8     *b = t;
9 }
10
11 // проверка упорядоченности массива
12 int isSorted(const int *a, size_t size) {
13     for (int i = 1; i < size; i++)
14         if (a[i - 1] > a[i])
15             return 0;
16     return 1;
17 }
18
19 // перемешивание элементов массива
20 void shuffleArray(int *a, size_t size) {
21     for (size_t i = 0; i < size; ++i)
22         swap(&a[i], &a[(rand() % size)]);
23 }
24
25 // сортировка, возвращает количество перемешиваний
26 int bogosort(int *a, size_t size) {
27     int count = 0;
28     while (!isSorted(a, size)) {
29         shuffleArray(a, size);
30         count += 1;
31     }
32     return count;
33 }
34
35 // эксперимент
36 int main() {
37     int a[] = {6, 4, 2, 5, 7, 1, 3};
38     int n = 7;
39
40     double s = 0;
41     double nExpr = 500000.0;
42     for (int i = 0; i < nExpr; i++) {
43         s += bogosort(a, n);
44         shuffleArray(a, n);
45     }
46
47     printf("%f", s / nExpr);
48
49     return 0;
50 }
```

Запустив его на своей машине, я получил значение 5033.897984 генерации массивов в среднем.

11.2 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 в небольшом количестве операций обмена. Массив разделяется на 2 части: отсортированную и неотсортированную. На i -ой итерации цикла происходит следующее:

- Выполняется поиск позиции minPos – позиции минимального элемента среди элементов в неотсортированной части (начиная с индекса i , заканчивая $n - 1$).
- Элементы на позиции i и minPos обмениваются.

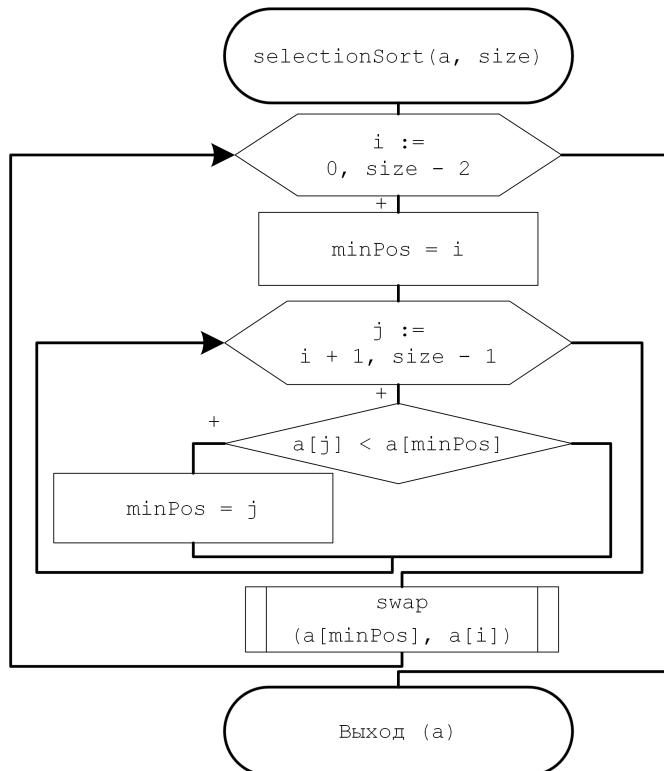


Рис. 11.2 – Блок-схема сортировки выбором

```

1 void selectionSort(int *a, const int size) {
2     for (int i = 0; i < size - 1; i++) {
3         int minPos = i;
4         for (int j = i + 1; j < size; j++)
5             if (a[j] < a[minPos])
6                 minPos = j;
7         swap(&a[i], &a[minPos]);
8     }
9 }
```

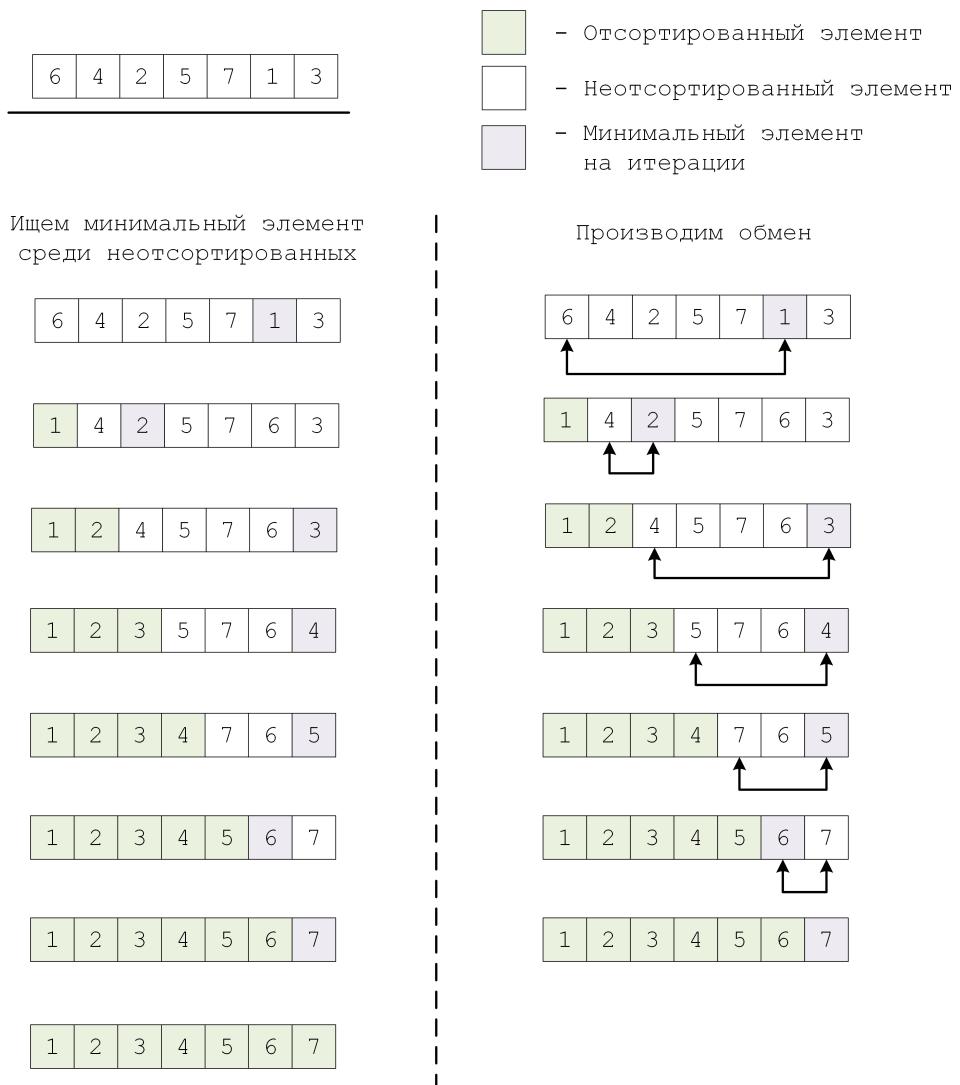


Рис. 11.3 – Сортировка выбором

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 $O(n^2)$.

11.3 Сортировка вставками

Сортировка вставками

Дан массив a размера n . Необходимо упорядочить элементы массива по неубыванию сортировкой вставками.

Сложность по времени: $\mathbf{O}(n^2)$, $\Omega(n)$.

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 11.5. Блок-схема на рисунке 11.4.

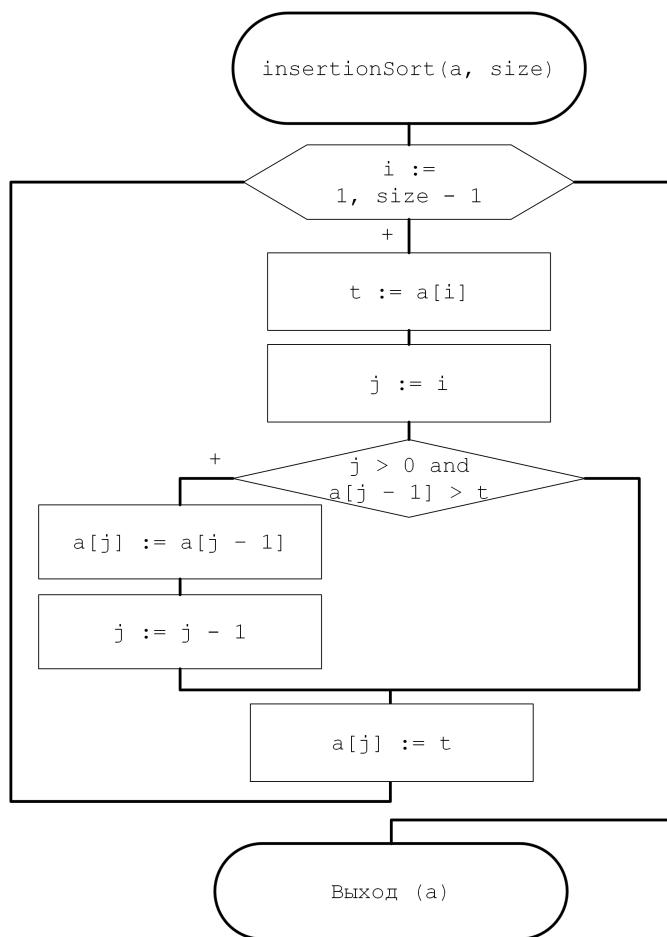


Рис. 11.4 – Блок-схема сортировки вставками

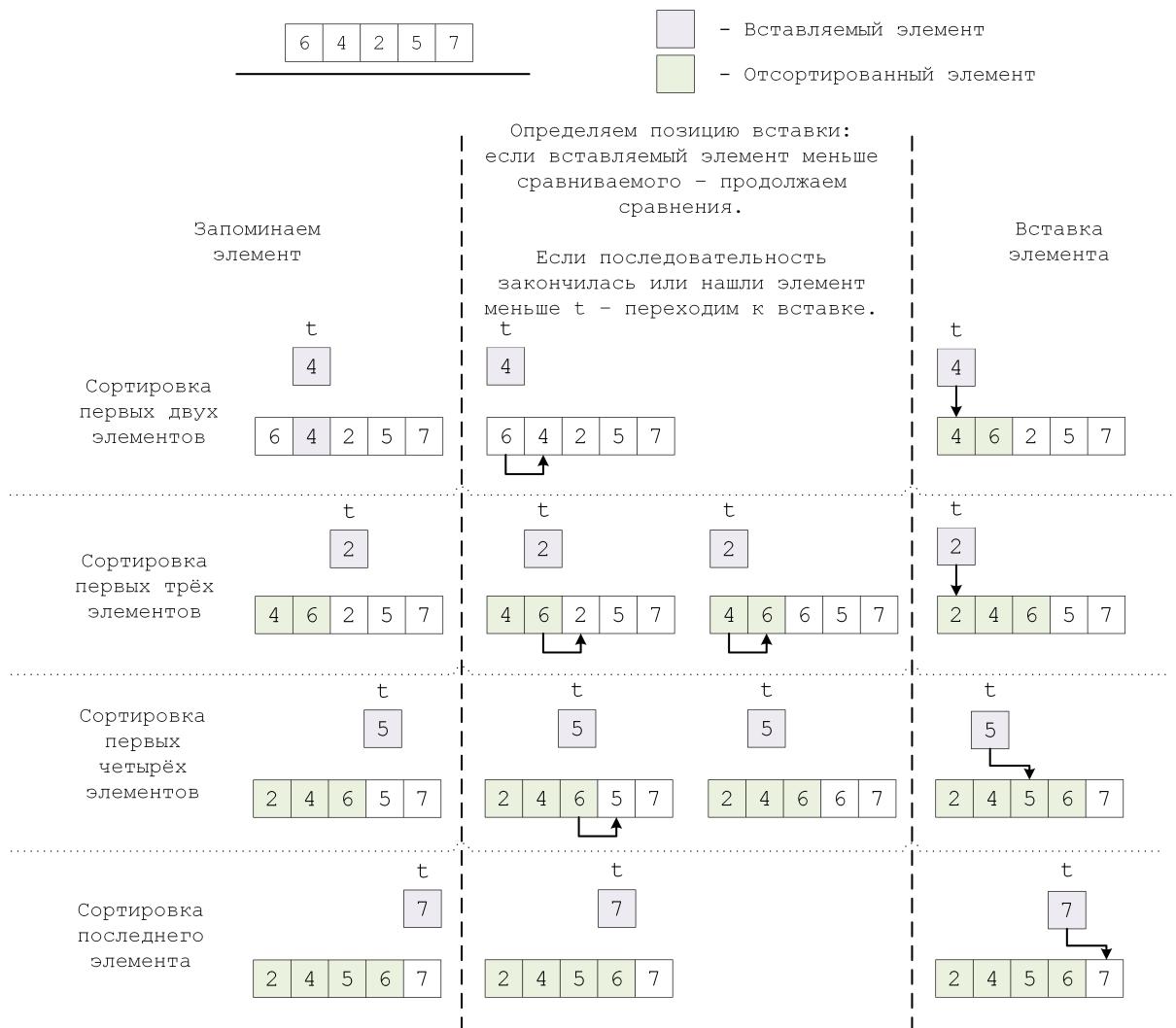


Рис. 11.5 – Сортировка вставками

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).

```

1 void insertionSort(int *a, const size_t size) {
2     for (size_t i = 1; i < size; i++) {
3         int t = a[i];
4         int j = i;
5         while (j > 0 && a[j - 1] > t) {
6             a[j] = a[j - 1];
7             j--;
8         }
9         a[j] = t;
10    }
11 }
```

11.4 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».

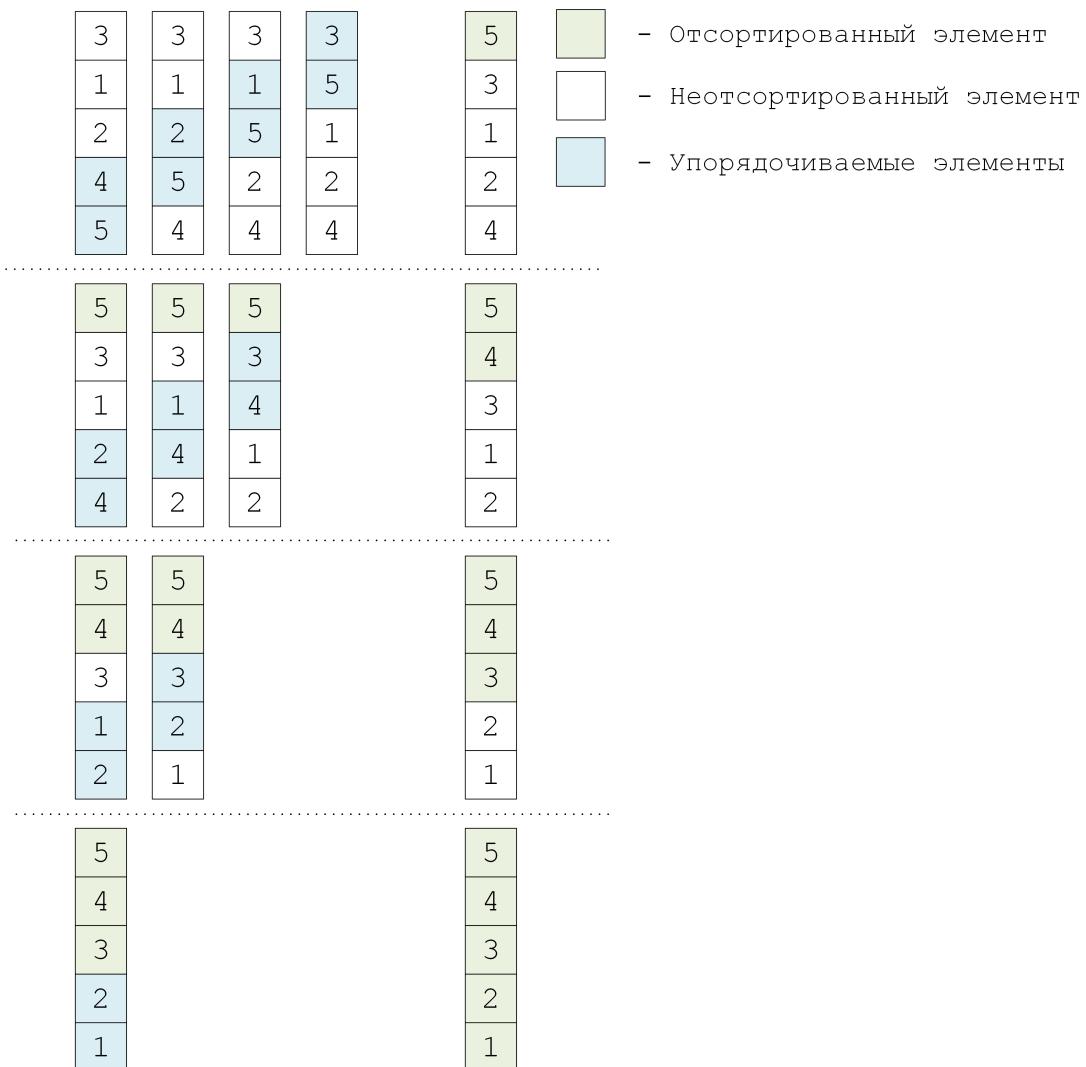


Рис. 11.6 – Идея пузырьковой сортировки

Возможная реализация (сдвигает минимум вниз):

```

1 void bubbleSort(int *a, size_t size) {
2     for (size_t i = 0; i < size - 1; i++)
3         for (size_t j = size - 1; j > i; j--)
4             if (a[j - 1] > a[j])
5                 swap(&a[j - 1], &a[j]);
6 }
```

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.

11.5 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:

$$k = \frac{1}{1 - \frac{1}{e^{-\phi}}}$$

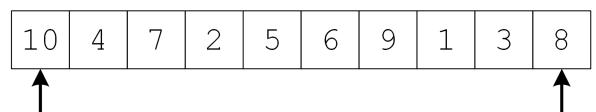
где e — основание натурального логарифма, а ϕ — золотое сечение.

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.

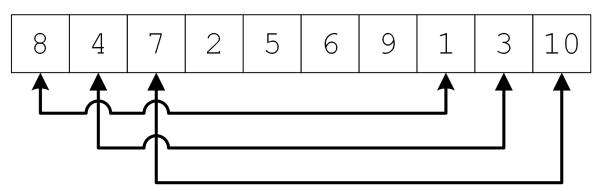
```

1 void combsort(int *a, const size_t size) {
2     size_t step = size;
3     int swapped = 1;
4     while (step > 1 || swapped) {
5         if (step > 1)
6             step /= 1.24733;
7         swapped = 0;
8         for (size_t i = 0, j = i + step; j < size; ++i, ++j)
9             if (a[i] > a[j]) {
10                 swap(&a[i], &a[j]);
11                 swapped = 1;
12             }
13     }
14 }
```

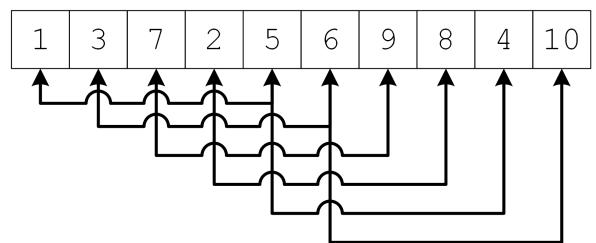
Шаг 1 ($t = n - 1 = 9$)



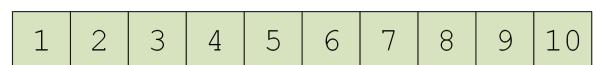
Шаг 2 ($t := t / 1.2473 = 7$)



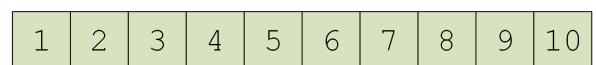
Шаг 3 ($t := t / 1.2473 = 5$)



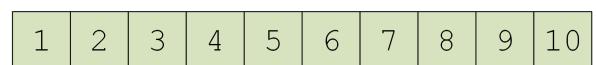
Шаг 4 ($t := t / 1.2473 = 4$)



Шаг 5 ($t := t / 1.2473 = 3$)



Шаг 6 ($t := t / 1.2473 = 2$)



Шаг 7 ($t := t / 1.2473 = 1$)

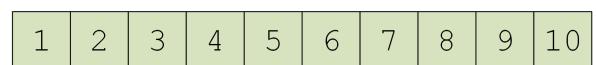


Рис. 11.7 – Изменение шага при сортировке расческой

11.6 Сортировка подсчётом

Сортировка подсчётом

Дан массив a размера n . Необходимо упорядочить элементы массива по неубыванию сортировкой подсчётом.

Сложность по времени: $\mathbf{O(n + k)}$,

Требуется дополнительная память: $\mathbf{O(k)}$, 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 множество 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 :

$$a = \{1, 1, 1, 2, 2, 2, 2, 3, 3, 3\}$$

```

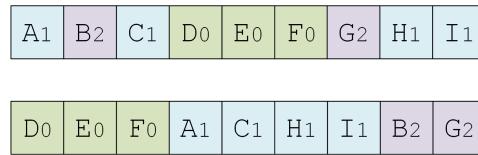
1 #include <stdlib.h>
2
3 void getMinMax(const int *a, size_t size, int *min, int *max) {
4     *min = a[0];
5     *max = a[0];
6     for (int i = 1; i < size; i++) {
7         if (a[i] < *min)
8             *min = a[i];
9         else if (a[i] > *max)
10            *max = a[i];
11    }
12 }
13
14 void countSort(int *a, const size_t size) {
15     int min, max;
16     getMinMax(a, size, &min, &max);
17     int k = max - min + 1;
18
19     // выделение памяти под динамический массив из k элементов,
20     // где каждый из элементов равен 0
21     int *b = (int *)calloc(k, sizeof(int));
22     for (int i = 0; i < size; i++)

```

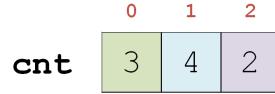
```

23     b[a[i] - min]++;
24
25     int ind = 0;
26     for (int i = 0; i < k; i++) {
27         int x = b[i];
28         for (int j = 0; j < x; j++)
29             a[ind++] = min + i;
30     }
31
32     // освобождение памяти, выделенной под динамический массив
33     free(b);
34 }
```

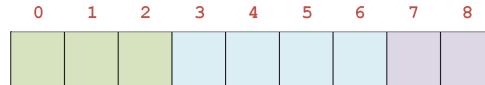
Рассмотрим такой случай. Пусть элементы массива – это не просто числа, а какие-то более сложные объекты, которые требуется отсортировать по ключу. От того, что мы отсортируем только одни ключи, не отсортируются объекты. Укажем объекты в виде букв, а в качестве индекса будет выступать значение ключа. Исходный массив и его отсортированная версия представлены на рисунке ниже:



Посчитаем, сколько раз встречался тот или иной ключ:



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 $p[i]$. Сортировка подсчётом полностью выглядит так:

```

1 countSort(a)
2     for i = 0..n-1:
3         k = a[i].key
4         cnt[k]++
5     p[0] = 0;
6     // m - разброс значений для ключей
7     for i = 1..m-1:
8         p[i] = p[i - 1] + cnt[i - 1]
9     for i = 0..n-1:
10        k = a[i].key
11        b[p[k]++] = a[i]

```

и последний этап представлен на рисунке 11.8.

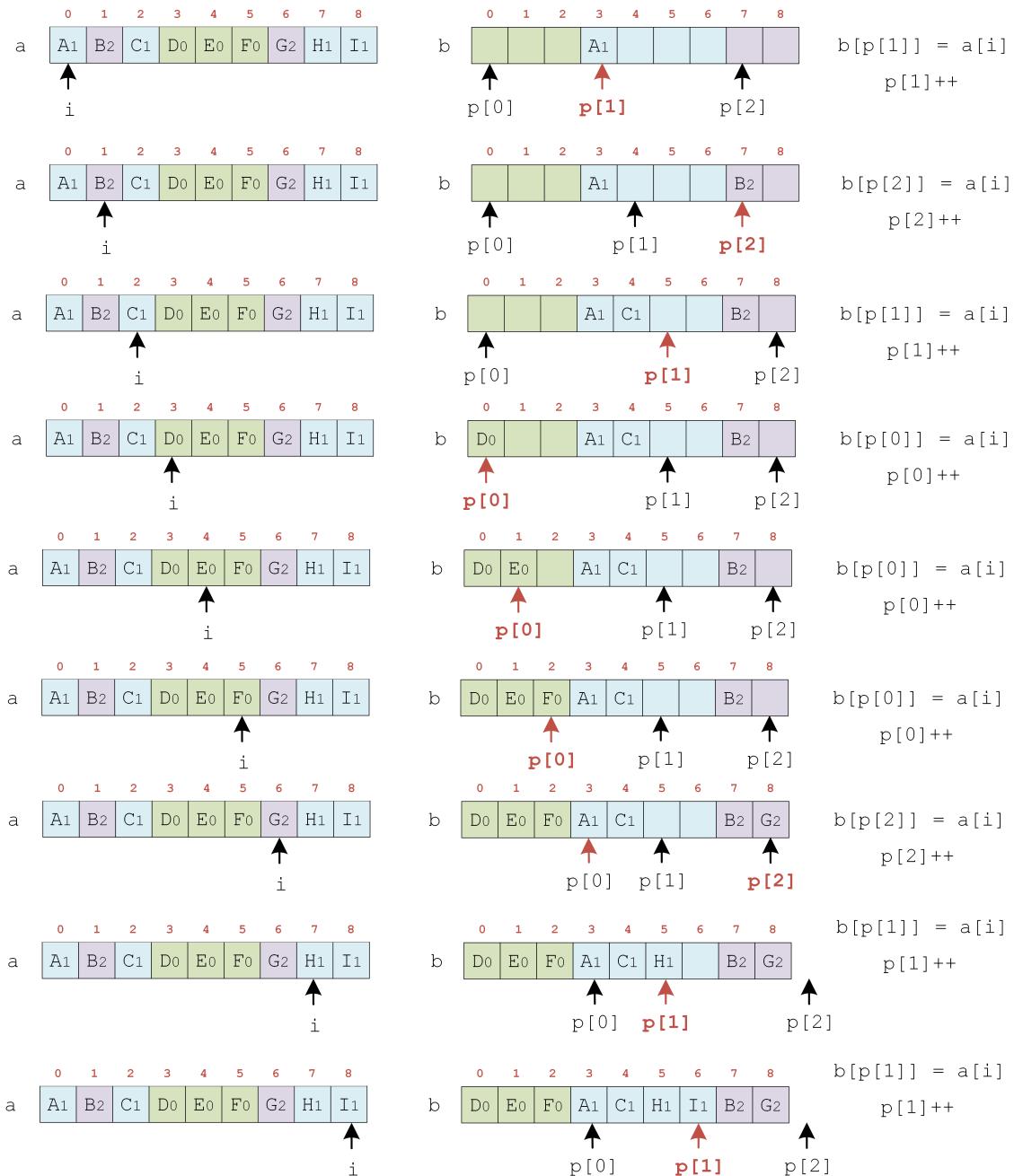


Рис. 11.8 – Последний этап сортировки

Важной особенностью сортировки является её устойчивость. Данный аспект будет использован для реализации цифровой сортировки.

11.7 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

0	1	2	3	4	5	6	7	8	9
01100011	11001010	01100110	11011001	11110000	10100101	00111100	00001010	01010110	10101111

cnt	0	1	2	3	4	5	6	7	8	9		
	2	2	4	2					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

cnt	0	1	2	3	4	5	6	7	8	9		
	2	3	3	2					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	
cnt	2	2	4	2		p	0	2	4	8
0	1	2	3	4	5	6	7	8	9	
110001010	00001010	01010110	11011001	01100011	10100101	01100110	10101111	11110000	00111100	
00	01	10	11			00	01	10	11	
cnt	2	3	2	3		p	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.

11.8 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.

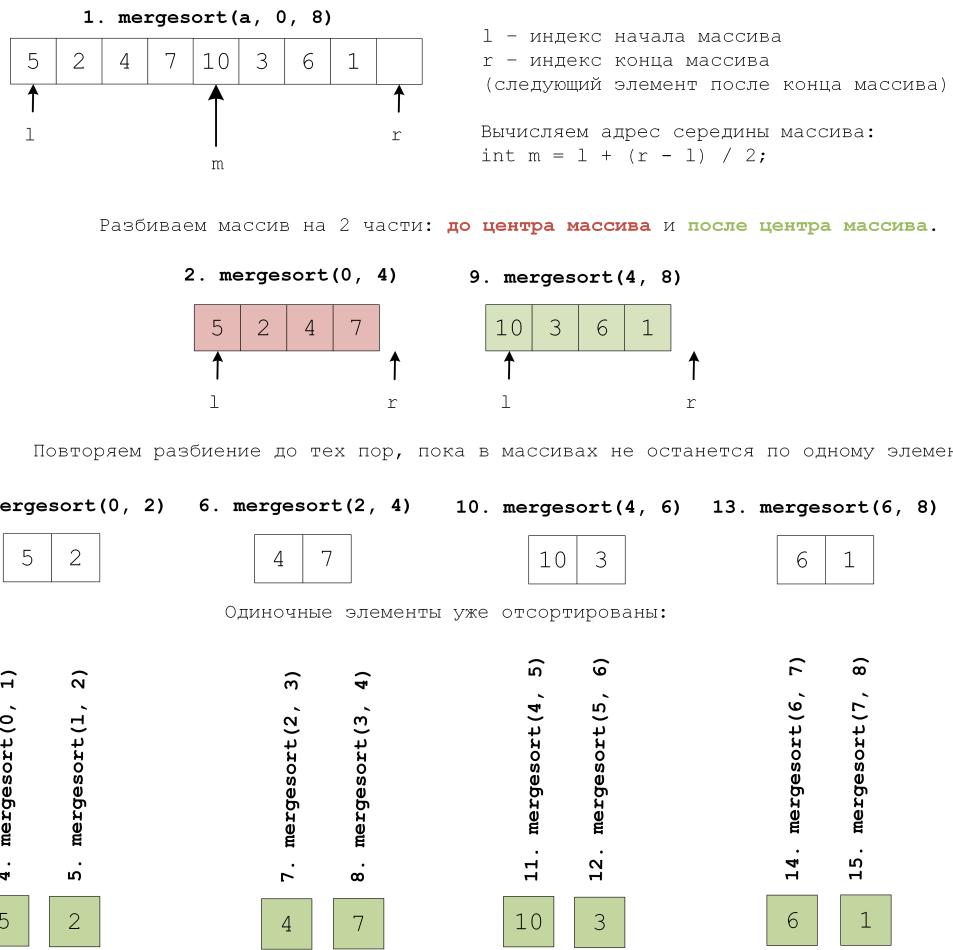
Опишем функцию, которая выполняла заявленные действия:

```

1 void merge(const int *a, const int n,
2            const int *b, const int m, int *c) {
3     int i = 0, j = 0;
4     while (i < n || j < m) {
5         if (j == m || i < n && a[i] < b[j]) {
6             c[i + j] = a[i];
7             i++;
8         } else {
9             c[i + j] = b[j];
10            j++;
11        }
12    }
13 }
```

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.

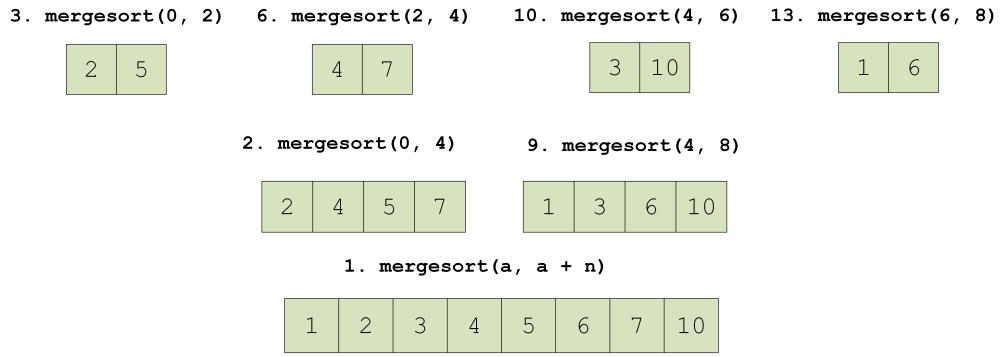


Рис. 11.9 – Сортировка слиянием

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 $O(n)$.

Реализация:

```

1 #include <stdio.h>
2 #include <malloc.h>
3 #include <memory.h>
4
5 void merge(const int *a, const int n,
6            const int *b, const int m, int *c) {
7     int i = 0, j = 0;
8     while (i < n || j < m) {
9         if (j == m || i < n && a[i] < b[j]) {
10             c[i + j] = a[i];
11             i++;
12         } else {
13             c[i + j] = b[j];
14             j++;
15         }
16     }
17 }
18
19 void mergeSort_(int *source, int l, int r, int *buffer) {
20     int n = r - l;
21     if (n <= 1)
22         return;
23
24     // определяем середину последовательности
25     // и рекурсивно вызываем функцию сортировки для каждой половины
26     int m = (l + r) / 2;
27     mergeSort_(source, l, m, buffer);
28     mergeSort_(source, m, r, buffer);
29
30     // производим слияние элементов, результат сохраняем в буфере
31     merge(source + l, m - l, source + m, r - m, buffer);
32     // переписываем сформированную последовательность с буфера
33     // в исходный массив
34     memcpy(source + l, buffer, sizeof(int) * n);
35 }
36
37 void mergeSort(int *a, int n) {
38     // создаём буфер из которого будут браться элементы массива
39     int *buffer = (int*)malloc(sizeof(int) * n);
40     mergeSort_(a, 0, n, buffer);
41     free(buffer);
42 }
43
44 int main() {
45     int a[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
46     size_t n = sizeof(a) / sizeof(int);
47
48     mergeSort(a, n);
49     for (int i = 0; i < n; i++)
50         printf("%d ", a[i]);
51
52     return 0;
53 }
```

11.9 Куча и сортировка кучей

Структура данных, которая хранит некоторое множество элементов. Поддерживает операции:

- `insert(x)` - объединить элемент с множеством.
- `getMin()` - возвращает минимальный элемент в множестве A .
- `removeMin()` - удаляет минимальный элемент из множества A . Если минимальный несколько, возвращается любое значение.

Рассмотрим несколько реализаций на массиве:

```

1 insert(x): // O(1)
2     a[n++] = x;
3
4 getMin(): // O(n)
5     res = infinity
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 11.10. Дерево заполнено по уровням. Заполнены будут все уровни (кроме, возможно, последнего).

