Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Институт информационных технологий

Кафедра информационных систем и технологий

А. И. Парамонов, А. Г. Савенко

АЛГОРИТМИЗАЦИЯ И КОМПЬЮТЕРНЫЕ ВЫЧИСЛЕНИЯ НА ЯЗЫКАХ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

*Рекомендовано УМО по образованию в области
информатики и радиоэлектроники в качестве
учебно-методического пособия для специальностей
6-05-0611-05 «Компьютерная инженерия»,
6-05-0612-01 «Программная инженерия»*

Минск БГУИР 2024

УДК 004.432(076)
ББК 32.973.2я73
П18

Р е ц е н з е н т ы:

кафедра информационных систем и технологий учреждения образования
«Белорусский государственный технологический университет»
(протокол № 10 от 25.05.2023);

заведующий кафедрой дискретной математики и алгоритмики
Белорусского государственного университета
доктор физико-математических наук, профессор В. М. Котов

Парамонов, А. И.
П18 Алгоритмизация и компьютерные вычисления на языках программирования высокого уровня : учеб.-метод. пособие / А. И. Парамонов, А. Г. Савенко. – Минск : БГУИР, 2024. – 212 с. : ил.
ISBN 978-985-543-738-4.

Предназначено для студентов, получающих общее высшее образование, интегрированное с образовательными программами среднего специального образования, по специальностям 6-05-0612-01 «Программная инженерия» и 6-05-0611-05 «Компьютерная инженерия». Содержит теоретический материал, методические указания с примерами по выполнению лабораторных, контрольной и курсовой работ по учебной дисциплине «Основы алгоритмизации и программирования».

Может быть использовано для самостоятельной работы студентов указанных специальностей любых форм обучения. Будет полезно студентам всех специальностей профиля образования «06. Информационно-коммуникационные технологии».

УДК 004.432(076)
ББК 32.973.2я73

ISBN 978-985-543-738-4

© Парамонов А. И., Савенко А. Г., 2024
© УО «Белорусский государственный университет информатики и радиоэлектроники», 2024

СОДЕРЖАНИЕ

Введение ... 5
Теоретическая часть
1 Основы алгоритмизации ... 9
1.1 Этапы решения задач на вычислительных машинах ... 9
1.2 Понятие алгоритмизации и алгоритма ... 11
1.3 Виды алгоритмов ... 12
1.4 Способы описания алгоритмов ... 14
1.5 Графическое описание алгоритмов по ГОСТ 19.701–90 ... 18
1.6 Оценка сложности алгоритмов ... 23
2 Основы языков программирования высокого уровня ... 39
2.1 Понятие и классификация языков программирования. Парадигмы программирования ... 39
2.2 Базовые понятия и элементы языка программирования высокого уровня ... 42
2.3 Типизация данных. Данные языков программирования высокого уровня ... 44
2.4 Модификаторы типов данных ... 50
2.5 Основные скалярные типы данных и их свойства в языках программирования Delphi и C/C++ ... 51
2.6 Операции, определенные над скалярными типами данных ... 54
2.7 Преобразование стандартных скалярных типов данных ... 56
2.8 Модификаторы доступа ... 59
2.9 Спецификаторы класса памяти ... 59
2.10 Понятия объявления, определения и инициализации ... 60
2.11 Операции и выражения ... 62
2.12 Приоритеты выполнения операций ... 70
2.13 Некоторые математические функции для работы с данными ... 73
2.14 Ввод и вывод данных ... 75
2.15 Операторы организации ветвления ... 82
2.16 Операторы организации циклов ... 87
2.17 Массивы ... 91
2.18 Методы (алгоритмы) сортировки массивов ... 97
2.19 Указатели ... 103
2.20 Динамическое выделение памяти ... 106
2.21 Строковые данные ... 109
2.22 Пользовательские подпрограммы (функции) ... 113
2.23 Рекурсия ... 122
2.24 Составные пользовательские типы данных. Записи (структуры) ... 123
2.25 Файлы ... 131
Практическая часть
Общие рекомендации по выполнению лабораторных работ ... 143
Лабораторная работа № 1. Разработка и программирование алгоритмов для разветвляющихся вычислительных процессов ... 144

Лабораторная работа № 2. Разработка вычислительных процессов
с использованием составных статических структур данных (массивы) 153
Лабораторная работа № 3. Разработка вычислительных процессов
с использованием динамической памяти (указатели) 164
Лабораторная работа № 4. Разработка программ с использованием
пользовательских функций 173
Лабораторная работа № 5. Реализация вычислительных алгоритмов
с использованием рекурсии 179
Лабораторная работа № 6. Программирование вычислительных процессов
на основе пользовательских структур данных и файлов 186
Контрольная работа. Составление алгоритмов. Блок-схемы алгоритмов 191
Курсовая работа 196
Приложение А. Таблицы кодировок 199
Приложение Б. Структура модуля программы Delphi 201
Приложение В. Перечень тем курсовых работ 203
Приложение Г. Шаблон листа задания на курсовую работу 205
Приложение Д. Шаблон титульного листа курсовой работы 207
Приложение Е. Рамки чертежей формата А3 208
Список рекомендуемой литературы 210
Список использованных источников 211

Введение

Пособие написано согласно учебным планам и программам учебных дисциплин «Основы алгоритмизации и программирования» для студентов специальностей 6-05-0612-01 «Программная инженерия» и 6-05-0611-05 «Компьютерная инженерия».

Учебно-методическое пособие направлено на формирование у студентов навыков алгоритмического мышления и программирования вычислительных процессов. С учетом того что освоение специальности «Программная инженерия» направлено на выработку компетенций и навыков по умелому использованию принципов информатики и компьютерных наук для их сочетания с инженерными подходами, разработанными для материального производства сложных корпоративных компьютерных систем, пособие для первой специальной дисциплины, изучаемой в системе подготовки специалистов, содержит обширный материал, знакомящий как с теоретическими основами алгоритмизации, так и с прикладными задачами в области разработки программ на языках программирования высокого уровня.

В данном учебно-методическом пособии первый раздел написан без привязки к конкретному языку программирования, второй раздел рассматривается на примере двух высокоуровневых языков, являющихся классическими для изучения основ алгоритмизации и программирования, Delphi и C/C++.

Наряду с теоретическим материалом учебно-методическое пособие включает типовые задания по контрольной и курсовой работам, рекомендации по выполнению лабораторных работ, тексты заданий по лабораторным работам, затрагивающие практически все основные темы учебной программы по дисциплине «Основы алгоритмизации и программирования». Приведены справочные материалы и примеры решения типовых задач.

Учебно-методическое пособие в первую очередь ориентировано на студентов заочной формы получения общего высшего образования, интегрированного с образовательными программами среднего специального образования, по специальностям 6-05-0612-01 «Программная инженерия» и 6-05-0611-05 «Компьютерная инженерия».

Пособие может быть полезно студентам всех специальностей профиля образования «06. Информационно-коммуникационные технологии», а также может быть использовано для профессиональной ориентации в ходе подготовки к освоению квалификации инженера-программиста.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1 ОСНОВЫ АЛГОРИТМИЗАЦИИ

1.1 Этапы решения задач на вычислительных машинах

В настоящее время термин «вычислительная машина» является более общим и широким, нежели «электронная вычислительная машина» (ЭВМ). ЭВМ подразумевает под собой использование электронных компонентов в качестве функциональных узлов, реализующих в конечном счете процесс вычисления. В результате развития технологий вычислительные машины могут использовать и другие способы реализации обработки информации: электронные, биологические, оптические, квантовые и их различные комбинации. Таким образом, в настоящее время вычислительная машина представляет собой комплекс технических, аппаратных и программных средств, реализующих процесс обработки информации (в том числе вычислений), автоматического управления и т. д. Синонимом понятия «вычислительная машина» является понятие «компьютер».

Для эффективности процесс решения прикладных задач на вычислительных машинах (в том числе на персональных компьютерах) разбит на ряд этапов. Классическим вариантом такого разбиения являются следующие семь этапов.

1 Постановка задачи. Данный этап включает в себя сбор и анализ информации о задаче, формулировку условия задачи, определение связей между исходными данными и искомым результатом, описание и типизацию данных.

Применительно к разработке программного обеспечения данный этап предполагает разработку технического задания, в соответствии с которым проектируется и разрабатывается программное обеспечение.

2 Моделирование. В общем случае данный этап может быть разделен на формальное построение модели задачи и построение математической модели.

Формальное построение модели задачи предполагает создание какой-либо модели (эвристической или натуральной) задачи (объекта, процесса, явления, концепции и т. д.) с характеристиками, адекватными реальной задаче, и на основании физического, информационного или иного принципа.

Построение математической модели предполагает математическую формализацию модели решения задачи. Математическая модель может как абсолютно точно, так и приближенно описывать решаемую задачу (например, численными методами). В большинстве реальных решаемых задач математическая модель строится с определенной точностью, какими-либо допущениями и ограничениями.

Для разработки программного обеспечения чаще всего сразу выполняется математическое моделирование технического задания.

3 Выбор и обоснование метода решения задачи. Поскольку чаще всего одну и ту же задачу можно решить несколькими различными методами, проводится определение оптимального метода решения по заданному критерию или ограничению. При выборе метода решения оценивается влияние различных фак-

торов и условий на конечный результат. Критериями оптимальности могут являться точность решения задачи, время решения задачи, аппаратные и архитектурные ресурсы вычислительной машины (например, требуемый объем памяти) и т. д.

4 Алгоритмизация решаемой задачи. На данном этапе определяется конкретная последовательность действий для решения поставленной задачи в соответствии с выбранным методом решения. Данная последовательность действий должна представлять собой ряд простых команд, приводящих в конечном счете к результату – решению поставленной задачи. Эта последовательность действий должна быть представлена каким-либо из способов, например, в виде блок-схемы, псевдокода или даже таблицы.

Алгоритмизация решаемой задачи и собственно алгоритм ее решения являются отправной точкой для работы любого программиста, поскольку алгоритм является универсальным «языком» и может быть реализован с применением разнообразных технологий и на различных языках программирования.

5 Составление программы. Данный этап предполагает выбор и обоснование выбора языка программирования, запись на нем разработанного алгоритма. В настоящее время существует множество языков программирования и каждый из них в различной степени подходит для решения той или иной задачи.

6 Тестирование и отладка программы. На данном этапе выполняется ручная или автоматическая проверка работоспособности программ, т. е. *тестирование*. Проверяется логика ее работы, а при выявлении каких-либо ошибок происходит их устранение, т. е. *отладка*.

Отладка программы предполагает проверку значений переменных в ходе выполнения (*трассировку*), определение правильности выбора ветвей вычислений при их наличии (должна быть протестирована каждая из ветвей вычислительного процесса). Во время тестирования, как правило, проверяется работа программы при допустимых, т. е. реально возможных, значениях исходных данных, пограничных случаях допустимых значений, а также заведомо недопустимых значениях исходных данных. Таким образом проверяется также обработка возникающих исключительных ситуаций при работе программы.

7 Непосредственно решение задачи, анализ результатов и уточнение модели при необходимости. Данный этап реализует решение задачи на вычислительной машине с использованием разработанной программы. Особенностью этапа является фактическое отсутствие эталона решения (даже несмотря на предусмотренные критерии точности или скорости получения решения). Поэтому после анализа полученных результатов может возникнуть ситуация, требующая внесения изменений в модель решаемой задачи и соответствующих корректировок в последующие этапы.

Таким образом, все приведенные этапы тесно связаны между собой. Анализ полученных результатов может привести к необходимости внесения изменений в программу, алгоритм, метод решения или даже в постановку самой решаемой задачи.

1.2 Понятие алгоритмизации и алгоритма

Как уже было отмечено, алгоритмизация – это один из важнейших этапов решения задач на вычислительных машинах.

Алгоритмизация – это сведение решаемой задачи к последовательности действий, выполняемых друг за другом так, что результаты предыдущих действий используются при выполнении последующих, приводя в конечном счете к определенному решению поставленной задачи.

Тогда алгоритмом будет являться система правил, четко описывающих последовательность действий для исполнителя, которые необходимо выполнить для решения поставленной задачи.

Алгоритм – это четкая и однозначная последовательность действий, приводящая к определенному результату (решению) за конечное число шагов.

Следует отметить, что любой алгоритм предназначен для конкретного исполнителя и учитывает его возможности. Исполнителем может быть как человек, так и машина.

Исходя из того что алгоритм – это определенная система правил, каждый алгоритм должен обладать следующими свойствами.

1 Дискретность. Алгоритм должен состоять из отдельных *шагов* – простых действий, называемых инструкцией или командой и выполняемых одно за другим в определенном порядке.

2 Детерминированность (определенность). Каждый шаг алгоритма должен быть точно и однозначно определен. Также однозначно должно быть определено, какая из команд будет выполняться следующей. Таким образом, один и тот же алгоритм при реализации разными исполнителями должен приводить к одному и тому же конечному результату при одинаковых исходных данных.

3 Результативность. Любой алгоритм должен давать определенный конечный результат (даже если этим результатом является ошибка).

4 Конечность. Любой алгоритм должен завершаться за определенное конечное число шагов.

5 Понятность. Любой алгоритм должен состоять из команд, понятных его исполнителю, т. е. входящих в систему команд исполнителя.

6 Массовость (универсальность). Алгоритм должен быть рассчитан на решение определенного класса задач, а не на одну конкретную задачу. Однако неправильно утверждать, что алгоритм обязан быть универсальным. Он может быть использован для решения однотипных задач со схожими исходными данными и не обязан подходить для решения любой другой задачи, пусть даже с аналогичными исходными данными.

При субъективной оценке алгоритмов можно еще выделить такое свойство, как **эффективность**. Это означает, что алгоритм должен быть выполнен не просто за конечное число шагов, а за разумно конечное, иначе он не будет считаться приемлемым.

1.3 Виды алгоритмов

Существуют три основных вида алгоритмов (базовые алгоритмические конструкции): линейные, ветвящиеся и циклические. Они в свою очередь составляют фундамент парадигмы структурного программирования.

Линейный алгоритм (линейный вычислительный процесс) – это алгоритм, у которого есть единственное направление его продолжения на любом шаге вычислений, а последовательность действий при этом не зависит ни от исходных данных, ни от результатов промежуточных вычислений.

Как правило, линейный алгоритм представляет собой последовательность следующих типов действий:

– ввод исходных данных;

– выполнение необходимых линейных вычислений;

– вывод конечного результата.

Ветвящийся алгоритм (ветвящийся вычислительный процесс) – это алгоритм, в котором в зависимости от какого-либо условия (исходных данных, результатов промежуточных вычислений или любого другого условия) последующие действия могут выполняться по двум или более возможным и предусмотренным вариантам направлений (ветвям), причем ветви являются параллельными и всегда выполняется только одна из них.

Условие, определяющее выбор дальнейшего направления вычислений, может быть простым (*дихотомическим* – определяющим одну из двух возможных ветвей вычислительного процесса) или более сложным (определяющим более двух возможных направлений вычислительного процесса или состоящим из нескольких простых условий). В случае если условие ветвления состоит из нескольких простых условий, то дальнейший выбор направления вычислительного процесса (ветви) осуществляется последовательной проверкой каждого из простых условий.

Таким образом, в ветвящемся алгоритме всегда будут выполняться не все описанные действия, а только те, которые не имеют альтернативы (линейные) и те альтернативные, которые определяются заданным условием.

Простейшим примером дихотомического ветвления может быть проверка равенства знаменателя дроби нулю: если знаменатель не равен нулю, выполняется вычисление, в противном случае выводится сообщение об ошибке, т. к. деление на нуль недопустимо.

Примером сложного ветвления можно назвать последовательность действий: проверку равенства знаменателя дроби нулю, затем, если равенство не выполняется, то последующие проверки знаменателя на принадлежность к положительной и отрицательной области значений. При этом для отрицательного и положительного значения знаменателя будет своя ветвь последующих вычислений.

Циклический алгоритм (циклический вычислительный процесс) – это алгоритм, в котором оператор (действие) или группа операторов (последовательность действий) может выполняться некоторое количество раз в зависимости от

условия, причем повторяющиеся операторы (действия) не дублируются линейно, а определены в алгоритме единожды, но могут при определенных условиях многократно повторяться.

Тело цикла – это команда (инструкция/оператор/действие) или группа команд (последовательность действий), многократно выполняемых при определенных условиях в циклическом алгоритме.

Итерация – это один проход (однократное выполнение) по телу цикла.

Количество итераций (повторений тела цикла) как раз определяется условием цикла.

Условие цикла – некоторое условие, проверяемое на каждой итерации и определяющее необходимость повторения цикла (выполнения еще одной итерации) или завершения цикла – перехода к последующим инструкциям (действиям), определенным после тела цикла.

Можно сказать, что циклический алгоритм в общем случае является ветвящимся, поскольку на каждой итерации проверяется *дихотомическое условие* – переход на следующую итерацию цикла или выход из цикла.