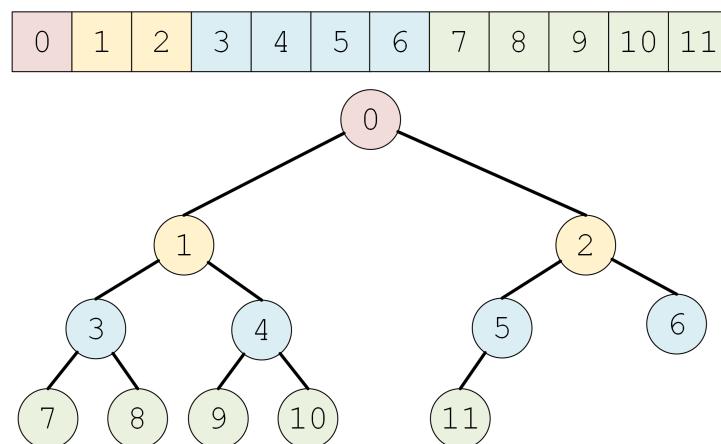


Рис. 11.10 – Соответствие элемента массива узлу дерева

Показанное дерево на рисунке 11.10 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 11.11).

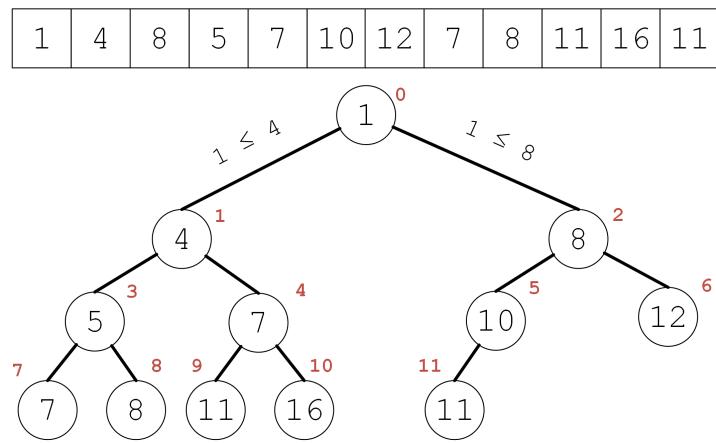


Рис. 11.11 – Пример кучи

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 $a[0]$:

```
1 getMin(): // O(1)
2     return a[0];
```

Опишем вставку. Добавим элемент в конец массива:

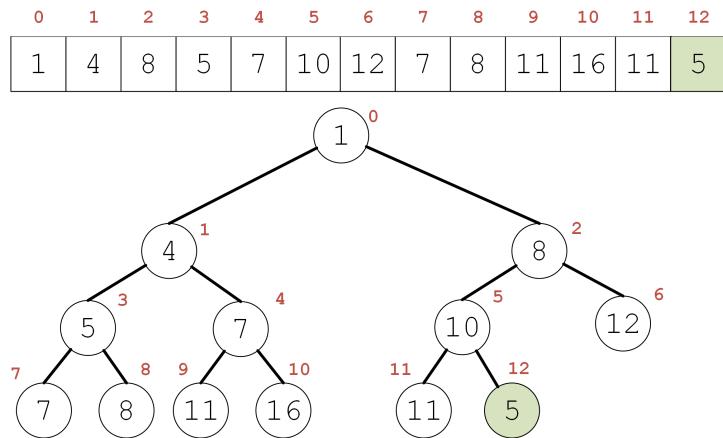


Рис. 11.12 – Состояние кучи после вставки в конец

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:

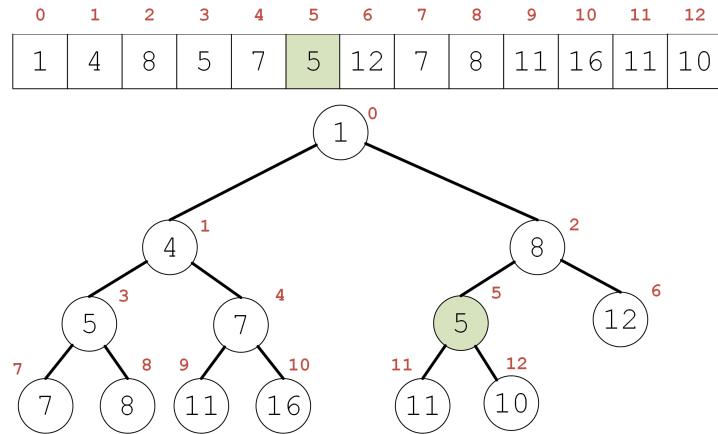


Рис. 11.13 – Обмен элементов на 5 и 12 позициях

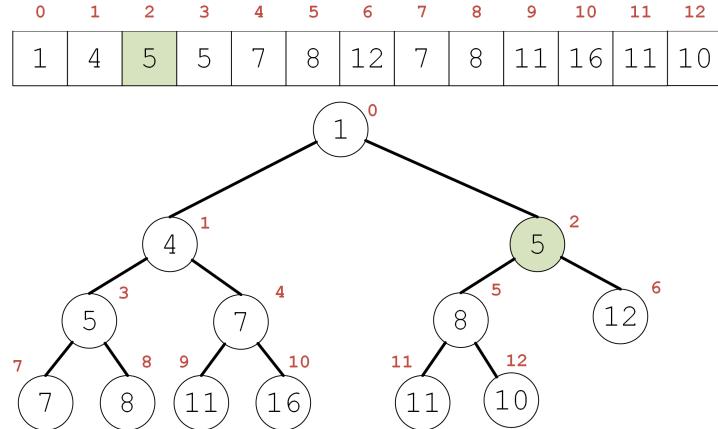


Рис. 11.14 – Обмен элементов на 2 и 5 позициях

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:

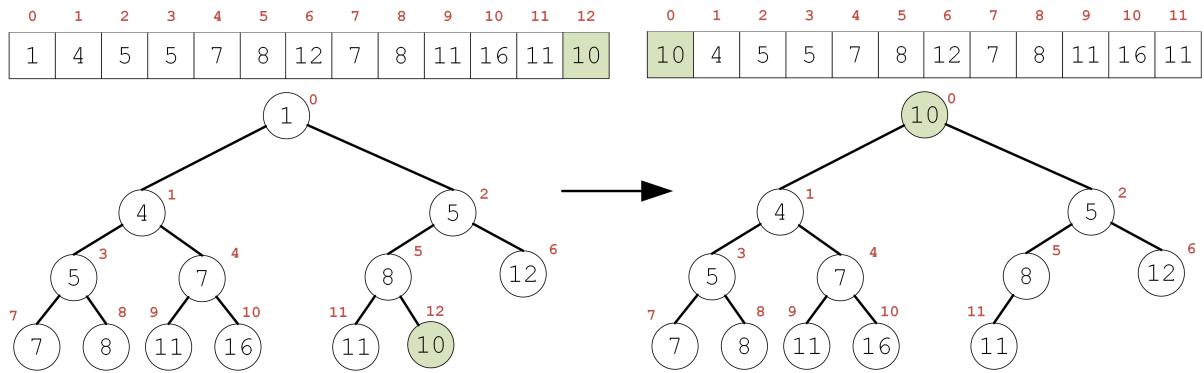


Рис. 11.15 – Перенос последнего элемента в корень

- Сравниваем вставляемое значение с потомками. Если находится потомок с меньшим значением - производится обмен:

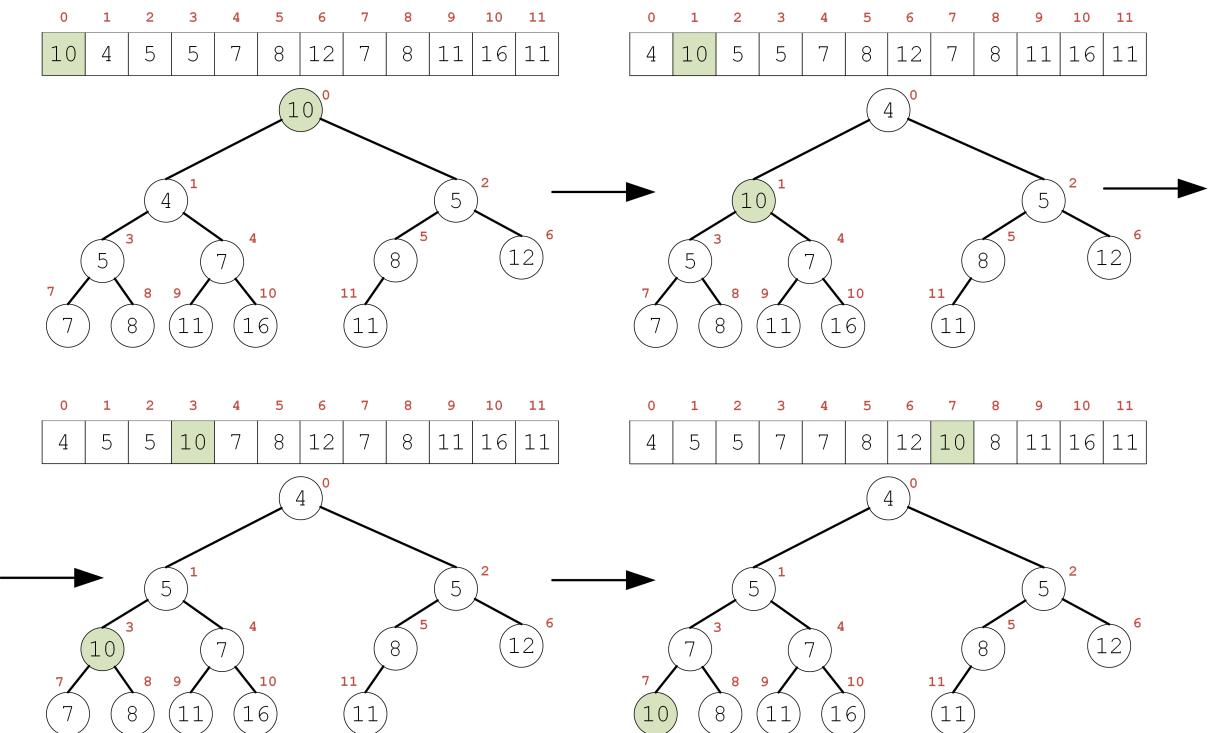


Рис. 11.16 – Проталкивание элемента

```

1 getMinChildIndex(parentIndex):
2     int leftChild = 2 * parentIndex + 1
3     int 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    int 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:

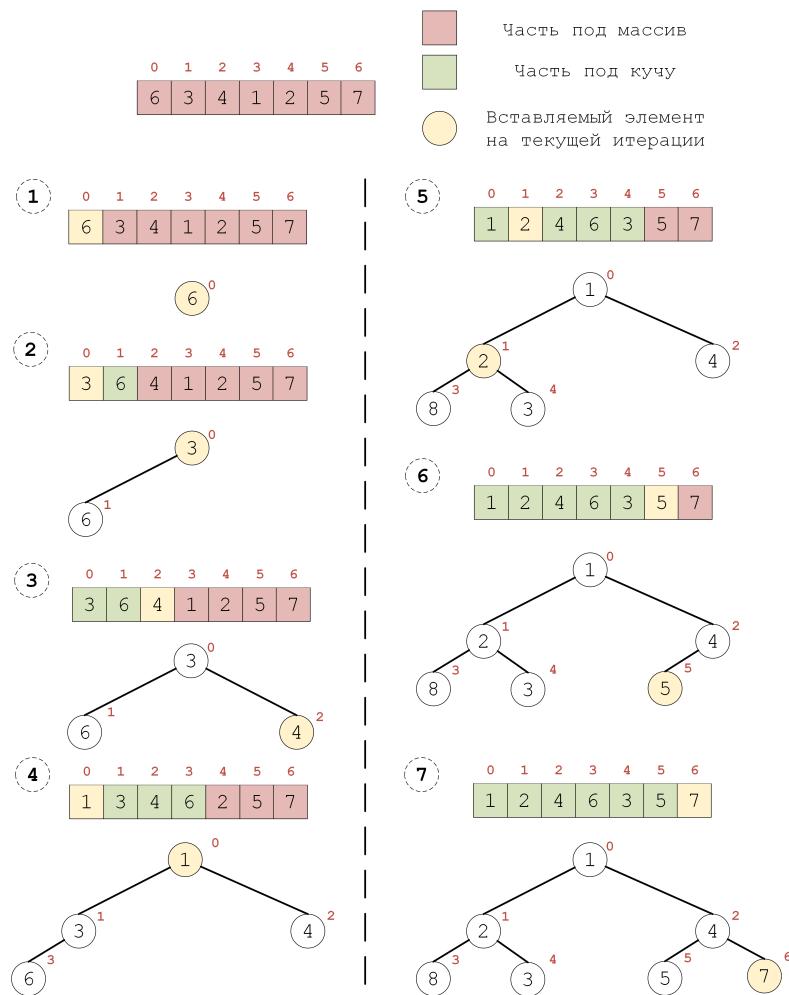


Рис. 11.17 – Процесс построения кучи на массиве a

После этого выполним удаление минимальных элементов из кучи, пока она не станет пуста. Иллюстрация этого представлена на рисунке ??.

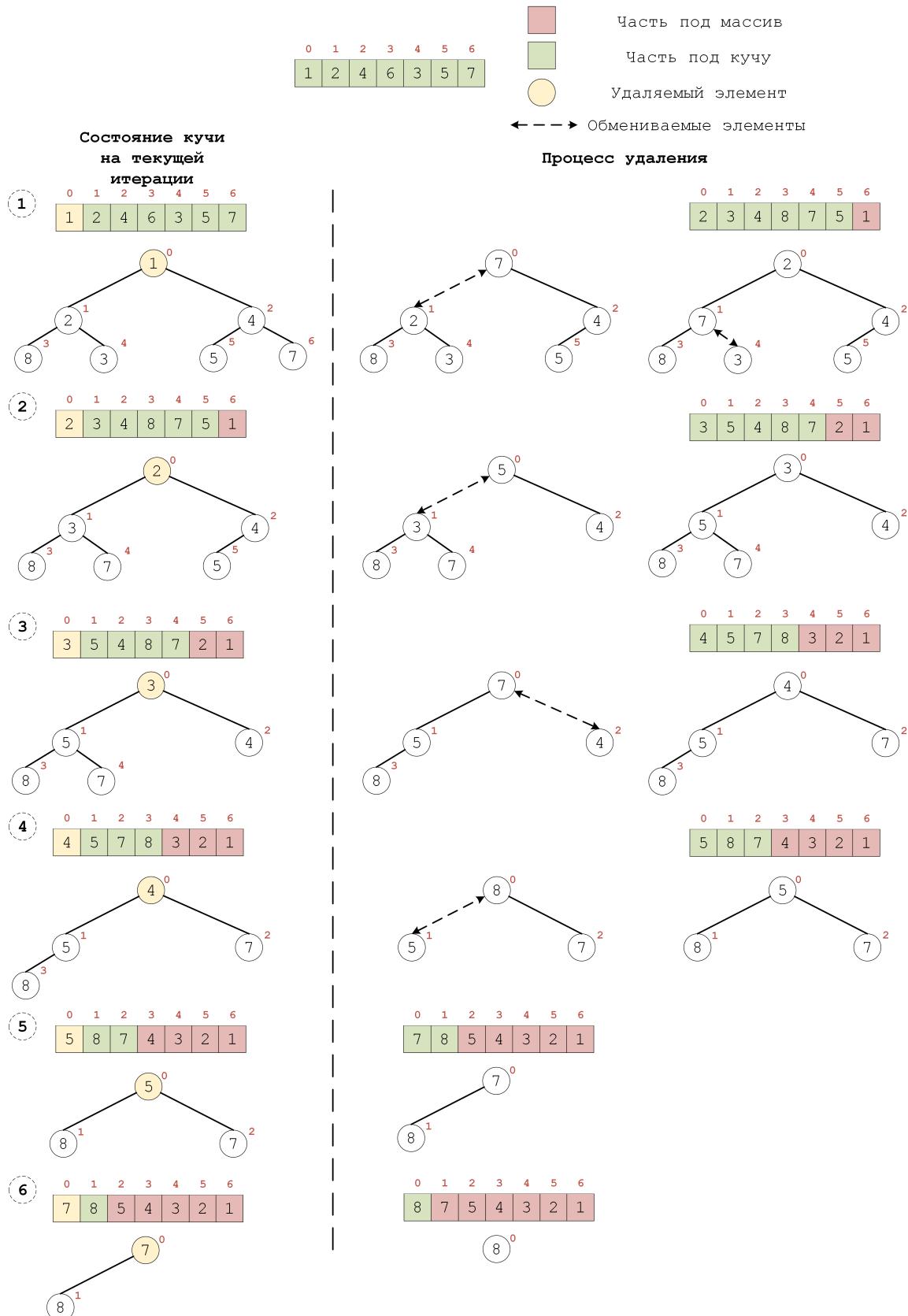


Рис. 11.18 – Процесс удаления элементов из кучи. Можно заметить, что массив сортируется по невозрастанию

Реализуем функцию вставки элемента в кучу, удаление элемента из кучи и алгоритм сортировки:

```

1 #include <stdbool.h>
2
3 void swap(int *a, int *b) {
4     int t = *a;
5     *a = *b;
6     *b = t;
7 }
8
9 void insertHeap(int *a, size_t *size, int x) {
10    a[(*size)++] = x;
11    size_t childIndex = *size - 1;
12    size_t parentIndex = (childIndex - 1) / 2;
13    while (a[childIndex] < a[parentIndex] && childIndex != 0) {
14        swap(&a[childIndex], &a[parentIndex]);
15        childIndex = parentIndex;
16        parentIndex = (childIndex - 1) / 2;
17    }
18 }
19
20 bool hasLeftChild(size_t parentIndex, size_t size) {
21     return 2 * parentIndex + 1 < size;
22 }
23
24 bool hasRightChild(size_t parentIndex, size_t size) {
25     return 2 * parentIndex + 2 < size;
26 }
27
28 size_t getLeftChildIndex(size_t parentIndex) {
29     return 2 * parentIndex + 1;
30 }
31
32 size_t getMinChildIndex(const int *a, size_t size, size_t parentIndex) {
33     size_t minChildIndex = getLeftChildIndex(parentIndex);
34     size_t rightChildIndex = minChildIndex + 1;
35     if (hasRightChild(parentIndex, size))
36         if (a[rightChildIndex] < a[minChildIndex])
37             minChildIndex = rightChildIndex;
38     return minChildIndex;
39 }
40
41 void removeMinHeap(int *a, size_t *size) {
42     *size -= 1;
43     swap(&a[0], &a[*size]);
44     size_t parentIndex = 0;
45     while (hasLeftChild(parentIndex, *size)) {
46         size_t minChildIndex = getMinChildIndex(a, *size, parentIndex);
47         if (a[minChildIndex] < a[parentIndex]) {
48             swap(&a[minChildIndex], &a[parentIndex]);
49             parentIndex = minChildIndex;
50         } else
51             break;
52     }
53 }
```

```

1 void heapSort(int *a, size_t size) {
2     size_t heapSize = 0;
3     while (heapSize != size)
4         insertHeap(a, &heapSize, a[heapSize]);
5     while (heapSize)
6         removeMinHeap(a, &heapSize);
7 }
```

11.10 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`¹.

Поиск k -ой порядковой статистики. Алгоритм Хоара

Идея алгоритма Хоара может быть использована для поиска k -ой порядковой статистики². Запишем алгоритм псевдокодом:

¹Вы можете легко в этом убедиться, если выполните сортировку для двух равных элементов.

² **k -ой порядковой статистикой** называется элемент, который будет стоять на k -ом месте в отсортированном массиве.

```

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.

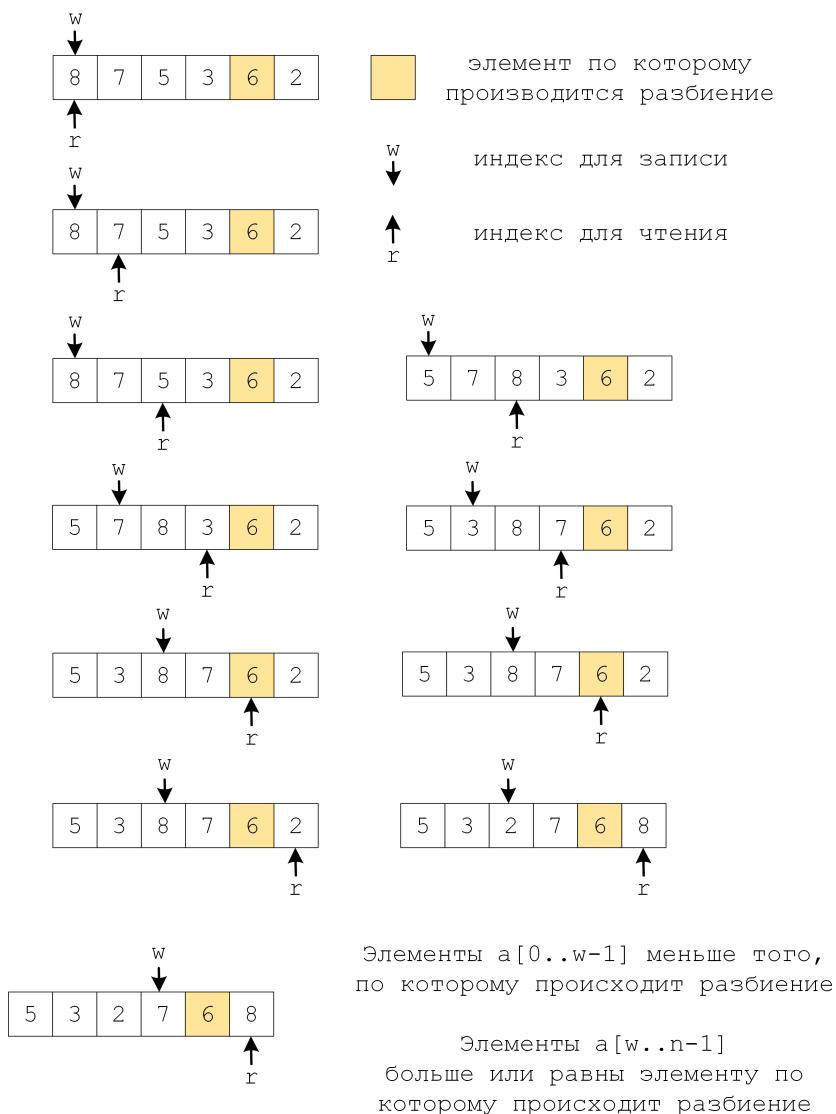


Рис. 11.19 – Разбиение последовательности функцией $split$

Глава 12

Система непересекающихся множеств

Пусть задано N множеств. Изначально каждое множество состоит из одного элемента. Назовём **лидером множества** некоторый элемент, который принадлежит данному множеству¹. Пусть N равно 6. Графически можно представить так:

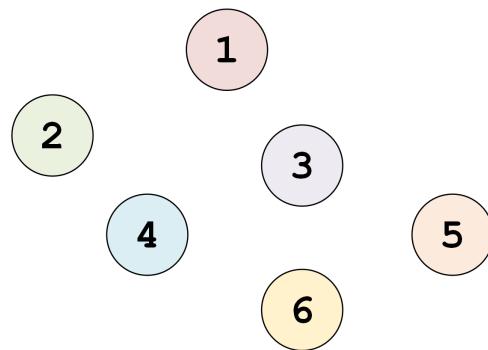


Рис. 12.1 – Шесть множеств из одного элемента. Каждый элемент является лидером своего множества

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 12.2). 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.

¹Принадлежность элемента множеству будем обозначать цветом. Лидеры множества выделены **полужирным** начертанием

12.1 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 12.2 можно было бы создать следующий массив:

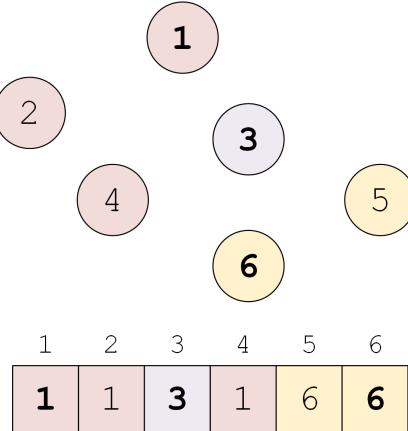


Рис. 12.2 – Три множества и значения массива p

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:

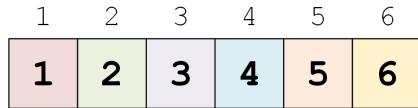


Рис. 12.3 – Инициализация массива p

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:

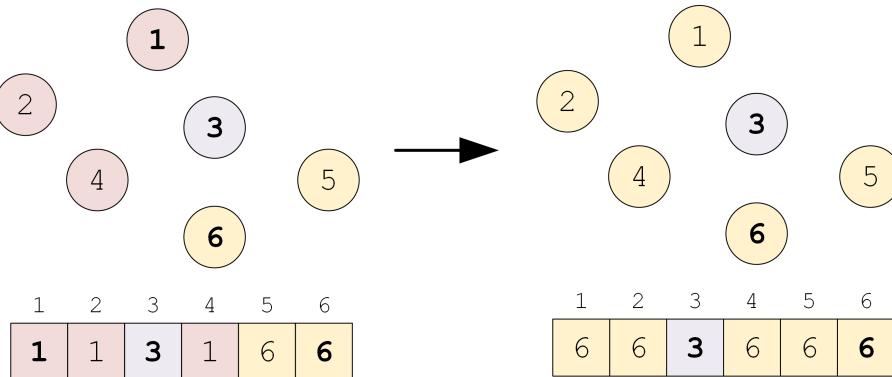


Рис. 12.4 – Выполнение операции $union(2, 6)$

Сложность такого алгоритма объединения $O(N)$. Рассмотрим ряд модификаций, которые могут ускорить решение.

12.2 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:

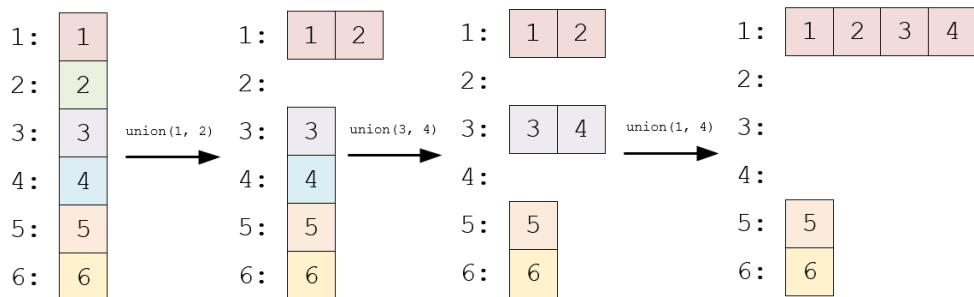


Рис. 12.5 – Процесс объединения множеств

Идейно 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 12.6 является более предпочтительным первый вариант, так как он потребует всего лишь одно обновление в массиве p вместо двух.

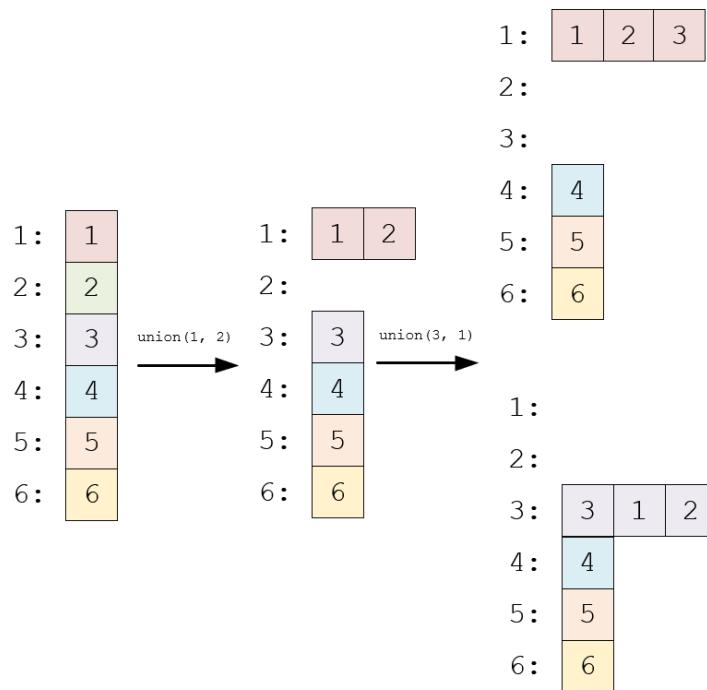


Рис. 12.6 – Варианты объединения

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 $O(\log N)$.

12.3 Модификация №2 - Переход к деревьям

Представим множества в виде деревьев. В качестве корня будем хранить лидера множества:

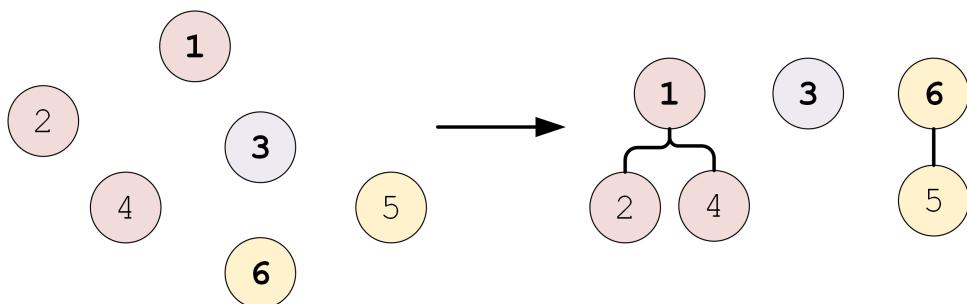


Рис. 12.7 – Представление в виде дерева

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
1	1	3	1	6	6

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):

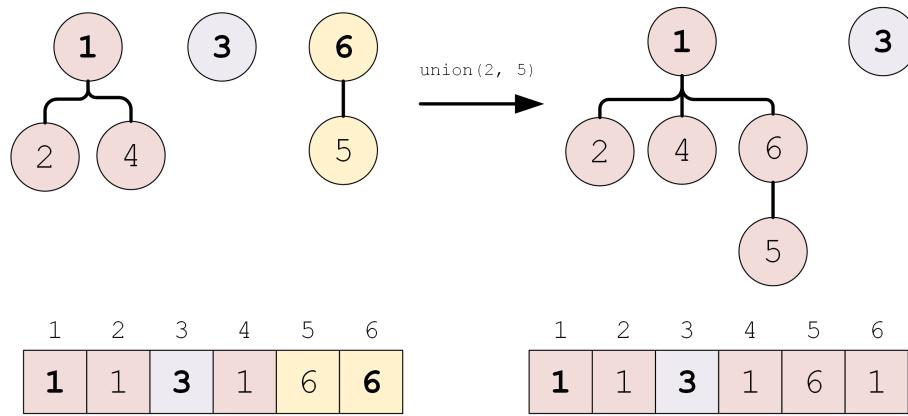


Рис. 12.8 – Объединение множеств

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:

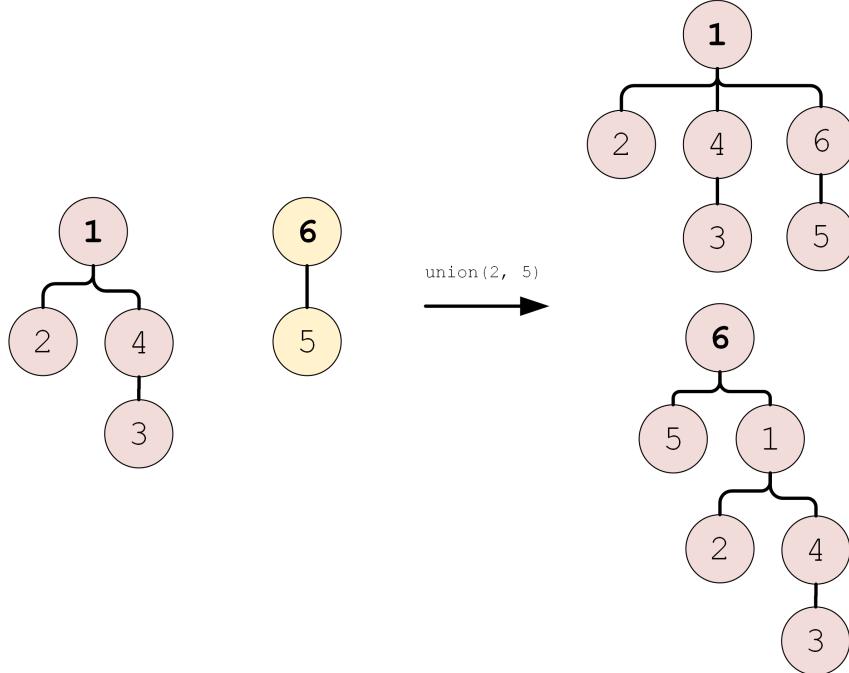


Рис. 12.9 – Варианты объединения множеств.