Циклические алгоритмы в свою очередь делятся на два типа.

1 *Циклы со счетчиком* (циклы с параметром) – это циклические процессы с заранее известным количеством повторений (итераций). В таких циклах задается начальное значение счетчика (переменной), конечное значение счетчика (переменной) и правило изменения счетчика (переменной). В таком случае условием цикла является проверка соответствия текущего значения счетчика (переменной) его конечному значению. При достижении конечного значения осуществляется выход из цикла. Таким образом, зная начальное и конечное значения счетчика и правило его изменения, можно определить количество итераций циклического алгоритма.

2 *Итерационные циклы* – это циклические процессы, количество итераций которых заранее не известно: они продолжаются до тех пор, пока не будет выполнено условие выхода из цикла. Как правило, итерационные циклы выполняются до тех пор, пока разность между двумя соседними, уточняемыми на каждой итерации, значениями какого-то параметра, проверяемого в условии цикла, не окажется меньше заранее заданной величины, точности, либо равной ей. Для итерационных процессов характерно, что значения, которые получены на текущем шаге итерации, используются, как правило, в качестве исходных данных для следующего шага итерации. Такое использование в большинстве случаев позволяет существенно повысить эффективность разрабатываемого алгоритма. Итерационные циклы являются основным механизмом нахождения решений численными методами высшей математики.

Также различают простой и сложный циклы (конструкции циклов).

Простой цикл – цикл, не содержащий в себе других конструкций цикла.

Сложный цикл – конструкция цикла, имеющая вложенные (внутренние) циклы (со своим условием цикла).

Следует отметить, что в реальных задачах, решаемых на вычислительных машинах, практически невозможно встретить в чистом виде только один из видов алгоритмов. Алгоритмы решения прикладных задач будут содержать как линейный вычислительный процесс, так и ветвление, и циклические процессы.

1.4 Способы описания алгоритмов

Для описания алгоритмов существует множество различных способов, обладающих своими преимуществами и недостатками. К основным способам описания алгоритмов относятся следующие.

1 ***Словесное (вербальное) описание*** – запись алгоритма на естественном (человеческом) языке. При таком способе описания алгоритм записывается в виде текста в последовательности его выполнения, например:

Шаг 1. Ввод начального значения диапазона изменения аргумента функции.

Шаг 2. Ввод конечного значения диапазона изменения аргумента функции.

Шаг 3. Ввод шага изменения текущего значения аргумента функции.

Шаг 4. Задание начального значения диапазона изменения аргумента функции в качестве текущего значения функции.

Шаг 5. Вычисление значения функции при текущем значении аргумента.

…

Шаг *N*. Вывод конечного результата.

Словесная форма описания алгоритмов используется в повседневной жизни, при описании алгоритмов приготовления напитков и блюд (рецепты), алгоритмов маршрутизации и т. д. Словесное описание алгоритма характеризуется многословностью, отсутствием наглядности вычислительного процесса и проблемой неоднозначности (многозначности). Однако следует отметить, что данным способом все равно можно описать любой сколь угодно сложный алгоритм.

2 ***Использование ограниченного языка*** **–** запись алгоритма с помощью специального набора символов. Например, описание алгоритма с помощью математических записей (формулы, системы уравнений и т. д.). Данный способ обеспечивает точное и лаконичное представление алгоритма, однако имеет недостаток – необходимость владения используемым ограниченным языком.

Пример записи алгоритма формирования кодированной записи алгебраического двоичного числа в дополнительном коде с помощью математической формулы:

$$[A]_{\text{дк}} = \begin{cases} 0.A,\ A \geq 0, \\ 1.(q^n + A),\ A < 0. \end{cases}$$

3 ***Описание алгоритма с помощью псевдокода*** – это запись алгоритма на естественном, но частично формализованном языке. Устанавливается система обозначений и правил единообразной записи алгоритма, а значение отдельных

специальных слов определено заранее и не изменяется. Такое описание отличается большей компактностью, нежели словесное описание, но предполагает понимание формализации, как правило, присущей большинству языков программирования. При этом строгих синтаксических правил описания псевдокода не существует. Примеры описания алгоритма с помощью псевдокода различного уровня формализации:

```
НАЧАЛО
  ЕСЛИ <вы счастливы> ТОГДА
      УЛЫБНИТЕСЬ
  В ПРОТИВНОМ СЛУЧАЕ
      НЕ УЛЫБАЙТЕСЬ
КОНЕЦ
```

```
BEGIN
  IF X = 0 THEN
     СООБЩЕНИЕ ОБ ОШИБКЕ
  ELSE
     Y = Z / X
END.
```

4 ***Графическое описание алгоритма*** – это описание алгоритма в виде геометрических фигур, обозначающих выполнение одного или нескольких определенных действий, и линий связи, обозначающих связь и направление вычислительного процесса.

Графическое описание – это наиболее распространенный и универсальный способ записи алгоритма. Его преимуществами являются лаконичность (компактность), наглядность и отсутствие необходимости знания специальных математических символов или конкретных языков программирования. Тем не менее у данного способа существует и недостаток – сложность графического описания некоторых непростых операций. Для устранения данного недостатка используется уточнение необходимых деталей посредством словесного или формального описания (например комментарии к геометрическим фигурам), что по сути является комбинированным способом описания алгоритма. В конечном счете графическое описание алгоритма представляет собой схему алгоритма.

Схема алгоритма (блок-схема) – это графическое представление алгоритма, в котором этапы процесса обработки информации и носители информации представлены в виде блочных символов (геометрических фигур), а последовательность процесса отражена, как правило, направлением линий переходов.

Существует множество различных способов графического описания алгоритмов.

1 Метод Дамке

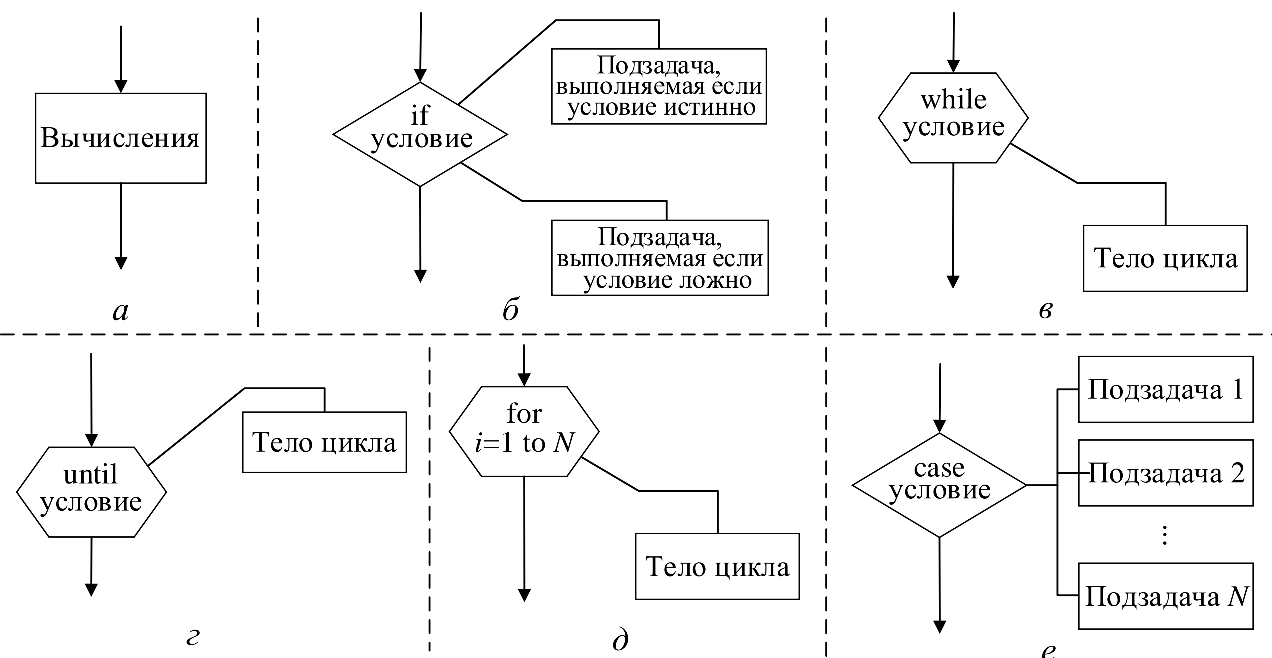
Основной принцип построения схемы алгоритма по методу Дамке – принцип декомпозиции: элементы, расположенные на схеме левее, представляют укрупненную структуру алгоритма, а расположенные справа – детализацию соответствующих блоков (например, слева может быть общий блок обозначающий цикл, а справа – непосредственно тело цикла).

На рисунке 1.1 представлены конструкции схем, построенных по методу Дамке.

Схема алгоритма, построенная по методу Дамке, предполагает три базовые конструкции:

– функциональный блок;
– конструкция if – then – else;
– конструкция цикла с предусловием.

Допускается также использование дополнительных конструкций:
– конструкция цикла с постусловием;
– конструкция цикла с параметром;
– конструкция множественного выбора (case).



а – функциональный блок; *б* – условие (if – then – else); *в* – цикл с предусловием; *г* – цикл с постусловием; *д* – цикл с параметром; *е* – множественный выбор (case)

Рисунок 1.1 – Конструкции записи алгоритма методом Дамке

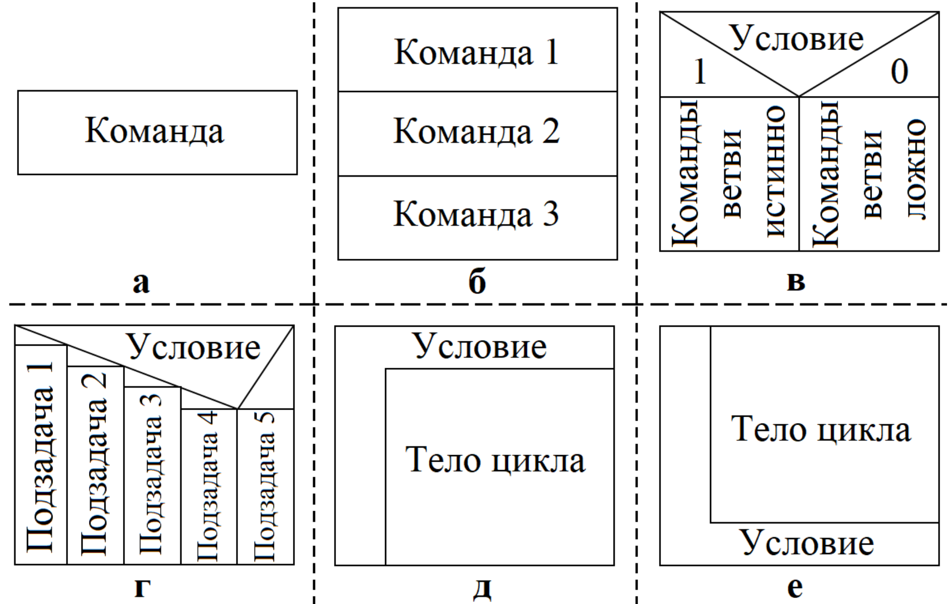
К преимуществам метода Дамке можно отнести следующие:
– не позволяет разработать схему неструктурированного алгоритма;
– удобно использовать для нисходящего проектирования;
– наглядность для больших программ;
– удобство коллективной разработки.

Недостатком является то, что схемы алгоритмов могут быть более громоздкими по сравнению со схемами, построенными по другим методам.

2 Схемы Насси – Шнейдермана (структурограммы)

Структурограммы Насси – Шнейдермана иллюстрируют структуру передачи управления с помощью вложенных блоков. Характерной особенностью такого метода является отсутствие линии, указывающей на передачу управления (направления вычислений). В данном случае команды, выполняемые друг за другом, следуют в блоках в нужном порядке.

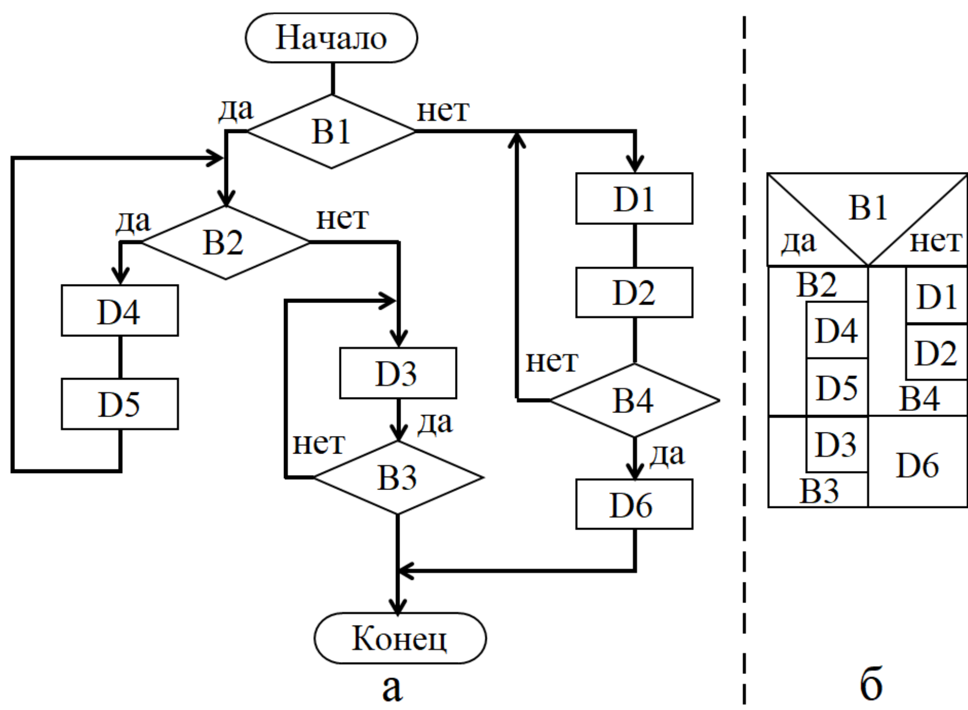
На рисунке 1.2 представлены базовые геометрические объекты для записи алгоритмов методом Насси – Шнейдермана.



а – функциональный блок; *б* – блок следования команд;
в – блок дихотомического решения; *г* – блок множественного выбора case;
д – цикл «пока»; *е* – цикл «до»

Рисунок 1.2 – Конструкции записи алгоритма методом Насси – Шнейдермана

Наглядное представление компактности записи алгоритма методом Насси – Шнейдермана (рисунок 1.3, *б*) в сравнении с блок-схемой (рисунок 1.3, *а*) продемонстрировано ниже.



а – блок-схема алгоритма; *б* – алгоритм методом Насси – Шнейдермана
Рисунок 1.3 – Сравнение записи алгоритма различными методами

3 Графическое описание по ГОСТ 19.701–90 (ISO 5807–85)

Запись алгоритма по ГОСТ 19.701–90 «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения» (разработан методом прямого применения международного стандарта ISO 5807–85) представляет собой блок-схему. Особенностью записи алгоритма блок-схемой является то, что таким образом может быть составлена схема алгоритма, не являющегося структурированным.

В стандарте определены символы, предназначенные для использования в документации по обработке данных, и приведено руководство по условным обозначениям и их применению в схемах программ, а также в схемах, отражающих данные, работу системы, взаимодействие программ, ресурсы системы.

Запись алгоритма в виде блок-схемы по ГОСТ 19.701–90 является наиболее популярным и распространенным способом во всем мире. Подробные правила записи алгоритма по ГОСТ 19.701–90 будут описаны в разделе 1.6.

Помимо рассмотренных способов описания алгоритмов существуют и другие, менее популярные, способы, такие как:

– при помощи граф-схем;

– при помощи сетей Петри;

– различные комбинации рассмотренных способов.

1.5 Графическое описание алгоритмов по ГОСТ 19.701–90

В соответствии со стандартом каждая операция изображается отдельной геометрической фигурой, а направление вычислений обозначается линиями.

Все символы разделены на четыре группы:

1) символы данных (основные и специфические);

2) символы процесса (основные и специфические);

3) символы линий (основные и специфические);

4) специальные символы.

Виды и назначение некоторых наиболее используемых символов, применяемых при составлении блок-схем программ, приведены в таблице 1.1.