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 12.10.

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:

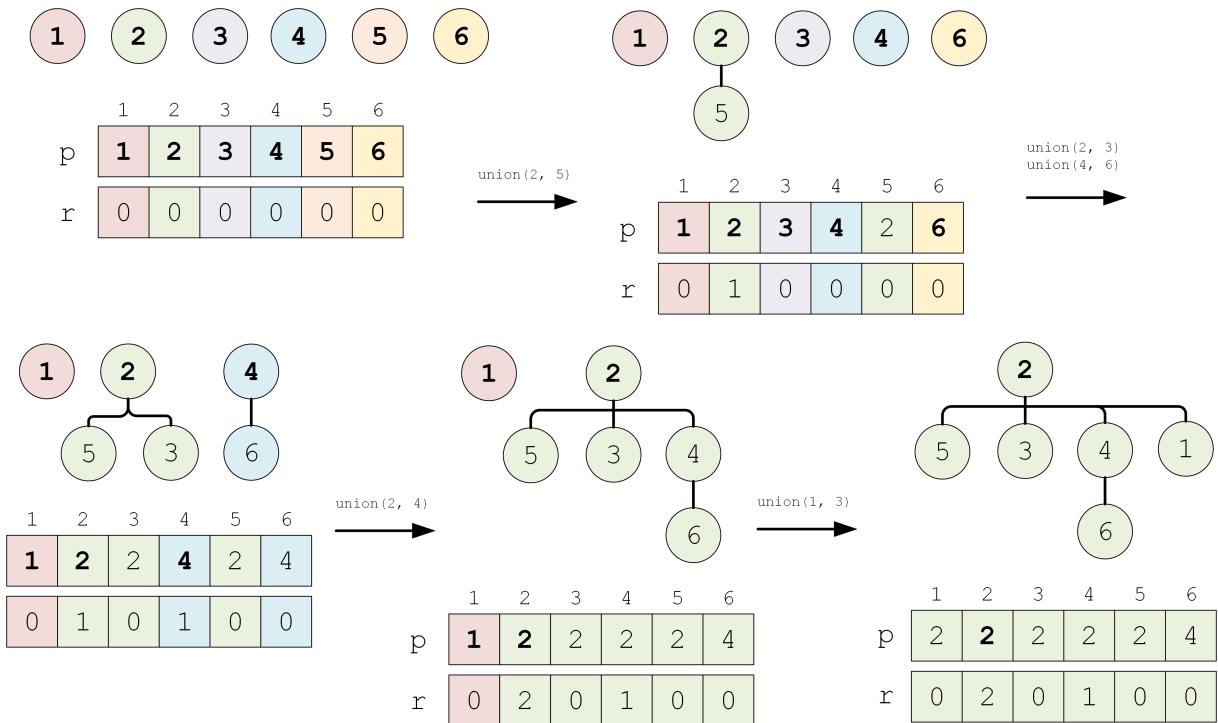


Рис. 12.10 – Объединение множеств

Можно найти ещё один способ ускорения времени работы функции `get`. Будем вычислять лидера рекурсивно. В процессе рекурсивного спуска ищем лидера, в процессе подъема (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 12.11). 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.

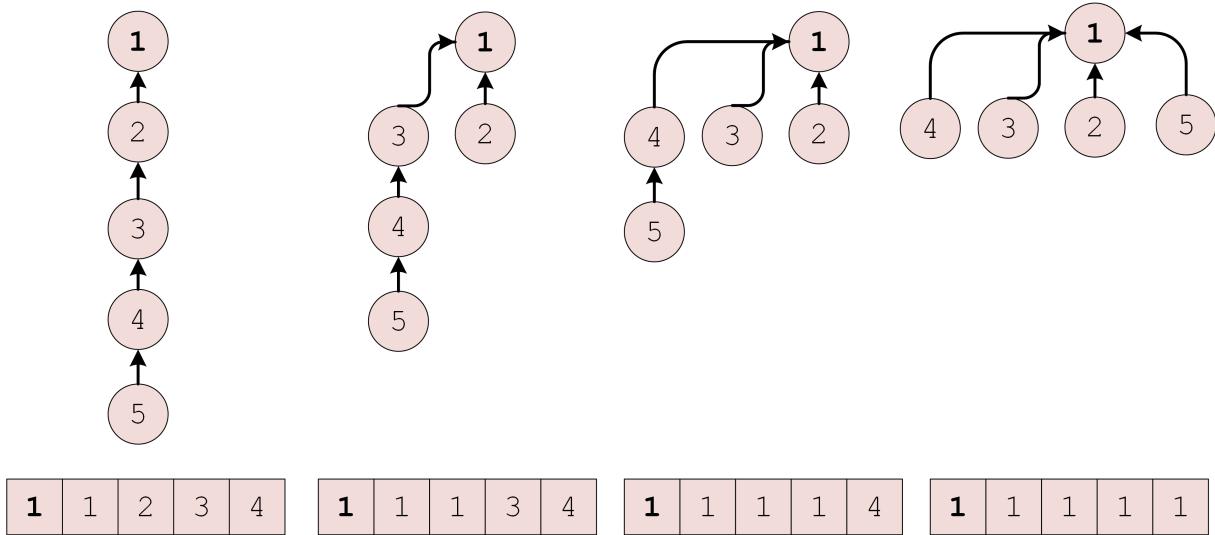


Рис. 12.11 – Рекурсивный подъем функции `get`

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])

```

12.4 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.

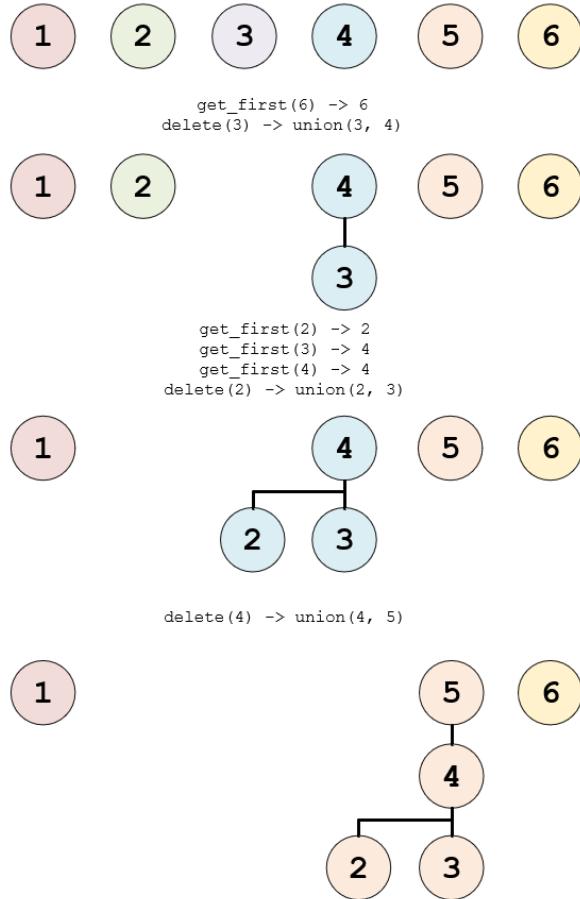


Рис. 12.12 – Примеры запросов

Но за счёт эвристики сжатия пути можно добиться амортизированного $O(\log N)$ на запрос².

12.5 Алгоритм Краскала

Алгоритм Краскала — эффективный алгоритм построения минимального остовного дерева³. Пример минимального остовного дерева представлен на рисунке 12.13.

²Хорошей 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.

³**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.

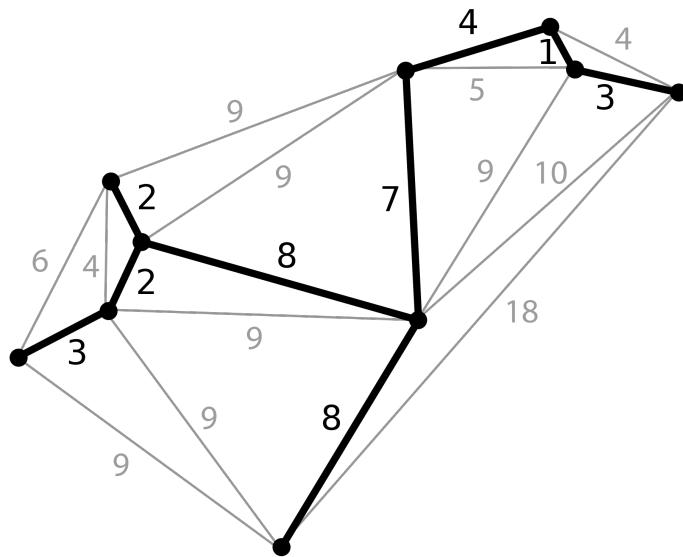
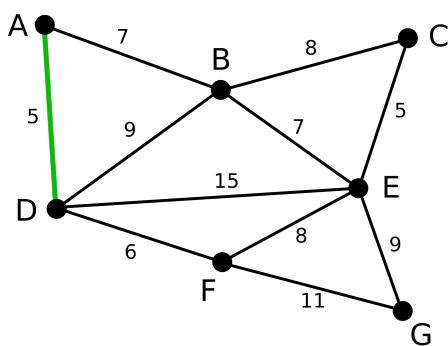


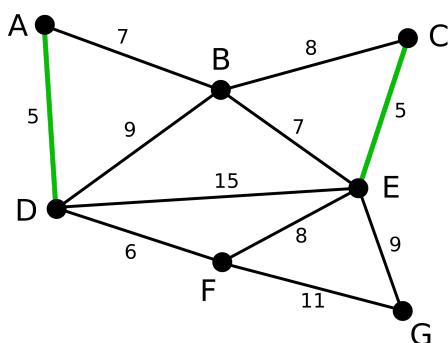
Рис. 12.13 – Минимальное оствовное дерево.

Данный алгоритм является жадным⁴. 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:

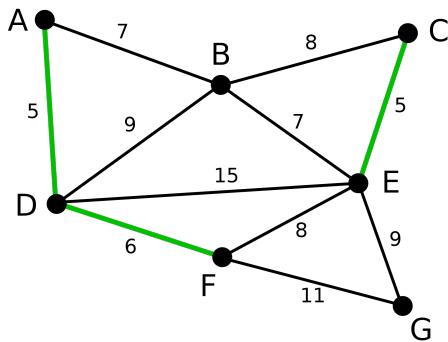


Имеем два ребра длины 5. Добавим ребро AD .

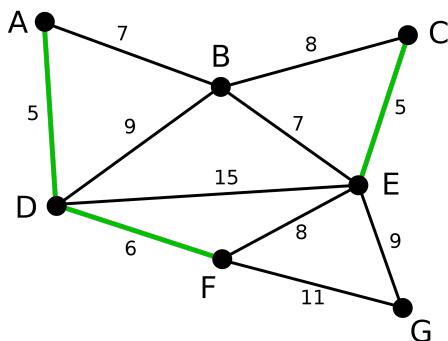


Следующее рассматриваемое ребро – CE . Так как оно соединяет вершины, между которыми ещё нет пути (путь пока что имеется только из вершины A в вершину D), добавляем ребро.

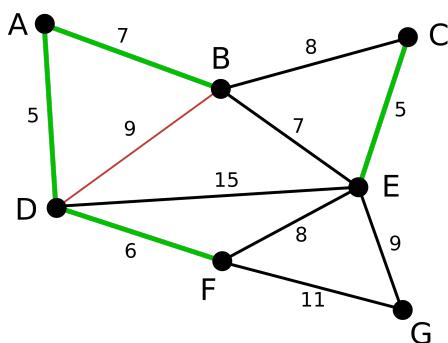
⁴Жадный алгоритм – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.



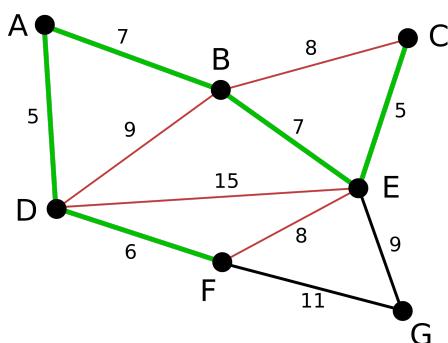
Рассмотрим ребро DF . Так как оно соединяет вершины, между которыми нет пути, добавляем ребро в дерево.



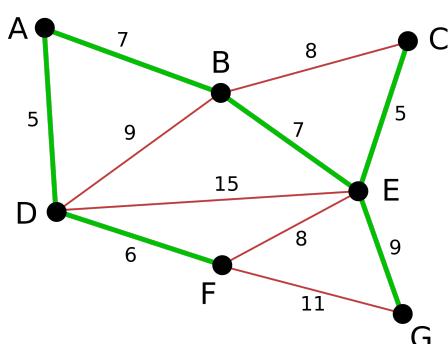
Рассмотрим ребро DF . Так как оно соединяет вершины, между которыми нет пути, добавляем ребро в дерево.



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 оствое 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 13

Структуры данных на одномерных массивах

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.

13.1 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:

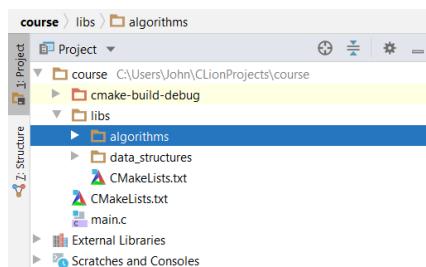
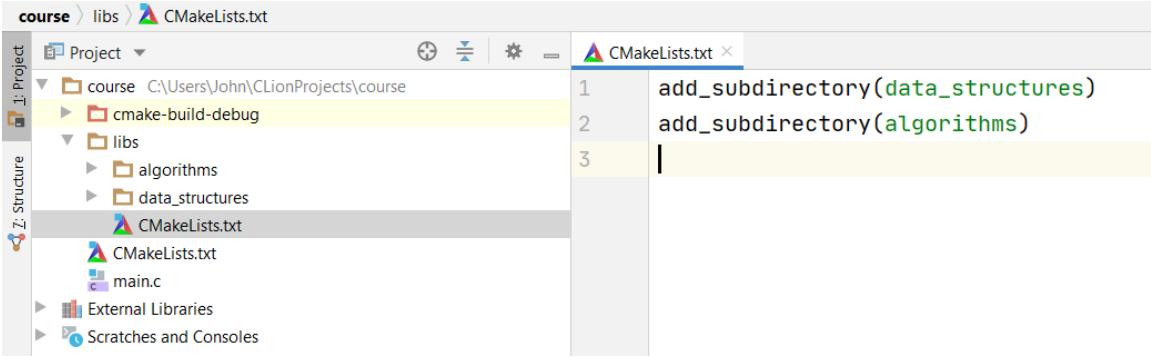


Рис. 13.1 – Структура проекта

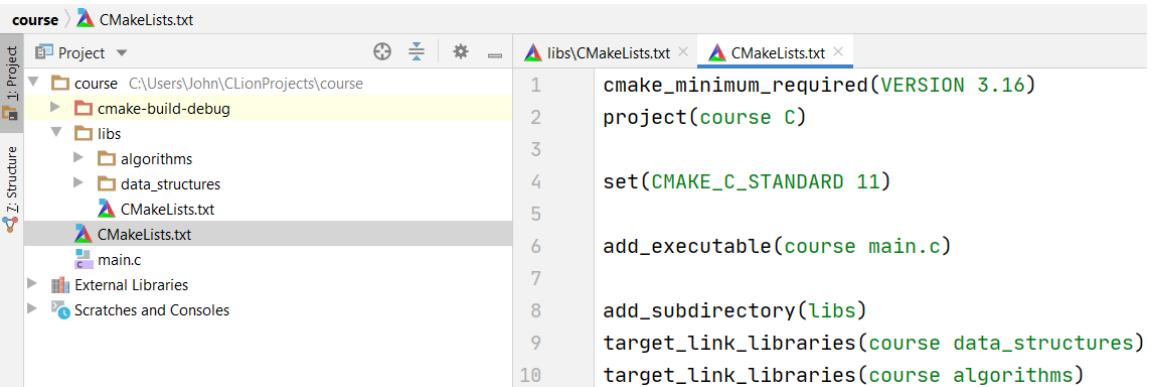
В *libs/CMakeLists* добавим наши поддиректории:



```
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)
```

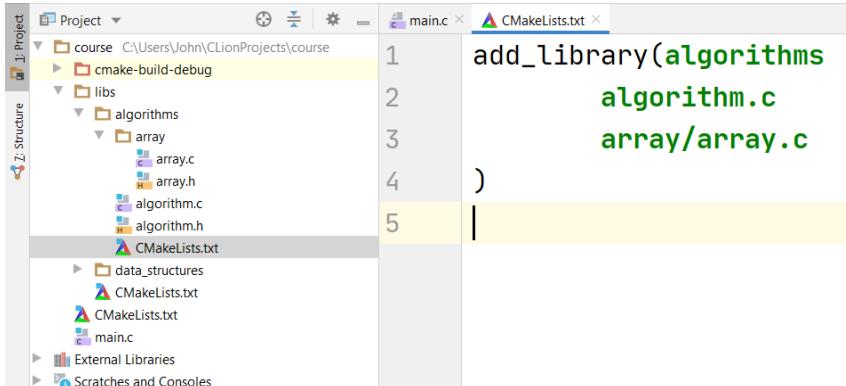
Обновим *CMakeLists* проекта:



```
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
```

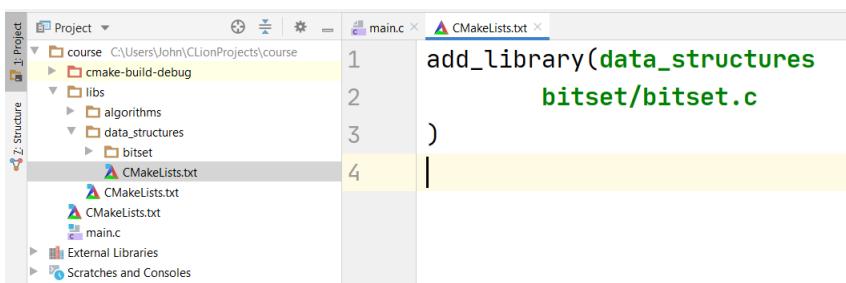
```
1 cmake_minimum_required(VERSION 3.16)
2 project(course C)
3
4 set(CMAKE_C_STANDARD 11)
5
6 add_executable(course main.c)
7
8 add_subdirectory(libs)
9 target_link_libraries(course data_structures)
10 target_link_libraries(course algorithms)
```

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
```

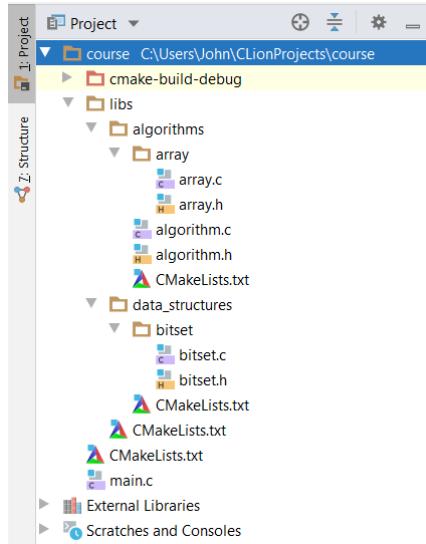
```
1 add_library(algorithms
2   algorithm.c
3   array/array.c
4 )
5 |
```



```
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
38 void deleteByPosSaveOrder_(int *a, size_t *n, const size_t pos) {
39     for (size_t i = pos; i < *n - 1; i++)
40         a[i] = a[i + 1];
41     (*n)--;
42 }
43
44 void deleteByPosUnsaveOrder_(int *a, size_t *n, size_t pos) {
45     a[pos] = a[*n - 1];
46     (*n)--;
47 }
48
49 size_t linearSearch_(const int *a, const size_t n, int x) {
50     for (size_t i = 0; i < n; i++)
51         if (a[i] == x)
52             return i;
53     return n;
54 }
55
56 int any_(const int *a, size_t n, int (*predicate)(int)) {
57     for (size_t i = 0; i < n; i++)
58         if (predicate(a[i]))
59             return 1;
60     return 0;
61 }
62
63 int all_(const int *a, size_t n, int (*predicate)(int)) {
64     for (size_t i = 0; i < n; i++)
65         if (!predicate(a[i]))
66             return 0;
67     return 1;
68 }
69
70 int countIf_(const int * const a, const size_t n, int (*predicate)(int))
71 {
72     int count = 0;
73     for (size_t i = 0; i < n; i++)
74         count += predicate(a[i]);
75     return count;
76 }
77
78 void deleteIf_(int * const a, size_t * const n, int (*deletePredicate)(int)) {
79     size_t iRead = 0;
80     while (iRead < *n && !deletePredicate(a[iRead]))
81         iRead++;
82     size_t iWrite = iRead;
83     while (iRead < *n) {
84         if (!deletePredicate(a[iRead])) {
85             a[iWrite] = a[iRead];
86             iWrite++;
87         }
88     }
89 }
90
91 void copy_(const int * const a, size_t n, int * const b) {
92     for (size_t i = 0; i < n; i++)
93         b[i] = a[i];
94 }
95
96 void reverse_(int * const a, size_t n) {
97     for (size_t i = 0; i < n / 2; i++)
98         std::swap(a[i], a[n - i - 1]);
99 }
100
101 void sort_(int * const a, size_t n) {
102     for (size_t i = 0; i < n - 1; i++)
103         for (size_t j = i + 1; j < n; j++)
104             if (a[i] > a[j])
105                 std::swap(a[i], a[j]);
106 }
107
108 void merge_(const int * const a, size_t n1, const int * const b, size_t n2,
109             int * const c, size_t n) {
110     size_t i = 0, j = 0, k = 0;
111     while (i < n1 && j < n2) {
112         if (a[i] < b[j])
113             c[k] = a[i];
114         else
115             c[k] = b[j];
116         i++;
117         j++;
118         k++;
119     }
120     while (i < n1) {
121         c[k] = a[i];
122         i++;
123         k++;
124     }
125     while (j < n2) {
126         c[k] = b[j];
127         j++;
128         k++;
129     }
130 }
131
132 void quickSort_(int * const a, size_t n) {
133     if (n < 2)
134         return;
135     size_t pivotIndex = n / 2;
136     int pivotValue = a[pivotIndex];
137     size_t i = 0, j = n - 1;
138     while (i < j) {
139         while (a[i] <= pivotValue)
140             i++;
141         while (a[j] > pivotValue)
142             j--;
143         if (i < j) {
144             std::swap(a[i], a[j]);
145         }
146     }
147     std::swap(a[i], a[pivotIndex]);
148     quickSort_(a, i);
149     quickSort_(a + i + 1, n - i - 1);
150 }
151
152 void bubbleSort_(int * const a, size_t n) {
153     for (size_t i = 0; i < n - 1; i++)
154         for (size_t j = 0; j < n - i - 1; j++)
155             if (a[j] > a[j + 1])
156                 std::swap(a[j], a[j + 1]);
157 }
158
159 void insertionSort_(int * const a, size_t n) {
160     for (size_t i = 1; i < n; i++) {
161         int key = a[i];
162         size_t j = i - 1;
163         while (j >= 0 && a[j] > key) {
164             a[j + 1] = a[j];
165             j--;
166         }
167         a[j + 1] = key;
168     }
169 }
170
171 void selectionSort_(int * const a, size_t n) {
172     for (size_t i = 0; i < n - 1; i++) {
173         size_t minIndex = i;
174         for (size_t j = i + 1; j < n; j++)
175             if (a[j] < a[minIndex])
176                 minIndex = j;
177         std::swap(a[i], a[minIndex]);
178     }
179 }
180
181 void heapSort_(int * const a, size_t n) {
182     for (size_t i = n / 2 - 1; i >= 0; i--)
183         heapify(a, n, i);
184     for (size_t i = n - 1; i >= 0; i--) {
185         std::swap(a[0], a[i]);
186         heapify(a, i, 0);
187     }
188 }
189
190 void heapify_(int * const a, size_t n, size_t i) {
191     size_t largest = i;
192     size_t left = 2 * i + 1;
193     size_t right = 2 * i + 2;
194     if (left < n && a[left] > a[largest])
195         largest = left;
196     if (right < n && a[right] > a[largest])
197         largest = right;
198     if (largest != i) {
199         std::swap(a[i], a[largest]);
200         heapify_(a, n, largest);
201     }
202 }
203
204 void shellSort_(int * const a, size_t n) {
205     for (size_t gap = n / 2; gap > 0; gap /= 2) {
206         for (size_t i = gap; i < n; i++) {
207             int temp = a[i];
208             size_t j;
209             for (j = i - gap; j >= 0 && a[j] > temp; j -= gap)
210                 a[j + gap] = a[j];
211             a[j + gap] = temp;
212         }
213     }
214 }
215
216 void cocktailShakerSort_(int * const a, size_t n) {
217     for (size_t i = 0, j = n - 1; i < j; i++, j--) {
218         for (size_t k = i; k < j; k++) {
219             if (a[k] > a[k + 1])
220                 std::swap(a[k], a[k + 1]);
221         }
222         for (size_t k = j; k > i; k--) {
223             if (a[k] < a[k - 1])
224                 std::swap(a[k], a[k - 1]);
225         }
226     }
227 }
228
229 void combSort_(int * const a, size_t n) {
230     size_t gap = n;
231     while (gap > 1) {
232         gap = gap / 1.25;
233         for (size_t i = 0; i < n - gap; i++) {
234             if (a[i] > a[i + gap])
235                 std::swap(a[i], a[i + gap]);
236         }
237     }
238 }
239
240 void bucketSort_(int * const a, size_t n) {
241     size_t max = a[0];
242     for (size_t i = 1; i < n; i++)
243         if (a[i] > max)
244             max = a[i];
245     size_t bucketCount = max + 1;
246     int * buckets = new int[bucketCount];
247     for (size_t i = 0; i < n; i++)
248         buckets[a[i]]++;
249     for (size_t i = 1; i < bucketCount; i++)
250         buckets[i] += buckets[i - 1];
251     for (size_t i = n - 1; i >= 0; i--) {
252         a[buckets[a[i]] - 1] = a[i];
253         buckets[a[i]]--;
254     }
255     delete[] buckets;
256 }
257
258 void radixSort_(int * const a, size_t n) {
259     size_t max = a[0];
260     for (size_t i = 1; i < n; i++)
261         if (a[i] > max)
262             max = a[i];
263     size_t digit = 1;
264     while (max / digit > 0) {
265         size_t count[10] = {0};
266         for (size_t i = 0; i < n; i++)
267             count[(a[i] / digit) % 10]++;
268         for (size_t i = 1; i < 10; i++)
269             count[i] += count[i - 1];
270         for (size_t i = n - 1; i >= 0; i--) {
271             a[count[(a[i] / digit) % 10] - 1] = a[i];
272             count[(a[i] / digit) % 10]--;
273         }
274         digit *= 10;
275     }
276 }
277
278 void bitonicSort_(int * const a, size_t n) {
279     for (size_t i = 0; i < n; i += 2) {
280         if (a[i] > a[i + 1])
281             std::swap(a[i], a[i + 1]);
282     }
283     for (size_t i = 0; i < n; i += 4) {
284         if (a[i] > a[i + 2])
285             std::swap(a[i], a[i + 2]);
286         if (a[i + 1] > a[i + 3])
287             std::swap(a[i + 1], a[i + 3]);
288     }
289     for (size_t i = 0; i < n; i += 8) {
290         if (a[i] > a[i + 4])
291             std::swap(a[i], a[i + 4]);
292         if (a[i + 1] > a[i + 5])
293             std::swap(a[i + 1], a[i + 5]);
294         if (a[i + 2] > a[i + 6])
295             std::swap(a[i + 2], a[i + 6]);
296         if (a[i + 3] > a[i + 7])
297             std::swap(a[i + 3], a[i + 7]);
298     }
299     for (size_t i = 0; i < n; i += 16) {
300         if (a[i] > a[i + 8])
301             std::swap(a[i], a[i + 8]);
302         if (a[i + 1] > a[i + 9])
303             std::swap(a[i + 1], a[i + 9]);
304         if (a[i + 2] > a[i + 10])
305             std::swap(a[i + 2], a[i + 10]);
306         if (a[i + 3] > a[i + 11])
307             std::swap(a[i + 3], a[i + 11]);
308         if (a[i + 4] > a[i + 12])
309             std::swap(a[i + 4], a[i + 12]);
310         if (a[i + 5] > a[i + 13])
311             std::swap(a[i + 5], a[i + 13]);
312         if (a[i + 6] > a[i + 14])
313             std::swap(a[i + 6], a[i + 14]);
314         if (a[i + 7] > a[i + 15])
315             std::swap(a[i + 7], a[i + 15]);
316     }
317     for (size_t i = 0; i < n; i += 32) {
318         if (a[i] > a[i + 16])
319             std::swap(a[i], a[i + 16]);
320         if (a[i + 1] > a[i + 17])
321             std::swap(a[i + 1], a[i + 17]);
322         if (a[i + 2] > a[i + 18])
323             std::swap(a[i + 2], a[i + 18]);
324         if (a[i + 3] > a[i + 19])
325             std::swap(a[i + 3], a[i + 19]);
326         if (a[i + 4] > a[i + 20])
327             std::swap(a[i + 4], a[i + 20]);
328         if (a[i + 5] > a[i + 21])
329             std::swap(a[i + 5], a[i + 21]);
330         if (a[i + 6] > a[i + 22])
331             std::swap(a[i + 6], a[i + 22]);
332         if (a[i + 7] > a[i + 23])
333             std::swap(a[i + 7], a[i + 23]);
334         if (a[i + 8] > a[i + 24])
335             std::swap(a[i + 8], a[i + 24]);
336         if (a[i + 9] > a[i + 25])
337             std::swap(a[i + 9], a[i + 25]);
338         if (a[i + 10] > a[i + 26])
339             std::swap(a[i + 10], a[i + 26]);
340         if (a[i + 11] > a[i + 27])
341             std::swap(a[i + 11], a[i + 27]);
342         if (a[i + 12] > a[i + 28])
343             std::swap(a[i + 12], a[i + 28]);
344         if (a[i + 13] > a[i + 29])
345             std::swap(a[i + 13], a[i + 29]);
346         if (a[i + 14] > a[i + 30])
347             std::swap(a[i + 14], a[i + 30]);
348         if (a[i + 15] > a[i + 31])
349             std::swap(a[i + 15], a[i + 31]);
350     }
351 }
352
353 void bitonicMerge_(int * const a, size_t n, size_t start, size_t end, size_t direction) {
354     if (start < end) {
355         size_t mid = (start + end) / 2;
356         if (direction == 1) {
357             for (size_t i = start; i < mid; i++)
358                 if (a[i] > a[i + 1])
359                     std::swap(a[i], a[i + 1]);
360             for (size_t i = mid; i < end; i++)
361                 if (a[i] > a[i + 1])
362                     std::swap(a[i], a[i + 1]);
363         } else {
364             for (size_t i = start; i < mid; i++)
365                 if (a[i] < a[i + 1])
366                     std::swap(a[i], a[i + 1]);
367             for (size_t i = mid; i < end; i++)
368                 if (a[i] < a[i + 1])
369                     std::swap(a[i], a[i + 1]);
370         }
371         bitonicMerge_(a, n, start, mid, direction);
372         bitonicMerge_(a, n, mid, end, direction);
373     }
374 }
375
376 void bitonicSort_(int * const a, size_t n) {
377     bitonicMerge_(a, n, 0, n, 1);
378     bitonicMerge_(a, n, 0, n, 0);
379 }
380
381 void bitonicSortParallel_(int * const a, size_t n) {
382     bitonicMergeParallel_(a, n, 0, n, 1);
383     bitonicMergeParallel_(a, n, 0, n, 0);
384 }
385
386 void bitonicMergeParallel_(int * const a, size_t n, size_t start, size_t end, size_t direction) {
387     if (start < end) {
388         size_t mid = (start + end) / 2;
389         if (direction == 1) {
390             for (size_t i = start; i < mid; i += 2)
391                 if (a[i] > a[i + 1])
392                     std::swap(a[i], a[i + 1]);
393             for (size_t i = mid; i < end; i += 2)
394                 if (a[i] > a[i + 1])
395                     std::swap(a[i], a[i + 1]);
396         } else {
397             for (size_t i = start; i < mid; i += 2)
398                 if (a[i] < a[i + 1])
399                     std::swap(a[i], a[i + 1]);
400             for (size_t i = mid; i < end; i += 2)
401                 if (a[i] < a[i + 1])
402                     std::swap(a[i], a[i + 1]);
403         }
404         bitonicMergeParallel_(a, n, start, mid, direction);
405         bitonicMergeParallel_(a, n, mid, end, direction);
406     }
407 }
408
409 void bitonicSortParallel_(int * const a, size_t n) {
410     bitonicMergeParallel_(a, n, 0, n, 1);
411     bitonicMergeParallel_(a, n, 0, n, 0);
412 }
413
414 void bitonicSortParallel_(int * const a, size_t n) {
415     bitonicMergeParallel_(a, n, 0, n, 1);
416     bitonicMergeParallel_(a, n, 0, n, 0);
417 }
418
419 void bitonicSortParallel_(int * const a, size_t n) {
420     bitonicMergeParallel_(a, n, 0, n, 1);
421     bitonicMergeParallel_(a, n, 0, n, 0);
422 }
423
424 void bitonicSortParallel_(int * const a, size_t n) {
425     bitonicMergeParallel_(a, n, 0, n, 1);
426     bitonicMergeParallel_(a, n, 0, n, 0);
427 }
428
429 void bitonicSortParallel_(int * const a, size_t n) {
430     bitonicMergeParallel_(a, n, 0, n, 1);
431     bitonicMergeParallel_(a, n, 0, n, 0);
432 }
433
434 void bitonicSortParallel_(int * const a, size_t n) {
435     bitonicMergeParallel_(a, n, 0, n, 1);
436     bitonicMergeParallel_(a, n, 0, n, 0);
437 }
438
439 void bitonicSortParallel_(int * const a, size_t n) {
440     bitonicMergeParallel_(a, n, 0, n, 1);
441     bitonicMergeParallel_(a, n, 0, n, 0);
442 }
443
444 void bitonicSortParallel_(int * const a, size_t n) {
445     bitonicMergeParallel_(a, n, 0, n, 1);
446     bitonicMergeParallel_(a, n, 0, n, 0);
447 }
448
449 void bitonicSortParallel_(int * const a, size_t n) {
450     bitonicMergeParallel_(a, n, 0, n, 1);
451     bitonicMergeParallel_(a, n, 0, n, 0);
452 }
453
454 void bitonicSortParallel_(int * const a, size_t n) {
455     bitonicMergeParallel_(a, n, 0, n, 1);
456     bitonicMergeParallel_(a, n, 0, n, 0);
457 }
458
459 void bitonicSortParallel_(int * const a, size_t n) {
460     bitonicMergeParallel_(a, n, 0, n, 1);
461     bitonicMergeParallel_(a, n, 0, n, 0);
462 }
463
464 void bitonicSortParallel_(int * const a, size_t n) {
465     bitonicMergeParallel_(a, n, 0, n, 1);
466     bitonicMergeParallel_(a, n, 0, n, 0);
467 }
468
469 void bitonicSortParallel_(int * const a, size_t n) {
470     bitonicMergeParallel_(a, n, 0, n, 1);
471     bitonicMergeParallel_(a, n, 0, n, 0);
472 }
473
474 void bitonicSortParallel_(int * const a, size_t n) {
475     bitonicMergeParallel_(a, n, 0, n, 1);
476     bitonicMergeParallel_(a, n, 0, n, 0);
477 }
478
479 void bitonicSortParallel_(int * const a, size_t n) {
480     bitonicMergeParallel_(a, n, 0, n, 1);
481     bitonicMergeParallel_(a, n, 0, n, 0);
482 }
483
484 void bitonicSortParallel_(int * const a, size_t n) {
485     bitonicMergeParallel_(a, n, 0, n, 1);
486     bitonicMergeParallel_(a, n, 0, n, 0);
487 }
488
489 void bitonicSortParallel_(int * const a, size_t n) {
490     bitonicMergeParallel_(a, n, 0, n, 1);
491     bitonicMergeParallel_(a, n, 0, n, 0);
492 }
493
494 void bitonicSortParallel_(int * const a, size_t n) {
495     bitonicMergeParallel_(a, n, 0, n, 1);
496     bitonicMergeParallel_(a, n, 0, n, 0);
497 }
498
499 void bitonicSortParallel_(int * const a, size_t n) {
500     bitonicMergeParallel_(a, n, 0, n, 1);
501     bitonicMergeParallel_(a, n, 0, n, 0);
502 }
503
504 void bitonicSortParallel_(int * const a, size_t n) {
505     bitonicMergeParallel_(a, n, 0, n, 1);
506     bitonicMergeParallel_(a, n, 0, n, 0);
507 }
508
509 void bitonicSortParallel_(int * const a, size_t n) {
510     bitonicMergeParallel_(a, n, 0, n, 1);
511     bitonicMergeParallel_(a, n, 0, n, 0);
512 }
513
514 void bitonicSortParallel_(int * const a, size_t n) {
515     bitonicMergeParallel_(a, n, 0, n, 1);
516     bitonicMergeParallel_(a, n, 0, n, 0);
517 }
518
519 void bitonicSortParallel_(int * const a, size_t n) {
520     bitonicMergeParallel_(a, n, 0, n, 1);
521     bitonicMergeParallel_(a, n, 0, n, 0);
522 }
523
524 void bitonicSortParallel_(int * const a, size_t n) {
525     bitonicMergeParallel_(a, n, 0, n, 1);
526     bitonicMergeParallel_(a, n, 0, n, 0);
527 }
528
529 void bitonicSortParallel_(int * const a, size_t n) {
530     bitonicMergeParallel_(a, n, 0, n, 1);
531     bitonicMergeParallel_(a, n, 0, n, 0);
532 }
533
534 void bitonicSortParallel_(int * const a, size_t n) {
535     bitonicMergeParallel_(a, n, 0, n, 1);
536     bitonicMergeParallel_(a, n, 0, n, 0);
537 }
538
539 void bitonicSortParallel_(int * const a, size_t n) {
540     bitonicMergeParallel_(a, n, 0, n, 1);
541     bitonicMergeParallel_(a, n, 0, n, 0);
542 }
543
544 void bitonicSortParallel_(int * const a, size_t n) {
545     bitonicMergeParallel_(a, n, 0, n, 1);
546     bitonicMergeParallel_(a, n, 0, n, 0);
547 }
548
549 void bitonicSortParallel_(int * const a, size_t n) {
550     bitonicMergeParallel_(a, n, 0, n, 1);
551     bitonicMergeParallel_(a, n, 0, n, 0);
552 }
553
554 void bitonicSortParallel_(int * const a, size_t n) {
555     bitonicMergeParallel_(a, n, 0, n, 1);
556     bitonicMergeParallel_(a, n, 0, n, 0);
557 }
558
559 void bitonicSortParallel_(int * const a, size_t n) {
560     bitonicMergeParallel_(a, n, 0, n, 1);
561     bitonicMergeParallel_(a, n, 0, n, 0);
562 }
563
564 void bitonicSortParallel_(int * const a, size_t n) {
565     bitonicMergeParallel_(a, n, 0, n, 1);
566     bitonicMergeParallel_(a, n, 0, n, 0);
567 }
568
569 void bitonicSortParallel_(int * const a, size_t n) {
570     bitonicMergeParallel_(a, n, 0, n, 1);
571     bitonicMergeParallel_(a, n, 0, n, 0);
572 }
573
574 void bitonicSortParallel_(int * const a, size_t n) {
575     bitonicMergeParallel_(a, n, 0, n, 1);
576     bitonicMergeParallel_(a, n, 0, n, 0);
577 }
578
579 void bitonicSortParallel_(int * const a, size_t n) {
580     bitonicMergeParallel_(a, n, 0, n, 1);
581     bitonicMergeParallel_(a, n, 0, n, 0);
582 }
583
584 void bitonicSortParallel_(int * const a, size_t n) {
585     bitonicMergeParallel_(a, n, 0, n, 1);
586     bitonicMergeParallel_(a, n, 0, n, 0);
587 }
588
589 void bitonicSortParallel_(int * const a, size_t n) {
590     bitonicMergeParallel_(a, n, 0, n, 1);
591     bitonicMergeParallel_(a, n, 0, n, 0);
592 }
593
594 void bitonicSortParallel_(int * const a, size_t n) {
595     bitonicMergeParallel_(a, n, 0, n, 1);
596     bitonicMergeParallel_(a, n, 0, n, 0);
597 }
598
599 void bitonicSortParallel_(int * const a, size_t n) {
600     bitonicMergeParallel_(a, n, 0, n, 1);
601     bitonicMergeParallel_(a, n, 0, n, 0);
602 }
603
604 void bitonicSortParallel_(int * const a, size_t n) {
605     bitonicMergeParallel_(a, n, 0, n, 1);
606     bitonicMergeParallel_(a, n, 0, n, 0);
607 }
608
609 void bitonicSortParallel_(int * const a, size_t n) {
610     bitonicMergeParallel_(a, n, 0, n, 1);
611     bitonicMergeParallel_(a, n, 0, n, 0);
612 }
613
614 void bitonicSortParallel_(int * const a, size_t n) {
615     bitonicMergeParallel_(a, n, 0, n, 1);
616     bitonicMergeParallel_(a, n, 0, n, 0);
617 }
618
619 void bitonicSortParallel_(int * const a, size_t n) {
620     bitonicMergeParallel_(a, n, 0, n, 1);
621     bitonicMergeParallel_(a, n, 0, n, 0);
622 }
623
624 void bitonicSortParallel_(int * const a, size_t n) {
625     bitonicMergeParallel_(a, n, 0, n, 1);
626     bitonicMergeParallel_(a, n, 0, n, 0);
627 }
628
629 void bitonicSortParallel_(int * const a, size_t n) {
630     bitonicMergeParallel_(a, n, 0, n, 1);
631     bitonicMergeParallel_(a, n, 0, n, 0);
632 }
633
634 void bitonicSortParallel_(int * const a, size_t n) {
635     bitonicMergeParallel_(a, n, 0, n, 1);
636     bitonicMergeParallel_(a, n, 0, n, 0);
637 }
638
639 void bitonicSortParallel_(int * const a, size_t n) {
640     bitonicMergeParallel_(a, n, 0, n, 1);
641     bitonicMergeParallel_(a, n, 0, n, 0);
642 }
643
644 void bitonicSortParallel_(int * const a, size_t n) {
645     bitonicMergeParallel_(a, n, 0, n, 1);
646     bitonicMergeParallel_(a, n, 0, n, 0);
647 }
648
649 void bitonicSortParallel_(int * const a, size_t n) {
650     bitonicMergeParallel_(a, n, 0, n, 1);
651     bitonicMergeParallel_(a, n, 0, n, 0);
652 }
653
654 void bitonicSortParallel_(int * const a, size_t n) {
655     bitonicMergeParallel_(a, n, 0, n, 1);
656     bitonicMergeParallel_(a, n, 0, n, 0);
657 }
658
659 void bitonicSortParallel_(int * const a, size_t n) {
660     bitonicMergeParallel_(a, n, 0, n, 1);
661     bitonicMergeParallel_(a, n, 0, n, 0);
662 }
663
664 void bitonicSortParallel_(int * const a, size_t n) {
665     bitonicMergeParallel_(a, n, 0, n, 1);
666     bitonicMergeParallel_(a, n, 0, n, 0);
667 }
668
669 void bitonicSortParallel_(int * const a, size_t n) {
670     bitonicMergeParallel_(a, n, 0, n, 1);
671     bitonicMergeParallel_(a, n, 0, n, 0);
672 }
673
674 void bitonicSortParallel_(int * const a, size_t n) {
675     bitonicMergeParallel_(a, n, 0, n, 1);
676     bitonicMergeParallel_(a, n, 0, n, 0);
677 }
678
679 void bitonicSortParallel_(int * const a, size_t n) {
680     bitonicMergeParallel_(a, n, 0, n, 1);
681     bitonicMergeParallel_(a, n, 0, n, 0);
682 }
683
684 void bitonicSortParallel_(int * const a, size_t n) {
685     bitonicMergeParallel_(a, n, 0, n, 1);
686     bitonicMergeParallel_(a, n, 0, n, 0);
687 }
688
689 void bitonicSortParallel_(int * const a, size_t n) {
690     bitonicMergeParallel_(a, n, 0, n, 1);
691     bitonicMergeParallel_(a, n, 0, n, 0);
692 }
693
694 void bitonicSortParallel_(int * const a, size_t n) {
695     bitonicMergeParallel_(a, n, 0, n, 1);
696     bitonicMergeParallel_(a, n, 0, n, 0);
697 }
698
699 void bitonicSortParallel_(int * const a, size_t n) {
700     bitonicMergeParallel_(a, n, 0, n, 1);
701     bitonicMergeParallel_(a, n, 0, n, 0);
702 }
703
704 void bitonicSortParallel_(int * const a, size_t n) {
705     bitonicMergeParallel_(a, n, 0, n, 1);
706     bitonicMergeParallel_(a, n, 0, n, 0);
707 }
708
709 void bitonicSortParallel_(int * const a, size_t n) {
710     bitonicMergeParallel_(a, n, 0, n, 1);
711     bitonicMergeParallel_(a, n, 0, n, 0);
712 }
713
714 void bitonicSortParallel_(int * const a, size_t n) {
715     bitonicMergeParallel_(a, n, 0, n, 1);
716     bitonicMergeParallel_(a, n, 0, n, 0);
717 }
718
719 void bitonicSortParallel_(int * const a, size_t n) {
720     bitonicMergeParallel_(a, n, 0, n, 1);
721     bitonicMergeParallel_(a, n, 0, n, 0);
722 }
723
724 void bitonicSortParallel_(int * const a, size_t n) {
725     bitonicMergeParallel_(a, n, 0, n, 1);
726     bitonicMergeParallel_(a, n, 0, n, 0);
727 }
728
729 void bitonicSortParallel_(int * const a, size_t n) {
730     bitonicMergeParallel_(a, n, 0, n, 1);
731     bitonicMergeParallel_(a, n, 0, n, 0);
732 }
733
734 void bitonicSortParallel_(int * const a, size_t n) {
735     bitonicMergeParallel_(a, n, 0, n, 1);
736     bitonicMergeParallel_(a, n, 0, n, 0);
737 }
738
739 void bitonicSortParallel_(int * const a, size_t n) {
740     bitonicMergeParallel_(a, n, 0, n, 1);
741     bitonicMergeParallel_(a, n, 0, n, 0);
742 }
743
744 void bitonicSortParallel_(int * const a, size_t n) {
745     bitonicMergeParallel_(a, n, 0, n, 1);
746     bitonicMergeParallel_(a, n, 0, n, 0);
747 }
748
749 void bitonicSortParallel_(int * const a, size_t n) {
750     bitonicMergeParallel_(a, n, 0, n, 1);
751     bitonicMergeParallel_(a, n, 0, n, 0);
752 }
753
754 void bitonicSortParallel_(int * const a, size_t n) {
755     bitonicMergeParallel_(a, n, 0, n, 1);
756     bitonicMergeParallel_(a, n, 0, n, 0);
757 }
758
759 void bitonicSortParallel_(int * const a, size_t n) {
760     bitonicMergeParallel_(a, n, 0, n, 1);
761     bitonicMergeParallel_(a, n, 0, n, 0);
762 }
763
764 void bitonicSortParallel_(int * const a, size_t n) {
765     bitonicMergeParallel_(a, n, 0, n, 1);
766     bitonicMergeParallel_(a, n, 0, n, 0);
767 }
768
769 void bitonicSortParallel_(int * const a, size_t n) {
770     bitonicMergeParallel_(a, n, 0, n, 1);
771     bitonicMergeParallel_(a, n, 0, n, 0);
772 }
773
774 void bitonicSortParallel_(int * const a, size_t n) {
775     bitonicMergeParallel_(a, n, 0, n, 1);
776     bitonicMergeParallel_(a, n, 0, n, 0);
777 }
778
779 void bitonicSortParallel_(int * const a, size_t n) {
780     bitonicMergeParallel_(a, n, 0, n, 1);
781     bitonicMergeParallel_(a, n, 0, n, 0);
782 }
783
784 void bitonicSortParallel_(int * const a, size_t n) {
785     bitonicMergeParallel_(a, n, 0, n, 1);
786     bitonicMergeParallel_(a, n, 0, n, 0);
787 }
788
789 void bitonicSortParallel_(int * const a, size_t n) {
790     bitonicMergeParallel_(a, n, 0, n, 1);
791     bitonicMergeParallel_(a, n, 0, n, 0);
792 }
793
794 void bitonicSortParallel_(int * const a, size_t n) {
795     bitonicMergeParallel_(a, n, 0, n, 1);
796     bitonicMergeParallel_(a, n, 0, n, 0);
797 }
798
799 void bitonicSortParallel_(int * const a, size_t n) {
800     bitonicMergeParallel_(a, n, 0, n, 1);
801     bitonicMergeParallel_(a, n, 0, n, 0);
802 }
803
804 void bitonicSortParallel_(int * const a, size_t n) {
805     bitonicMergeParallel_(a, n, 0, n, 1);
806     bitonicMergeParallel_(a, n, 0, n, 0);
807 }
808
809 void bitonicSortParallel_(int * const a, size_t n) {
810     bitonicMergeParallel_(a, n, 0, n, 1);
811     bitonicMergeParallel_(a, n, 0, n, 0);
812 }
813
814 void bitonicSortParallel_(int * const a, size_t n) {
815     bitonicMergeParallel_(a, n, 0, n, 1);
816     bitonicMergeParallel_(a, n, 0, n, 0);
817 }
818
819 void bitonicSortParallel_(int * const a, size_t n) {
820     bitonicMergeParallel_(a, n, 0, n, 1);
821     bitonicMergeParallel_(a, n, 0, n, 0);
822 }
823
824 void bitonicSortParallel_(int * const a, size_t n) {
825     bitonicMergeParallel_(a, n, 0, n, 1);
826     bitonicMergeParallel_(a, n, 0, n, 0);
827 }
828
829 void bitonicSortParallel_(int * const a, size_t n) {
830     bitonicMergeParallel_(a, n, 0, n
```

```

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;
}

```

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 }

```

13.2 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.

13.2.1 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. 13.2 – 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`:

```
1 // 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:

$$A \cup \{x\} \Rightarrow A := A \mid (1 \ll x)$$



Удаление элемента из множества

Для удаления элемента из множества, необходимо выставить бит для соответствующего элемента в ноль:

$$A \setminus \{x\} \Rightarrow A := A \& \sim(1 \ll x)$$

Используемые биты для представления множества																													
Неиспользуемые биты																													
A																													
$A \setminus \{6\}$																													

$$A \setminus \{6\} = \{1, 2, 3, 4, 6, 10\} \setminus \{6\} = \{1, 2, 3, 4, 10\}$$

Объединение

Объединение множеств реализуется за счёт применения побитового ИЛИ:

$$A \cup B \Rightarrow A | B$$

Используемые биты для представления множества																													
Неиспользуемые биты																													
A																													
B																													
$A \cup B$																													

$$A \cup B = \{1, 2, 3, 4, 6, 10\} \cup \{1, 4, 5, 6\} = \{1, 2, 3, 4, 5, 6, 10\}$$

Пересечение

Пересечение множеств реализуется через побитовое И:

$$A \cap B \Rightarrow A \& B$$

Используемые биты для представления множества																													
Неиспользовуемые биты																													
A																													
B																													
$A \cap B$																													

$$A \cap B = \{1, 2, 3, 4, 6, 10\} \cap \{1, 4, 5, 6\} = \{1, 4, 6\}$$

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				
<i>A</i>	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	1	0	0	0	0
<i>B</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0
<i>A \ B</i>	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	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				
<i>A</i>	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0	0	0	0	0
<i>B</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0
<i>A \ B</i>	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	0	0

$$A \Delta B = \{1, 2, 3, 4, 6, 10\} \Delta \{1, 4, 5, 6\} = \{2, 3, 5, 10\}$$

Дополнение

$$\overline{A} \Rightarrow \sim A$$

	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				
<i>A</i>	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0	0	0	0	0
<i>Ā</i>	1	1	1	1	1	1	1	...	1	1	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0

$$\overline{A} = U \setminus \{1, 2, 3, 4, 6, 10\} = \{0, 5, 7, 8, 9, 11, \dots, 25\}$$

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 сбросыванием 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, мы вызовем

```
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 $O(1)$.

13.2.2 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 байт¹. 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 }
```

¹Всегда 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 отличии 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 }
```

13.2.3 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:

A:	1	5	10	7				
B:	7	5	6					
		- неиспользуемые элементы, потенциально могут быть заняты при выполнении операций						

Порядок элементов в массиве зависит от порядка их включения в множество. Если в множество A добавляется элемент, он появится в самом конце:

A:	1	5	10	7	2					
----	---	---	----	---	---	--	--	--	--	--

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 множества:

```

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;
}
```

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.

13.2.4 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 14

Лабораторные работы

14.1 Лабораторная работа №1 «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);
```

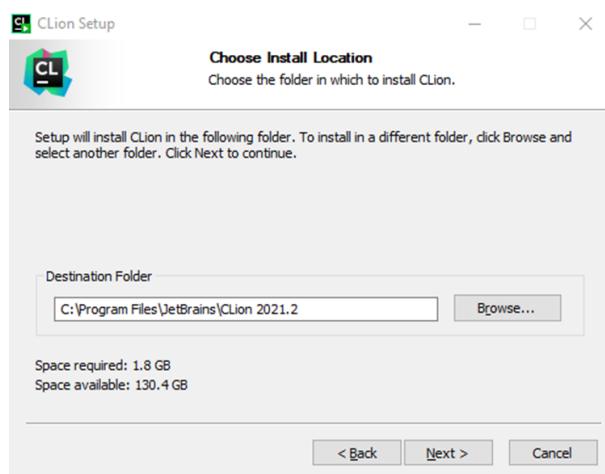
2. Какое значение вернет функция `printf("%4.2f", 345.789)` при успешном выполнении?
3. Какое значение вернет функция `scanf("%3f %4s %c", &f, s, &c)` при успешном выполнении?
4. Какие значения будут присвоены переменным соответствующих типов `f, s, c` после вызова функции `scanf("%3f %8s %c", &f, s, &c)`, если на клавиатуре набрано: 1234.56 Belgorod 308012?

Пояснения к лабораторной работе

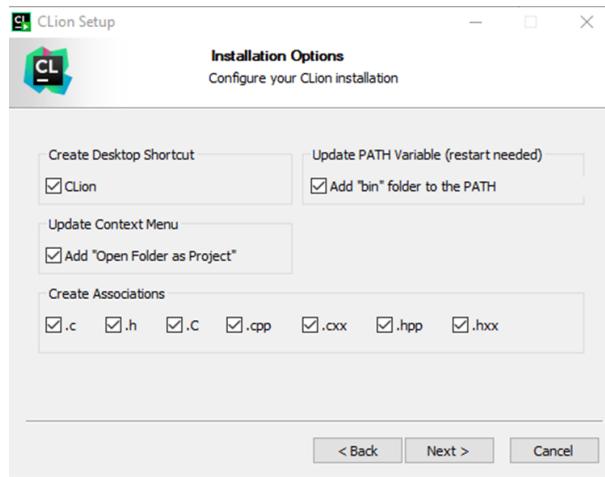
Для выполнения последующих работ вам потребуется компилятор и среда разработки¹.

Установка CLion:

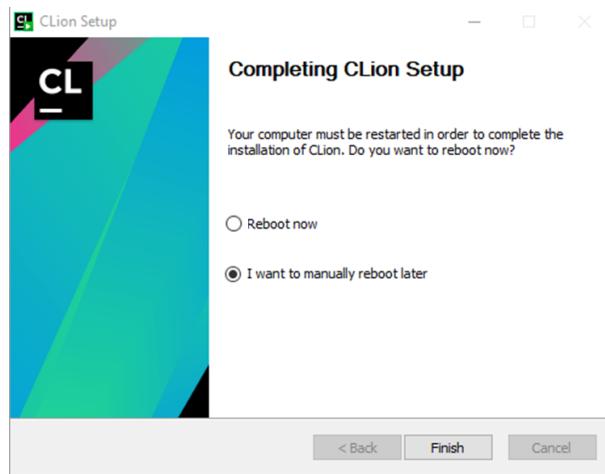
1. Выбираем свою систему и скачиваем установщик CLion с официального сайта: <https://www.jetbrains.com/clion/download/> (Далее будут представлены инструкции для системы Windows)
2. Проходим процедуру установки:



¹Инструкция любезно предоставлена Николотовым Александром

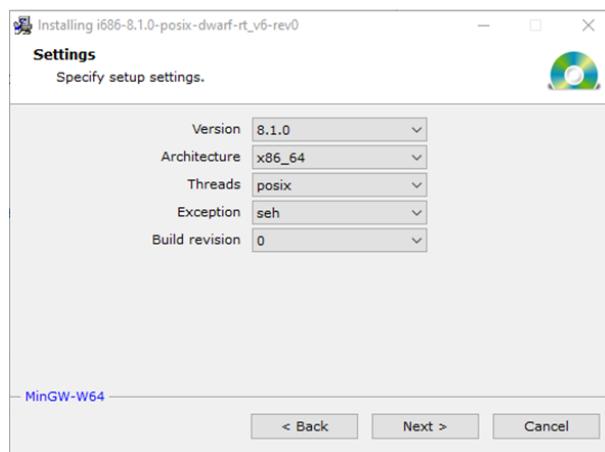


- Закрываем установщик. Перезагрузку мы выполним после установки компилятора.



Установка компилятора:

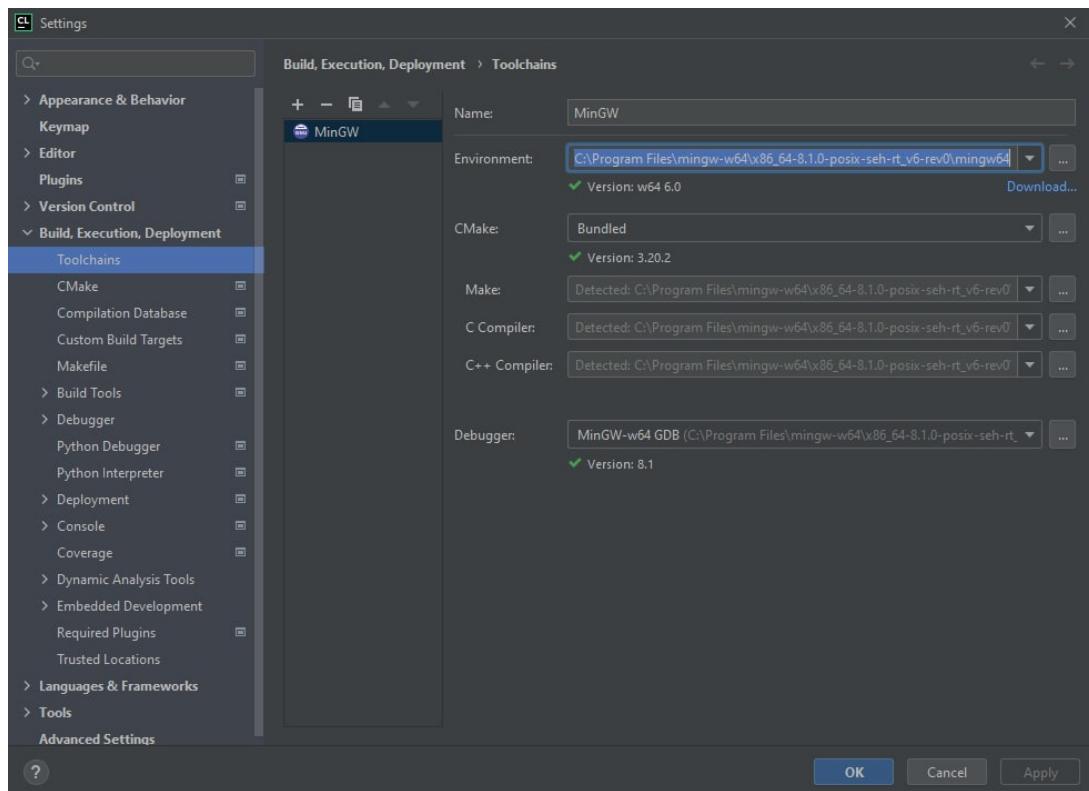
- Скачиваем установщик MinGw для своей системы: <https://www.mingw-w64.org/downloads/> (Для windows выбираем “MingW-W64-builds”).
- При установке выбираем “Architecture” - “x86_64”, в случае если у вас установлена 64-битная версия ОС, или “I686” - в противном случае.



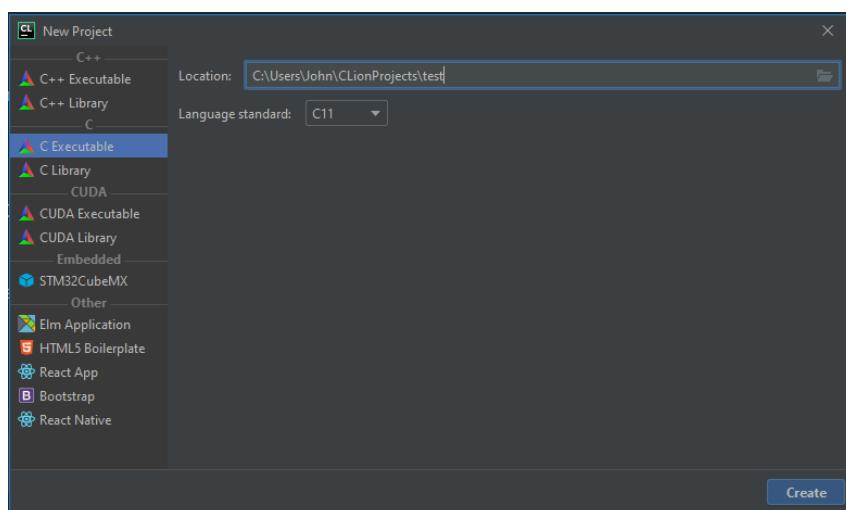
- После установки, перезагружаем ПК.

Старт работы в CLion:

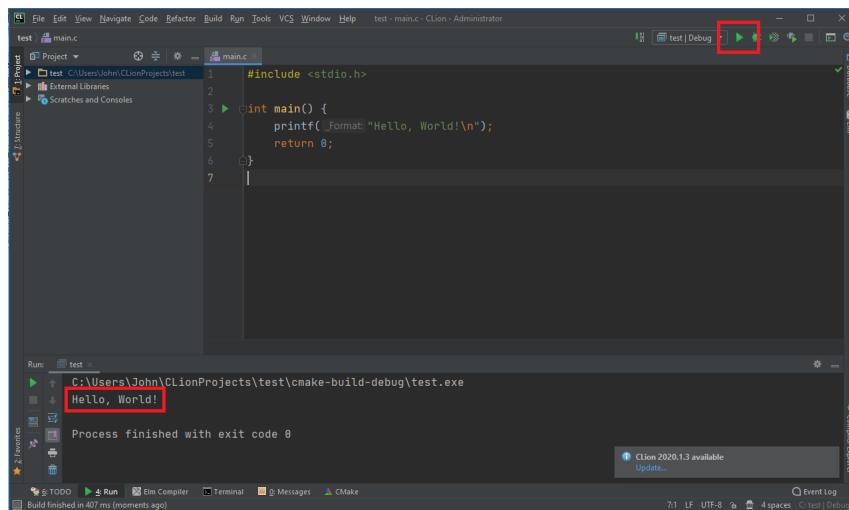
1. Запускаем CLion, через меню переходим в: File / Settings / Build, Execution, Deployment / Toolchains.
2. Выбираем “MinGW” и указываем путь к папке “mingw64”, например: “C:\Program Files\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0\mingw64”.



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 сниппеты. Важно чтобы **вам нравился ваш редактор и вы умели в нем ориентироваться**.
- **Как можно скорее научитесь пользоваться отладчиком.**

14.2 Лабораторная работа №2а «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 со звездочкой приложить блок-схему.
 - Исходный код.
 - Скриншот с *codeforces* с указанием вердикта тестирующей системы.
- Вывод.

Перечень задач:²

1. Минуты до Нового года (1283A)³
2. Опять двадцать пять! (630A)⁴
3. Игра (513A)⁵
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)⁶
11. *Покупка воды (1118A)
12. Блэкджек (104A)

²Щелчок по названию задачи переведёт на условие.

³Разбор задачи имеется в примере оформления на странице 337.

⁴Решения некоторых задач проще, чем кажется на первый взгляд. Если у вас возникает желание использовать циклы – это плохая идея.

⁵Для решения некоторых задач вам могут потребоваться не все исходные данные. Иногда достаточно воспользоваться частью.

⁶Если 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:

```

1 #include <stdio.h>
2
3 int main() {
4     int n_sets;
5     scanf("%d", &n_sets);
6
7     for (int set_number = 1; set_number <= n_sets; set_number++) {
8         // код для одного случая
9     }
10
11    return 0;
12 }
```

Только в этом случае допускается использовать циклы. Во всех других ситуациях использование циклов **категорически запрещено**.

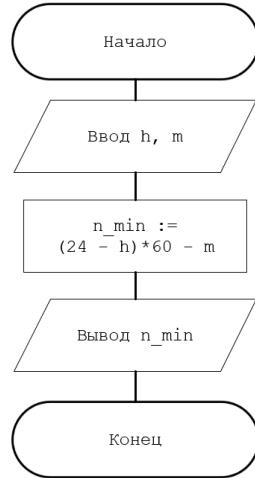
Рассмотрим в качестве примера первую задачу (Минуты до Нового года). В описании входных данных указано следующее:

Первая строка входных данных содержит одно целое число t ($1 \leq t \leq 1439$) – количество наборов входных данных.