Таблица 1.1 – Виды и назначение символов для составления блок-схем

Наименование символа	**Обозначение символа**	**Назначение символа**
1.1 Символы данных основные		
Данные		Применяется для общего обозначения ввода/вывода данных (носитель данных не определен)

Продолжение таблицы 1.1

Наименование символа	**Обозначение символа**	**Назначение символа**
1.2 Символы данных специфические		
Ручной ввод		Применяется для обозначения данных, вводимых вручную с устройств ввода любого типа (например, ввод с клавиатуры)
Дисплей		Применяется для обозначения вывода данных на экран в человекочитаемой форме (например, вывод результата или сообщения об ошибке на экран монитора)
2.1 Символы процесса основные		
Процесс		Применяется для обозначения обработки данных любого вида (действия, вычислений и т. д.), изменяющих значение, форму представления или расположение данных
2.2 Символы процесса специфические		
Предопределенный процесс		Применяется для обозначения процесса, предопределенного в другом месте (в подпрограмме, модуле и т. д.), т. е., по сути, обозначает вызов подпрограммы
Решение		Применяется для обозначения выбора направления дальнейшего хода выполнения алгоритма в зависимости от решения условия, указанного в блоке. Имеет один вход и не менее двух выходов (ветвей), только один из которых будет активирован после разрешения условия
Границы цикла		Символ, состоящий из двух частей и применяемый для обозначения начала и конца циклического процесса. Тело цикла помещается между первой и второй частями данного символа. Условия инициализации счетчика, правила его изменения и условия прекращения цикла записываются внутри первой или второй части символа в зависимости от расположения условия (т. е. предусловия или постусловия). При вложенности нескольких циклов обе части одного и того же цикла обозначаются одинаковыми буквами или цифрами
Подготовка		Применяется для обозначения модификации команды (группы команд) для воздействия на некоторую последующую функцию. Используется как правило в блок-схемах алгоритмов для языков программирования низкого уровня

Продолжение таблицы 1.1

Наименование символа	**Обозначение символа**	**Назначение символа**
Параллельные действия		Применяется для обозначения начала и конца параллельного процесса. Символы процессов и действий (двух и более), выполняемых не последовательно, а одновременно и независимо друг от друга, т. е. параллельно, помещаются между двумя данными символами, обозначающими начало и конец параллельного процесса
3.1 Символы линий основные		
Линия		Применяется для отображения потока данных или управления (т. е. для обозначения направления хода выполнения алгоритма)
3.2 Символы линий специфические		
Пунктирная линия		Применяется для обведения аннотированного участка блок-схемы (например, группы символов, связанных между собой определенным смыслом, функционалом и т. д.)
4 Специальные символы		
Терминатор		Применяется для обозначения начала и конца блок-схемы алгоритма (схемы программы)
Соединитель		Применяется для обозначения выхода из части схемы и входа в другую часть схемы, т. е. для обрыва линии и продолжения ее в другой части схемы. Пара символов, обозначающая выход-вход, должна быть обозначена одним и тем же уникальным буквенным или цифровым идентификатором
Комментарий		Применяется для добавления комментариев и пояснений к другим символам. Текст комментария записывается справа внутри квадратной скобки, а пунктирная линия соединяется с комментируемым символом (или объединенными в группу пунктирной линией группой символов). При этом пунктирная линия данного символа не считается входом или выходом
Пропуск	——···—— |···|	Символ (три точки) применяется только в разрыве символа линии или между двумя символами линии (как в горизонтальной, так и вертикальной ориентации) для обозначения пропуска какого-либо символа или группы символов (как правило, для изображения решения с заранее неизвестным числом повторения)

Общие правила применения символов и рекомендации по составлению блок-схем алгоритмов:

– допускается нумерация блоков (символов) алгоритма в разрыве линии или проставление над линией в верхнем левом углу символа. Нумерация блоков осуществляется в порядке сверху вниз и слева-направо (рисунок 1.4, *а*, *б*);

– следует придерживаться минимального числа линий;

– размеры символов должны позволять включать текст внутрь символа;

– внутри символа следует помещать минимальное количество текста, необходимое для понимания его функции. Текст внутри символа записывается слева направо и сверху вниз (рисунок 1.4, *а*, *б*);

– если текст не помещается внутри символа, следует использовать символ «комментарий» (рисунок 1.4, *б*);

– нельзя изменять углы и другие параметры формы символа;

– линии показывают направление потока управления. Естественным направлением потока управления, т. е. выполнения алгоритма, являются направления сверху вниз и слева направо. При естественном направлении стрелки не ставятся (рисунок 1.4, *а*, *б*);

– если направление потока отличается от стандартного (т. е. снизу вверх и справа налево), оно должно указываться стрелками (рисунок 1.4, *а*, *б*);

– каждый блок (символ) может иметь только один вход (кроме терминатора, обозначающего начало алгоритма, и соединителя, обозначающего вход из другой части схемы, т .к. они вообще не имеют входа) (рисунок 1.4, *а*, *б*);

– каждый блок (символ) может иметь только один выход (кроме терминатора, обозначающего конец алгоритма, соединителя, обозначающего выход из части схемы, т. к. они вообще не имеют выхода, и блока решения, т. к. он всегда имеет более одного выхода) (рисунок 1.4, *а*, *б*);

– линии должны входить в блок (символ) только сверху или слева;

– линии должны выходить из блока (символа) только снизу или справа;

– линии (входящие или выходящие) должны быть направлены к центру блока (символа);

– пересечение линий без их дальнейшего объединения в одну (в соответствии с алгоритмом) не допускается. В случае если невозможно развести такие линии, а также чтобы уменьшить длину линий, необходимо использовать соединитель (рисунок 1.4, *б*);

– две и более входящие линии могут объединяться в одну исходящую, при этом место их объединения должно быть смещено (на рисунке 1.4, *а* перед блоком № 7 и на рисунке 1.4, *б* перед блоком № 5);

– символы должны быть расположены на схеме алгоритма равномерно, а сама блок-схема заполнять примерно 80 % листа;

– по возможности символы должны быть одного размера (рисунок 1.4).

Правила обозначения блока (символа) решения при множественном количестве (более двух) выходов приведены на рисунке 1.5.

Пример использования символа «пропуск» приведен на рисунке 1.5 (между выходами блока «Y =»).

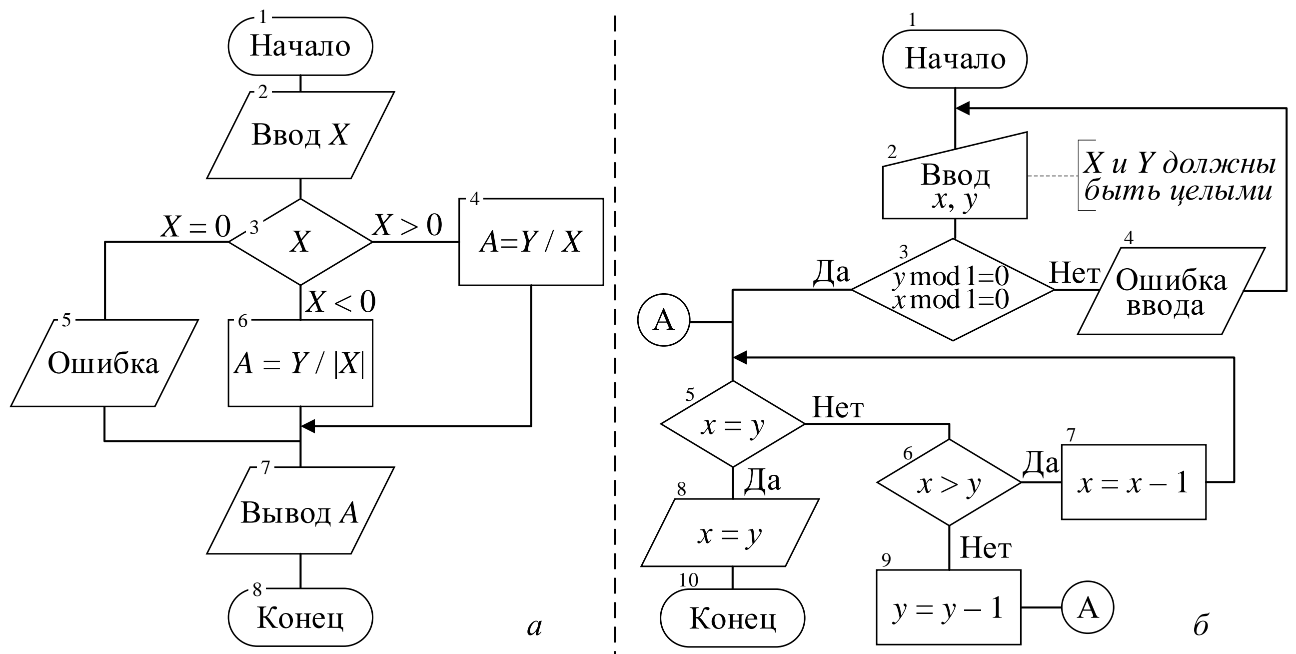


Рисунок 1.4 – Примеры блок-схем алгоритмов

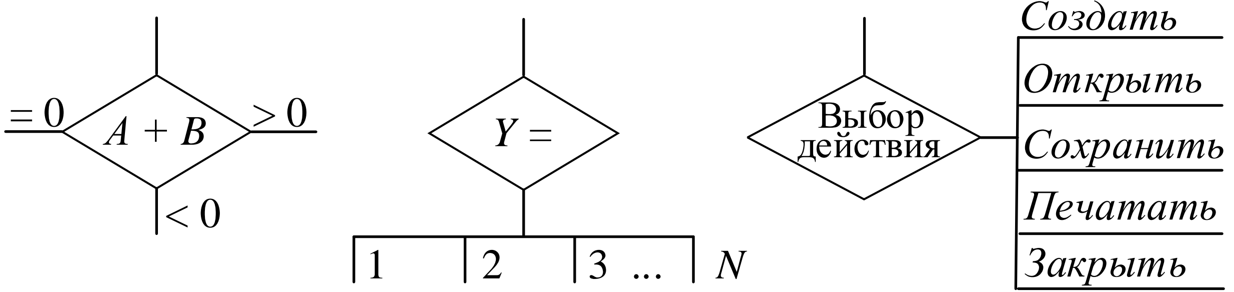


Рисунок 1.5 – Пример блоков решения при множественном количестве выходов

Наиболее частые ошибки в схемах алгоритмов приведены на рисунке 1.6.

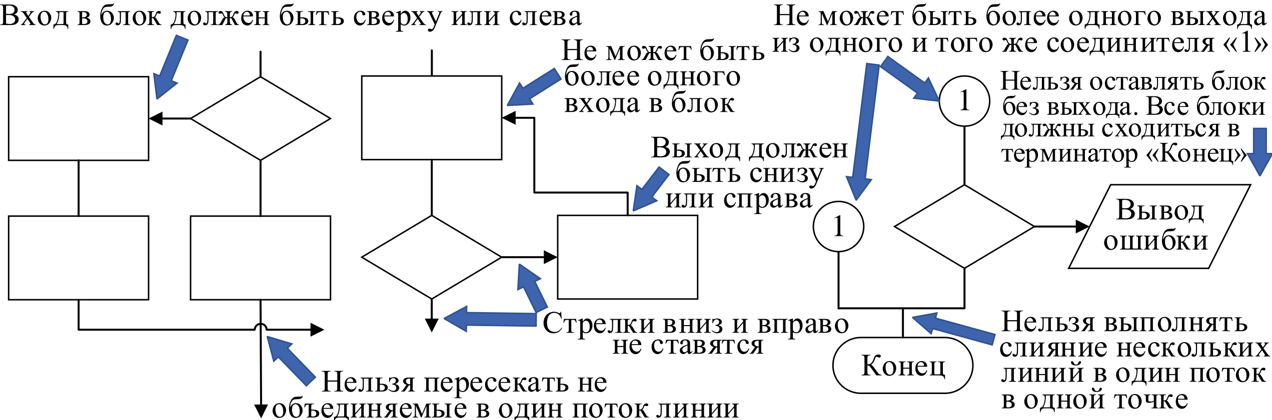


Рисунок 1.6 – Наиболее распространенные ошибки на блок-схемах алгоритмов

1.6 Оценка сложности алгоритмов

При разработке программного обеспечения существует потребность в планировании ресурсов: человеческих, необходимых для осуществления разработки, и вычислительных – для исполнения программы.

Оценкой вычислительных ресурсов занимается теория вычислительной сложности, которая оценивает алгоритмы по количеству требуемого времени выполнения и необходимому объему памяти. Теория вычислительной сложности является математической дисциплиной, основы которой будут изложены далее.

Главный вопрос, на который пытается ответить теория вычислительной сложности: сколько времени и памяти необходимо для выполнения данного алгоритма при данном размере входных данных?

Математически это можно сформулировать как задачу построения функции, которая определяет количество времени и памяти для заданного алгоритма в зависимости от размера входных данных. Размер входных данных формально определяется как количество бит в битовой строке, где они закодированы. Конечно, точное количество бит зависит от способа кодирования, поэтому для каждого конкретного случая определяется отдельный специфический для задачи способ измерять размер входных данных. К примеру, для задачи поиска элемента в массиве чисел размером входных данных целесообразно считать количество частей массива.

Память и время зачастую измеряются в абстрактных единицах, поскольку необходимо иметь оценку, не привязанную к конкретному аппаратному обеспечению. Память измеряется аналогично размеру входных данных, т. е. выбирается некоторая характерная для задачи величина. Например, одна ячейка массива может считаться одной единицей памяти. Время измеряется количеством элементарных операций, которые должен выполнить процессор для выполнения всего алгоритма. Существуют и формальные определения: память как количество ячеек машины Тьюринга и время как количество шагов, выполняемых машиной Тьюринга (хотя стоит отметить, что их использование целесообразно только лишь в теоретических задачах). Очевидно, что существует взаимно однозначное превращение формальных единиц в абстрактные неформальные, хотя, как будет показано далее, на практике определять такое преобразование не нужно. Подсчет количества шагов алгоритма основан на абстрактной модели компьютера, которая называется машиной с произвольным доступом к памяти (Random Access Machine, или RAM-машиной).

Согласно этой модели вычислительная машина работает таким образом:

– выполнение любой арифметической или булевой операции (+, *, -, =, if – else, % и т. п.) осуществляется за одну единицу времени;

– циклы и функции не являются элементарными операциями, поэтому их временная сложность является суммой сложности элементов алгоритма, которые формируют тело цикла или функции;

– в случае циклов дополнительно учитывается то, что его тело выполняется многократно, т. е. количество элементарных шагов всего цикла рассчитывается как сумма количества шагов при всех проходах тела;

– считывание значений из памяти или записи значений в память выполняется за одну единицу времени;

– память считается бесконечной и однородной, т. е. не учитываются разные типы памяти, такие как кэш процессора, диски и прочее.

Прямой подсчет количества операций для выполнения алгоритма

Для наглядности рассмотрим ход выполнения оценки алгоритма по принципу прямого подсчета количества элементарных операций на примере. Проведем анализ алгоритма сортировки вставками для массива чисел, который можно описать следующими образом:

1) имеется массив, первые n чисел которого уже упорядочены;

2) считываем $(n + 1)$-е число массива;

3) находим позицию среди первых n чисел массива, такую что все числа перед ней меньше нового числа или равны ему, а все числа после – больше;

4) сдвигаем все числа после этой позиции на одну ячейку вперед;

5) сохраняем $(n + 1)$-е число в эту позицию;

6) теперь имеем массив, первые $(n + 1)$ чисел которого уже упорядочены, и далее переходим к шагу 1.

Пример реализации этого алгоритма на языках C++ и Python приведен в листингах 1.1 и 1.2.

Листинг 1.1 – Реализация алгоритма сортировки вставками на языке C++

```
1   void insertions_sort (std::vector<int> &array) {
2       for(size_t n_sorted = 0; n_sorted < array.size(); ++n_sorted) {
3           size_t insert_position = n_sorted;
4           int x = array[n_sorted];
5           // поиск позиции для вставки
6           while ((insert_position > 0) && (array[insert_position-1] > x)){
7               insert_position--;
8           }
9           // смещение элементов массива
10          for (size_t i = n_sorted; i > insert_position; --i) {
11              array[i] = array[i-1];
12          }
13          // вставка элемента в правильную позицию
14          array[insert_position] = x;
15      }
16  }
```

Листинг 1.2 – Реализация алгоритма сортировки вставками на языке Python

```
1    def insertions_sort(array):
2        for n_sorted in range(len(array)):
3            insert_position = n_sorted
4            x = array[n_sorted]
5            # поиск позиции для вставки
6            while insert_position > 0 and array[insert_position-1] > x:
7                insert_position -= 1
8            # смещение элементов массива
9    for i in range(n_sorted, insert_position, -1):
10               array[i] = array[i-1]
11           # вставка элемента в правильную позицию
12           array[insert_position] = x
```

Вычислим, сколько элементарных шагов потребуется для сортировки небольшого массива с помощью данного алгоритма.

Внимание! Способ поиска позиции в упорядоченной части реализован не оптимально – менее эффективная, но более простая реализация используется для большей наглядности демонстрации идеи рассматриваемого способа оценки.

Определим число операций, которые потребуются для каждой конструкции на языке Python в предложенном решении задачи (листинг 1.2).

Функция `len` считывает длину списка. В Python длина хранится в отдельной ячейке памяти, поэтому можно считать, что это одна операция.

Цикл `for` с использованием `range` является эквивалентом цикла `for` С-подобных языков программирования, т. е. он создает одну переменную, присваивает ей первоначальное значение, а на каждой итерации увеличивает значение переменной на единицу и сравнивает его со значением конца цикла. Поэтому можно считать, что на каждой итерации выполняются две операции (сравнение и запись в переменную) и еще две при выходе из цикла.