Следующие t строк 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:

```

1 int h, m;
2 scanf("%d %d", &h, &m);
3
4 int n_min = (24 - h)*60 - m;
5
6 printf("%d\n", n_min);
  
```

Вставим полученный фрагмент в функцию *main*:

```

1 #include <stdio.h>
2
3 int main() {
4     int n_sets;
5     scanf("%d", &n_sets);
6
7     for (int set_number = 1; set_number <= n_sets; set_number++) {
8         int h, m;
9         scanf("%d %d", &h, &m);
10
11         int n_min = (24 - h)*60 - m;
12
13         printf("%d\n", n_min);
14     }
15
16     return 0;
17 }
  
```

На странице с задачей выбираем пункт "Отослать":

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		
Исходный код:	<pre> 1 #include <stdio.h> 2 3 int main() { 4 int n_sets; 5 scanf("%d", &n_sets); 6 7 for (int set_number = 1; set_number <= n_sets; set_number++) { 8 int h, m; 9 scanf("%d %d", &h, &m); 10 int n_min = (24 - h)*60 - m; 11 printf("%d\n", n_min); 12 } 13 14 return 0; 15 }</pre>		

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
126443394	19.08.2021 15:26	ispritchin	A - Минуты до Нового года	GNU C11	Полное решение	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 конструкций:

```

1 // if (<логическое выражение1>
2 //      <оператор1>
3 // else {
4 //     if (<логическое выражение2>
5 //         <оператор2>
6 //     else
7 //         <оператор3>
8 // }
```

вместо:

```

1 // if (<логическое выражение1>
2 //      <оператор1>
3 // else if (<логическое выражение2>
4 //         <оператор2>
5 // else
6 //     <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:

```

1 // if (<логическое выражение1>
2 //     <оператор1>
3 // else if (<логическое выражение2>
4 //     <оператор2>
5 // else
6 //     <оператор3>
```

но не так:

```

1 // if (<логическое выражение1>
2 //     <оператор1>
3 // if (<логическое выражение2>
4 //     <оператор2>
5 // if (<логическое выражение3>
6 //     <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 `vзде`, где это необходимо:

```
1 x += a
```

вместо

```
1 x = x + a
```

- Не стоит отделять объявление и инициализацию переменных:

```

1 int a = 10; // правильно
2
3 int b; // неправильно
4 b = 10
```

- Не объявляйте с инициализацией две и более переменных:

```

1 int a = 10; // правильно
2 int b = 20;
3
4 int a = 10, b = 20; // неправильно
```

- Имеются несколько способов округлить результат деления вверх (рекомендую остановиться на втором или третьем варианте):

- При помощи функции `ceil`, которая находится в стандартной библиотеке `math.h`⁷:

⁷Вариант может создавать погрешности, так как не работает с большими числами в силу приведения типа из `long long` в `double`. Не рекомендуется к использованию.

```
1 int x = ceil((double)a / b);
```

2. При помощи условного оператора *if – else*, тернарного оператора:

```
1 // через if-else
2 int x;
3 if (a % b)
4     x = a / b + 1;
5 else
6     x = a / b;
7
8 // через тернарный оператор ?:;
9 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:

```
1 x = a / b + (a % b != 0)
```

4. Пользуясь свойством⁸

$$\left\lceil \frac{a}{b} \right\rceil \equiv (a + b - 1) \text{ div } b$$

- После импортирования библиотек оставляйте пустую строку. Дополнительно рекомендуется вставлять пустые строки, если это улучшает читаемость кода, например, разделить ввод, обработку, и вывод:

```
1 #include <stdio.h>
2
3 int main() {
4     int n_sets;
5     scanf("%d", &n_sets);
6     // пустая строка, так как закончен ввод
7     for (int set_number = 1; set_number <= n_sets; set_number++) {
8         int h, m;
9         scanf("%d %d", &h, &m);
10        // пустая строка, так как закончен ввод
11        int n_min = (24 - h)*60 - m;
12        // пустая строка, так как закончилась обработка
13        printf("%d\n", n_min);
14    }
15    // пустая строка перед return 0
16    return 0;
17 }
```

- По возможности не осуществляйте вывод в ветках. Делайте так:

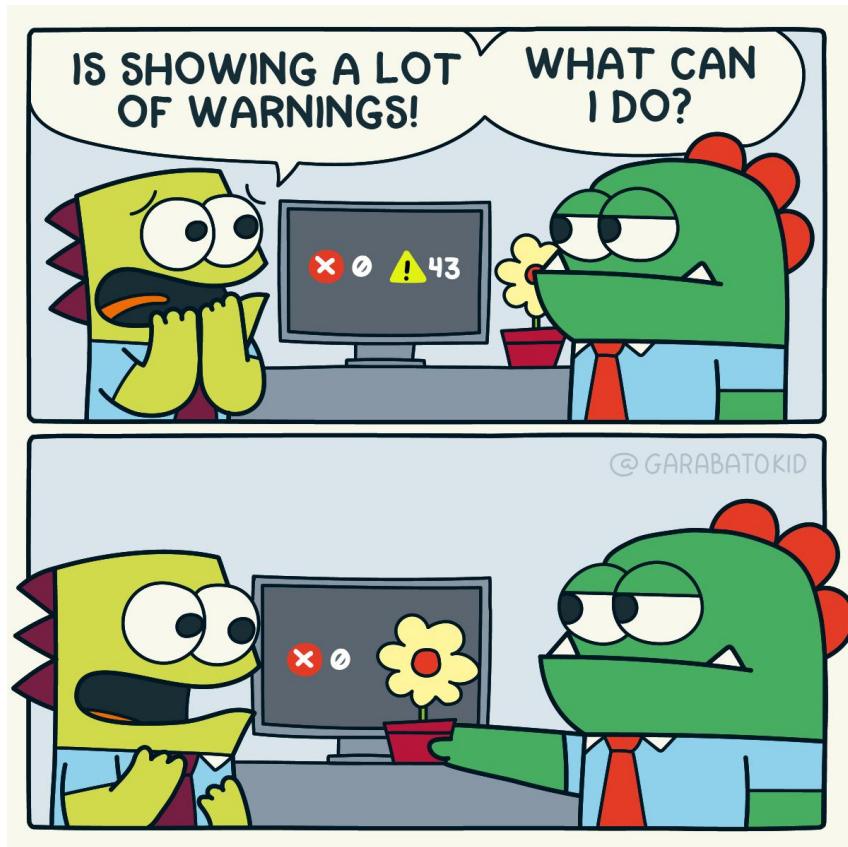
```
1 int max;
2 if (a > b)
3     max = a;
4 else
5     max = b;
6
7 printf("%d", max);
```

Вместо

```
1 if (a > b)
2     printf("%d", a);
3 else
4     printf("%d", b);
```

⁸ $\lceil a \rceil$ – обозначение округления в большую сторону для выражения a

- 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:

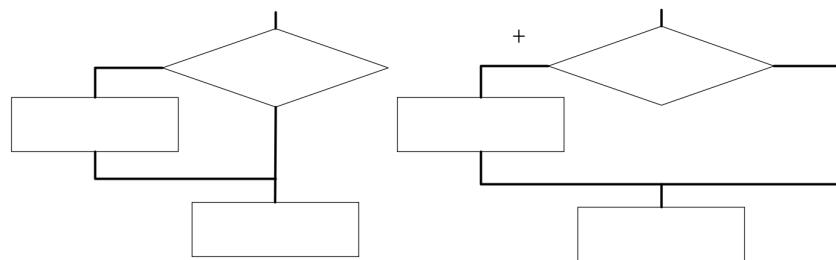
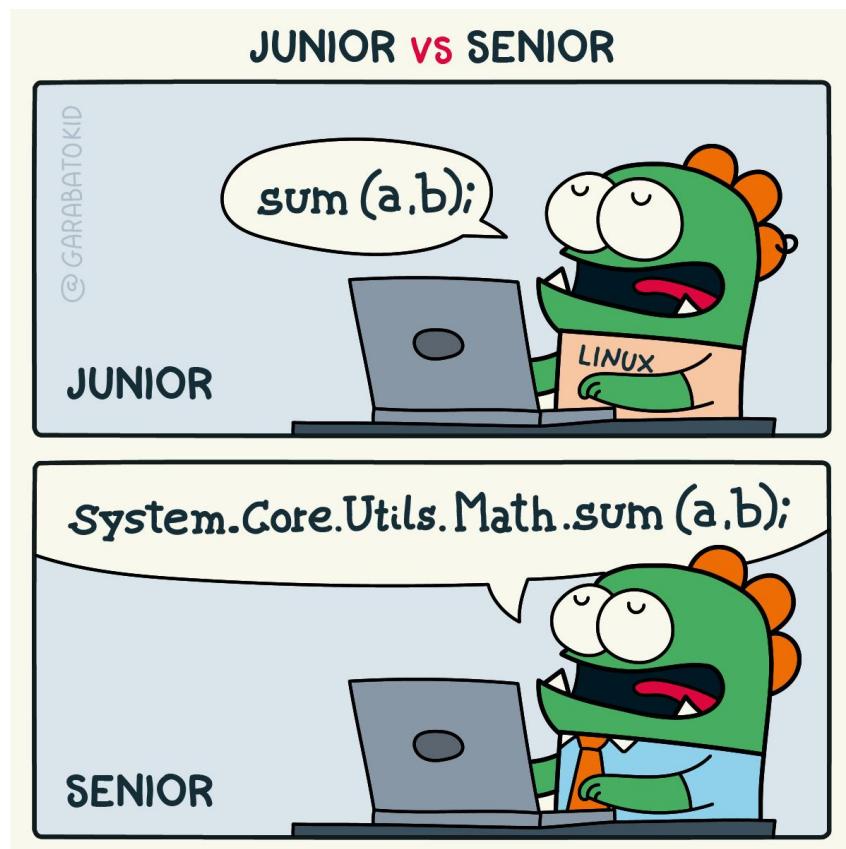


Рис. 14.1 – Вариант оформления справа является правильным

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 №2а «Алгоритмы разветвляющейся структуры»

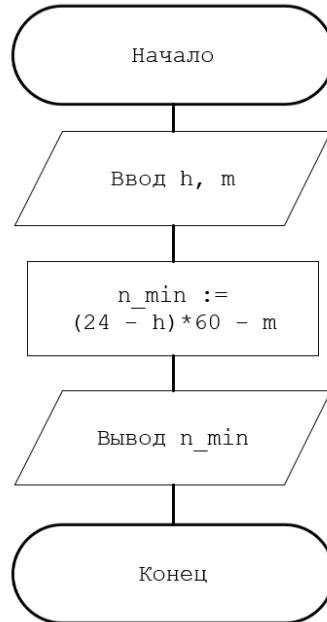
Цель работы: получение навыков написания линейных и разветвляющихся алгоритмов.

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:

```

1 #include <stdio.h>
2
3 int main() {
4     int nSets;
5     scanf("%d", &nSets);
6
7     for (int setNumber = 1; setNumber <= nSets; setNumber++) {
8         int h, m;
9         scanf("%d %d", &h, &m);
10
11        int nMin = (24 - h) * 60 - m;
12
13        printf("%d\n", nMin);
14    }
15
16    return 0;
17 }
```

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).

14.3 Лабораторная работа №2b «Алгоритмы разветвляющейся 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:⁹

1. Спасти Люка (624A)¹⁰
2. Поликарп и монеты (1551A)
3. Номер этажа (1426A)
4. Два кролика (1304A)¹¹
5. **Странная таблица (1506A)
6. Разделение последовательности (1102A)¹²

⁹Рекомендации:

- 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.

¹⁰В нескольких решениях задач, вместо явного приведения к *double* использовалось неявное: 1.0 * Не делайте так.

¹¹Избегайте дублирования вычислений. Создавайте вспомогательные переменные. Если видите, что какой-то фрагмент вычислений повторяется - создайте переменную.

¹²Попробуйте 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)¹³
10. Паша и палка (610A)
11. Не NP (805A)
12. ExAb И нОд (1325A)
13. Три кучки с конфетами (1196A)
14. Театральная площадь (1A)¹⁴
15. Найти Амира (804A)
16. * Пицца, пицца, пицца!!! (979A)¹⁵
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)¹⁶
26. Чунга-Чанга (1181A)
27. Отопление (1260A)¹⁷
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)¹⁸

¹³В 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.

¹⁴При необходимости округлить значения, следует использовать функции.

¹⁵Хорошее именование переменных может привести к исключению дублирования вычислений.

¹⁶Для вывода результата используйте `printf("%02d:%02d", hours, minutes);`

¹⁷Особое внимание именованию переменных в данной задаче.

¹⁸Допускается 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:

```

1 // возвращает минимальное значение переменных a и b
2 long long min2(long long a, long long b) {
3     return a < b ? a : b;
4 }
5
6 // возвращает максимальное значение переменных a и b
7 long long max2(long long a, long long b) {
8     return a > b ? a : b;
9 }
10
11 // возвращает максимальное значение среди переменных a, b, c
12 long long max3(long long a, long long b, long long c) {
13     long long max;
14     if (a > b && a > c)
15         max = a;
16     else
17         max = b > c ? b : c;
18     return max;
19 }
20
21 // возвращает значение дроби num / denum округлённое вверх
22 // примеры: 5 / 3 -> 2, 8 / 2 -> 4, 7 / 4 -> 2;
23 // пример вызова функции: ceil_frac(5, 3);
24 long long ceilFrac(long long a, long long b) {
25     return a % b ? a / b + 1 : a / b;
26 }
27
28 // возвращает модуль числа x
29 long long abs(long long x) {
30     return x >= 0 ? x : -x;
31 }
```

- 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:

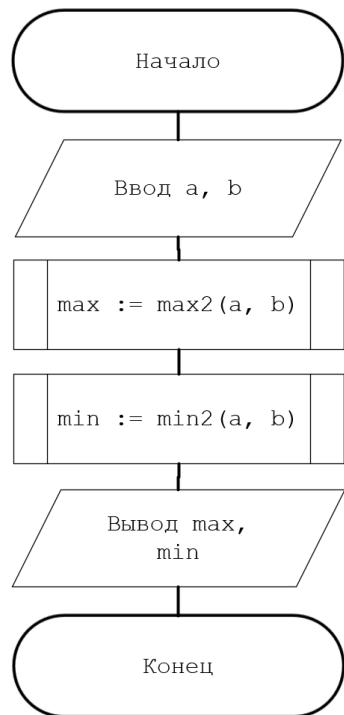


Рис. 14.2 – Пример оформления блок-схемы с вызовами функций

Дополнительные материалы

Рекомендуется прочесть главу 15 'Условные операторы' из книги Макконнелла 'Совершенный код'.

14.4 Лабораторная работа №3а «Циклы»

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 задач¹⁹:

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:

```
1 #include <stdio.h>
2 #include <windows.h>
3
4 int main() {
5     SetConsoleOutputCP(CP_UTF8);
6 }
7 }
```

3. С клавиатуры вводятся n ($n > 0$) чисел. Найти индекс первого минимального значения. Нумерация элементов - с нуля²⁰.
4. С клавиатуры вводятся n ($n > 0$) чисел. Найти индекс последнего максимального значения. Нумерация элементов - с нуля²¹.
5. С клавиатуры вводятся n ($n > 0$) чисел. Найти количество минимальных значений²².

¹⁹Проведите качественное тестирование приложений. Это указано в целях работы.

²⁰Вариант для именования результирующей переменной *firstMinIndex*. В ходе проверки работ было замечено, что в процессе решения выделяются лишние переменные.

²¹Переменную для индекса можно назвать *lastMaxIndex*.

²²Рекомендуемое имя для переменной-результата *minCount*. *Count* в конце намекает, что будет подсчитываться количество *min*.

6. * С клавиатуры вводятся n ($n > 0$) чисел. Найти разность между максимальным и минимальным значением.²³
7. С клавиатуры вводится последовательность. Признак конца ввода - 0. Найти сумму четных чисел²⁴.
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 числа²⁵.
11. Дано целое число n ($n > 0$). Проверить, входит ли в запись числа n данная цифра *digit k* раз²⁶.
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'²⁷. Определить количество букв²⁸.
15. С клавиатуры вводится символы. Признак конца ввода - символ перехода на новую строку '\n'. Определить количество согласных букв²⁹.
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 групп³⁰.
17. С клавиатуры вводятся символы. Признак конца ввода – точка. Определить сумму введенных цифр.

²³Проектируя решение, подумайте над вопросом: может ли вводимое значение быть больше максимума и меньше минимума одновременно. Подсказка: на каждой итерации цикла должно проверяться 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.

²⁴Вы хотите больше хороших имен? Возьмите `evenSum`. Больше о хорошем стиле именования переменных на странице 354.

²⁵Не используйте магические константы в духе 48 ('0') и 32 (' ' = 'a' - 'A'). **Магическими константами** называют плохую практику программирования, когда в исходном тексте встречается числовое значение и неочевиден его смысл.

²⁶Рекомендация по тестированию: проверяйте крайние случаи, когда количество цифр в числе равно k , меньше на 1, больше на 1.

²⁷Клавиша *Enter*.

²⁸В задачах, в которых идёт оперирование символами, используйте функцию `getchar`. Буквы вводятся в одном регистре. Рекомендуется (не требуется) выделение функции под эту задачу.

²⁹Предполагается, что буквы принадлежат одному регистру и одному языку. Мне кажется крайне удачной идеей использовать флаги `isLetter` и `isVowel` или `isConsonant`.

³⁰Подсказка: в данной задаче очень легко допустить ошибку. Рекомендуется использовать три флага для поиска решения.

18. ** С клавиатуры вводятся символы (пробелы и цифры). Признак конца ввода – точка. Определить сумму введенных чисел.
19. ** С клавиатуры вводятся вещественные числа³¹. 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 14.1.

Таблица 14.1: 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, но 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.

Некоторые задачи могут иметь несколько компонентов вводимых данных. Каждую компоненту описывайте с новой строки³² (пример для задачи 11):

Входные данные	Ожидаемый результат	Пояснение
$n = 14445$	'YES'	
$digit = 4$		Цифра встречается ровно требуемое количество раз.
$k = 3$		

³¹ Вещественные числа нельзя сравнивать на равенство. Правильный механизм сравнения находится в пояснениях к лабораторной работе.

³² Если смысл компоненты неочевиден, укажите её имя из условия задачи.

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 $fcompare^{33}$:

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define EPS 1e-12
5
6 int fcompare(double a, double b) {
7     return fabs(a - b) < EPS;
8 }
9
10 int main() {
11     double a, b;
12     scanf("%lf %lf", &a, &b);
13
14     // не равно - !fcompare(a, b);
15     if (!fcompare(a, b))
16         printf("Equal");
17     else
18         printf("Not equal");
19
20     return 0;
21 }
```

В задачах, где используется символьный ввод до какого-то признака конца ввода (например, `\n`), можно использовать следующий прием:

```

1 char symbol;
2 while ((symbol = getchar()) != '\n') {
3     // ...
4 }
```

вместо

```

1 char symbol = getchar();
2 while (symbol != '\n') {
3     // ...
4     symbol = getchar();
5 }
```

³³Специально использован префикс f намекающий на сравнение вещественных чисел.

Рекомендации по именованию:

- Общие:
 - Придерживайтесь единства.
 - 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`³⁴, даже если первое слово является именем собственным. Примеры: `lastMinIndex`, `moscowPosition`.
 - Избегайте общих имён `result`, `index`, `variable`, `data`, `temp`, `tmp`, 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 нулём (`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 `number0fCustomers` - 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`.
 - 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:

```

1 int setsTotal;
2 scanf("%d", &setsTotal);
3
4 for (int setIndex = 1; setIndex <= setsTotal; setIndex++) {
5
6 }
```

³⁴На самом деле, для языка C должен применяться `snake_case`, но в силу того, что дальше ожидается переход на C++, принято решение использовать `camelCase`-стиль сразу.

Особенно стоит придерживаться данного правила для вложенных циклов.

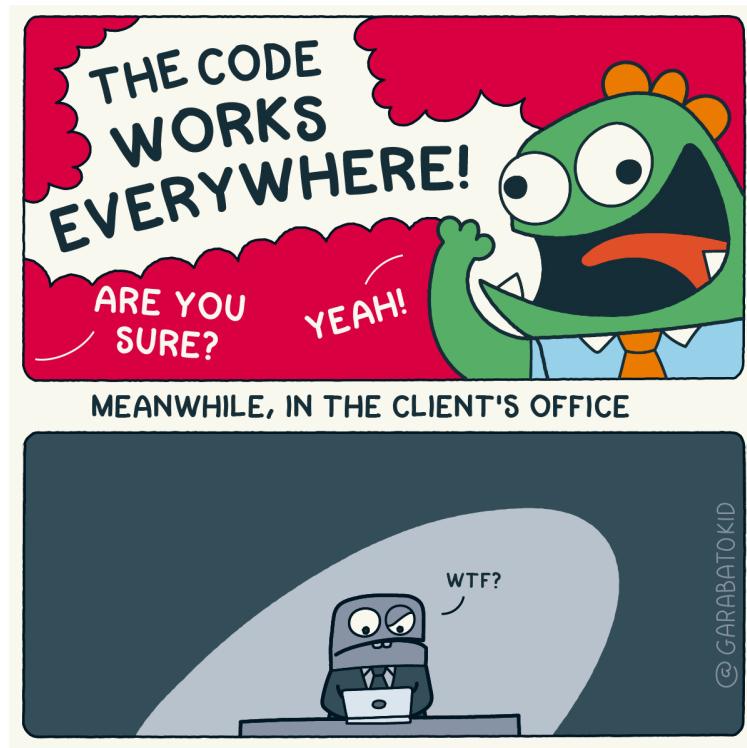
- Переменные статуса не должны содержать в себе слово *flag*. Имя флага не должно включать фрагмент *flag*, потому что он ничего не говорит о его сути. Рекомендуется использовать префиксы *is* для данных переменных: `isError`, `isFound`, `isProcessingComplete`, что превращается в вопрос. Переменные-флаги 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`.



Пример оформления отчета

Лабораторная работа №3а «Циклы»

Цель работы: получение навыков написания циклических алгоритмов и проведения ручного тестирования.

Содержание отчета:

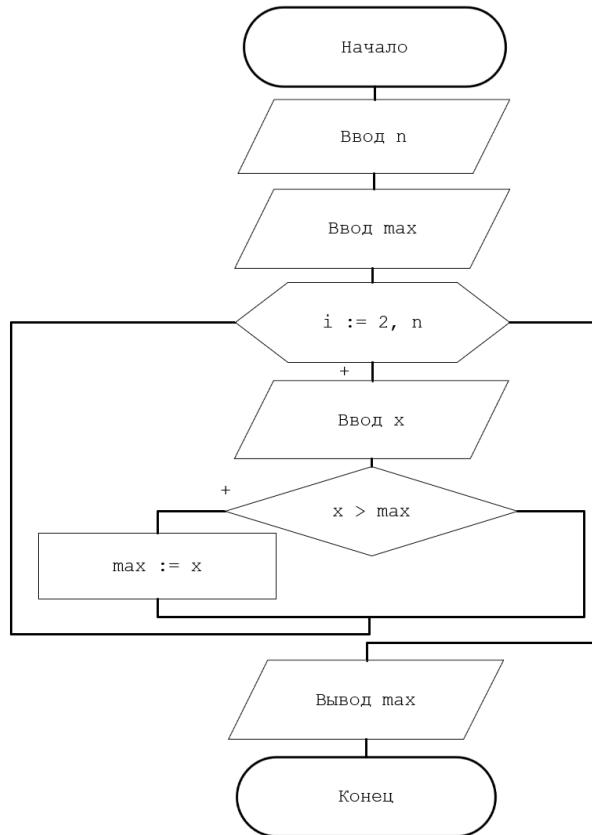
- Тема лабораторной работы
- Цель лабораторной работы
- Решения задач (пример оформления задачи представлен на следующей странице). Для каждой задачи указать:
 - Условие задачи.
 - Для задач с звездочкой приложить блок-схему.
 - Задачи с двумя звездочками допускается пропустить.
 - Тестовые данные, на 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:

```

1 #include <stdio.h>
2
3 int main () {
4     int n;
5     scanf ("%d", &n);
6
7     int max;
8     scanf ("%d", &max);
9     for (int i = 2; i <= n; i++) {
10         int x;
11         scanf ("%d", &x);
12
13         if (x > max)
14             max = x;
15     }
16
17     printf ("%d", max);
18
19     return 0;
20 }
```

Вывод: в ходе работы получены навыки написания циклических алгоритмов, получены навыки проведения ручного тестирования.

14.5 Лабораторная работа №3b «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 задач³⁵:

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)

³⁵Во 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.

14.6 Лабораторная работа №4а «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 данные³⁶.
 - Исходный код функции и её спецификацию.
- Задачи с двумя звездочками допускается не решать.

Требования к лабораторной работе:

- Требования к оформлению спецификаций:



³⁶ 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)³⁷.
- **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:

```

1 if (< логическое выражение >)
2     return 1;
3 return 0;
1 if (< логическое выражение >)
2     return 1;
3 else
4     return 0;

```

вместо:

```
1 return < логическое выражение >
```

- При возврате выражения

```
1 return < выражение >
```

не ставьте скобки:

```
1 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 .
2. Напишите функцию `max2`, которая возвращает максимальное значение из двух целочисленных переменных типа `int`³⁸.
3. Напишите функцию `max3`, которая возвращает максимальное значение из трёх целочисленных переменных типа `int`. Используйте при решении функцию `max2`.

³⁷Отличный шрифт для параметров необходим для того, чтобы они не сливались с остальным текстом назначения.

³⁸Тело функций для задач 1, 2, 3, 6 должны содержать одну строку и не использовать вспомогательные переменные.

4. Напишите функцию *getDistance*, которая вычисляет расстояние между двумя точками, заданными целочисленными координатами (x_1, y_1) , (x_2, y_2) ³⁹

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 сообщение⁴⁰.

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 значения⁴¹.

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)⁴².

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.

³⁹Ещё 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`).

⁴⁰Обратите внимание, что в задании имеется слово 'выводит', следовательно, вывод внутри функции допустим.

Не дублируйте вычисление квадратного корня. Не извлекайте корень из отрицательных чисел.

⁴¹Грамотное назначение для данной функции: обменивает значения переменных по адресам *a* и *b*.

⁴²Не 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.

12. Напишите функцию *isPrime*, которая возвращает значение 'истина', если число является простым, иначе – 'ложь'. Приложите 3 вариации⁴³:

- (а) Без оптимизаций
- (б) С оптимизацией перебора до \sqrt{N} .
- (с) С оптимизацией перебора до \sqrt{N} и шагом 2.

13. Напишите функцию *deleteOctNumber*, которая удаляет цифру *digit* в записи данного восьмеричного числа *x*⁴⁴:

Входные данные	Выходные данные
$3179_{10} = 110'001'101'011_2 = 6\underline{1}53_8$ <i>digit</i> = 1	$653_8 = 110'101'011_2 = 427_{10}$
$9_{10} = \underline{1}'001_2 = \underline{1}1_8$ <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 изменения⁴⁵.

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.

Входные данные	Выходные данные
$77_{10} = 100'1101_2 = 4D_{16}$	$D4_{16} = 1101'0100_2 = 212_{10}$
$2732_{10} = 1010'1010'1100_2 = AAC_{16}$	$CAA_{16} = 1100'1010'1010_2 = 3242_{10}$

⁴³Извлечение 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.

⁴⁴Исходное 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.

⁴⁵Решение должно быть выполнено без подсчёта количества битов.

16. Напишите функцию *isBinPoly*, которая возвращает значение 'истина', если число *x* является палиндромом в двоичном представлении, иначе - 'ложь'⁴⁶.

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
$sh_1 = 0000000000001100_2 = 12_{10}$	$00000000000000000000000000000000110100100_2 = 420_{10}$
$sh_2 = 000000000010010_2 = 18_{10}$	
$sh_1 = 011111100000000_2 = 32512_{10}$	$00101010101010000000000000000000_2 = 715784192_{10}$
$sh_2 = 000000000000000_2 = 0_{10}$	

18. Вывести восьмеричное представление записи числа x^{47} .
 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
$x = 27_{10} = 11011_2$ $k = 1$	$10111_2 = 23_{10}$
$x = 27_{10} = 11011_2$ $k = 2$	$1111_2 = 15_{10}$
$x = 42_{10} = 101010_2$ $k = 1$	$10101_2 = 21_{10}$

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
$x = 1_{10} = 1_2$	$1_2 = 1_{10}$
$x = 2_{10} = 10_2$	$1_2 = 1_{10}$
$x = 3_{10} = 11_2$	$1_2 = 1_{10}$
$x = 4_{10} = 100_2$	$10_2 = 2_{10}$
$x = 10_{10} = 1010_2$	$11_2 = 3_{10}$
$x = 40_{10} = 101000_2$	$110_2 = 6_{10}$

⁴⁶Решение должно быть выполнено без подсчёта количества битов.

⁴⁷ Вариантом `printf("%o", x)` пользоваться запрещено. Решение должно достигаться за один цикл. Будет ошибкой получить десятичное представление числа и выводить его.

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 задач⁴⁸:

1. **Новогодняя гирлянда (1279A)
2. **Квадрат? (1351B)
3. **Поход за едой (876A)
4. **Пакеты с монетами (1037A)

⁴⁸Просто 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`.

```

1 // возвращает среднее суммы модулей значений a и b.
2 double absAverage(int a, int b) {
3     if (a < 0)
4         a = -a;
5     if (b < 0)
6         b = -b;
7     return (a + b) / 2.0;
8 }
```

14.7 Лабораторная работа №4б «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.⁴⁹ 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.

⁴⁹Если 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 функции⁵⁰:

- 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 позиций⁵¹.
- ** 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'.

⁵⁰ Для задач поиска, возвращать значение -1 , если элемент не найден.

Если реализуется вариант с функцией-предикатом, решение простого варианта необязательно.

⁵¹ 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*.

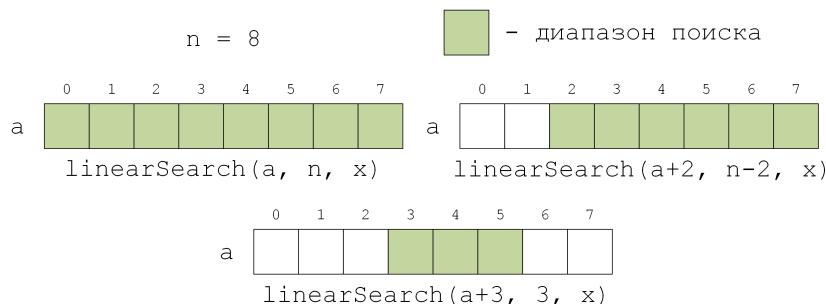
Для задач без ** выполните самостоятельную проверку реализованных функций (сверившись с пособием).

2. Перечень задач:

- Если возможно, то упорядочить данный массив размера *n* по убыванию, иначе массив **оставить без изменения**.⁵² 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:

```
1 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 последовательности⁵³, 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

⁵²Массив не может быть упорядочен по убыванию, если содержит одинаковые элементы. Вспомогательным массивом не пользоваться

⁵³В 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 геометрическое⁵⁴ положительных членов.⁵⁵

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 второй⁵⁶.

Входные данные	Выходные данные
$a = \{1, 2, 4\}$	$c = \{2, 1\}$
$b = \{4\}$	
$a = \{1, 2, 2, 4\}$	$c = \{2, 2\}$
$b = \{1, 4\}$	
$a = \{1, 2, 2, 4\}$	$c = \{4, 2, 2, 1\}$
$b = \{3\}$	
$a = \{1, 3, 3, 4, 6, 8, 8, 9, 10, 12, 12, 13, 15, 15\}$	$c = \{1, 4, 6, 10, 12, 12, 13, 15, 15\}$
$b = \{0, 3, 5, 5, 8, 9, 9, 11, 14, 14\}$	

⁵⁴ Средним геометрическим нескольких положительных вещественных чисел называется такое число, которым можно заменить каждое из этих чисел так, чтобы их произведение не изменилось.

⁵⁵ Функции проверки на упорядоченность не должны содержать лишние проверки. Если на каком-то этапе можно сказать, что массив неупорядочен - необходимо прервать работу функции.

⁵⁶ 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.

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 неубыванию⁵⁷.

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 сверху⁵⁸.

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 ∞
1 1 5 5 5 $x = 3$	1 5

8. Данна целочисленная последовательность. Получить массив из уникальных элементов последовательности⁵⁹.

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. Разрешимое допущение: знаменатель прогрессии – целое число.

⁵⁷ Для решения используйте произвольную функцию сортировки и функцию *reverse*. Использование двух сортировок будет считаться ошибкой.

⁵⁸ Не использовать функции сортировки. Выполнить поиск за один проход с использованием лишь одной функции.

⁵⁹ 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	"Yes"
1 2 5	"No"
1 1	"Yes"
0 1	"No"
1 3 0	"No"
-1 -4 -16 2 8	"Yes"
1 2 -4 -8 -16	"No"
1 1 -1 -1 -1	"Yes"
1 1 1 1 -1	"No"
0 0 0	"No"
1 2 4 4 4 4 8	"No"
1 1 -1	"Yes"

10. **Найти сумму четных цифр элементов массива из положительных чисел⁶⁰.

Пример оформления задачи из первого блока

Задача №1. Ввод массива a размера n .

Код функции:

```

1 // ввод массива a размера n
2 void inputArray(int * const a, const size_t n) {
3     for (size_t i = 0; i < n; i++)
4         scanf("%d", &a[i]);
5 }
```

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 239. Он содержит:

1. Тестовые данные.
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).

⁶⁰Попробуйте использовать функцию `forEach`.

14.8 Лабораторная работа №4с «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

```
1 void qsort(void *ptr, size_t count, size_t size,
2           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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define SIZE 5
5
6 int compare_ints(const void* a, const void* b) {
7     int arg1 = *(const int*)a;
8     int arg2 = *(const int*)b;
```

```

10     if (arg1 < arg2) return -1;
11     if (arg1 > arg2) return 1;
12     return 0;
13 }
14
15 int main() {
16     int a[SIZE] = {5, 2, 4, 3, 1};
17
18     qsort(a, SIZE, sizeof(int), compare_ints);
19
20     for (size_t i = 0; i < SIZE; i++)
21         printf("%d ", a[i]);
22
23     return 0;
24 }
```

Перечень задач:

1. Двоичный поиск.
2. Ближайшее слева.
3. Ближайшее справа.
4. Быстрый поиск в массиве.
5. Веревочки.
6. Очень Легкая Задача.
7. Гамбургеры.

Для чтения исходных данных воспользуйтесь следующим кодом:

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX_RECIPE_LENGTH 100
5
6 void getRecipe(char *recipe, int *nBread, int *nSausage, int *
7     nCheese) {
8     // strlen - функция, определенная в string.h для вычисления длины строки
9     int nIngredients = strlen(recipe);
10    *nBread = 0;
11    *nSausage = 0;
12    *nCheese = 0;
13    for (int ingredientIndex = 0; ingredientIndex < nIngredients;
14        ingredientIndex++) {
15        switch (recipe[ingredientIndex]) {
16            case 'B':
17                (*nBread)++;
18                break;
19            case 'S':
20                (*nSausage)++;
21                break;
22            case 'C':
23                (*nCheese)++;
24                break;
25        }
26    }
27 }
```

```

27 int main() {
28     char recipe[MAX_RECIPE_LENGTH + 1]; // +1 - под ноль-символ
29
30     int nBread, nSausage, nCheese;
31     gets(recipe);
32     getRecipe(recipe, &nBread, &nSausage, &nCheese);
33
34     //...
35     return 0;
36 }
```

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. **Спасти больше мышек.
18. **Slay the Dragon.
19. **Детский праздник.⁶¹
20. **Удаление двух элементов⁶²

⁶¹Очень 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.

⁶²Эта задача (Уверен, что её можно решить без бинарного поиска. Вспоминается высказывание: я предполагаю, что если единственный инструмент, который вы имеете – молоток, то заманчиво рассматривать всё как гвозди.)

14.9 Лабораторная работа №4d «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 обёртку, 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:

```

1 #include <ctype.h>
2
3 //...
4 isdigit('8');    // 'истина'
5 isspace(' ');    // 'истина'
6 isspace('\t');   // 'истина'
7 isspace('1');    // 'ложь'
8 isupper('A');    // 'истина'
```

Условия задач:

1. Определить количество цифр в тексте, вводимом с клавиатуры. Текст заканчивается символом перехода на новую строку `\n`.⁶³
2. Вывести данное натуральное число в восьмеричной системе счисления.
3. Дан знаменатель и первый член геометрической прогрессии. Вычислить n -й член прогрессии.

⁶³Текст 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**⁶⁴ 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 неубыванию⁶⁵.
6. *Найти номер первого вхождения минимального значения в последовательность длины n (линейный поиск)⁶⁶.
7. Даны натуральные числа a и b . Определить, могут ли эти числа быть **соседними** членами последовательности Фибоначчи. Последовательность Фибоначчи задается следующим образом⁶⁷:

$$f_1 = f_2 = 1 \quad f_i = f_{i-1} + f_{i-2} \text{ для } i > 2$$

8. Вывести в обратном порядке символы данного текста, вводимого с клавиатуры, которые не являются цифрами. Текст заканчивается символом перехода на новую строку `\n`⁶⁸.
9. Дан n -й член арифметической прогрессии, ее разность и значение n . Вычислить первый член прогрессии⁶⁹.
10. * С клавиатуры вводятся положительные вещественные числа a_1, a_2, \dots, a_n . Признак конца ввода – отрицательное число⁷⁰. 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`⁷¹, 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.

⁶⁴Решение каким-то образом должно использовать упорядоченность элементов в массиве. Ответьте себе на вопрос: "Каким образом я использовал упорядоченность элементов массива?" Обратитесь к следующему примеру: $a = \{10, 9, 8, 2, 0\}$, $x = 3$.

⁶⁵Функцию-обёртку не использовать. Решение может быть записано с использованием не более чем одного `if-else`. Функция не должна содержать больше двух формальных параметров.

⁶⁶Рекурсивная функция не должна использовать более четырёх параметров.

⁶⁷Саму последовательность чисел Фибоначчи не хранить.

⁶⁸Функция должна содержать ровно один рекурсивный вызов.

⁶⁹Рекурсивная функция содержит не более четырёх строк и не создаёт внутри никаких вспомогательных переменных.

⁷⁰Вводимую последовательность в массиве не хранить.

⁷¹И 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.

14.10 Лабораторная работа №4е «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:

```

1 struct point {
2     double x;
3     double y;
4 };
5
6 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`⁷².
Заголовок функции ввода: `void inputPoints(point *p, int n)`.
Заголовок функции вывода: `void outputPoints(point *p, int n)`.

⁷²Функция 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`.⁷³

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

$$ax + by + c = 0$$

```

1 struct line {
2     double a;
3     double b;
4     double c;
5 };

```

- (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)`.

- (d) Напишите код для создания линии из точек, без явного создания структур `p1` и `p2`.

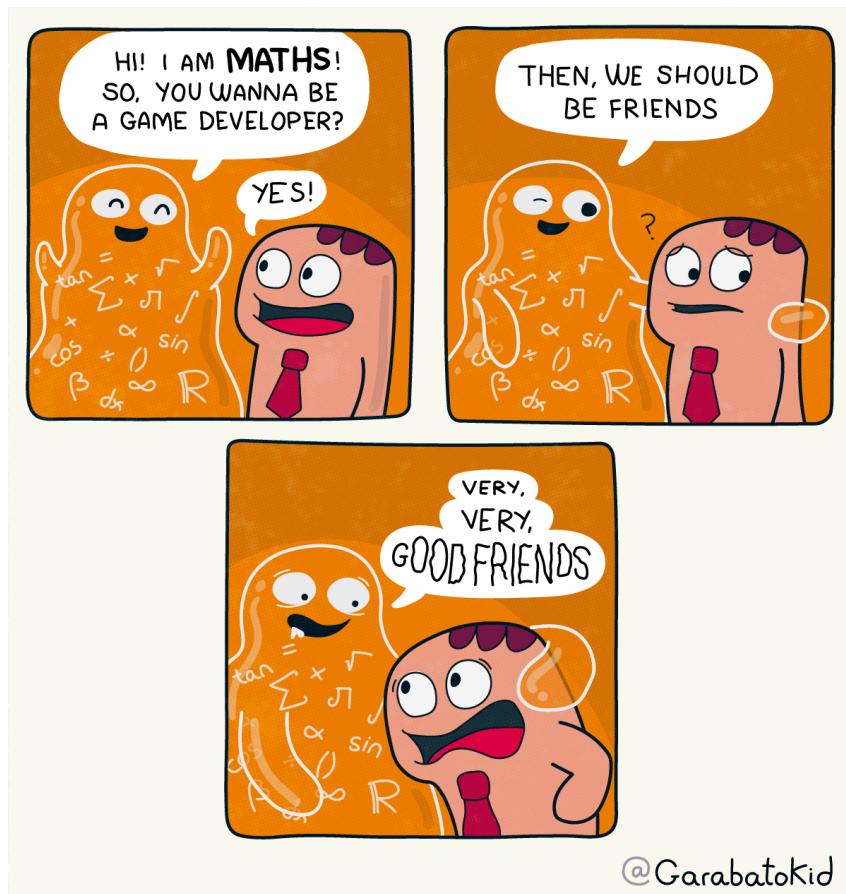
- (e) Реализуйте функцию `outputLineEquation` вывода уравнения прямой `line`.

Заголовок: `void outputLineEquation(line line)`.

⁷³Используйте функции `isEqualPoint` и `getMiddlePoint`.

Рекомендуемая реализация:

```
1 void outputLineEquation(line line) {
2     printf("%+.21fx%+.21fy%+.21f = 0", line.a, line.b, line.c);
3 }
```



- (f) Реализуйте функцию `isParallel`, которая возвращает значение 'истина' если прямые `line1` и `line2` параллельны, 'ложь' – в противном случае.
Заголовок: `int isParallel(line 11, line 12)`.
- (g) Реализуйте функцию `isPerpendicular`, которая возвращает значение 'истина' если прямые `line1` и `line2` перпендикулярны, 'ложь' – в противном случае.
Заголовок: `int isPerpendicular(line 11, line 12)`.
- (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 11, line 12)`.

3. Опишите структуру `circle`, которая задаёт окружность посредством центра окружности `center(x0, y0)`, и радиуса `r`.

```
1 struct circle {
2     point center; // центр окружности
```

```

3     double r;      // радиус
4 };

```

- (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`:

```

1 struct fraction {
2     int numerator;    // числитель
3     int denominator; // знаменатель
4 };

```

- (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 кратное⁷⁴.

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`⁷⁵.

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`⁷⁶.

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 оценки⁷⁷. Вывести фамилии оставшихся студентов. Указание: используйте структуру

```

1 #define N_MARKS 5
2
3 struct student {
4     char surname[20];
5     int marks[N_MARKS];
6 };

```

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.

⁷⁴Функция должна использовать в себе вызов `gcd`.

⁷⁵Функция должна использовать в себе вызов `mulFractions`.

⁷⁶Функция должна использовать в себе вызов `addFractions`.

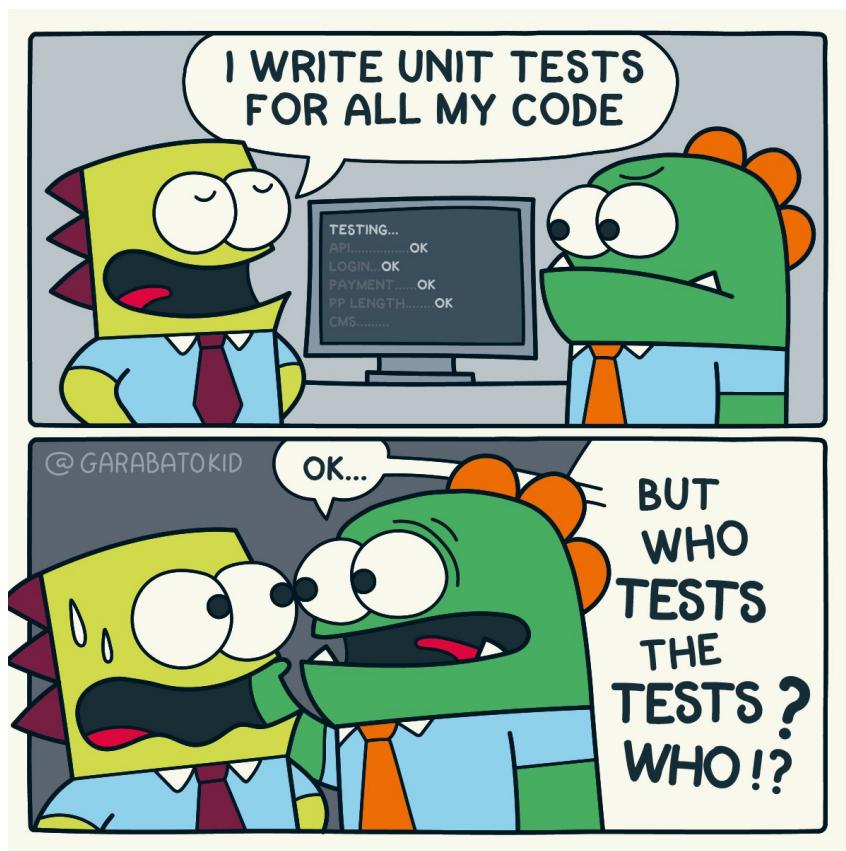
⁷⁷Используйте алгоритм однопроходного удаления. Выделите функцию `isGoodStudent`, которая возвращает значение 'истина', если студент 'хороший', иначе – 'ложь'.

14.11 Лабораторная работа №5а «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 a 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

```

⁷⁸ В реализации используйте функцию *unordered_array_set_insert*. Тело функции не должно содержать более 4 строк.

⁷⁹ 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

```

⁸⁰Элементы 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

⁸¹Запрещено использовать функцию `unordered_array_set_deleteElement`.

⁸²Перед получением ответа проверьте, что множество `set` является подмножеством `universumSet`.

⁸³Существует 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

```

⁸⁴Решение должно использовать функцию `memcmp`.

⁸⁵Особое 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. Шедрый 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 }
```

14.12 Лабораторная работа №5b «Реализация структуры данных «Вектор»

Цель работы: усовершенствование навыков в создании библиотек, получение навыков работы с системой контроля версий *git*.⁸⁶



Содержание отчета:⁸⁷

- Тема лабораторной работы.
- Цель лабораторной работы.
- Ссылка на открытый репозиторий с решением.
- Исходный код файлов⁸⁸:
 - `vector.h` / `vector.c`
 - `*vectorVoid.h` / `vectorVoid.c`
 - `main.c`
- Результат выполнения команд⁸⁹

⁸⁶Было время, когда я трижды переделывал большую работу с базами данных, будучи студентом, так как ломал проект. Если бы я тогда пользовался системами контроля версий....

⁸⁷Фрагменты отчета, помеченные звёздочкой требуются для получения максимального балла.

⁸⁸Используйте такой шрифт, чтобы строки кода умещались по ширине.

⁸⁹Оформление выполняется исключительно скриншотом.

```
1 git log --stat -- libs/data_structures/vector/ main.c
2 *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. 14.3 – Внешний вид страницы сайта

- Выберите свою операционную систему (далее все пояснения будут для обладателей лучшей операционной системы *Windows*);



Рис. 14.4 – Внешний вид страницы сайта

3. Выберите установщик в соответствии с разрядностью вашей системы;

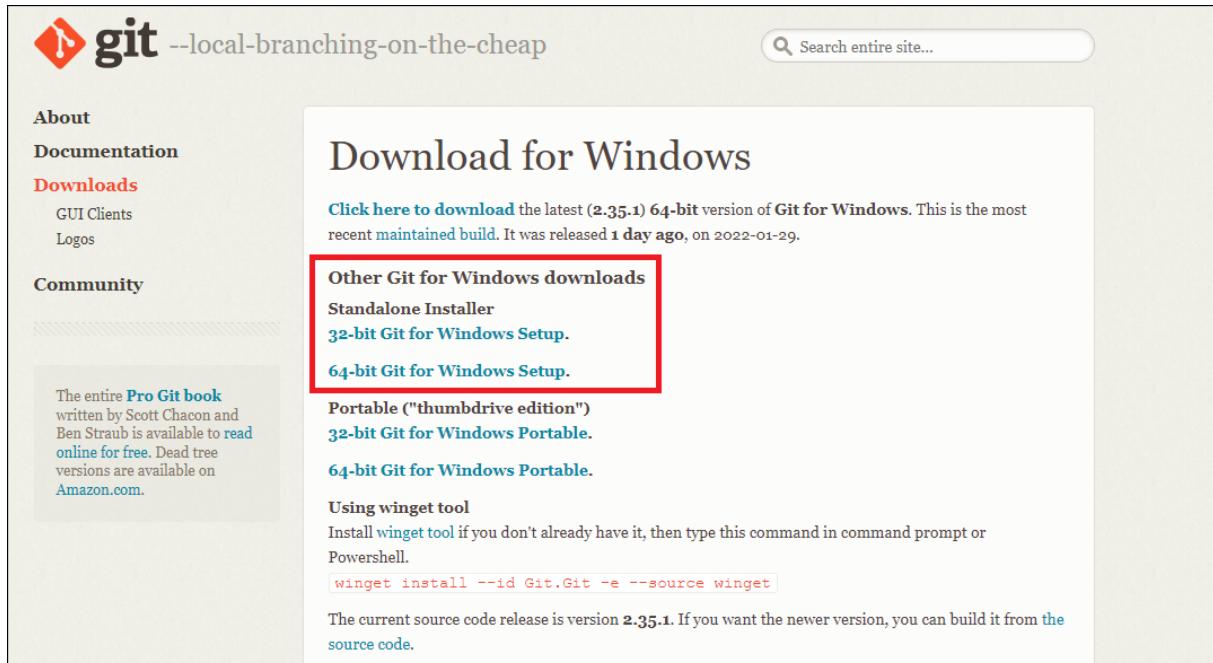


Рис. 14.5 – Выбор установщика *Git*

4. Проходим процедуру установки. **Не меняйте** никакие параметры при установке. При желании, можно изменить **только** расположение файлов установщика;

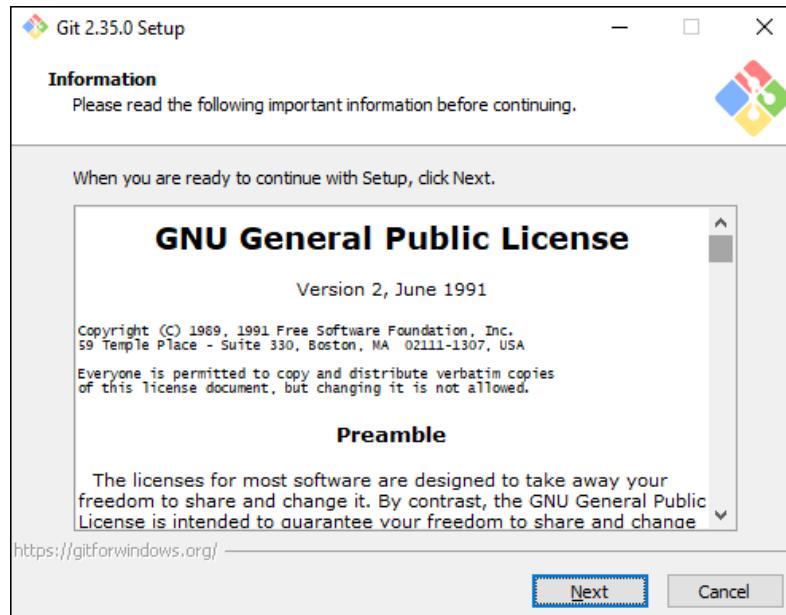


Рис. 14.6 – Начальное окно установки

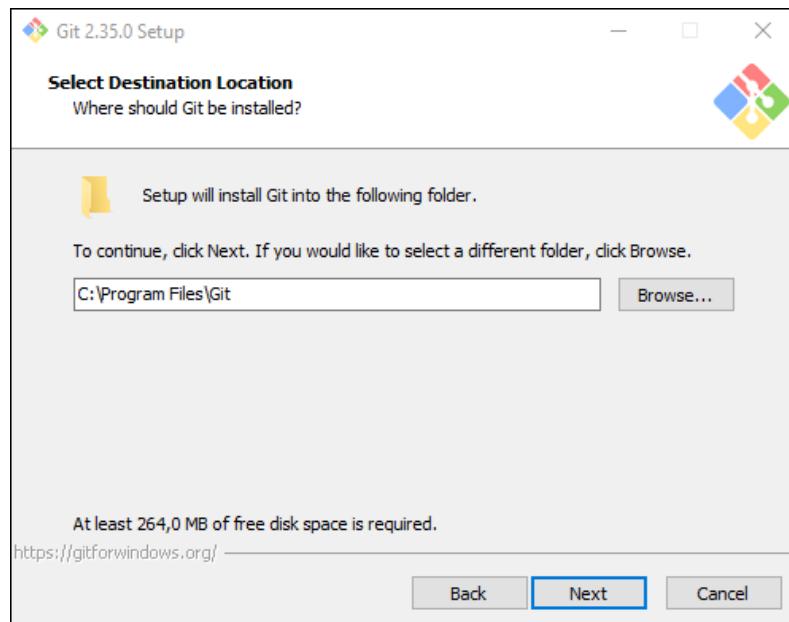


Рис. 14.7 – Окно выбора расположения файлов установщика

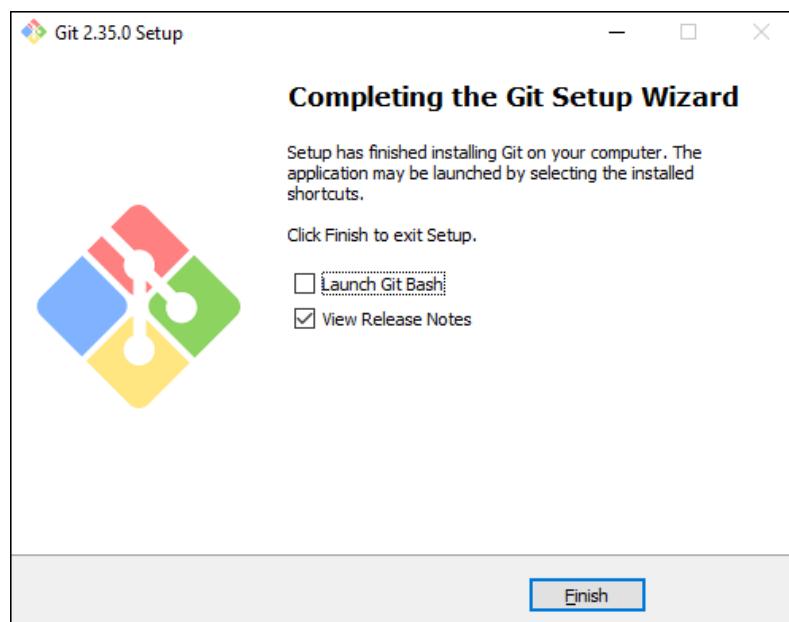
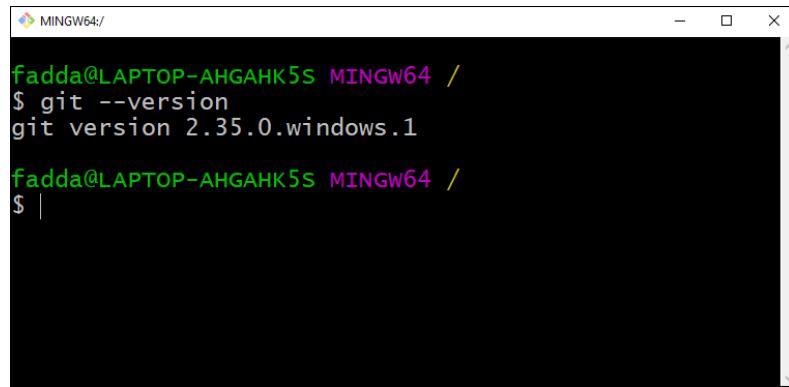


Рис. 14.8 – Окно завершения установки *Git*

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;

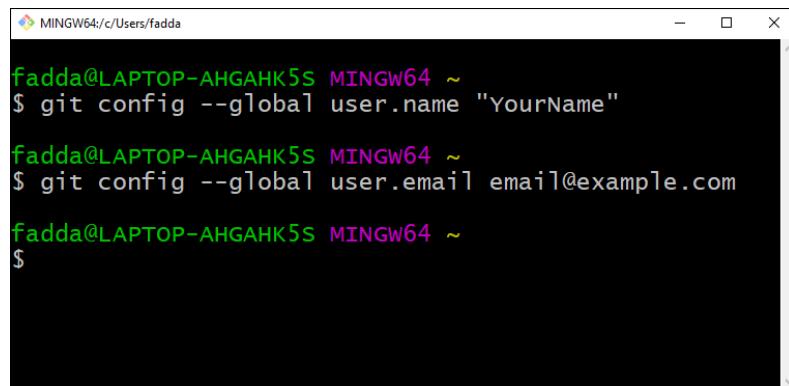


```
Fadda@LAPTOP-AHGAHK5S MINGW64 /
$ git --version
git version 2.35.0.windows.1
```

Рис. 14.9 – Окно *Git Bash*

8. Введите данные команды, чтобы указать ваше имя и адрес электронной почты:

```
1 git config --global user.name "Ваш< псевдоним>"
2 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 ~
$
```

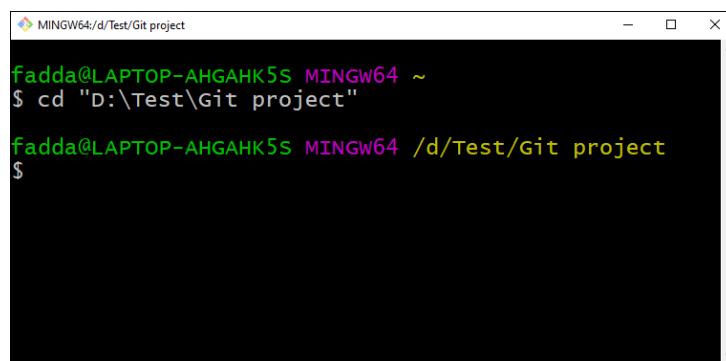
Рис. 14.10 – Окно *Git Bash*

9. Существует 2 способа перехода к определённой директории в *Git Bash*:

- 1) Использовать команду в *Git Bash* для перехода в определённую директорию:

```
1 cd "<расположение директории>"
```

Для того, чтобы вставить скопированный текст, используйте комбинацию клавиш (*Shift + Ins*), или через ПКМ по окну консоли.



```
Fadda@LAPTOP-AHGAHK5S MINGW64 ~
$ cd "D:\Test\Git project"
Fadda@LAPTOP-AHGAHK5S MINGW64 /d/Test/Git project
$
```

Рис. 14.11 – Окно *Git Bash*

- 2) Перейти в папку с нужной директорией, нажать на неё ПКМ и выбрать "Git Bash Here":

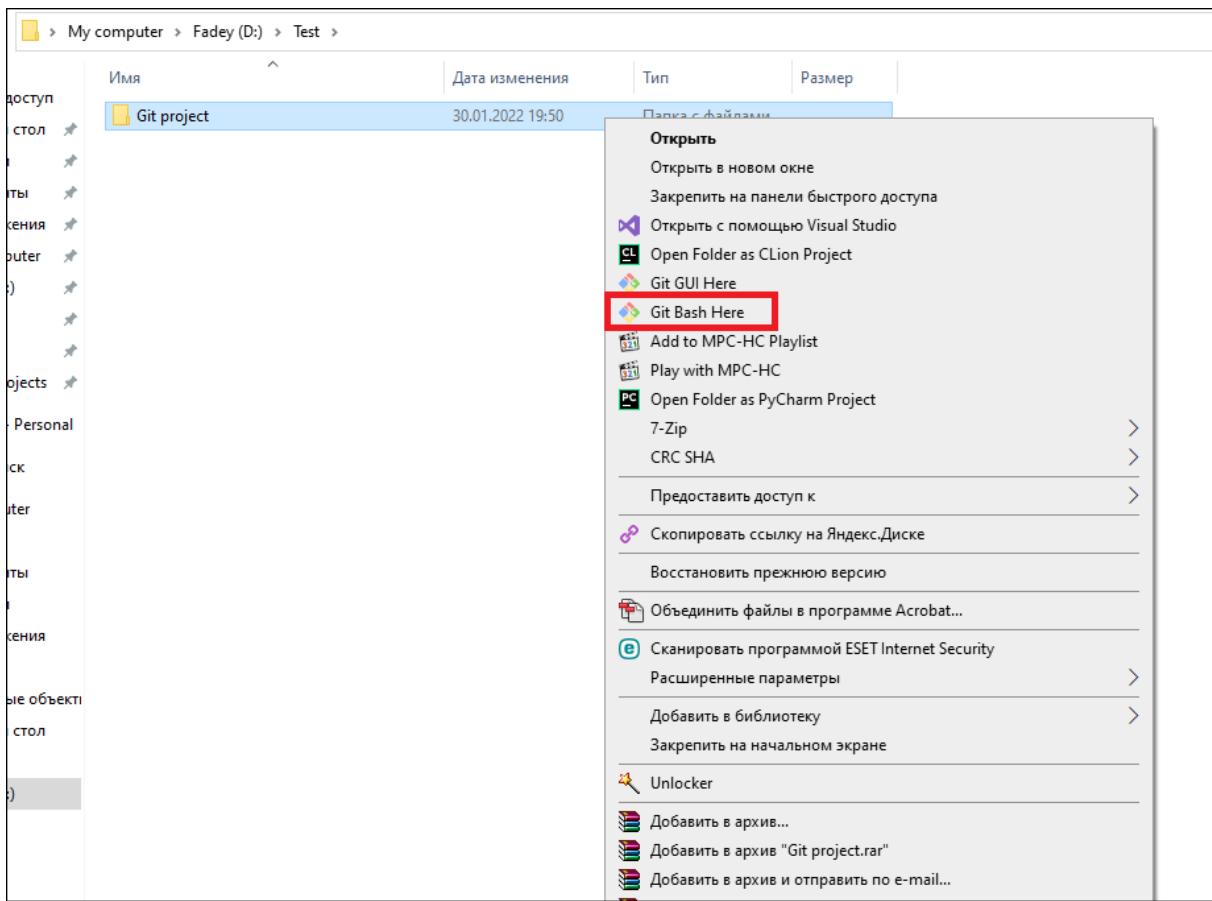
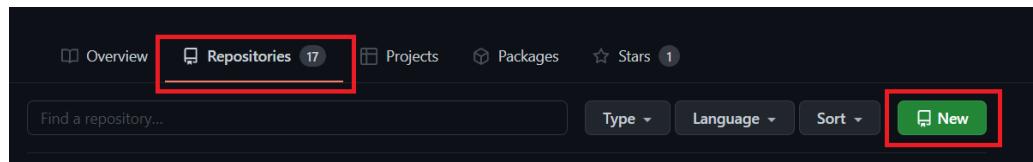


Рис. 14.12 – Окно проводника

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](#).

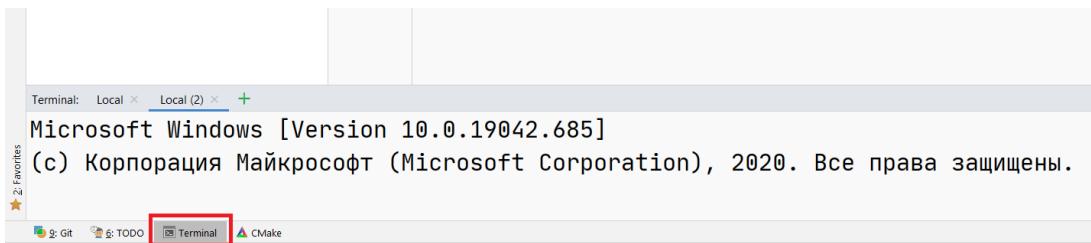
Add .gitignore
Choose which files not to track from a list of templates. [Learn more](#).

Choose a license
A license tells others what they can and can't do with your code. [Learn more](#).

Create repository

Рис. 14.13 – 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. 14.14 – 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`:

```

1 add_library(data_structures
2     // прочие файлы библиотеки
3     vector/vector.c
4 )

```

3. В заголовочном файле `vector.h` объявите структуру вектор:

```

1 typedef struct vector {
2     int *data;           // указатель на элементы вектора
3     size_t size;         // размер вектора
4     size_t capacity;    // вместимость вектора
5 } vector;

```

4. Вектор представляет собой размещенный в динамической памяти массив, который автоматически расширяется по мере необходимости. Может быть представлен так:

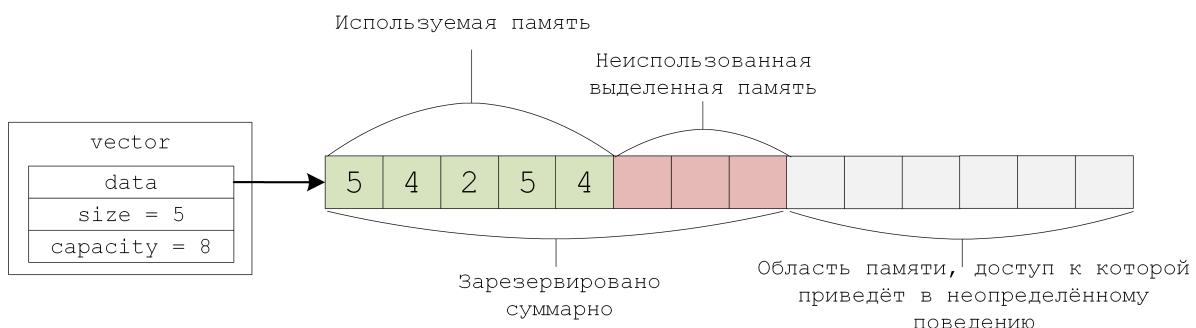
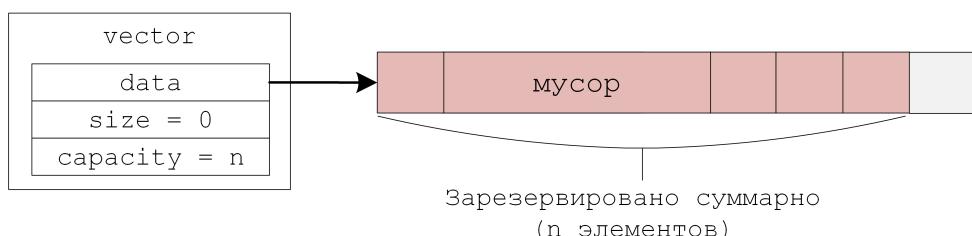


Рис. 14.15 – Вектор из 5 элементов и вместимостью в 8 элементов

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:

```

1 fprintf(stderr, "bad alloc");
2 exit(1);

```

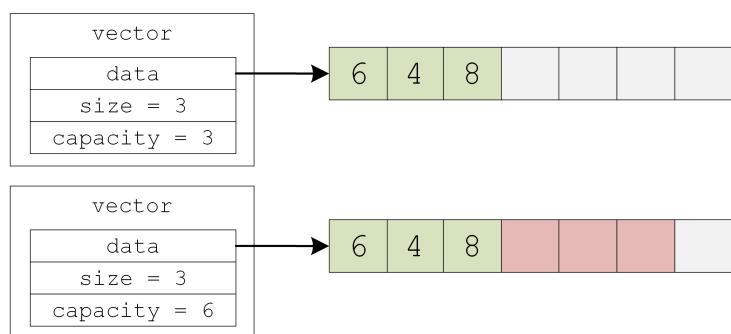
По окончанию реализации функции выполните следующий код:

```

1 #include <stdio.h>
2 #include <stdint.h>
3 #include "libs/data_structures/vector/vector.h"
4
5 int main() {
6     vector v = createVector(SIZE_MAX);
7
8     return 0;
9 }
```

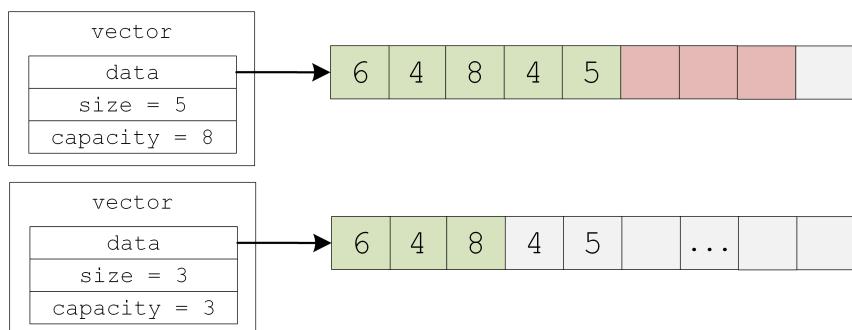
Результаты работы функции должны будут отобразиться в консоли. Сделайте соответствующие выводы.

- `void reserve(vector *v, size_t newCapacity)` – изменяет количество памяти, выделенное под хранение элементов вектора. Пример, когда новая вместимость больше:

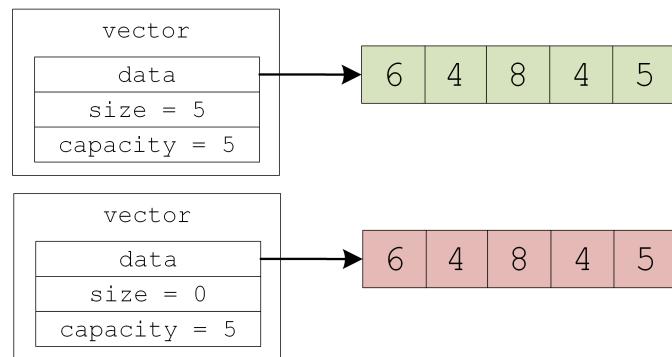


Должны соблюдаться следующие правила:

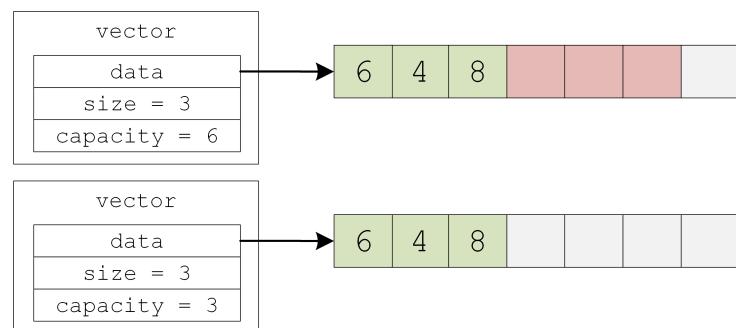
- Если в качестве `newCapacity` указано значение 0, в `data` должен быть записан `NULL`.
- Если значение `newCapacity` стало меньше `size`, тогда значение `size` должно равняться `newCapacity`:



- 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 вектору.

5. Сделайте коммит в репозитории:

```
1 git add .
2 git commit -m "memory usage of vector"
3 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 кода⁹⁰.