Присвоение значений локальным переменным или в ячейке списка является одной операцией.

Считывание значения из ячейки массива также одна операция.

Инкременты и декременты переменных будем считать одной операцией.

Арифметические операции над индексами и сравнение значений массива также по одной операции.

Здесь снова стоит отметить, что принятое определение количества операций весьма условно.

Выполним трассировку алгоритма вручную и подсчитаем общее количество операций для входного исходного массива размерностью четыре элемента со значениями [1, 4, 2, 3]. Пошаговая трассировка алгоритма приведена в таблице 1.2.

Таблица 1.2 – Пошаговая оценка алгоритма прямым подсчетом

Действия алгоритма	Число выполненных операций
Начальное состояние массива: [1, 4, 2, 3]	**0**
Начало внешнего цикла `for`, `n_sorted = 0`	**3** (считывание длины массива и две операции цикла for)
Записали локальные переменные `x` и `insert_position`	**6** (+ 3: две записи в переменную и одно считывание из массива)
Цикл поиска завершается, ни разу не выполнив тело цикла, потому что `n_sorted = 0`	**7** (+ 1: одно сравнение)
Смещение завершается нулем итераций	**9**
Вставляем значение на позицию, состояние массива: [1, 4, 2, 3]	**10** (+ 1: одно присвоение)
Начало наружного цикла `for, n_sorted = 1`	**12**
Записали локальные переменные `x` и `insert_position`	**15**
Цикл поиска позиции завершается, ни разу не выполнив тело цикла, т. к. значение 4 больше, чем последнее значение отсортированной части массива `n_sorted = 1`	**20** (+ 5: в условии цикла выполнено сравнение, арифметическая операция, 1 считывание, еще сравнение и логическая операция)
Смещение завершается нулем итераций	**22**
Вставляем значение на позицию, состояние массива: [1, 4, 2, 3]	**23**
Начало наружного цикла `for, n_sorted = 2`	**25**
Записали локальные переменные `x` и `insert_position`	**28**
Итерация цикла поиска позиции для вставки: текущее значение 1	**34** (+ 5 операций в условии цикла, + 1 декремент переменной)
Цикл поиска позиции завершен: результат 1	**39** (+ 5 операций в условии цикла)
Смещение на позиции 2, состояние массива: [1, 4, 4, 3]	**44** (+ 5: 2 операции цикла for, арифметика над индексом, считывание и запись в массив)
Сдвиг завершен	**46**
Вставляем значение на позицию, состояние массива: [1, 2, 4, 3]	**47**
Начало наружного цикла `for`, `n_sorted = 3`	**49**
Записали локальные переменные `x` и `insert_position`	**52**
Итерация цикла поиска позиции для вставки: текущее значение 2	**58**
Цикл поиска позиции завершен: результат 2	**63**
Смещение на позиции 3, состояние массива: [1, 2, 4, 4]	**68**
Сдвиг завершен	**70**
Вставляем значение на позицию, состояние массива: [1, 2, 3, 4]	**71**

Продолжение таблицы 1.2

Действия алгоритма	Число выполненных операций
Выход из внешнего цикла `for`	**73**
Алгоритм завершен	

Таким образом, в соответствии с указанными выше договоренностям о количествах операций, которые потребуются для выполнения определенных конструкций языка Python, для выполнения предложенного алгоритма сортировки на исходном массиве [1, 4, 2, 3] нужно выполнить 73 элементарные операции.

Упражнение для самостоятельной работы по теме: попробуйте самостоятельно подсчитать аналогичным образом количество операций при сортировке массивов [1, 2, 3] и [3, 2, 1].

Правильные ответы для самоконтроля: 38 и 63 соответственно.

Анализ лучшего, худшего и среднего случаев

Как видно из результатов упражнения, предложенного в конце предыдущего подраздела, количество выполняемых операций зависит не только от размерности исходных данных, но и от содержания входных данных. При одинаковой длине массивов количество элементарных шагов для их сортировки отличается примерно в полтора раза. Чтобы понять, что это не случайность, можно подсчитать операции для сортировки массивов длиной 5: [1, 5, 2, 4, 3] и [5, 4, 3, 2, 1] (потребуется 108 и 158 операция соответственно). Поэтому с целью охарактеризовать сам алгоритм в целом, а не частные случаи использования этого алгоритма, определяют количество операций для случая с простейшими входными данными для данного алгоритма, для случая с самыми сложными данными, и рассчитывают среднее количество операций для всех возможных входных данных определенного размера.

Определим, какие входные данные являются лучшими, худшими и средними для алгоритма сортировки вставками (листинги 1.1, 1.2). Проанализируем, от чего зависит количество итераций трех циклов в этом алгоритме и количество операций внутри этих циклов. При этом обозначим буквой N количество элементов массива.

Наружный цикл `for` выполняется ровно N раз. Это означает, что N раз выполнятся эти две операции, реализующие цикл `for` плюс две операции при завершении цикла плюс одна операция для считывания длины массива. В теле внешнего цикла `for` сначала выполняются присваивания локальным переменным (три операции), затем два внутренних цикла и одно присвоение в конце – все эти операции выполняются независимо от содержания массива.

Определим сложность цикла поиска позиции для вставки. Очевидно, что она зависит от содержания массива. В итерации цикла выполняется ровно одна операция. Условие цикла может выполняться за одну операцию, если переменная `insert_position` равна нулю, или за пять операций, если эта переменная

больше нуля. Несложно увидеть, что эта переменная никогда не принимает отрицательные значения.

К минимальному количеству итераций этого цикла приводит случай, при котором элемент на позиции `n_sorted` (значение которого сохранено в переменной `x`) больше, чем последний элемент упорядоченной части массива. В этом случае цикл будет выполнен нуль раз. Условие цикла будет выполнено за одну операцию, если `n_sorted` равно нулю, и за пять операций в других случаях. Таким образом, в лучшем случае цикл поиска позиции для вставки выполняется за одну операцию во время первой итерации внешнего цикла и за пять операций во время остальных итераций, т. е. $1 + 5 \cdot (N - 1) = 5N - 4$ операций в целом.

Худшим случаем для цикла поиска позиции является ситуация, когда элемент нужно вставлять в начало упорядоченной части массива, потому что цикл должен пройти весь упорядоченный массив полностью. Можно увидеть, что это соответствует массиву, упорядоченному в обратном порядке. При этом цикл начинается при определенном значении переменной `insert_position`, равном `n_sorted`, и заканчивается при значении `insert_position` = 0, т. е. `n_sorted` раз проверка условия цикла за пять операций и один раз – за одну операцию. Таким образом, за весь алгоритм при худших входных этот цикл добавляет $(1 + (1 + 6 \times 1) + (1 + 6 \cdot 2) + \ldots + (1 + 6 \cdot (N - 1)))$ операций. Эта сумма является арифметической прогрессией, поэтому сокращенная формула суммы $N + 6N \times (N - 1) / 2$.

Подсчитаем средний случай для цикла поиска позиции. Можно заметить, что в среднем цикл должен просматривать половину упорядоченной части массива, т. е. (`n_sorted` / 2) элементов. Аналогично худшему, за все время выполнения алгоритма в среднем случае этот цикл будет добавлять $(1 + (1 + 6 / 2) + (1 + 6 \cdot 2 / 2) + \ldots + (1 + 6 \cdot (N - 1) / 2)) = (N + 6N \cdot (N - 1) / 4)$ операций.

Применив такой же подход для цикла сдвига, можно вывести формулы лучшего, худшего и среднего случаев:

– лучший: 2 за итерацию, $2 \cdot N$ в целом;

– худший: 2 · (`n_sorted` + 1) + 3 · `n_sorted` за итерацию, $5N \cdot (N - 1) / 2 + 2N$ в целом;

– средний: `n_sorted` + 2 + 3 · `n_sorted` / 2 за итерацию, $5N \cdot (N - 1) / 4 + 2N$ в целом.

За исключением двух внутренних циклов внешний цикл выполняется при $6N + 3$ операций независимо от содержания массива. Объединяя формулы для частей алгоритма вместе, получим формулы для трех случаев:

– лучший: $6N + 3 + 5N - 4 + 2N = 13N - 1$;

– худший: $6N + 3 + N + 6N \cdot (N - 1) / 2 + 5N \cdot (N - 1) / 2 + 2N = (11/2)N^2 + (7/2)N + 3$;

– средний: $6N + 3 + N + 6N \cdot (N - 1) / 4 + 5N \cdot (N - 1) / 4 + 2N = (11/4)N^2 + (25/4)N + 3$.

На практике проводить такой трудоемкий анализ нецелесообразно.

Далее рассмотрим, почему нужно и как можно упрощать анализ.

Асимптотический анализ и *O*-нотация

Анализ сложности алгоритмов выполняется для того, чтобы оценить, будет ли определенный алгоритм требовать приемлемое количество времени и памяти при выполнении на большом объеме данных, а также для сравнения алгоритмов между собой. В примере, приведенном в предыдущем подразделе, много усилий было потрачено на вывод точной формулы сложности при условии определенной договоренности относительно «стоимости» каждой элементарной операции. Поскольку нам интересно, как изменится сложность программы именно при большом количестве данных, выполним подсчет временной сложности алгоритма для разных значений N с помощью полученной ранее формулы для худшего случая. Отдельно выполним подсчет вклада, который добавляется каждым из внутренних циклов, и вклада простых операций в каждой итерации цикла `for` (таблица 1.3).

Таблица 1.3 – Влияние разных частей алгоритма на общую сложность в худшем случае при разных размерах входных данных

Выполняемые действия	**Формула оценки сложности**	$N = 10$	$N = 1000$	$N = 100\,000$
простые операции во внешнем цикле for	$6N + 3$	63, 11,7 %	6003, 0,01 %	600003, 0,001 %
цикл поиска позиции для вставки	$N + 6N \cdot (N - 1) / 2$	280, 47,6 %	2998000, 54,5 %	29999800000, 54,5 %
цикл сдвига	$5N \cdot (N - 1) / 2 + 2N$	245, 41,7 %	2499500, 45,4 %	24999950000, 45,5 %
весь алгоритм вместе	$(11/2)N^2 + (7/2)N + 3$	588, 100 %	5503503, 100 %	55000350003, 100 %

Видно, что вклад в общее количество операций, созданный простыми операциями внешнего цикла `for`, растет значительно медленнее, чем вклад внутренних двух циклов, и в конце концов становится столь малым от общей суммы, что им можно пренебречь. Очевидно, это происходит потому, что сложность внутренних циклов описывается полиномом более высокого порядка. Из этого можно заключить, что при оценке сложности на больших данных имеет значение лишь слагаемые формулы, имеющие наивысший порядок роста.

Давайте представим, что произойдет, если переписать наш алгоритм на другой язык программирования. Могут возникнуть различные ситуации, зависящие от особенностей языка. Например, может понадобиться другое количество операций для поддержки цикла `for`, или доступ к ячейке памяти в массиве будет осуществляться за разное число операций и т. д. Легко убедиться, что такие особенности приводят только к изменению коэффициентов в формуле. Коэффициенты могут играть важную роль при выборе между двумя алгоритмами (обычно алгоритм, имеющий более простую логику или меньшие коэффициенты), но они

никак не влияют на то, как будет быстро расти количество операций при увеличении входных данных, т. е. не определяют порядок роста формулы.

Сочетание двух идей (пренебрежение слагаемыми с меньшим порядком роста и пренебрежение константными коэффициентами) образует методику оценки алгоритмов, которая называется *асимптотическим анализом*. Цель этого анализа – определить порядок роста сложности алгоритма. Результат такого анализа записывается в так называемой O-нотации в виде формулы, заключенной в скобки с буквой O. Например, O-нотацией сложности наихудшего случая алгоритма сортировки, рассмотренного выше, будет $O(N^2)$. Читается как «о большое от эн квадрат».

О большое показывает верхнюю границу зависимости между входными данными и количеством операций, которые необходимо выполнить для реализации алгоритма. Классы сложности алгоритмов, наиболее часто встречаемые на практике, приведены в таблице 1.4.

Важно! При сравнении эффективности алгоритмов всегда сравнивают функции для худшего случая выполнения алгоритма. Поэтому сложность алгоритмов практически всегда дается для верхней асимптотической границы.

Таблица 1.4 – Классы асимптотической сложности алгоритмов

Класс сложности	**Асимптотика**
Постоянный (константный)	$O(1)$
Логарифмический	$O(log(N))$
Линейный	$O(N)$
Квазилинейный	$O(N \cdot log(N))$
Квадратичный	$O(N^2)$
Кубический	$O(N^3)$
Полиномиальный	$O(N^m)$
Экспоненциальный	$O(2^N)$
Факториальный	$O(N!)$

Пример. Асимптотический анализ оптимизированной версии алгоритма сортировки вставками.

Рассмотрим другой вариант реализации алгоритма сортировки вставками на языках C++ и Python (листинги 1.3, 1.4).

Листинг 1.3 – Реализация алгоритма сортировки вставками на языке C++ (улучшенная версия)

```
1   void insertions_sort(std::vector<int> &array) {
2       for(size_t n_sorted = 0; n_sorted < array.size(); ++n_sorted){
3           // поиск позиции для вставки
4           int x = array[n_sorted];
5           size_t left = 0, right = n_sorted;
6           while (left < right) {
```

```
7               size_t mid = (left + right) / 2;
8               if (array[mid] > x) {
9                   right = mid;
10              } else {
11                  left = mid + 1;
12              }
13          }
14          size_t insert_position = left;
15          // смещение элементов массива
16          for (size_t i = n_sorted; i > insert_position; --i){
17              array[i] = array[i-1];
18          }
19          // вставка элемента в правильную позицию:
20          array[insert_position] = x;
21      }
22  }
```

Листинг 1.4 – Реализация алгоритма сортировки вставками на языке Python (улучшенная версия)

```
1   def insertions_sort(array):
2       for n_sorted in range(len(array)):
3           # поиск позиции для вставки
4           x = array[n_sorted]
5           left, right = 0, n_sorted
6           while left < right:
7               mid = (left + right) // 2
8               if array[mid] > x:
9                   right = mid
10              else:
11                  left = mid + 1
12          insert_position = left
13          # смещение элементов массива
14          for i in range(n_sorted, insert_position, -1):
15              array[i] = array[i-1]
16          # вставка елемента
17   array[insert_position] = x
```

В отличие от предыдущей версии реализации (листинги 1.1, 1.2) в этом алгоритме используется другой способ поиска позиции для вставки. Идея этого способа поиска состоит в том, что необязательно просматривать каждый элемент массива, если известно, что он уже упорядочен. Вместо этого можно смотреть сразу внутрь массива и, сравнивая значение элемента посередине со значением целевого элемента, определять, находится ли правильная позиция перед серединой или после. Таким образом, размер части массива, в которой осуществляется поиск, за одну итерацию уменьшается вдвое. Завершается поиск тогда, когда переменные `left` и `right` приобретают одинаковые значения (т. е. размер части массива для поиска уменьшился до одного элемента). Из этого следует, что

количество операций при поиске зависит только от размера упорядоченной части массива, а не от его содержания. Поскольку одна итерация уменьшает размер массива вдвое, начинается поиск при размере `n_sorted`, а завершается при размере, равном 1, очевидно, что количество итераций примерно равно $\log_2$(`n_sorted`). Поскольку в условии и в итерации цикла присутствуют только простые операции, то и количество операций одного выполнения этого цикла пропорционально $\log_2$(`n_sorted`) с точностью до константных слагаемых.

Цикл сдвига по-прежнему имеет количество операций, пропорциональное `n_sorted` в среднем и худшем случаях, и константное количество операций в лучшем случае.

Внутренние циклы выполняются N раз при изменении значений переменной `n_sorted` от 0 до $N - 1$.

Таким образом, общее количество операций алгоритма в лучшем случае пропорционально величине

$$\sum_{\text{n_sorted} = 0}^{N-1} (1 + \log_2(\text{n_sorted}) + 1) \approx N + \text{N} \cdot \log_2(N) \approx N \cdot \log_2(N).$$

В среднем и худшем случаях:

$$\sum_{\text{n_sorted} = 0}^{N-1} (1 + \log_2(\text{n_sorted}) + N) \approx N + N \cdot \log_2(N) + N^2 \approx N^2.$$

Внимание! В O-нотации обычно не пишут основу логарифмов, поскольку логарифмы с разной основой и одинаковым аргументом считаются пропорциональными друг другу.

Таким образом, можно записать, что данный алгоритм имеет сложность $O(N \cdot \log(N))$ в лучшем случае и сложность $O(N^2)$ в среднем и худшем случаях.

При такой же оценке первая версия алгоритма имеет сложность $O(N)$ в лучшем случае и $O(N^2)$ в среднем и худшем случаях.

Разница в асимптотике разных версий реализации одного и того же алгоритма заключается в том, что оптимизированный поиск позиции значительно уменьшает константный множитель в среднем и худшем случаях ценой не очень больших потерь в лучшем случае. Если известна дополнительная информация про обрабатываемые данные (например, что алгоритм будет всегда работать с почти отсортированными данными), тогда предпочтение можно отдать первой версии алгоритма.

Пример. Асимптотический анализ рекурсивных алгоритмов.

Выполним анализ сложности известного алгоритма «быстрой» сортировки. Его идея состоит в том, чтобы передвинуть большие элементы в конец, а маленькие – в начало, а затем отсортировать начало и конец в отдельности.

Чтобы решать, является ли элемент большим или маленьким, выбирают так называемый «опорный» элемент: все элементы меньшие его являются «маленькими», а все бóльшие – «большими». Разделив массив один раз, больше нет надобности передвигать элементы из одной половины в другую, поэтому если затем отсортировать два образованных подмассива, то весь массив также будет упорядочен. Пример реализации этого алгоритма на языках C++ и Python приведен в листингах 1.5 и 1.6.

Листинг 1.5 – Реализация алгоритма «быстрой» сортировки на языке C++

```
 1  void sort_subarray (std::vector<int> &array, size_t first, size_t last){
 2      // выбор опорного элемента
 3      size_t mid = (first + last) / 2;
 4      int pivot = array[mid];
 5      /* разделение массива: элементы меньше опорного передвигаются в
 6      начало, а элементы больше опорного – передвигаются в конец */
 7      size_t left = first, right = last;
 8      while (left < right) {
 9      // пропускаем элементы, если они уже в своих половинах массива
10          while (array[left] < pivot) {
11              left++;
12          }
13          while (array[right] > pivot) {
14              right--;
15          }
16          //если есть элементы не в своих подмассивах, меняем их местами
17          if (left < right) {
18              int tmp = array[right];
19              array[right] = array[left];
20              array[left] = tmp;
21              left++;
22              right--;
23          }
24      }
25      // рекурсивно сортируем разделенные подмассивы
26      if ((first < right) && (last != right)) {
27          _sort_subarray(array, first, right);
28      }
29      if ((left < last) && (left != first)) {
30          _sort_subarray(array, left, last);
31      }
32  }
33
34  void quicksort(std::vector<int> &array)
35  {
36      if (array.size() > 0) {
37          _sort_subarray(array, 0, array.size()-1);
38      }
39  }
```

Листинг 1.6 – Реализация «быстрой» сортировки на языке Python

```
1    def quicksort(array):
2        def sort_subarray(first, last):
3            # выбор опорного элемента
4            mid = (first + last) // 2
5            pivot = array[mid]
6     # разделение массива: элементы меньше опорного передвигаются в начало
7     #   а элементы больше опорного – в конец
8            left, right = first, last
9            while left < right:
10    # пропускаем элементы, если они уже в своих половинах массива
11               while array[left] < pivot:
12                   left += 1
13               while array[right] > pivot:
14                   right -= 1
15    # если нашли элементы не в своих половинах массива, меняем их местами
16               if left < right:
17                   array[left], array[right] = array[right], array[left]
18                   left += 1
19                   right -= 1
20    # рекурсивно сортируем разделенные подмассивы
21           if first < right and last != right:
22               sort_subarray(first, right)
23           if left < last and left != first:
24               sort_subarray(left, last)
25       sort_subarray(0, len(array)-1)
```

Оценим сложность выполнения процедуры разделения. Она зависит от длины той части массива, которая сортируется на данном этапе рекурсии. Переменные `left` и `right` сначала указывают на начало и конец отрезка массива, соответственно, а затем «сближаются» благодаря инкрементам `left` и декрементам `right`, пока не достигнут друг друга. Эти операции выполняются примерно столько же, сколько элементов имеем на отрезке массива. При этом одновременно могут выполняться обмены элементов массива (очевидно, что обменов не может быть больше половины от длины отрезка). В лучшем случае не будет никакого обмена (т. е. массив уже упорядочен), в худшем случае потребуются менять все элементы (т. е. массив упорядочен в обратном порядке), и в среднем случае нужно будет обменивать половину элементов. Таким образом, происходит разделение сложности порядка N, и в зависимости от содержания массива изменяется константа.

Данный алгоритм имеет два рекурсивных вызова, т. е. вызовы образуют дерево с нулем, одной или двумя ветвями в каждом узле. Попробуем проанализировать, какой размер может иметь дерево и как оно может быть сбалансировано. Разделение отрезка на две части не обязательно разделяет массив на две части одинакового размера – соотношение размера частей зависит от того, какую позицию будет занимать опорный элемент после упорядочения. Интуиция дает

подсказку, что лучшим случаем будет разделение массива на две одинаковые части. Если на каждом этапе рекурсии опорный элемент разделяет массив на две одинаковые части, длины отрезка уменьшаются в два раза на каждом уровне рекурсии, из чего следует, что дерево будет сбалансированным и будет иметь глубину $\log_2(N)$. На каждом уровне дерева есть $2d$ узлов, где d – глубина рекурсии, и в каждом вызове на этом уровне осуществляется ($N / 2d$) операций, т. е. по N операций на каждом уровне дерева. Поскольку дерево имеет $\log_2(N)$ уровня, то сложность алгоритма в лучшем случае $O(N \cdot \log_2(N))$.

Оценим худший случай. Очевидно, что наиболее неблагоприятным случаем является ситуация, когда дерево вызовов максимально несбалансированно. Это соответствует случаю, когда опорным выбирается наибольший или наименьший элемент в отрезке, тогда происходит один рекурсивный вызов для отрезка длиной на единицу меньше. Таким образом, последовательность вызовов имеет длину N и на каждом вызове выполняется процедура разделения по ($N – d$) операций. Таким образом, худший случай дает сложность $O(N^2)$.

Средний случай соответствует промежуточному варианту между идеальным разделением на две одинаковые части каждого рекурсивного вызова и вырожденным случаем разделения на $N – 1$ и 0 элементов. Можно доказать, что математическое ожидание разделения случайного массива является разделением в пропорции 75 и 25 %. Это образует несбалансированное дерево с максимальной глубиной рекурсии $\log_{4/3}(N)$. Повторяя рассуждения аналогичные тем, что приведены в анализе лучшего случая, можно доказать, что на каждом уровне дерева вызовов выполняется в целом N операций, т. е. общее число операций описывается как $O(N \cdot \log_{4/3}(N))$ – такой же порядок роста, как и в лучшем случае, но с большей константой.

Таким образом, получим сложность $O(N \cdot \log(N))$ в лучшем и среднем случаях и $O(N^2)$ – в худшем случае.

Упражнение для самостоятельной работы по теме: оценить временнýю сложность алгоритма разложения на простые множители (листинги 1.7, 1.8).

Листинг 1.7 – Реализация алгоритма разложения на простые множители на языке программирования C++

```
1   std::vector< std::pair<unsigned int, unsigned int> >
2   factorize(unsigned int n) {
3       std::vector< std::pair<unsigned int, unsigned int> > result;
4       unsigned p_two = 0;
5       while (n % 2 == 0) {
6           n /= 2;
7           p_two++;
8       }
9       if (p_two > 0)
10          result.push_back(std::make_pair(2, p_two));
11      unsigned x = 3;
12      while (x * x <= n) {
```

```
13          unsigned p_x = 0;
14          while (n%x == 0) {
15              n /= x;
16              p_x++;
17          }
18          if (p_x > 0)
19              result.push_back(std::make_pair(x, p_x));
20          x += 2;
21      }
22      if (n != 1)
23          result.push_back(std::make_pair(n, 1));
24      return result;
25  }
```

Листинг 1.8 – Реализация алгоритма разложения на простые множители на языке программирования Python

```
1   def factorize(n):
2       result = []
3       p_two = 0
4       while n % 2 == 0:
5           n //= 2
6           p_two += 1
7       if p_two > 0:
8           result.append((2, p_two))
9       x = 3
10      while x*x <= n:
11          p_x = 0
12          while n % x == 0:
13              n //= x
14              p_x += 1
15          if p_x > 0:
16              result.append((x, p_x))
17          x += 2
18      if n != 1:
19          result.append((n, 1))
20      return result
```

Правильный ответ для самоконтроля: $O(\log(N))$ – в лучшем случае, $O(\mathrm{sqrt}(n))$ – в среднем и худшем случае.

Асимптотический анализ пространственной сложности

Кроме анализа времени, которое необходимо для выполнения программы, также бывает важно оценивать количество памяти, которое может быть задействовано при выполнении алгоритма. Для того чтобы описывать алгоритмы по количеству необходимой памяти, также используется O-нотация, при этом обычно речь идет об использовании именно дополнительной памяти, т. е. общее

использование памяти, за исключением памяти, хранящей входные данные алгоритма. Можно сказать, что на практике анализ использования памяти несколько проще, чем анализ времени, поскольку в большинстве алгоритмов используемая память описывается явно при описании вспомогательных структур данных.

Во многих операционных системах, после того как программа запросила у системы порцию памяти, она не обязательно возвращается сразу после завершения использования, а может остаться в распоряжении процесса для дальнейшего повторного использования. Это приводит к «утечке памяти». Недостаток памяти приводит к отказу программы. Поэтому при разработке алгоритмов важно оптимизировать худший случай, т. е. максимально уменьшить возможное использование алгоритмом дополнительной памяти.

Пример 1. Вычисление биномиальных коэффициентов.

Примеры реализации вычислений на языках программирования C++ и Python приведены в листингах 1.9 и 1.10.

Листинг 1.9 – Реализация алгоритма на языке программирования C++

```
1   unsigned int binomial_coefficient(unsigned n, unsigned k) {
2       std::vector<unsigned> coefs;
3       coefs.assign(n+1, 1);
4       if (k > (n - k)) {
5           k = n - k;
6       }
7       for (unsigned i = 0; i < k; ++i) {
8           for (unsigned j = 1; j < n-i; ++j) {
9               coefs[j] += coefs[j-1];
10          }
11      }
12      return coefs[n-k];
13  }
```

Листинг 1.10 – Реализация алгоритма на языке программирования Python

```
1  def binomial_coefficient(n, k):
2      coefs = [1] * (n+1)
3      if k > n - k:
4          k = n - k
5      for i in range(k):
6          for j in range(1, n-i):
7              coefs[j] += coefs[j-1]
8      return coefs[n-k]
```

Массив `coefs` хранит одну строку треугольника Паскаля (если изобразить треугольник в виде таблицы, в углу которой расположена вершина треугольника). В строке 2 приведенной реализации создается массив длиной $n + 1$, и при выполнении алгоритма используется только без добавления дополнительных

элементов, т. е. без расширения массива. Поэтому пространственную сложность данного алгоритма можно оценить как $O(N)$.

Пример 2. Алгоритм «быстрой сортировки».

Для анализа рассмотрим предложенный ранее рекурсивный алгоритм «быстрой» сортировки (листинги 1.5 и 1.6).

Рекурсивные методы имеют особенный аспект использования памяти. Дело в том, что каждый вызов функции (в любом языке программирования) использует дополнительную память, для того чтобы сохранить значение локальных переменных и место, куда программа должна вернуться после завершения вызываемой функции. Эта информация записывается в специально предназначенный для этого стек. Таким образом, количество памяти, использованной этим стеком, пропорционально текущей глубине рекурсии. Ранее мы выяснили, что в лучшем и среднем случае глубина дерева вызовов пропорциональна логарифму длины сортируемого массива, т. е. максимальное использование стека оценивается как $O(\log(N))$. В худшем случае, когда дерево вызовов вырождается в последовательность длины N, использование стека также становится $O(N)$. На практике это очень нежелательное поведение, т. к. обычно максимальная длина стека ограничена, а переполнение стека приводит к отказу программы. Поэтому приведенная нами реализация алгоритма «быстрой» сортировки не подходит для прикладного использования. Для эффективного промышленного программирования разработаны уже усовершенствованные версии быстрой сортировки, которые меньше используют стек и не имеют обозначенной проблемы.

2 ОСНОВЫ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

2.1 Понятие и классификация языков программирования. Парадигмы программирования

Язык программирования – это формализованный язык, предназначенный для записи текста программ для вычислительных машин и определяющий набор лексических, синтаксических и семантических правил, формирующих внешний вид этих программ и возможные действия вычислительной машины под их управлением.

По сути, язык программирования является таким же человеческим языком, как и многие языки народов мира, только со своими правилами применения и отсутствием необходимости вербального использования.

Существует множество классификаций языков программирования по различным критериям. Наиболее обобщающая и широкая классификация представлена на рисунке 2.1.

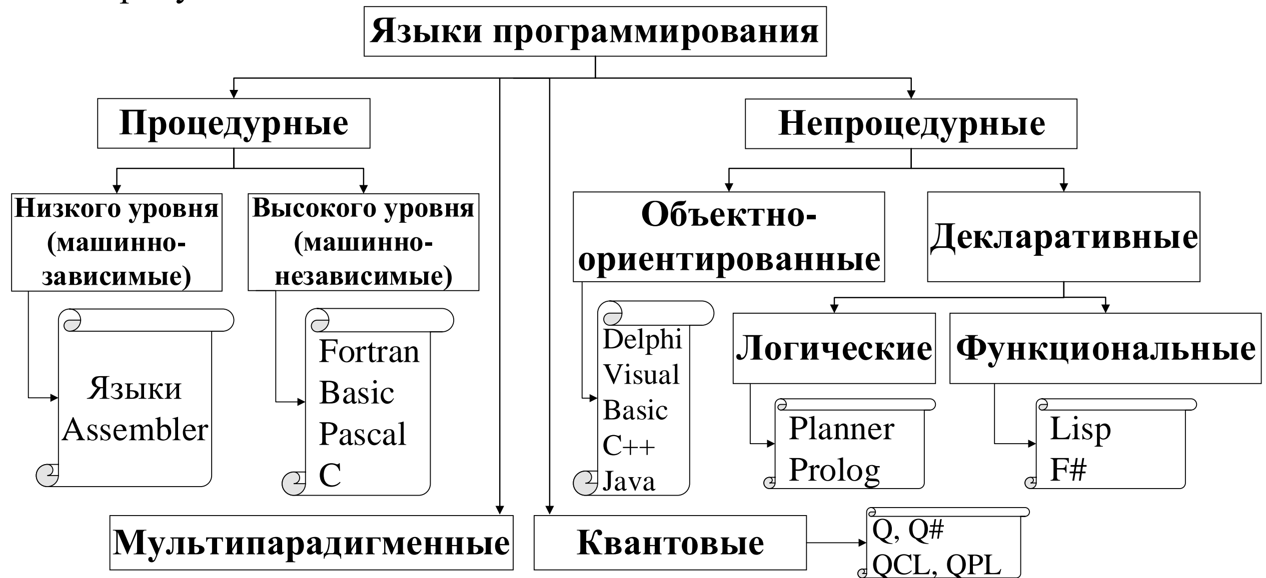


Рисунок 2.1 – Классификация языков программирования

Отдельную классификацию языков программирования представляет зависимость от аппаратной части вычислительной машины.

Изначально вычислительные машины программировались с помощью двоичных машинных кодов. Такой способ программирования колоссально трудоемок и зависим от системы команд конкретного процессора вычислительной машины, однако весьма изящен и эффективен с точки зрения требований к аппаратным ресурсам вычислительной машины.

Для упрощения программирования вычислительных машин позже были разработаны языки программирования низкого уровня (Assembler). Они являются машинно-зависимыми и программа, написанная на таком языке, в существенной степени определяется системой команд и архитектурой конкретного

процессора. При переносе программы на другую машину ее необходимо практически полностью переписывать. Программа, написанная на языке Assembler, переводится на язык машинных кодов при помощи транслятора.

Фактически все языки программирования, кроме Assembler, являются машинно-независимыми, т. е. высокого уровня, и глобально отличаются парадигмами программирования.

Парадигма программирования – это концепция, определяемая совокупностью идей и подходов к написанию компьютерных программ, организацией вычислений, структурированием работы программиста и вычислительной машины.

Перечислим наиболее известные парадигмы программирования.

1 Императивная. Характеризуется последовательностью выполнения команд, использованием именованных переменных и оператора присваивания, записью результатов и их чтением из памяти. Реализуется в первую очередь машинными кодами, языками ассемблеров, является фундаментом структурной, процедурной и даже объектно-ориентированной парадигмы.

2 Структурная. Характеризуется иерархической структурой блоков программы, в основе которой лежат три вида управляющих конструкций: последовательность, ветвление, цикл. Для выделения блоков применяются специальные слова и знаки (`begin` и `and`, `{` и `}`). Дополняет императивную парадигму. Реализуется многими высокоуровневыми языками.

3 Процедурная. Характеризуется объединением участков кода в подпрограммы (процедуры и функции) для многократного повторного использования с целью сокращения исходного кода программы. Дополняет императивную и структурную парадигму. Реализуется многими высокоуровневыми языками.