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:

- `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 14.16.

Протестируйте следующие случаи:

- Вектор пустой `void test_pushBack_emptyVector()`.
- Вектор заполнен `void test_pushBack_fullVector()`.

⁹⁰ Вектор пуст, если в нём нет 'полезных' элементов, хотя вместимость не обязана равняться нулю.
Вектор полон, когда используется вся доступная вектору вместимость.

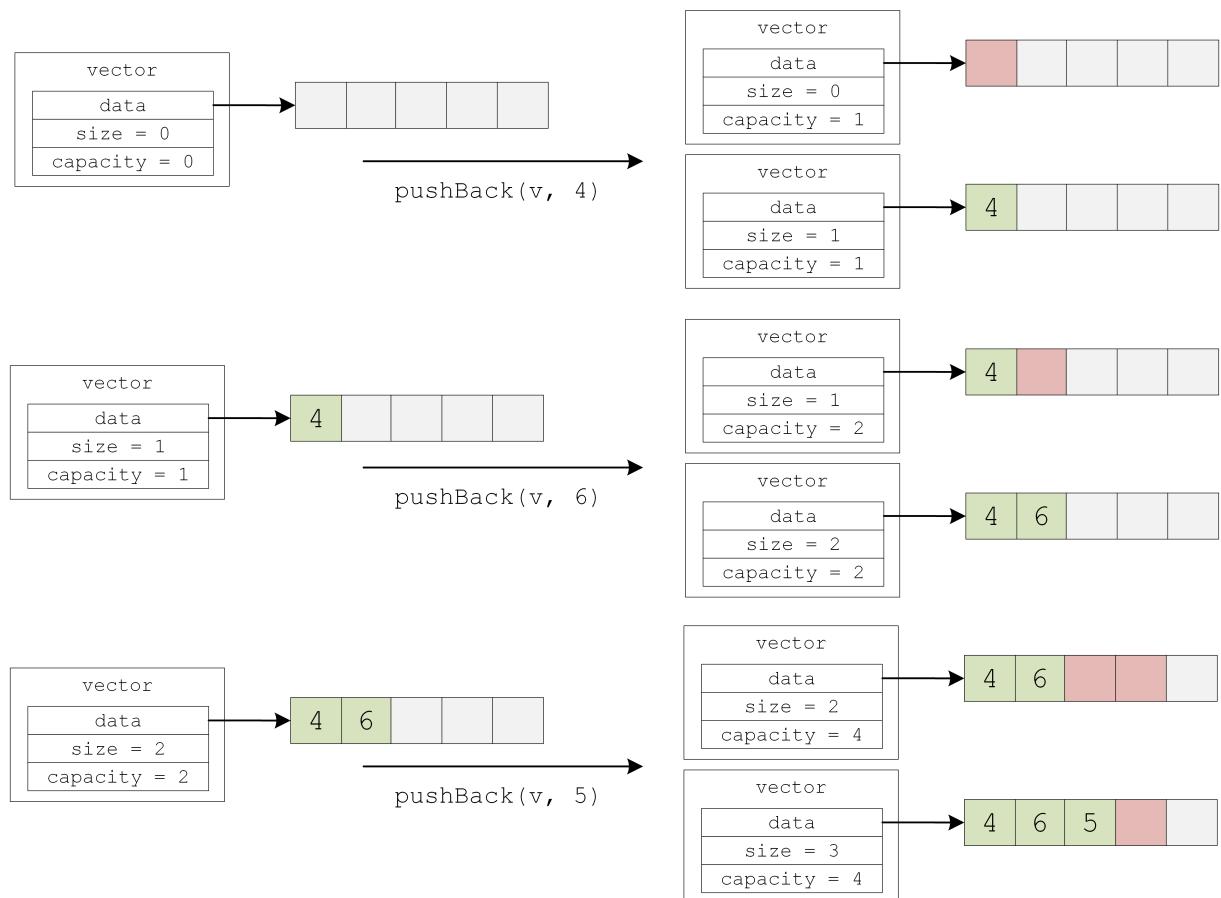


Рис. 14.16 – Процесс заполнения элементов вектора

Обратите внимание на правила именования тестов. Последовательность составляющих следующая:

- Слово `test`;
- Разделяющее нижнее подчёркивание `_`;
- Имя тестируемой функции, например, `pushBack`;
- Разделяющее нижнее подчёркивание `_`;
- Краткое описание случая `emptyVector`.

В файле `main.c` создайте функцию `test` для тестов. Все добавляемые тесты должны вызываться из неё.

```

1 void test() {
2     test_pushBack_emptyVector();
3     test_pushBack_fullVector();
4     // последующие тесты
5 }
```

- `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:

```

1 void test_popBack_notEmptyVector() {
2     vector v = createVector(0);
3     pushBack(&v, 10);
4
5     assert(v.size == 1);
```

```

6     popBack(&v);
7     assert(v.size == 0);
8     assert(v.capacity == 1);
9 }
```

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()91;`
- `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".
14. Выполните команду `git log --stat -- libs/data_structures/vector/ main.c` и приложите в отчёт результат выполнения. Пример на рисунке 14.17.

⁹¹Часто ошибки заседают на пограничных случаях. Проверяйте их.

```

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. 14.17 – 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

```

1 #ifndef INC_VECTORVOID_H
2 #define INC_VECTORVOID_H
3
4 #include <limits.h>
5
6 typedef struct vectorVoid {
7     void *data;           // указатель на нулевой элемент вектора

```

```

8     size_t size;           // размер вектора
9     size_t capacity;      // вместимость вектора
10    size_t baseTypeSize;  // размер базового типа:
11        // например, если вектор хранит int -
12        // то поле baseTypeSize = sizeof(int)
13        // если вектор хранит float -
14        // то поле baseTypeSize = sizeof(float)
15 } vectorVoid;
16
17 #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:


```

1 char *source = (char *) v->data + index * v->baseTypeSize;
2 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:

```

1 int main() {
2     size_t n;
3     scanf("%zd", &n);
4
5     vectorVoid v = createVectorV(0, sizeof(int));
6     for (int i = 0; i < n; i++) {
7         int x;
8         scanf("%d", &x);
9
10        pushBackV(&v, &x);
11    }
12
13    for (int i = 0; i < n; i++) {
14        int x;
15        getVectorValueV(&v, i, &x);
16
17        printf("%d ", x);
18    }
19
20    return 0;
21 }
```

и в варианте:

```

1 int main() {
2     size_t n;
3     scanf("%zd", &n);
4
5     vectorVoid v = createVectorV(0, sizeof(float));
6     for (int i = 0; i < n; i++) {
7         float x;
8         scanf("%f", &x);
9
10        pushBackV(&v, &x);
11    }
12
13    for (int i = 0; i < n; i++) {
14        float x;
15        getVectorValueV(&v, i, &x);
16
17        printf("%f ", x);
18    }
19
20    return 0;
21 }
```

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

```
1 git log --stat -- libs/data_structures/vectorVoid/
```

и приложите в отчёт результат выполнения.

14.13 Лабораторная работа №5d «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.

Рекомендации:

- Возможно, для ваших решений потребуется доступ к библиотеке `algorithms`, реализованной ранее. Чтобы сборка прошла успешно, для `CMakeLists.txt` в папке `data_structures` используйте:

```

1 add_library(data_structures
2         matrix/matrix.c
3 )
4
5 target_link_libraries(data_structures algorithms)

```

Задания к лабораторной работе:

1. В заголовочном файле `libs\data_structures\matrix\matrix.h` объявите структуру 'матрица' и 'позиция':

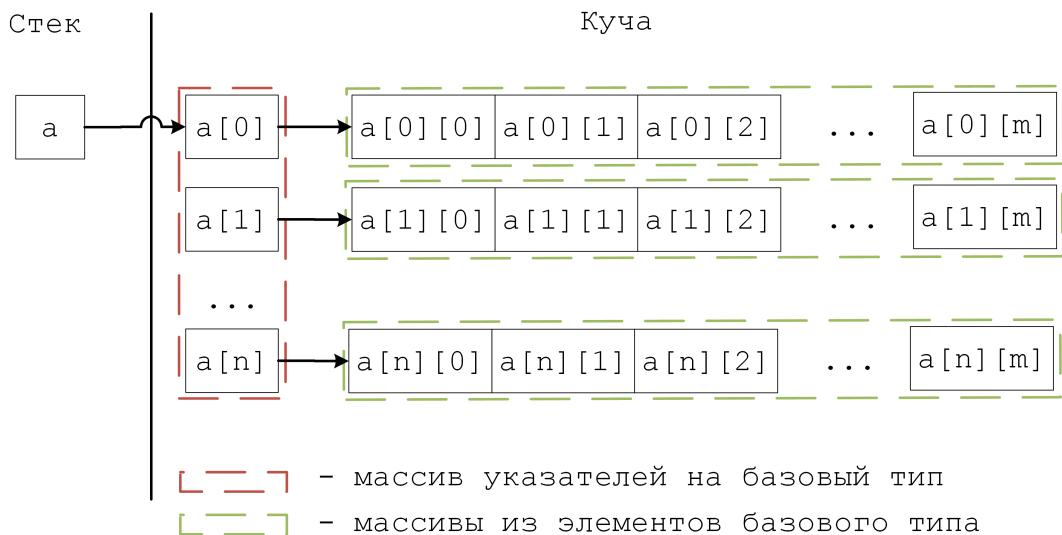
```

1 typedef struct matrix {
2     int **values;    // элементы матрицы
3     int nRows;       // количество рядов
4     int nCols;       // количество столбцов
5 } matrix;
6
7 typedef struct position {
8     int rowIndex;
9     int colIndex;
10} 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:

```

1  matrix getMemMatrix(int nRows, int nCols) {
2      int **values = (int **) malloc(sizeof(int*) * nRows);
3      for (int i = 0; i < nRows; i++)
4          values[i] = (int *) malloc(sizeof(int) * nCols);
5      return (matrix){values, nRows, nCols};
6 }
```

- (б) `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:

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."

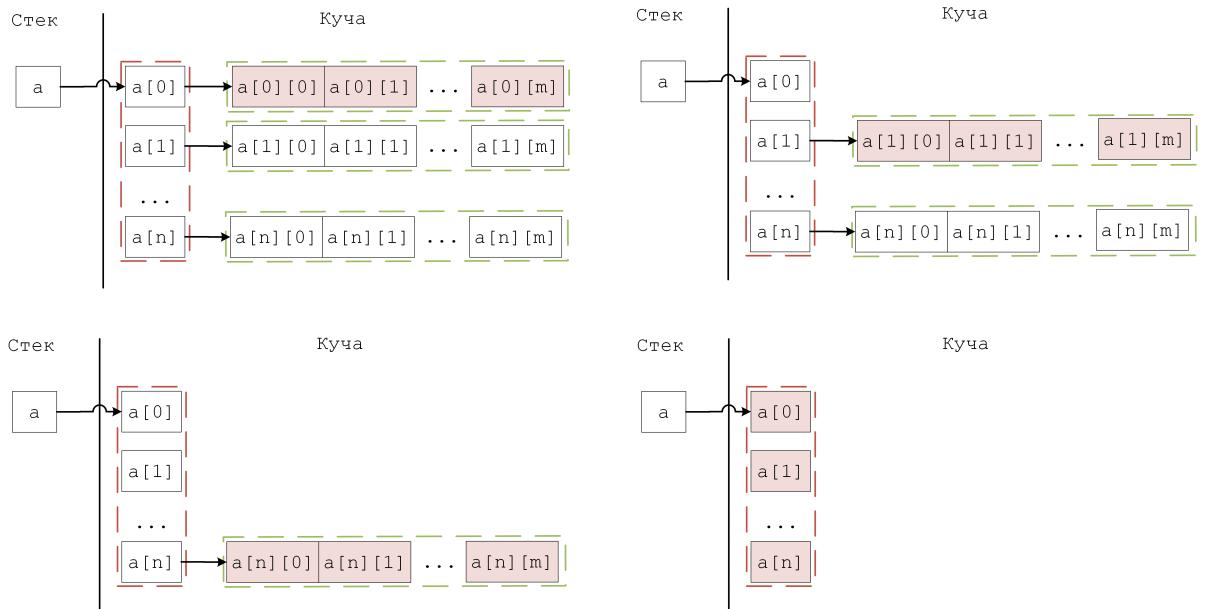


Рис. 14.18 – Освобождение динамической памяти

```

1 matrix *getMemArrayOfMatrices(int nMatrices,
2                                int nRows, int nCols) {
3     matrix *ms = (matrix*) malloc(sizeof(matrix) * nMatrices);
4     for (int i = 0; i < nMatrices; i++)
5         ms[i] = getMemMatrix(nRows, nCols);
6     return ms;
7 }
```

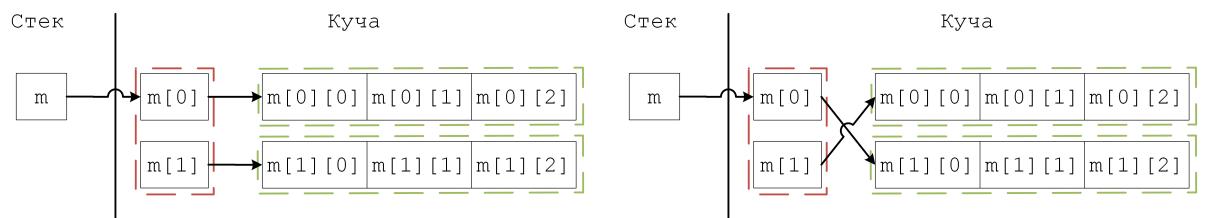
- (c) `void freeMemMatrix(matrix *m)` – освобождает память, выделенную под хранение матрицы `m`. Процесс освобождения изображен на рисунке 14.18.
- (d) `void freeMemMatrices(matrix *ms, int nMatrices)` – освобождает память, выделенную под хранение массива `ms` из `nMatrices` матриц.

3. В библиотеке `matrix` реализуйте функции для ввода и вывода матриц;

- (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 $O(1)$.

- (b) `void swapColumns(matrix m, int j1, int j2)` – обмен колонок с порядковыми номерами j_1 и j_2 в матрице m .

Обмен колонок будет заключаться в обмене $a[i][j_1]$ и $a[i][j_2]$ 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 строк⁹². 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.

```
1 getSum(int *a, int n);
```

- Данная функция передаётся как критерий в функцию сортировки матрицы:

```
1 insertionSortRowsMatrixByRowCriteria(m, getSum);
```

- 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 14.19.

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 $O(nx + n^2)$.

⁹²В 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`.

1. Вычисление сумм

						S
3	5	4	3	6		
7	5	5	5	8		
5	3	4	3	5		

→

3	5	4	3	6	21	
7	5	5	5	8	30	
5	3	4	3	5	20	

2. Параллельная сортировка и массива сумм, и соответствующих им строк матрицы

			S		S
3	5	4	3	6	21
7	5	5	5	8	30
5	3	4	3	5	20

→

3	5	4	3	6	21	
7	5	5	5	8	30	
5	3	4	3	5	20	

→

3	5	4	3	6	21	
5	3	4	3	5	20	
7	5	5	5	8	30	

→

5	3	4	3	5	20	
3	5	4	3	6	21	
7	5	5	5	8	30	

Рис. 14.19 – 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 столбцов⁹³.

Выполним оценку сложности процесса, исходя из того, что n – количество строк матрицы, m – количество столбцов. Пусть сложность вычисления 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 случае⁹⁴.
- `bool areTwoMatricesEqual(matrix *m1, matrix *m2)` – возвращает значение 'истина', если матрицы `m1` и `m2` равны, ложь – в противном случае.⁹⁵

⁹³В 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.

⁹⁴Функция должна содержать не более одной строки.

⁹⁵Функция должна содержать вызов `memcmp`.

- `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.⁹⁶

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`.

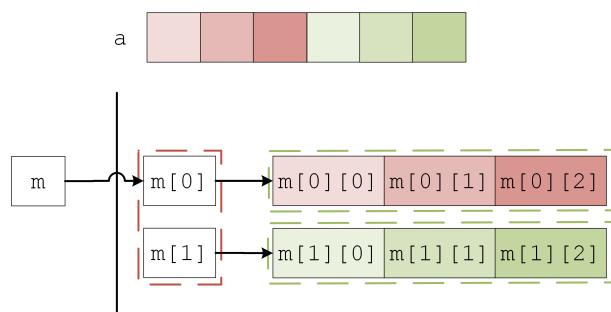
9. Дополните библиотеку функциями для тестирования:

- `matrix createMatrixFromArray(const int *a, size_t nRows, size_t nCols)` – возвращает матрицу размера `nRows` на `nCols`, построенную из элементов массива `a`:

```

1 matrix createMatrixFromArray(const int *a,
2                               int nRows, int nCols) {
3     matrix m = getMemMatrix(nRows, nCols);
4
5     int k = 0;
6     for (int i = 0; i < nRows; i++)
7         for (int j = 0; j < nCols; j++)
8             m.values[i][j] = a[k++];
9
10    return m;
11 }
```

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:

⁹⁶Функция не должна содержать лишних сравнений элементов друг с другом. Например, элементы диагонали не должны оцениваться как таковые.

```

1 void test_countZeroRows() {
2     matrix m = createMatrixFromArray(
3         (int[]) {
4             1, 1, 0,
5             0, 0, 0,
6             0, 0, 1,
7             0, 0, 0,
8             0, 1, 1,
9         },
10        5, 3
11    );
12
13    assert(countZeroRows(m, 5, 3) == 2);
14
15    freeMemMatrix(m, 5);
16}

```

- `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`:

```

1 matrix *createArrayOfMatrixFromArray(const int *values,
2                                     size_t nMatrices, size_t nRows, size_t nCols) {
3
4     matrix *ms = getMemArrayOfMatrices(nMatrices, nRows, nCols)
5         ;
6
7     int l = 0;
8     for (int k = 0; k < nMatrices; k++)
9         for (int i = 0; i < nRows; i++)
10            for (int j = 0; j < nCols; j++)
11                ms[k].values[i][j] = values[l++];
12
13     return ms;
14 }

```

При помощи реализованных функций выполнить решения следующих задач⁹⁷:

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)⁹⁸.

- `matrix mulMatrices(matrix m1, matrix m2)`
- `void getSquareOfMatrixIfSymmetric(matrix *m)`

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 ($A = B^{-1}$).⁹⁹

- (a) `bool isMutuallyInverseMatrices(matrix m1, matrix m2)`

⁹⁷Каждое из заданий снабжено рекомендуемыми для выделения функциями (сверх тех, которые были реализованы ранее). Дополнительно содержится информация, на что следует обратить внимание.

⁹⁸Симметричной называют квадратную матрицу, элементы которой симметричны относительно главной диагонали.

⁹⁹Проверьте, что при работе функции не происходит утечек памяти.

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 $n + m - 1$ элементов¹⁰⁰.

- (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:

10	7	5	6
3	11	8	9
4	1	12	2

6	8	9	2
7	12	3	4
10	11	5	1

Нижний элемент области – максимальный элемент матрицы¹⁰¹.

- (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

$$d = \sqrt{x_1^2 + x_2^2 + x_3^2 + \dots + x_m^2}$$

¹⁰⁰Попробуйте выполнить решение, при котором осуществляется проход по матрице слева направо и сверху вниз, а не по псевдодиагоналям. При этом однозначно потребуется вспомогательный массив.

¹⁰¹Проверять элементы матрицы на уникальность не требуется. Это гарантируется условием задачи.

- 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.

7	1
2	7
5	4
4	3
1	6
8	0

- (a) int cmp_long_long(const void *pa, const void *pb)
 (b) int countNUnique(long long *a, int n)
 (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	7
12	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.

1	2	3
4	5	6
7	8	1



1	2	3
1	4	7
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):

<table border="1"><tr><td>7</td><td>1</td></tr><tr><td>1</td><td>1</td></tr></table>	7	1	1	1	<table border="1"><tr><td>1</td><td>6</td></tr><tr><td>2</td><td>2</td></tr></table>	1	6	2	2	<table border="1"><tr><td>5</td><td>4</td></tr><tr><td>2</td><td>3</td></tr></table>	5	4	2	3	<table border="1"><tr><td>1</td><td>3</td></tr><tr><td>7</td><td>9</td></tr></table>	1	3	7	9
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 строк¹⁰²:

<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td></tr></table>	0	1	1	0	0	0	<table border="1"><tr><td>1</td><td>1</td></tr><tr><td>2</td><td>1</td></tr><tr><td>1</td><td>1</td></tr></table>	1	1	2	1	1	1	<table border="1"><tr><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td></tr><tr><td>4</td><td>7</td></tr></table>	0	0	0	0	4	7	<table border="1"><tr><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td></tr></table>	0	0	0	1	0	0	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>0</td><td>2</td></tr><tr><td>0</td><td>3</td></tr></table>	0	1	0	2	0	3
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																																	

- (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:

2	3	5	5	4
6	2	3	8	12
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)`

¹⁰²Функцию вывода протестировать невозможно, если матрицы заранее не собираются в какой-нибудь массив. Достаточно приложить тесты только для `countZeroRows`.

- (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)

14.14 Лабораторная работа №5е «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:

```
1 a[i]      ->      *(a + i)
```

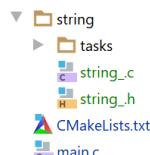
Однако в некоторых задачах допускается такая замена, но только для `i = 1`.

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).¹⁰³ Разработку организуйте через тестирование. Пишите тесты до написания функций.

Займёмся созданием библиотеки для работы со строками¹⁰⁴:

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:



¹⁰³По 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.

¹⁰⁴Вы можете дополнить библиотеку своими функциями. Вероятно некоторые задачи не потребуют никаких функций из библиотеки.

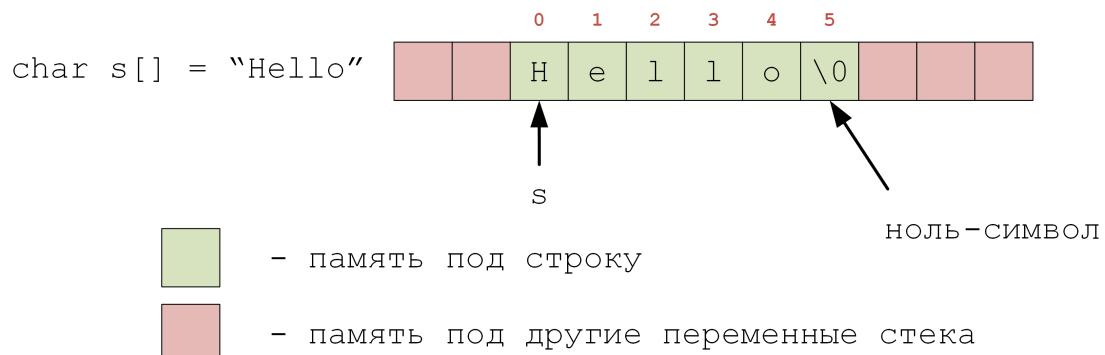
И обновите CMakeList:

```

1 cmake_minimum_required(VERSION 3.16)
2 project(project C)
3
4 set(CMAKE_C_STANDARD 11)
5
6 # определение точки входа. Будет запущен файл main.c.
7 # указывается произвольная метка, в данном случае - project
8 add_executable(project main.c)
9
10 # создаём библиотеку
11 add_library(str string/string_.h string/string_.c
12             # string/tasks/digitToStartTransform.h
13             # string/tasks/reverseWords.h
14             # string/tasks/replaceDigitsBySpaces.h
15             # ...
16             # < файл с решением задачи >
17             # ...
18             # string/tasks/hasPairOfWordsWithEqualLetterSet.h
19             # string/tasks/printWordsNonEqualLastWord.h
20         )
21
22 # описываем, что для запуска project потребуется сборка tasks
23 target_link_libraries(project str)

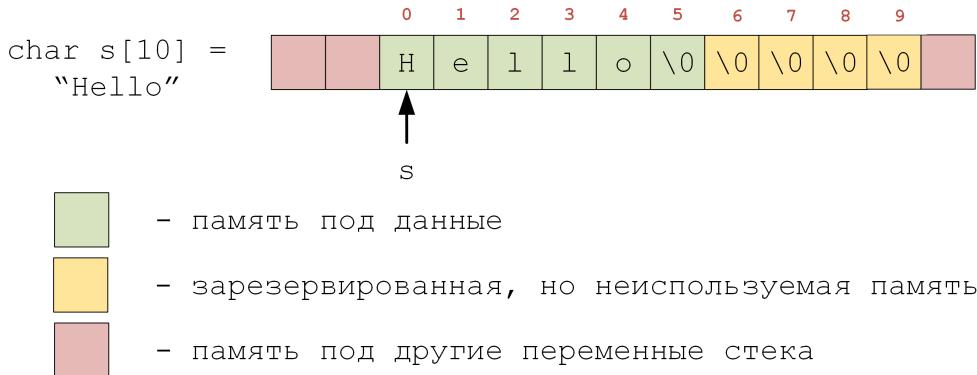
```

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:

```

1 // требует #include <string.h>
2
3 char *s1 = "Hi";
4 char s2[10] = "\tHello\t";
5
6 printf("%u\n", strlen(s1)); // 2
7 printf("%u\n", strlen(s2)); // 7

```

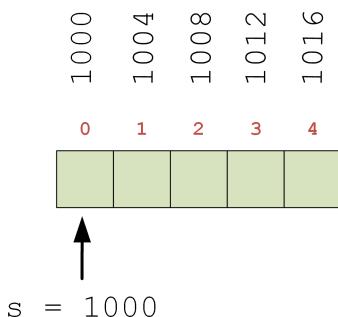
Наша задача заключается в том, чтобы определить, сколько символов имеется от начала строки до первого ноль-символа. Опишу три способа сделать это. Пойдём от худшего к лучшему. Первый вариант – используем индексы:

```

1 size_t strlen1(char *s) {
2     int i = 0;
3     while (s[i] != '\0')
4         i++;
5
6     return i;
7 }

```

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 неё 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:

$$\&s[i] = \&s[0] + \text{sizeof}(\text{basetype}) * i$$

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:

```

1 size_t strlen2(char *s) {
2     int i = 0;
3     while (*s != '\0') {
4         i++;
5         s++;
6     }
7
8     return i;
9 }
```

однако имеется избыточное количество сложений с `i`. В окончательном варианте вам стоит использовать арифметику указателей:

```

1 size_t strlen_(const char *begin) {
2     char *end = begin;
3     while (*end != '\0')
4         end++;
5     return end - begin;
6 }
```

Как это работает? Адрес i -го элемента массива может быть найден так:

$$\&s[i] = s + i$$

Выразим i из предыдущего уравнения:

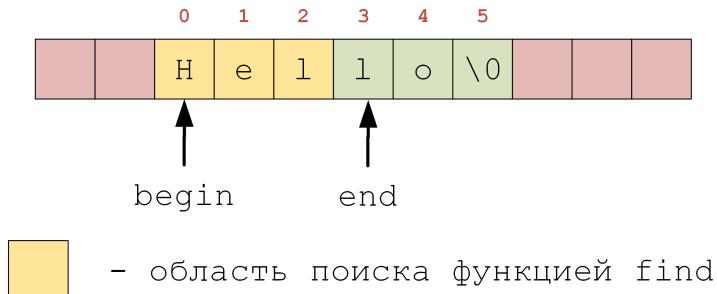
$$i = \&s[i] - s$$

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 цикла¹⁰⁵:



```

1 char* find(char *begin, char *end, int ch) {
2     while (begin != end && *begin != ch)
3         begin++;
4
5     return begin;
6 }
```

- (b) `char* findNonSpace(char *begin)` – возвращает указатель на первый символ, отличный от пробельных¹⁰⁶, расположенный на ленте памяти, начиная с `begin` и заканчивая ноль-символом. Если символ не найден, возвращается адрес первого ноль-символа¹⁰⁷.
- (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`.¹⁰⁸
- (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 14.20:

¹⁰⁵Если 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.

¹⁰⁶Не только пробел относится к пробельным символам. Используйте функцию `isspace` из `<ctype.h>`.

¹⁰⁷У 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.

¹⁰⁸В данном случае только по `rend` мы можем понять, где располагается начало строки

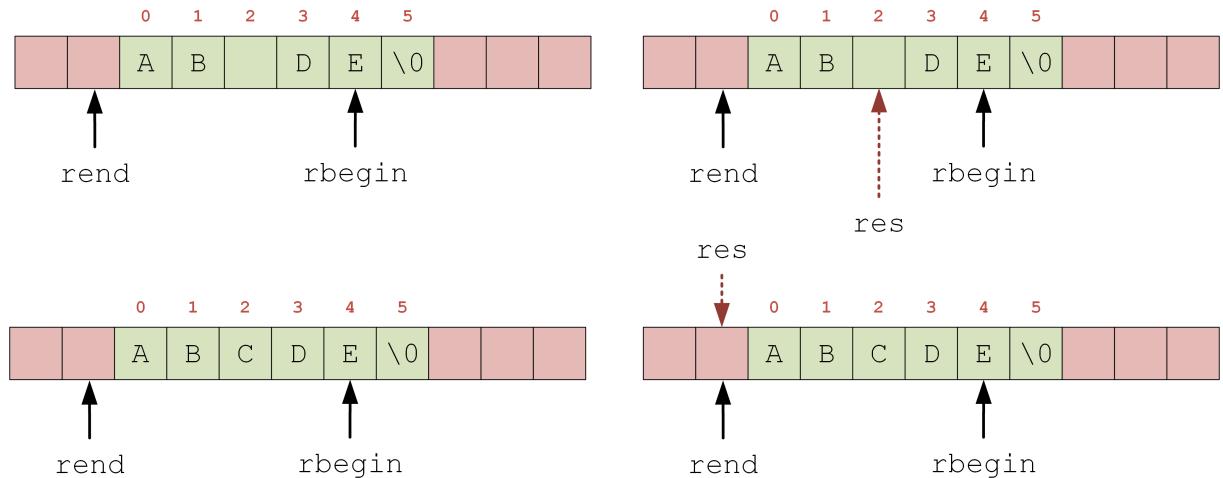


Рис. 14.20 – Работа функции *findSpaceReverse* для случаев, когда пробел имелся в исходной строке и нет

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:



Документация гласит следующее:

```
1 int strcmp(const char *lhs, const char *rhs)
```

Функция возвращает отрицательное значение, если `lhs`¹⁰⁹ располагается до `rhs` в лексикографическом порядке (как в словаре), значение 0, если `lhs` и `rhs` равны, иначе – положительное значение.