4 Объектно-ориентированная. Характеризуется представлением программы в виде совокупности взаимодействующих объектов, являющихся экземплярами своих классов, которые образуют иерархию наследования. Основными принципами парадигмы является инкапсуляция (объединение данных и кода, который их обрабатывает), наследование (возможность создавать новые типы данных путем расширения существующих) и полиморфизм (возможность использовать различные типы данных взаимозаменяемо). В своем роде дополняет предыдущие парадигмы. Реализуется объектно-ориентированными высокоуровневыми языками.

5 Функциональная. Характеризуется организацией вычислительного процесса как вычисления математических функций. Кардинально отличается даже само понятие функции от императивного и процедурного программирования. Вычисление значений функций происходит по входным данным или результатам других функций, как правило не предполагает явного хранения состояния программы (не содержит переменных и оператора присваивания) и всегда идентично при аналогичных аргументах (выходные данные зависят только от входных и не учитывают прочих состояний алгоритма). Существенно отличается от рассмотренных выше парадигм. Реализуется языками функционального

программирования и частично многими императивными высокоуровневыми языками.

6 Логическая. Основана на использовании математической логики: задании логических утверждений и аксиом и правил получения выводов по ним (дизъюнктов). Широко применяются логические механизмы древовидного представления структур данных и автоматического перебора с возвратом, сопоставления с образцом и др. Языки программирования не содержат переменных и оператора присваивания. Является специфической парадигмой, применяемой для класса задач, рассматривающих структурированные объекты и отношения между ними, экспертные системы. Реализуется логическими языками программирования.

7 Квантовые вычисления. Характеризуются вычислением кубитов (квантовых битов), способных одновременно принимать оба двоичных значения (нуль и единица), просчитывая таким образом одновременно все возможные состояния. Квантовые вычисления основываются на понятиях квантовой механики: квантовая суперпозиция, квантовая запутанность, квантовый параллелизм и т. д. Концептуально существенно отличается от других парадигм. Реализуется квантовыми высокоуровневыми языками.

К мультипрадигменным языкам относится целый ряд языков программирования, на которых чаще всего можно реализовывать императивную (в том числе структурную и процедурную) парадигму, объектно-ориентированную и частично функциональную. К таким языкам в большей или меньшей степени относятся Go, JavaScript, Phyton, Ruby и др.

Все языки программирования могут быть по-разному реализованы. Существуют три основных типа реализации.

1 Компилируемый язык. Исходный код программы на языке программирования преобразуется компилятором в язык машинных кодов для конкретного процессора и в конечном счете представляет собой отдельную неизменяемую исполняемую программу (файл). Необходимость изменения программы, изменения управляющей операционной системы или типа процессора влечет за собой необходимость изменения исходного кода и перекомпиляции. Скорость выполнения программ выше по сравнению с интерпретируемыми языками.

2 Интерпретируемый язык. Исходный код программы на языке программирования непосредственно сразу интерпретируется процессору без предварительного перевода на язык машинных кодов. Часто программы на интерпретируемых языках могут работать под управлением различных операционных систем и различных машинах.

3 Встраиваемые языки. По своей сути зависимы от другого транслируемого языка, в который они встроены. Часто базовый язык, в который встраивается другой, называют *метаязыком*. Довольно часто базовый, или встраиваемый, язык является визуальным. Это сделано для удобства его использования категорией лиц, для которых он разработан (не программистов). Наиболее известным встраиваемым языком является визуальный язык VBA (Visual Basic Application)

пакета MS Office и другого программного обеспечения. Он компилируется в промежуточный байт-код, выполняемый виртуальной машиной, которая, в свою очередь, управляется основным приложением.

Следует отметить, что реализации языков программирования достаточны условны. Нередко один и тот же язык может и компилироваться, и интерпретироваться, а встраиваемые языки частично интерпретируются.

2.2 Базовые понятия и элементы языка программирования высокого уровня

Любой язык программирования высокого уровня имеет свое лексическое, синтаксическое и семантическое описание.

Лексическое описание – это множество всех лексем (слов), используемых в языке программирования.

Синтаксическое описание – это совокупность правил составления лексем во фразах и конструкциях языка программирования, разметки программы и т. д.

Семантическое описание определяет логику, т. е. смысловое значение фраз и конструкций в исходном коде программы.

Любой язык программирования имеет свой алфавит, служебные слова, операторы и т. д.

Алфавит языка программирования – это набор возможных символов, используемых для составления лексем (инструкций, идентификаторов, служебных слов и т. д.

Лексема – неделимая последовательность символов алфавита (или один символ), имеющая(-ий) в программе определенный языком программирования или программистом смысл. Лексемами могут являться служебные слова, идентификаторы, литералы, выражения.

Алфавит языка программирования может включать:

– строчные и прописные латинские буквы ('A'...'Z', 'a'...'z');

– десятичные арабские цифры (0...9);

– знак нижнего подчеркивания ('_');

– специальные символы, такие как разделители ('.', ',', ';', ':', '"'), математические знаки ('+', '-', '*', '/', '=', '<', '>', '%'), различные скобки ('(', ')', '{', '}', '[', ']') и прочие символы ('!', '@', '#', '$', '&', '^', '?', пробел и др.).

С точки зрения применения алфавита языки программирования бывают:

– регистронезависимые (Basic, Fortran, Pascal, Lisp, Delphi и др.). Не различают написание букв в верхнем и нижнем регистрах (заглавных и строчных). Например, лексема будет иметь один и тот же смысл при следующих написаниях: for / FOR / For / fOr / foR / FOr / fOR;

– регистрозависимые (C++, C#, Java, Phyton, PHP и др.). Определяют по-разному одну и ту же букву в различных регистрах, что будет влиять на восприятие лексем. Например, лексемы «AX» и «ax» являются разными.

Служебные слова (*ключевые слова*) – это зарезервированные лексемы, состоящие только из букв, используемые языком программирования с заранее определенным смыслом и недопустимые для использования в качестве идентификаторов. Служебные слова могут обозначать совершенно различные лексемы (по смыслу, функциям, применению, значению и т. д.). Например, определять тип данных, способ их организации, последовательность выполнения операторов и т. п.

В разных языках программирования может быть от нескольких десятков до сотен служебных слов. Пример служебных слов для языков программирования Delphi и C++ представлен в таблице 2.1.

Таблица 2.1 – Служебные слова языков программирования Delphi и C++

Язык программирования Delphi	**Язык программирования C++**
`and, asm, array, begin, break, case, class, const, continue, div, do, downto, else, end, exit, file, for, function, goto, if, implementation, in, interface, label, mod, nil, not, object, of, or, procedure, program, record, repeat, set, string, then, unto, type, unit, until, uses, var, virtual, while, with, xor`	`auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, struct, switch, typedef, union, unsigned, void, volatile, while`

Идентификатор – это уникальное имя (лексема) конкретного объекта (переменной, пользовательского типа данных и т. д.), определяемое программистом и используемое средой программирования для связи с адресом пространства памяти, по которому физически будет располагаться именуемый объект.

Идентификатор позволяет однозначно определить единственный объект из множества всех объектов.

Правила составления идентификаторов:

– может состоять только из латинских букв любого регистра, десятичных чисел, знака нижнего подчеркивания;

– может начинаться только с буквы или знака нижнего подчеркивания;

– имеет ограниченное количество символов, воспринимаемых компилятором, или в целом ограниченное количество символов (длину идентификатора);

– не должен совпадать со служебными словами, именами функций из библиотек.

Кроме того, в тексте программы, помимо лексем, могут использоваться комментарии. Комментарии делают чтение исходного кода программы более понятным для программиста.

Комментарий – это пояснение, написанное на естественном языке, не являющееся лексемой и не предназначенное для выполнения программой.

Комментарий на естественном языке следует после лексемы (символа или группы символов), означающей начало комментария (например, // или { или /*).

Комментарии могут быть однострочными и многострочными. *Однострочный комментарий* имеет только лексему, обозначающую его начало, а *многострочный* – лексемы обозначающие и начало, и конец комментария. Примеры комментариев (обозначены курсивным начертанием) в синтаксисе языков программирования Delphi и C/C++ представлены в таблице 2.2.

Таблица 2.2 – Примеры комментариев

Тип комментария	**В синтаксисе Delphi**	**В синтаксисе C/C++**
Однострочный	*//Текст однострочного комментария*	*//Текст однострочного комментария*
Многострочный	*{Текст многострочного комментария}* *(* Текст многострочного комментария *)*	*/* Текст многострочного комментария */*

Таким образом, исходный текст программы состоит из лексем, пробелов и, возможно, комментариев.

2.3 Типизация данных. Данные языков программирования высокого уровня

Классификация языков программирования с точки зрения типизации данных представлена на рисунке 2.2.

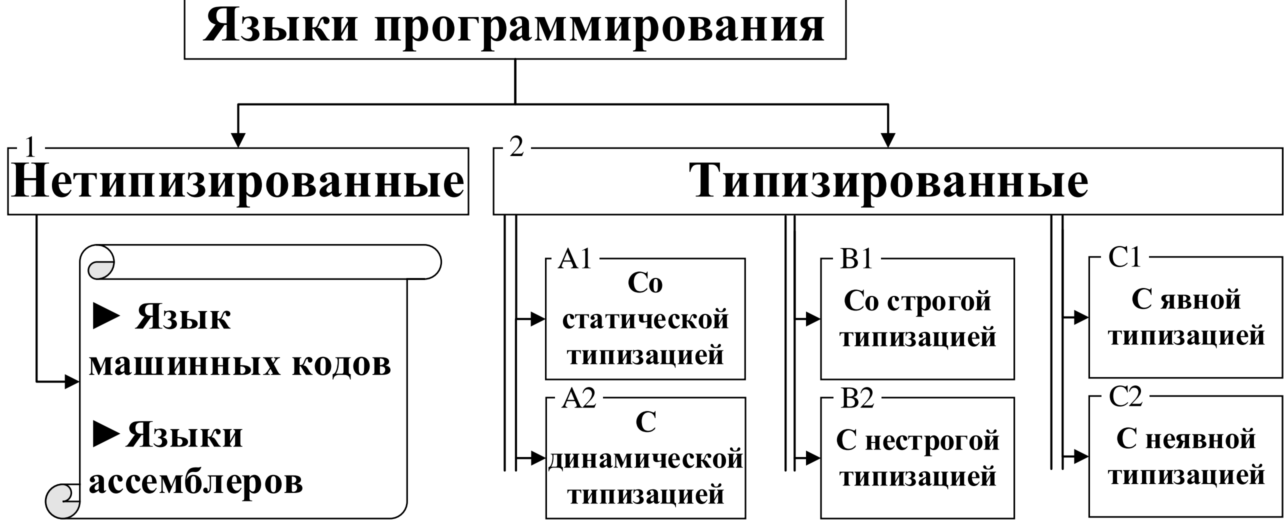


Рисунок 2.2 – Классификация языков программирования по типизации данных

Как видно из рисунка 2.2, языки программирования глобально бывают нетипизированные (1) и типизированные (2). Типизированные имеют три ветви классификации (*A*, *B*, *C*) с бинарными параллельными вариантами (1…2) – пересекающимися множествами в рамках ветвей *A*, *B*, *C*.

1 Нетипизированные языки. Все данные описываются последовательностью бит различной длины. Преимуществом данного подхода является эффективность кода программы, недостатком является сложность представления составных структур данных и отсутствие даже тривиальных проверок со стороны системы программирования. Нетипизированными языками являются машинный код и низкоуровневые языки программирования.

2.А1 Типизированные языки со статической типизацией. Проверки совместимости типов выполняются на этапе компиляции программы. К преимуществам данного подхода можно отнести то, что ошибки совместимости типов определяются сразу на этапе компиляции, что способствует увеличению скорости выполнения готовой программы. Статическая типизация присуща языкам программирования Pascal, Delphi, C, C#, Java и др.

2.А2 Типизированные языки с динамической типизацией. Проверки совместимости типов выполняются на этапе выполнения программы. Преимуществом данного подхода можно назвать удобство описания обобщенных алгоритмов (например, сортировки массива, которая не будет привязана к типу элементов). Недостатком являются более высокие требования к вычислительным мощностям машины, обусловленные необходимостью типизации во время выполнения программы, а также большее время выполнения программы по тем же причинам. Динамическую типизацию данных поддерживают языки программирования Python, JavaScript, Ruby и др.

Язык программирования C++ не относится явно к статической или динамической типизации, и, скорее, занимает промежуточное значение.

2.В1 Типизированные языки со строгой типизацией. Семантика языка программирования не позволяет смешивать в выражениях различные типы данных и выполнять автоматические преобразования типов. Преимуществами данного подхода можно назвать надежность и определенность получаемого результата (гарантированность отсутствия погрешностей автоматического преобразования типов от сложных к простым), понимание программистом сути преобразований. Недостатком, или, скорее, особенностью, является необходимость написания кода явного преобразования типов для совместимости выполняемых операций. Строгую типизацию поддерживают такие языки программирования, как Pascal, Delphi, Java, Python, Haskell, Lisp и др.

2.В2 Типизированные языки с нестрогой типизацией. Система программирования сама выполняет необходимые преобразования типов (как от более простых типов к более сложным, так и наоборот, но уже с очевидной потерей данных). Преимуществами данного подхода являются удобство вычисления смешанных выражений, лаконичность записи инструкции и акцентирование внимания на решаемой задаче, а не на типах данных. Недостатками являются потеря точности вычислений или полная потеря смысла данных, которая не вызовет формальной ошибки. Нестрогую типизацию поддерживают языки программирования C, C++, JavaScript, Visual Basic, PHP и др.

2.С1 Типизированные языки с явной типизацией. Характеризуются тем, что тип объявляемых данных (переменных, функций и их аргументов и т. д.) необходимо задавать в явном виде. Преимущество: явное понимание сути реализации необходимых процессов. Явную типизацию реализуют языки программирования Pascal, Delphi, C, C++, C#, D и др.

2.С2 Типизированные языки с неявной типизацией. Возлагают задачу задания типов объявляемых данных на компилятор или интерпретатор. Преимущества: сокращение исходного кода и устойчивость к вносимым изменениям (в первую очередь это касается аргументов и значений функций). Неявную типизацию реализуют языки программирования JavaScript, PHP, Lua и др.

Следует отметить, что варианты типизации языков в рамках ветвей *A*, *B*, *C* могут взаимопересекаться. Таким образом, классификация по типизации данных наиболее популярных типизированных языков программирования представлена в таблице 2.3.

Таблица 2.3 –Типизации данных в языках программирования

Язык программирования	**Типизация**		
Delphi	Статическая	Строгая	Явная
C	Статическая	Нестрогая	Явная
C++	Статическая	Нестрогая	Явная
C#	Статическая	Нестрогая	Явная
Java	Статическая	Строгая	Явная
JavaScript	Динамическая	Нестрогая	Неявная
PHP	Динамическая	Нестрогая	Неявная
Perl	Динамическая	Нестрогая	Неявная
Ruby	Динамическая	Строгая	Неявная
Python	Динамическая	Строгая	Неявная
Haskell	Статическая	Строгая	Неявная
Lisp	Динамическая	Строгая	Неявная
D	Статическая	Строгая	Явная

Язык программирования Delphi является типизированным со статической строгой явной типизацией, а языки программирования C/C++ – типизированными со статической нестрогой явной типизацией.

Изменение значений данных, именованных идентификаторами, в процессе выполнения программы может быть допустимо (переменные) или запрещено (константы).

Переменная – это данные, обозначенные идентификатором (именем), значения которых могут модифицироваться (изменяться) в процессе выполнения программы.

Константа – это данные, обозначенные самим значением (литерал) или идентификатором (именем), значения которых не могут быть изменены в процессе выполнения программы.

Выражение – это лексема, которая может состоять из определенной последовательности идентификаторов, являющихся операндами, и символов алфавита языка программирования, являющихся операторами.

В типизированных языках программирования (Delphi и C/C++) как переменная, так и константа должны иметь определенный тип хранимых данных. Даже выражение, состоящее из переменных и/или констант, в том числе разных типов данных, имеет определенный тип. Тип выражения определяется типом результата его вычисления.

Тип данных определяет:

– множество значений, которые могут принимать объекты (переменные, константы и т. д.) данного типа;

– формат хранения данных в памяти компьютера;

– свойства данных значений (например, количество значащих разрядов после запятой в вещественном типе данных);

– операции, допустимые над объектами данного типа.

Типы данных в типизированных языках программирования высокого уровня (Delphi/C++) можно в общем виде классифицировать, как показано на рисунке 2.3.

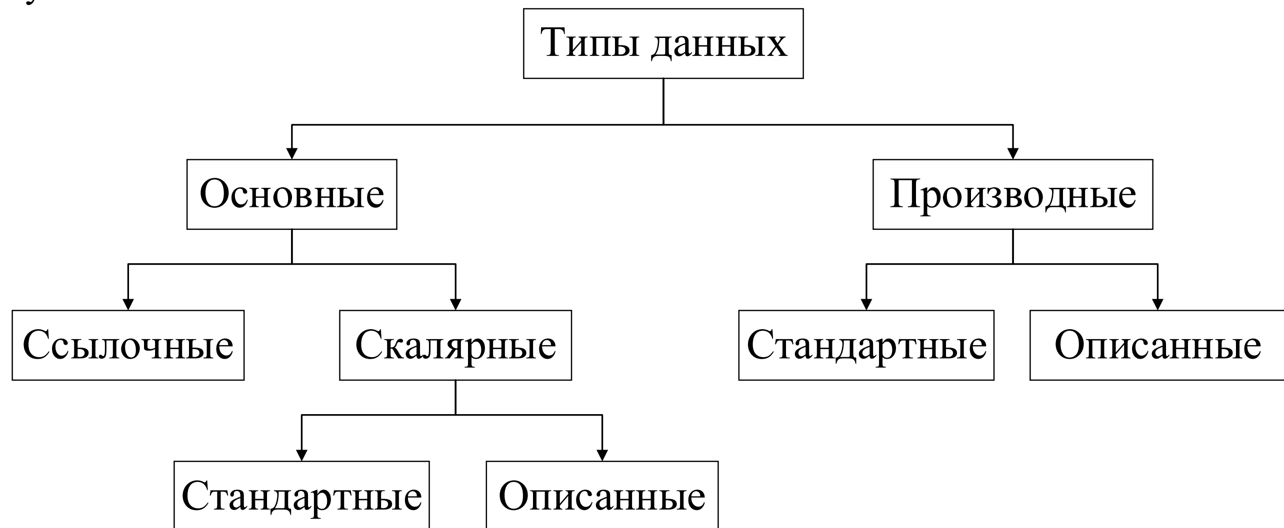


Рисунок 2.3 – Общая классификация данных в типизированных языках программирования высокого уровня

В соответствии с этой классификацией основными являются типы, которые состоят из одного единственного элемента данных и имеют тривиальную структуру данных. Производные в свою очередь основываются на других типах, причем как основных, так и других производных. Скалярные типы относятся к основным. Все, кроме ссылочных, являются типами значений. Объекты (переменные, константы и т. д.) данных типов непосредственно хранят свои значения. Объекты ссылочного типа хранят ссылки на их данные и, как правило, хранятся в стеке или куче.

Как основные скалярные типы, так и производные делятся на стандартные и описанные. Стандартными типами являются типы данных, предопределенные

языком программирования на уровне синтаксиса, т. е. название типа является служебным (зарезервированным) словом языка программирования. Описанные типы определяются программистом (например, перечисления (скалярный), массив, структура (производные)).

Большинство скалярных типов являются перенумерованными, т. е. упорядоченными линейно таким образом, что можно однозначно утверждать, какой элемент типа предшествует другому.

По сути хранимых в памяти данных типы данных языков программирования Delphi и C++ можно классифицировать, как показано на рисунке 2.4.

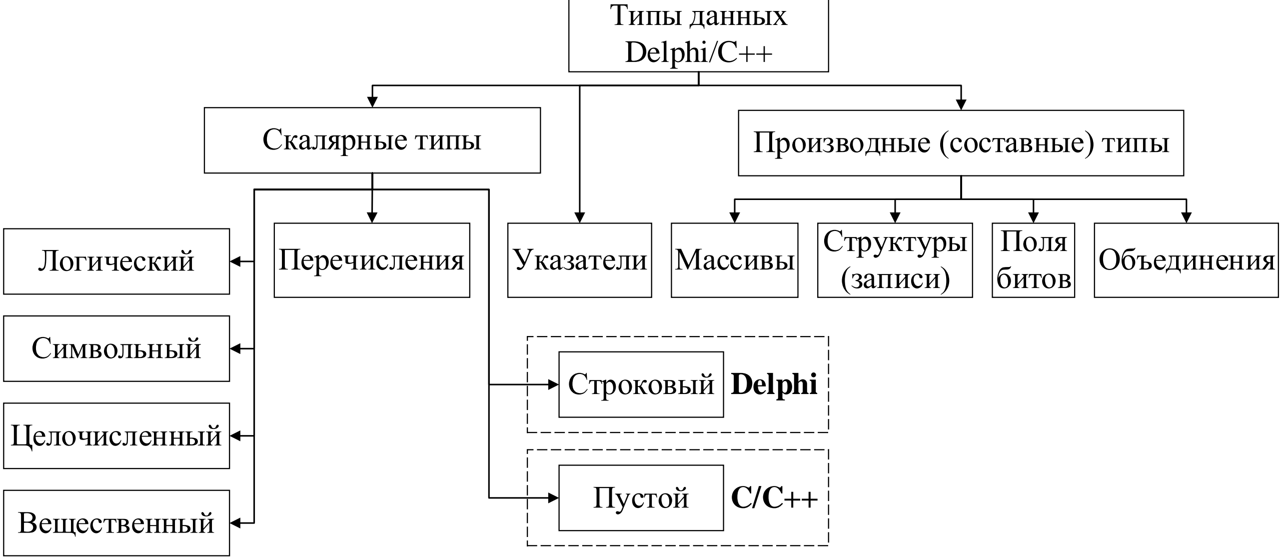


Рисунок 2.4 – Классификация типов данных языков программирования высокого уровня Delphi/C++

На рисунке 2.4 различия типов данных языков программирования Delphi и C++ обозначены штрихпунктирной линией. В языке программирования Delphi, в отличие от языков C/C++, есть отдельный строковый тип данных (string). В языке программирования C/C++ существует понятие символьной строки, представляющей собой массив символов, последним элементом которого является признак конца строки – нуль-терминатор (нуль-символ) «\0». В языке программирования C/C++, в отличие от языка Delphi, существует пустой тип данных (void).

Пустой тип void может использоваться в следующих случаях:

– указание о невозвращении значения функцией, когда у функции указан void-тип возвращаемого результата. В этом случае функция не может возвращать результат (т. е. использование оператора return недопустимо);

– указание о неполучении параметров функцией, когда в качестве типа аргументов (параметров) функции указан void;

– создание нетипизированных указателей, когда заранее неизвестно, на какой тип данных будет указывать указатель, и для получения данных, на которые ссылается void-указатель, необходимо в конечном счете выполнить приведение указателя к корректному типу данных, находящихся по адресу, содержащемуся в void-указателе.

Как было указано ранее, данные могут быть переменными и константными. Причем константные данные могут задаваться значением (литералом) или переменной.

Литерал – это безымянная константа, записанная в синтаксисе языка программирования и имеющая фиксированное значение.

Литералы бывают различных типов: числовые, символьные, строковые и логические. Числовыми литералами могут являться не только десятичные целые и дробные (вещественные) числа, но и восьмеричные, и шестнадцатеричные. Символьные и строковые литералы могут содержать не только латинские буквы и различные специальные знаки, но и символы других (национальных) алфавитов, в том числе кириллицу.

Примеры записи различных типов литералов в синтаксисе языков программирования Delphi и C/C++ представлены в таблице 2.4.

Таблица 2.4 – Примеры записи литералов

<table>
<tr><th colspan="4" rowspan="2">Тип литерала</th><th colspan="2">В синтаксисе Delphi</th><th colspan="2">В синтаксисе C/C++</th></tr>
<tr><th>Пример 1</th><th>Пример 2</th><th>Пример 1</th><th>Пример 2</th></tr>
<tr><td rowspan="5">Число-вой</td><td rowspan="4">десятичное число</td><td colspan="2">целочисленное</td><td>23</td><td>-594</td><td>23</td><td>-594</td></tr>
<tr><td rowspan="2">веще-ствен-ное</td><td>с фиксирован-ной точкой</td><td>-0.33333</td><td>3.14</td><td>-0.33333</td><td>3.14</td></tr>
<tr><td>с плавающей точкой</td><td>1.7E-5</td><td>3.14E00</td><td>1.7E-5</td><td>3.14E00</td></tr>
<tr><td colspan="3">шестнадцатеричное целочисленное</td><td>$49FC2</td><td>-$A3D</td><td>0x49FC2</td><td>-0xA3D</td></tr>
<tr><td colspan="3" rowspan="4">Символьный</td><td colspan="1">латиница</td><td>'q'</td><td>'Q'</td><td>'q'</td><td>'Q'</td></tr>
<tr><td>кириллица</td><td>'ы'</td><td>'Ы'</td><td>'ы'</td><td>'Ы'</td></tr>
<tr><td>число как символ</td><td>'5'</td><td>'0'</td><td>'5'</td><td>'0'</td></tr>
<tr><td>другие символы</td><td>'?'</td><td>' '</td><td>'?'</td><td>' '</td></tr>
<tr><td colspan="4">Строковый</td><td>'total = '</td><td>'Ввод x'</td><td>"total = "</td><td>"Ввод x"</td></tr>
<tr><td colspan="4">Логический</td><td>true</td><td>false</td><td>true*[1]</td><td>false*[1]</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td colspan="8">*[1] логические литералы (true и false) используются только в языке программирования C++ (начиная с 11-й версии стандарта). В языке программирования C для обозначения значения «ложь» используется целочисленный литерал «0» (нуль), а для обозначения значения «истина» – любой целочисленный литерал, отличный от нуля</td></tr>
</table>

Отдельно следует отметить непечатные литералы (управляющие символы), используемые для форматирования и разметки выводимых данных. Такие литералы в языке программирования Delphi могут задаваться с помощью номера символа, соответствующего кодировке, а в языке программирования C/C++ – с помощью управляющей последовательности (escape-последовательности), представляющей собой группу символов, идущих друг за другом и имеющих не значение последовательности индивидуальных смыслов, а один новый общий смысл (таблица 2.5).

Таблица 2.5 – Примеры непечатных литералов

Значение непечатного литерала	**В синтаксисе Delphi**	**В синтаксисе C/C++**
Перевод на новую строку	`#10`	`'\n'`
Горизонтальная табуляция	`#9`	`'\t'`
Обратное перемещение (Backspace)	`#8`	`'\b'`
Возврат каретки	`#13`	`'\r'`
Нуль-терминатор	`#0`	`'\0'`
Escape (Esc)	`#27`	`'\e'`
Вертикальная табуляция	Нет	`'\v'`

Также следует отметить сложность использования в литералах печатных символов «одинарная кавычка» (в Delphi и C/C++) и «двойная кавычка» (в C/C++), поскольку они будут восприняты как конец обозначения литерал, а не как его часть. Данная проблема обрабатывается аналогичным образом с использованием кодов соответствующих символов в Delphi и с помощью управляющих последовательностей в C/C++. Так символ «одинарная кавычка» может быть записан: `#39` (в Delphi) или '\'' (в C/C++). Символ «двойная кавычка» может быть записан как `'\''` (в C/C++).

2.4 Модификаторы типов данных

Помимо самих типов данных формат хранения данных, диапазон принимаемых значений и свойства типов могут определять модификаторы типов, используемые в языках программирования C/C++.

Модификаторы типа нельзя использовать с пустым типом void.

В языках программирования C/C++ существуют следующие модификаторы типов.

1 Модификатор типа **unsigned** (беззнаковый) может применяться с символьным и целочисленными типами и предназначен для исключения из диапазона значений типа области отрицательных значений, при этом нижней границей диапазона принимаемых значений является нуль, а верхняя граница диапазона увеличивается на значение исключенного множества отрицательных значений.

2 Модификатор типа **signed** (знаковый) может применяться с символьным и целочисленными типами и по сути является избыточным (поскольку символьные и целочисленные типы без модификатора априори являются знаковыми), но не запрещенным. Диапазон принимаемых значений включает в себя как область отрицательных значений, так и область положительных значений, причем эти области делятся поровну (нуль входит в положительную область).

Отличие знаковых целочисленных данных от беззнаковых отличается в том числе представлением значений в памяти. Старший бит знаковых данных используется для обозначения знака числа, а в беззнаковых данных – используется для хранения значения данных, как и другие младшие биты. Пример

представления данных символьного типа char с модификаторами типа unsigned (беззнаковый), signed (знаковый) представлен на рисунке 2.5.

Модификатор типа	Пример объявления данных и их значение	Пример представления в памяти

signed	`signed char a=127;`
	`//или аналогично`
	`char a=127;`
		0 1 1 1 1 1 1 1
		знак хранимое значение a

	`signed char b=-127;`
	`//или аналогично`
	`char b=-127;`
		1 1 1 1 1 1 1 1
		знак хранимое значение b

unsigned	`unsigned char c='я';`
	`//аналогично`
	`unsigned char c=255;`
		1 1 1 1 1 1 1 1
		хранимое значение c

	`unsigned char d=0;`	0 0 0 0 0 0 0 0 хранимое значение d

Рисунок 2.5 – Пример представления знаковых и беззнаковых данных в памяти

3 Модификатор типа **short** (короткий) может применяться с целочисленными типами и используется для сокращения вдвое размера выделяемой под данные памяти, тем самым сокращая диапазон принимаемых значений.

4 Модификатор типа **long** (длинный) может применяться с целочисленными типами и вещественным типом double и используется для увеличения вдвое размера выделяемой под данные памяти, тем самым соответственно увеличивая диапазон принимаемых значений.

Допускается префиксное использование модификаторов типа, например можно комбинировать знаковые / беззнаковые и короткие / длинные модификаторы типа: unsigned long int, signed long int, unsigned short int, signed short int.

Кроме того, в языке программирования C++ допускается использование сокращенной записи модификаторов типа без указания типа int, являющегося базовым для модификаторов, например:

```
unsigned x; //эквивалентно unsigned int x
short y; //эквивалентно short int y или short signed int y
```

2.5 Основные скалярные типы данных и их свойства в языках программирования Delphi и C/C++

В языке программирования Delphi не существуют модификаторы типа, зато предусмотрены отдельные типы данных, которые являются знаковыми (signed), беззнаковыми (unsigned), короткими (short) и длинными (long).

Основные скалярные типы данных и их свойства языка программирования Delphi представлены в таблице 2.6.

Таблица 2.6 – Основные типы данных языка программирования Delphi

Символьные типы				
Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Кодировка**	**Размер выделяемой памяти**
AnsiChar	0	255	ANSI	1 байт
Char	0	255	ANSI	1 байт
WideChar	0	65 535	Unicode	2 байта

Логические типы			
Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Размер выделяемой памяти**
Boolean	false	true	1
Bytebool	false	true	1
Bool	false	true	2
Wordbool	false	true	2
Longbool	false	true	4

Целочисленные типы				
Знак	**Тип**	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Размер выделяемой памяти**
Знаковые (Signed)	ShortInt	–128	127	1 байт
	SmallInt	–32 768	32 767	2 байта
	Integer	–2 147 483 648	2 147 483 647	4 байта
	LongInt	–2 147 483 648	2 147 483 647	4 байта
	Int64	-2^{63}	$2^{63}-1$	8 байт
Беззнако-вые (Unsigned)	Byte	0	255	1 байт
	Word	0	65 535	2 байта
	Longword	0	4 294 967 295	4 байта
	Cardinal	0	4 294 967 295	4 байта

Вещественные типы				
Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Значащих цифр после запятой**	**Размер выделяемой памяти**
Single	$1{,}5 \times 10^{-45}$	$3{,}4 \times 10^{38}$	7...8	4 байта
Real	5×10^{-324}	$1{,}7 \times 10^{308}$	15...16	8 байт
Double	5×10^{-324}	$1{,}7 \times 10^{308}$	15...16	8 байт
Extended	$3{,}6 \times 10^{-4951}$	$1{,}1 \times 10^{4932}$	19...20	10 байт
Comp	-2^{63}	$2^{63}-1$	19...20	8 байт
Currency	±922 337 203 685 477,5807		19...20	8 байт

Строковые типы				
Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Кодировка**	**Размер выделяемой памяти**
ShortString	0	255	ANSI	1 байт
String	0	255	ANSI	1 байт
LongString	0	Ограничено объемом свободной оперативной памяти	ANSI	Ограничено объемом свободной оперативной памяти
WideString	0		Unicode	

Основные скалярные типы данных (и их модификаторы), их свойства в языке программирования С++ представлены в таблице 2.7.