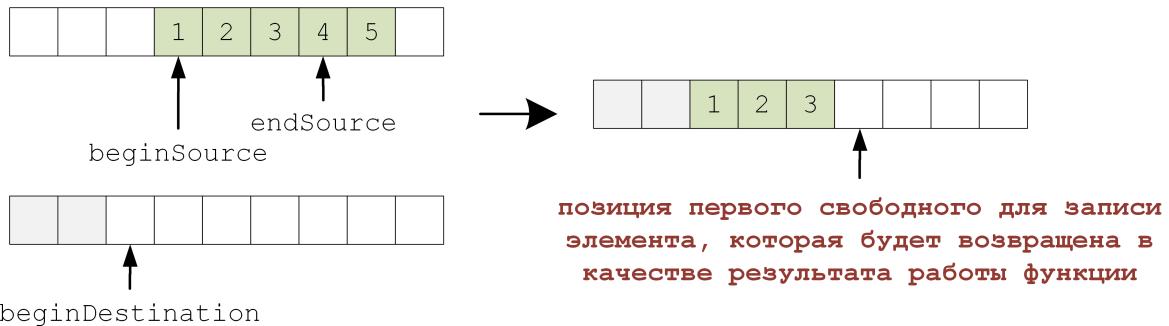
¹⁰⁹ `lhs` – *left hand size* – аргумент левой руки, `rhs` – *right hand size* – аргумент правой руки. В стандартных библиотеках часто используется такая нотация для функций из двух однородных аргументов.

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`¹¹⁰. Возвращает указатель на следующий свободный фрагмент памяти в `destination`¹¹¹:

- | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  – копируемый фрагмент
 – занятый фрагмент памяти в <code>destination</code>
 – свободный для записи фрагмент памяти в <code>destination</code> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

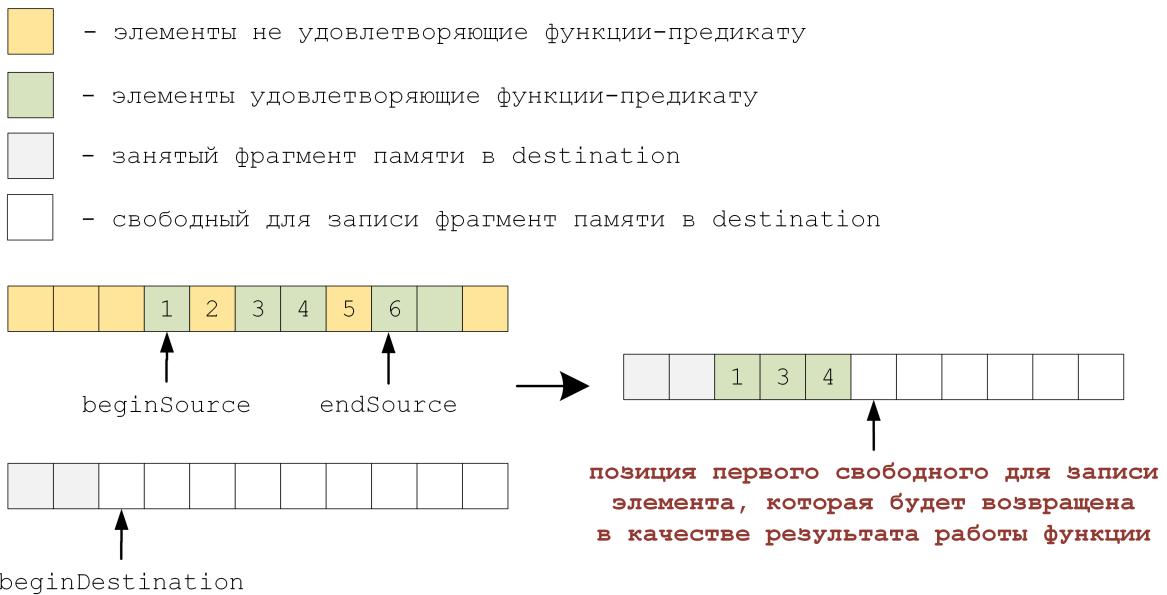


По окончанию работы функции ноль-символ не записывается.

- `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 в памяти.

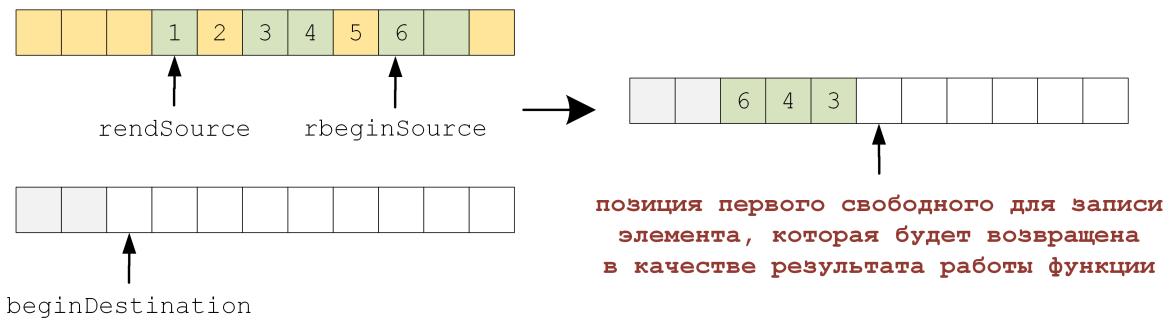
¹¹⁰При реализации используйте `memcpy` из `<memory.h>`

¹¹¹Приём 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++.



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.

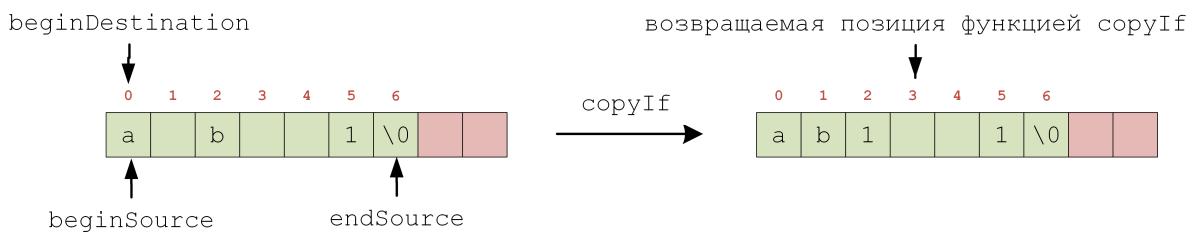


По окончанию работы функции ноль-символ не записывается.

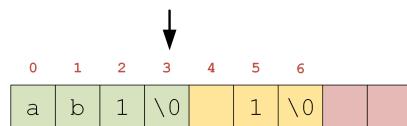
Список задач:

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:

```

1 void removeNonLetters(char *s) {
2     char *endSource = getEndOfString(s);
3     char *destination = copyIf(s, endSource, s, isgraph);
4     *destination = '\0';
5 }
```

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`:

```

1 void assertString(const char *expected, char *got,
2                   char const *fileName, char const *funcName,
3                   int line) {
4     if (strcmp_(expected, got)) {
5         fprintf(stderr, "File %s\n", fileName);
6         fprintf(stderr, "%s - failed on line %d\n", funcName, line);
7         fprintf(stderr, "Expected: \"%s\"\n", expected);
8         fprintf(stderr, "Got: \"%s\"\n", got);
9     } else
10        fprintf(stderr, "%s - OK\n", funcName);
11 }
```

Вы уже тоже чувствуете удобство этого интерфейса. Теперь можно писать:

```

1 assertString(s1, s2, "digitToStartTransform.h",
2              "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:

```
1 #define ASSERT_STRING(expected, got) assertString(expected, got, \
2                                __FILE__, __FUNCTION__, __LINE__)
```

Что нам это даёт? Предположим, что вы написали такой тест:

```
1 void test_digitToStartTransform_oneWord() {
2     char s[] = "Hi123 ";
3     digitToStartTransform(s);
4     ASSERT_STRING("321Hi", s);
5 }
```

На этапе препроцессирования будет выполнена макроподстановка:

```
1 void test_digitToStartTransform_oneWord() {
2     char s[] = "Hi123 ";
3     digitToStartTransform(s);
4     assertString("321Hi", s,
5                  "C:\Users\...\digitToStartTransform",
6                  "test_digitToStartTransform_oneWord", 4);
7 }
```

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:

- 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.

Классификация задач заняла львиную долю создания работы.

При тестировании функций на строки обязательно проверяйте случай с пустой строкой.

Потребуется функция для считывания слов. Спроектируем её с таким интерфейсом, чтобы позволить следующий приём:

```

1 char *beginSearch = beginString;
2
3 WordDescriptor word; // считываемое слово
4 while (getWord(beginSearch, &word)) {
5     // обработка слова
6     // ...
7     beginSearch = word.end;
8 }
```

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:

```

1 typedef struct WordDescriptor {
2     char *begin; // позиция начала слова
3     char *end;   // позиция первого символа, после последнего символа слова
4 } WordDescriptor;
```

Предлагаемый заголовок:

```
1 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 14.21).

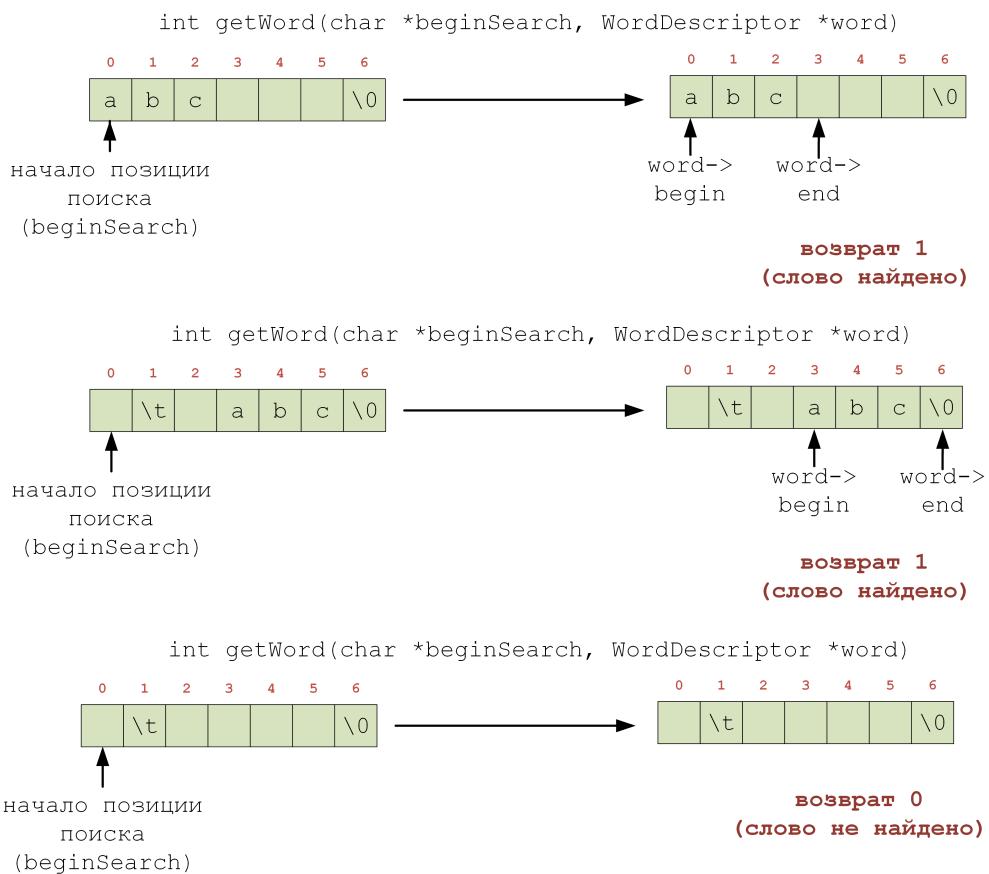
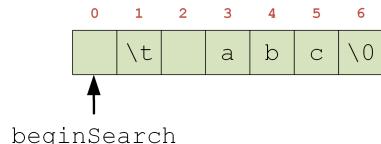


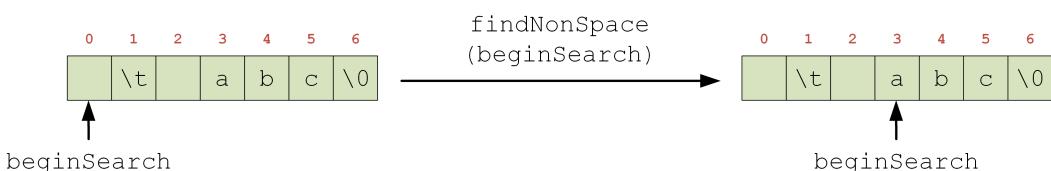
Рис. 14.21 – Результат работы функции `getWord`

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:

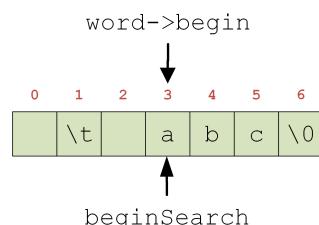
Пусть дана строка:



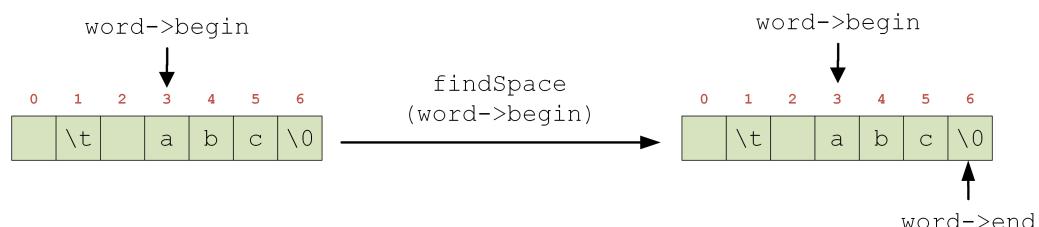
1. Выполняем поиск первого непробельного символа:



2. Сохраняем позицию начала слова (если, конечно не встретили '\0'):



3. Выполняем поиск конца слова, начиная с `beginWord`:



Если выразить кодом:

```

1 int getWord(char *beginSearch, WordDescriptor *word) {
2     word->begin = findNonSpace(beginSearch);
3     if (*word->begin == '\0')
4         return 0;
5
6     word->end = findSpace(word->begin);
7
8     return 1;
9 }
```

Если функция не найдёт слово, она вернёт значение 0. В функции, которая обрабатывает строки важно не забыть переместить позицию начала поиска:

```

1 char *beginSearch = beginString;
2 WordDescriptor word;
```

```

3 while (getWord(beginSearch, &word)) {
4     // обработка слова
5     beginSearch = word.end;
6 }
```

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 14.22.

```

1 void digitToStart(WordDescriptor word) {
2     char *endStringBuffer = copy(word.begin, word.end,
3                                   _stringBuffer);
4     char *recPosition = copyIfReverse(endStringBuffer - 1,
5                                       _stringBuffer - 1,
6                                       word.begin, isdigit);
7     copyIf(_stringBuffer, endStringBuffer, recPosition, isalpha);
8 }
```

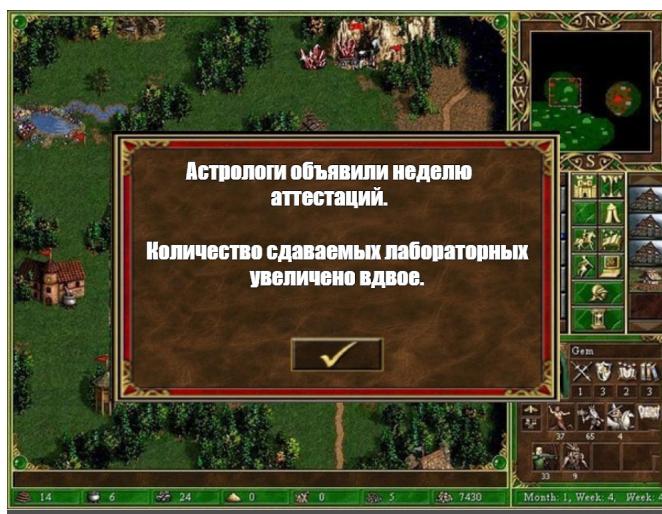
В `string_.h` объявите следующую глобальную переменную под буфер:

```
1 char _stringBuffer[MAX_STRING_SIZE + 1];
```

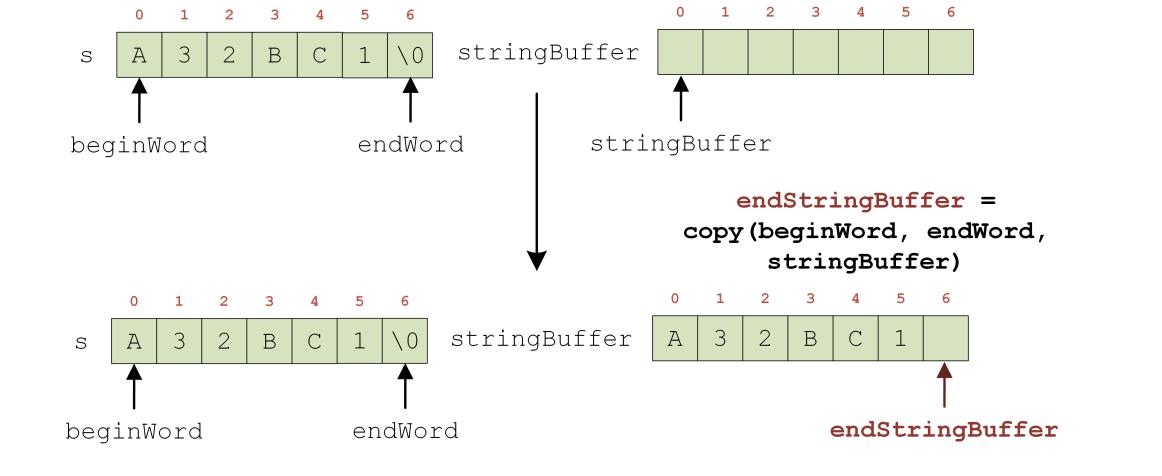
Для закрепления материала решите любую задачу из предложенных ниже:

- 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.

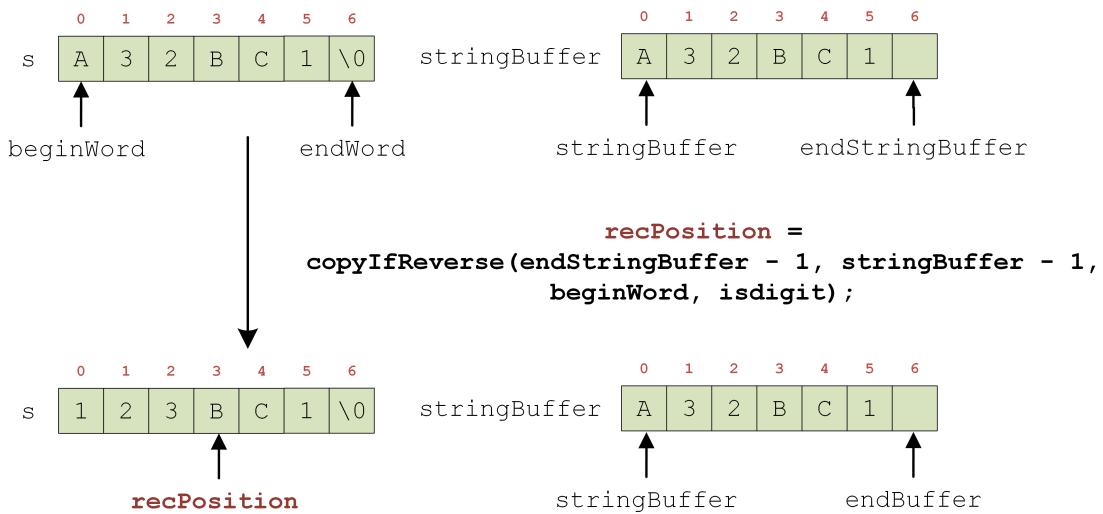
Дополнительно к этому реализуйте функцию
`bool getWordReverse(char *rbegin, char *rend, WordDescriptor *word)`
для считывания слова с конца строки. Механизм работы должен быть схож с `getWord`.



1. Копируем строку в буфер (промежуточное хранилище)



2. Копируем цифры из буфера, обрабатывая его справа налево:



3. Копируем буквы из буфера, обрабатывая его слева направо:

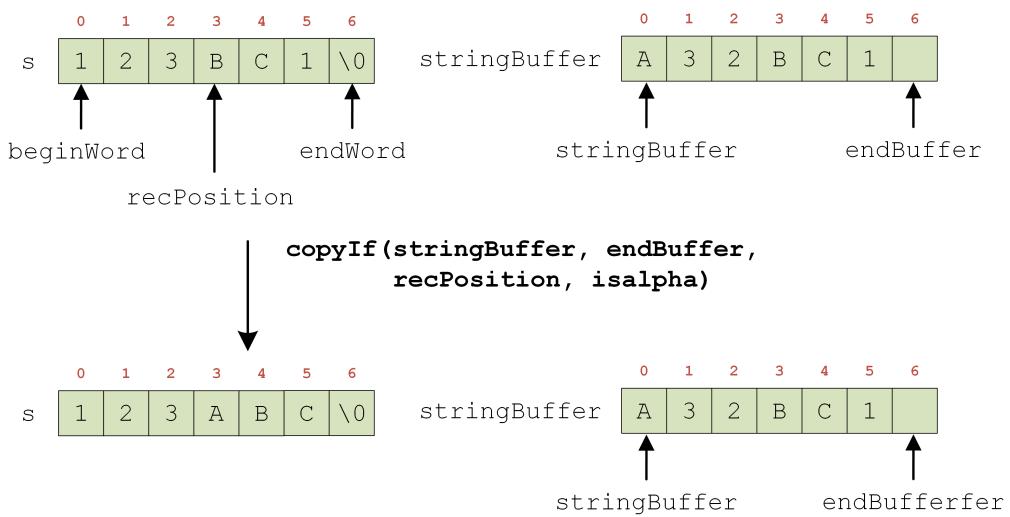
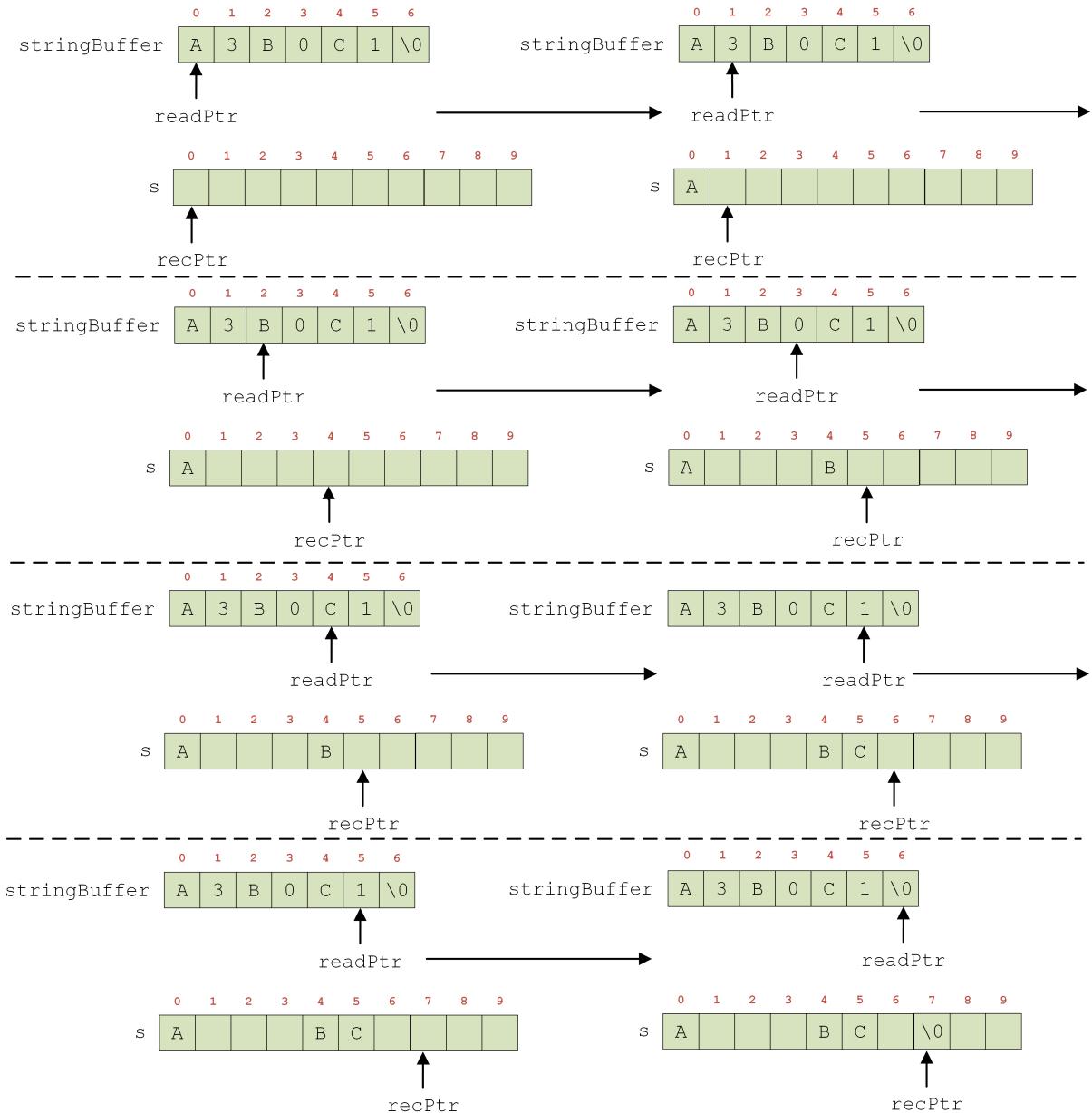


Рис. 14.22 – Работа функции `digitToStart`

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`:

```
1 #define MAX_STRING_SIZE 100
2 #define MAX_N_WORDS_IN_STRING 100
3 #define MAX_WORD_SIZE 20
```

Решение организуйте так: скопируйте исходную строку в буфер, а потом получайте ответ на строке `s`:



5. Заменить все вхождения слова w_1 на слово w_2 .

Порассуждаем о задаче. Если слово w_1 будет не более длинным, чем w_2 , тогда все преобразования можно сделать на исходной строке. Однако если это не так, возможно (если встретим w_1), придётся задействовать буфер. Для упрощения задачи будем считать, что слово w_1 всегда присутствует в строке, следователь-

но, копирование в буфер будет оправдано. Напишите функцию, которая решает данную задачу. Я помогу лишь с её началом:

```

1 void replace(char *source, char *w1, char *w2) {
2     size_t w1Size = strlen_(w1);
3     size_t w2Size = strlen_(w2);
4     WordDescriptor word1 = {w1, w1 + w1Size};
5     WordDescriptor word2 = {w2, w2 + w2Size};
6
7     char *readPtr, *recPtr;
8     if (w1Size >= w2Size) {
9         readPtr = source;
10        recPtr = source;
11    } else {
12        copy(source, getEndOfString(source), _stringBuffer);
13        readPtr = _stringBuffer;
14        recPtr = source;
15    }
16
17    // продолжение функции
18 }
```

Указание: выполните копирование строки в буфер и записывайте результат в исходную строку.

6. Определить, упорядочены ли лексикографически 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.

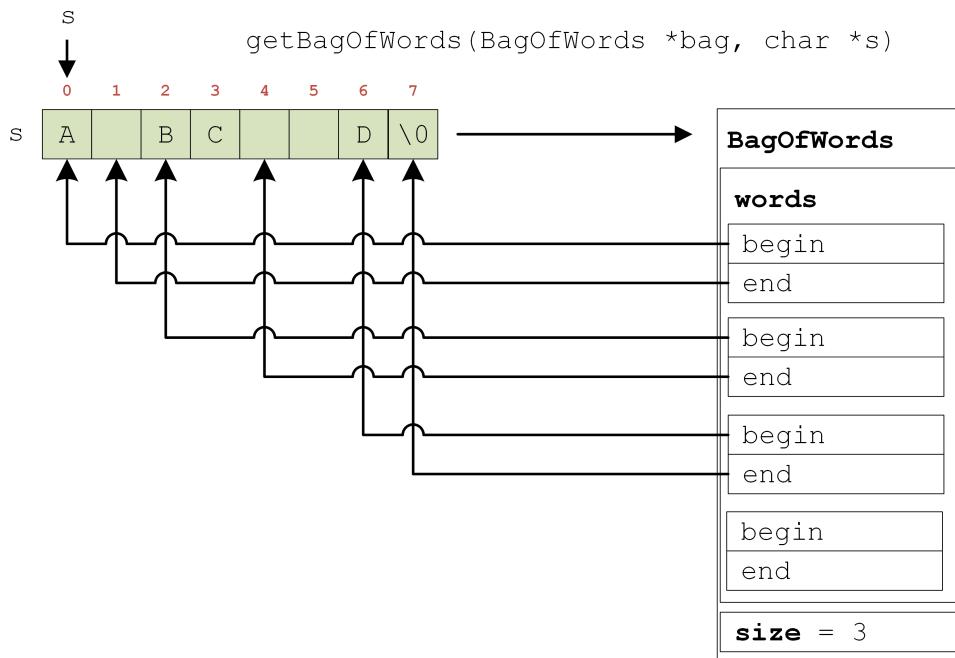
7. Вывести слова данной строки в обратном порядке по одному в строке экрана.

Указание: заведём вспомогательную структуру, которая будет хранить начало и конец каждого слова исходной строки:

```

1 typedef struct BagOfWords {
2     WordDescriptor words[MAX_N_WORDS_IN_STRING];
3     size_t size;
4 } BagOfWords;
```

Опишите функцию `void getBagOfWords(BagOfWords *bag, char *s)`, которая получает позиции начала и конца каждого слова строки:



Чтобы не создавать переменные `BagOfWords` внутри функций (и экономить на памяти), заведём в `string_.h` две глобальные переменные:

```

1 BagOfWords _bag;
2 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 строк. 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:

```

1 WordDescriptor word1, word2;
2 bool isW1Found, isW2Found;
3 char *beginSearch1 = s1, *beginSearch2 = s2;
4 while ((isW1Found = getWord(beginSearch1, &word1)),
5         (isW2Found = getWord(beginSearch2, &word2)),
6         isW1Found || isW2Found) {
7
8     //...
9 }

```

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`:

```

1 typedef enum WordBeforeFirstWordWithAReturnCode {
2     FIRST_WORD_WITH_A ,
3     NOT_FOUND_A_WORD_WITH_A ,
4     WORD_FOUND ,
5     EMPTY_STRING
6 } WordBeforeFirstWordWithAReturnCode;
7
8 // заголовок функции
9 WordBeforeFirstWordWithAReturnCode getWordBeforeFirstWordWithA(
10     char *s, char **beginWordBefore, char **endWordBefore)

```

Вместо возврата непонятных кодов можно будет написать:

```
1 return FIRST_WORD_WITH_A;
```

А тесты могут быть оформлены так:

```

1 void testAll_getWordBeforeFirstWordWithA() {
2     char *beginWord, *endWord;
3
4     char s1[] = "";
5     assert(
6         getWordBeforeFirstWordWithA(s1, &beginWord, &endWord)
7         == EMPTY_STRING
8     );
9
10    char s2[] = "ABC";
11    assert(
12        getWordBeforeFirstWordWithA(s2, &beginWord, &endWord)
13        == FIRST_WORD_WITH_A
14    );
15
16    char s3[] = "BC A";
17    assert(
18        getWordBeforeFirstWordWithA(s3, &beginWord, &endWord)
19        == WORD_FOUND
20    );
21    char got[MAX_WORD_SIZE];
22    copy(beginWord, endWord, got);
23    got[endWord - beginWord] = '\0';
24    ASSERT_STRING("BC", got);
25
26    char s4[] = "B Q WE YR OW IUWR";
27    assert(getWordBeforeFirstWordWithA(s4, &beginWord, &endWord) ==
28        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, перевода `WordDescriptor` в `char*`. Реализуйте для этого функцию `void wordDescriptorToString(WordDescriptor word, char *destination)`.

Тогда фрагмент теста будет выглядеть так:

```
1 WordDescriptor word =
2     lastWordInFirstStringInSecondString(s1[i], s2[i]);
3 wordDescriptorToString(word, string);
4 ASSERT_STRING(expected, string);
```

13. Определить, есть ли в данной строке одинаковые слова.

14. Определить, есть ли в данной строке пара слов, составленных из одинаковых 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. Задачи на поиск (любую одну на выбор):

- Даны две строки. Определить последнее из слов первой строки, которое есть во второй строке.
- 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 строки слова, совпадающие с последним словом.
- 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.