Таблица 2.7 – Основные типы данных языка программирования С++

Символьные типы

Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Кодировка**	**Размер выделяемой памяти**
char	–128	127	ANSI	1 байт
unsigned char	0	255	ANSI	1 байт
signed char	–128	127	ANSI	1 байт

Логический тип

Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Размер выделяемой памяти**
Bool	false	true	1 байт

Целочисленные типы

Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Размер выделяемой памяти***
int	–2 147 483 648	2 147 483 647	4 байта
unsigned int	0	4 294 967 295	4 байта
signed int	–2 147 483 648	2 147 483 647	4 байта
short int	–32 768	32 767	2 байта
unsigned short int	0	65 535	2 байта
signed short int	–32 768	32 767	2 байта
long int	–2 147 483 648	2 147 483 647	4 байта
unsigned long int	0	4 294 967 295	4 байта
signed long int	–2 147 483 648	2 147 483 647	4 байта

Вещественные типы

Тип	**Нижняя граница диапазона**	**Верхняя граница диапазона**	**Значащих цифр после запятой**	**Размер выделяемой памяти**
float	$3{,}4 \times 10^{-38}$	$3{,}4 \times 10^{38}$	7	4 байт
double	$1{,}7 \times 10^{-308}$	$1{,}7 \times 10^{308}$	15	8 байт
long double	$3{,}4 \times 10^{-4932}$	$1{,}1 \times 10^{4932}$	19	10 байт

* *Следует отметить, что размер памяти для типов int и unsigned int не определены в языке С/С++ и зависят от разрядности процессора, т. е. размер выделяемой памяти для типа int соответствует размеру машинного слова. При обработке на 16-разрядной ЭВМ тип int составляет 16 бит, или 2 байта (слово), на 32-разрядной – тип int составляет 32 бита, или 4 байта (слово) и т. д.*

2.6 Операции, определенные над скалярными типами данных

Тип данных определяет в том числе и операции, которые допустимо выполнять над данным типом.

Над целочисленными данными определены арифметические, поразрядные и логические операции, операции сдвига и сравнения (таблица 2.8).

Таблица 2.8 – Операции, определенные над целочисленными типами

<table>
<tr><th colspan="2">Оператор в синтаксисе</th><th rowspan="2">Вид оператора</th><th rowspan="2">Описание</th><th rowspan="2">Тип получаемого результата</th></tr>
<tr><th>Delphi</th><th>C++</th></tr>
<tr><td colspan="5">1 Арифметические операции</td></tr>
<tr><td colspan="2">+</td><td>Унарный</td><td>Сохранение знака</td><td>Целый</td></tr>
<tr><td colspan="2">−</td><td>Унарный</td><td>Отрицание знака</td><td>Целый</td></tr>
<tr><td colspan="2">+</td><td>Бинарный</td><td>Сложение</td><td>Целый</td></tr>
<tr><td colspan="2">−</td><td>Бинарный</td><td>Вычитание</td><td>Целый</td></tr>
<tr><td colspan="2">*</td><td>Бинарный</td><td>Умножение</td><td>Целый</td></tr>
<tr><td colspan="2">/</td><td>Бинарный</td><td>Деление</td><td>Вещественный</td></tr>
<tr><td>div</td><td>/ [*1]</td><td>Бинарный</td><td>Целочисленное деление</td><td>Целый</td></tr>
<tr><td>mod</td><td>%</td><td>Бинарный</td><td>Остаток целочисленного деления</td><td>Целый</td></tr>
<tr><td colspan="5">[*1] Целочисленное деление выполняется обычным оператором деления, при этом полученный результат присваивается переменной целочисленного типа. Результат деления целого числа на целое – это всегда целое число, остаток при делении отбрасывается!</td></tr>
<tr><td colspan="5">2 Поразрядные операции</td></tr>
<tr><td>not</td><td>~</td><td>Унарный</td><td>Поразрядное дополнение целого</td><td>Целый</td></tr>
<tr><td>and</td><td>&</td><td>Бинарный</td><td>Поразрядное логическое умножение (И)</td><td>Целый</td></tr>
<tr><td>or</td><td>|</td><td>Бинарный</td><td>Поразрядное логическое сложение (ИЛИ)</td><td>Целый</td></tr>
<tr><td>xor</td><td>^</td><td>Бинарный</td><td>Поразрядное логическое Исключающее ИЛИ</td><td>Целый</td></tr>
<tr><td colspan="5">3 Логические операции</td></tr>
<tr><td>not</td><td>!</td><td>Унарный</td><td>Отрицание</td><td>Логический</td></tr>
<tr><td>and</td><td>&&</td><td>Бинарный</td><td>Логическое умножение (И)</td><td>Логический</td></tr>
<tr><td>or</td><td>||</td><td>Бинарный</td><td>Логическое сложение (ИЛИ)</td><td>Логический</td></tr>
<tr><td>xor</td><td>нет [*2]</td><td>Бинарный</td><td>Логическое Исключающее ИЛИ</td><td>Логический</td></tr>
<tr><td colspan="5">[*2] Оператор Исключающее ИЛИ может быть реализован также через комбинацию операторов «&&», «||» и «!». Например, result = (x || y) && !(x && y);</td></tr>
</table>

Продолжение таблицы 2.8

Оператор в синтаксисе		**Вид оператора**	**Описание**	**Тип получаемого результата**
Delphi	**C++**			
4 Операции сдвига				
shl	<<	Бинарный	*i оператор сдвига влево j* – сдвиг влево значения *i* на *j* бит	Целый
shr	>>	Бинарный	*i оператор сдвига вправо j* – сдвиг вправо значения *i* на *j* бит	Целый
5 Операции сравнения				
=	==	Бинарный	Равно	Логический
<>	!=	Бинарный	Не равно	Логический
<		Бинарный	Меньше	Логический
>		Бинарный	Больше	Логический
<=		Бинарный	Меньше или равно	Логический
>=		Бинарный	Больше или равно	Логический

Над вещественными типами определены арифметические операции и операции сравнения (таблица 2.9).

Таблица 2.9 – Операции, определенные над вещественными типами

Оператор в синтаксисе		**Вид оператора**	**Описание**	**Тип получаемого результата**
Delphi	**C++**			
1 Арифметические				
+		Унарный	Сохранение знака	Вещественный
−		Унарный	Отрицание знака	Вещественный
+		Бинарный	Сложение	Вещественный
−		Бинарный	Вычитание	Вещественный
*		Бинарный	Умножение	Вещественный
/		Бинарный	Деление	Вещественный
2 Операции сравнения				
=	==	Бинарный	Равно	Логический
<>	!=	Бинарный	Не равно	Логический
<		Бинарный	Меньше	Логический
>		Бинарный	Больше	Логический
<=		Бинарный	Меньше или равно	Логический
>=		Бинарный	Больше или равно	Логический

Над символьными типами определены только операции сравнения (таблица 2.10).

Таблица 2.10 – Операции, определенные над символьными типами

<table>
<tr><th colspan="2">Оператор в синтаксисе</th><th rowspan="2">Вид оператора</th><th rowspan="2">Описание</th><th rowspan="2">Тип получаемого результата</th></tr>
<tr><th>Delphi</th><th>C++</th></tr>
<tr><td colspan="5">Операции сравнения</td></tr>
<tr><td>=</td><td>==</td><td>Бинарный</td><td>Равно</td><td>Логический</td></tr>
<tr><td><></td><td>!=</td><td>Бинарный</td><td>Не равно</td><td>Логический</td></tr>
<tr><td colspan="2"><</td><td>Бинарный</td><td>Меньше</td><td>Логический</td></tr>
<tr><td colspan="2">></td><td>Бинарный</td><td>Больше</td><td>Логический</td></tr>
<tr><td colspan="2"><=</td><td>Бинарный</td><td>Меньше или равно</td><td>Логический</td></tr>
<tr><td colspan="2">>=</td><td>Бинарный</td><td>Больше или равно</td><td>Логический</td></tr>
</table>

Над логическими типами определены логические операции и операции сравнения (таблица 2.11).

Таблица 2.11 – Операции, определенные над логическими типами

<table>
<tr><th colspan="2">Оператор в синтаксисе</th><th rowspan="2">Вид оператора</th><th rowspan="2">Описание</th><th rowspan="2">Тип получаемого результата</th></tr>
<tr><th>Delphi</th><th>C++</th></tr>
<tr><td colspan="5">1 Логические</td></tr>
<tr><td>not</td><td>!</td><td>Унарный</td><td>Отрицание</td><td>Логический</td></tr>
<tr><td>and</td><td>&&</td><td>Бинарный</td><td>Логическое умножение (И)</td><td>Логический</td></tr>
<tr><td>or</td><td>||</td><td>Бинарный</td><td>Логическое сложение (ИЛИ)</td><td>Логический</td></tr>
<tr><td>xor</td><td>нет[*1]</td><td>Бинарный</td><td>Логическое Исключающее ИЛИ</td><td>Логический</td></tr>
<tr><td colspan="5">[*1] Оператор Исключающее ИЛИ может быть реализован также через комбинацию операторов «&&», «||» и «!». Например, result = (x || y) && !(x && y);</td></tr>
<tr><td colspan="5">2 Операции сравнения</td></tr>
<tr><td>=</td><td>==</td><td>Бинарный</td><td>Равно</td><td>Логический</td></tr>
<tr><td><></td><td>!=</td><td>Бинарный</td><td>Не равно</td><td>Логический</td></tr>
<tr><td colspan="2"><</td><td>Бинарный</td><td>Меньше</td><td>Логический</td></tr>
<tr><td colspan="2">></td><td>Бинарный</td><td>Больше</td><td>Логический</td></tr>
<tr><td colspan="2"><=</td><td>Бинарный</td><td>Меньше или равно</td><td>Логический</td></tr>
<tr><td colspan="2">>=</td><td>Бинарный</td><td>Больше или равно</td><td>Логический</td></tr>
</table>

2.7 Преобразование стандартных скалярных типов данных

Как было отмечено ранее, язык программирования Delphi является типизированным со статической строгой явной типизацией. Это означает, что при использовании в выражении разных типов или при присвоении значения одного типа переменной другого типа, необходимо выполнять преобразование типов отличающихся данных.

Функции преобразования типов языка программирования Delphi представлены в таблице 2.12.

Таблица 2.12 – Функции преобразования типов Delphi

Функция преобразования	**Описание**
`StrToInt(s)`	Преобразует строку *S* в целое число
`StrToFloat(s)`	Преобразует строку *S* в вещественное число
`IntToStr(n)`	Преобразует целое число *N* в строку
`FloatToStr(x)`	Преобразует вещественное число *X* в строку
`FloatToStrF(x,format,i,j)`	Преобразует вещественное число *X* в строку заданного формата[*1], где *i* задает общее количество десятичных цифр мантиссы, *j* – количество цифр в дробной части
[*1] *Форматы преобразований представлены в таблице 2.13*	

Таблица 2.13 – Форматы преобразований типов Delphi

Значение параметра format	**Описание**
`fffExponent`	Научная форма представления с множителем *eXX*, где *i* задает общее количество десятичных цифр мантиссы, *j* – количество цифр в десятичном порядке *XX*
`ffFixed`	Формат с фиксированным положением разделителя целой и дробной частей, где *i* задает общее количество десятичных цифр в представлении числа, *j* – количество цифр в дробной части
`ffGeneral`	Универсальный формат, использующий наиболее удобную для чтения форму представления вещественного числа (≡ ffFixed, если количество цифр в целой части <= *i*, а само число >= 0,00001, иначе ≡ ffExponent)
`ffNumber`	Отличается от ffFixed использованием символа-разделителя тысяч (пробел) при выводе больших чисел
`ffCurrency`	Денежный формат. Соответствует ffNumber, + конце строки ставится символ денежной единицы (например, «р.»)

Языки программирования C/C++ являются типизированными со статической нестрогой явной типизацией. Нестрогая типизация означает, что допустимо в выражениях использовать различные типы данных без предварительного приведения к одному типу. Однако это не означает, что приведение типов не выполняется вовсе. В языках программирования C/C++ существует неявное и явное преобразование типов.

Неявное преобразование типов предполагает автоматическое приведение компилятором разных типов в выражении к более высокому типу (без потери

данных) или наоборот (с возможной потерей данных). Иерархия типов языков программирования C/C++ представлена на рисунке 2.6.

Более низкие типы — Более высокие типы

char → short int → int → long int → float → double → long double

bool ← → int

Рисунок 2.6 – Иерархия типов языков программирования C/C++

Неявное преобразование типов без проблем и потери данных может быть выполнено компилятором от более низких в иерархии типов к более высоким. Кроме того, в обе стороны без проблем выполняется неявное преобразования типов bool и int: значение «false» преобразуется в нуль, а значение «true» преобразуется в единицу при преобразовании от bool к int, и при преобразовании в обратную сторону нуль преобразуется в «false» и любое, отличное от нуля значение – в «true».

Пример неявного преобразования типов (листинг 2.1):

Листинг 2.1 – Пример неявного преобразования типов

```
1     int x = 3;
2     double y = 5.8;
3     y = x; //происходит неявное приведение типа int к типу double
4     x = y; //происходит неявное приведение типа double к типу int
```

Явное преобразование типов совершается самим программистом для исключения возможных ошибок автоматического неявного преобразования компилятором. Явное преобразование типов может быть реализовано в языке программирования C++ несколькими способами, в том числе следующими:

1 Явная спецификация типа операндов. Например, необходимо разделить целочисленный литерал на другой целочисленный литерал. Для исключения потери данных можно выполнить спецификацию типа одного из операндов (листинг 2.2).

Листинг 2.2 – Спецификация типа одного из операндов

```
1     float(7)/2; //результат будет 3.5, т. к. приведя один из
2     // литералов к вещественному типу, результат будет
3     //автоматически приведен также к вещественному типу
4     float(c)=7/2; //при этом такая запись выражения выдаст
5     // результат целочисленного деления, т. е. 3
```

2 Использование унарной операции приведения типа следующего вида:

```
static_cast </*тип данных*/> (/*переменная или число*/);
```

Пример приведен в листинге 2.3.

Листинг 2.3 – Использование операции приведения типа
```
1   static_cast<float>(7)/2; //результат будет равен 3.5
2   //или
3   int x=7;
4   static_cast<float>(x)/2; //результат будет равен 3.5
```

2.8 Модификаторы доступа

В языках программирования С/С++ существуют два типа модификаторов, используемых для контроля за модификацией данных const и volatile.

1 Модификатор доступа **const** используется для запрета изменения значений переменных в процессе выполнения программы, например:

```
const int x; //компилятор не допустит изменений значений
             //переменной x
```

Модификатор доступа const, как правило, используется при инициализации переменных и помимо инициализации данная переменная уже не может быть модифицирована в процессе выполнения программы.

2 Модификатор доступа **volatile** используется для разрешения изменения значения данных способами неопределенными в программе. Например, адрес объявленной в программе глобальной переменной может быть передан в подпрограмму часов операционной системы и значение переменной будет изменяться без оператора присваивания в самой программе. Пример использования модификатора доступа volatile:

```
volatile int w;
```

Оба модификатора доступа могут использоваться совместно в ряде случаев. Например, когда необходимо установить запрет на изменение адреса порта (например 0x32) в программе, при этом разрешив его изменение внешним устройством, например:

```
const volatile unsigned char *port = 0x32;
```

2.9 Спецификаторы класса памяти

В языках программирования С/С++ существует четыре спецификатора класса памяти, определяющих способ распределения памяти для переменной и область программы, где переменную можно использовать: **auto** (автоматический), **static** (статический), **extern** (внешний) и **register** (регистровый). Спецификатор класса памяти может отсутствовать, тогда класс памяти определяется по контексту инструкции.

1 Данные, описанные со спецификатором класса памяти **auto**, доступны лишь в той функции, в которой они объявлены. Эти данные являются локальными, они создаются при входе в блок и разрушаются при выходе из этого блока.

С развитием языка С++ спецификатор auto получил новые функции – он позволяет автоматически выводить тип переменной. При использовании спецификатора в данном случае обязательно должна выполняться инициализация (задание начального значения) переменной. Тогда спецификатор auto автоматически задает тип переменной аналогичный типу инициализатора (листинг 2.4):

Листинг 2.4 – Автоматическое задание типа переменной модификатором auto

```
1   auto i=10; //из значения 10 выводится тип int, который задается
2              //переменной i
3   auto d=3.14; //из значения 3.14 выводится тип double, который
4                //задается переменной d
```

2 Данные, описанные со спецификатором класса памяти **static**, могут быть локальными и глобальными. Локальные данные доступны в любой из функций нижнего уровня по отношению к функции, в которой они объявлены. Эти локальные данные хранятся все время выполнения программы. Глобальные данные доступны в пределах файла, в котором они описаны со спецификатором класса памяти static.

3 Данные, описанные со спецификатором класса памяти **extern**, имеют внешнюю привязку, т. е. компилятору сообщается только тип данных, а память под них может быть выделена в любом другом месте программы. Данные должны быть описаны до начала описания основной функции. Внешние данные доступны к использованию в любой из функций файла, а также в других файлах программы.

4 Данные, описанные со спецификатором класса памяти **register**, изначально предполагались для хранения в регистрах процессора (более быстрый доступ к данным, нежели из оперативной памяти). В настоящее время спецификатор класса памяти register сигнализирует компилятору о необходимости оптимизации хранения данных. Символьные и целочисленные данные могут храниться в регистрах процессора при условии наличия свободных регистров, другие данные могут быть определены в кэш-память. Однако использование спецификатора класса памяти register не означает обязательной оптимизации хранения.

2.10 Понятия объявления, определения и инициализации

Следует различать три основных понятия, применяемых к объектам данных: объявление, определение и инициализация.

Все данные (и функции в том числе) перед их использованием должны быть как минимум объявлены, чтобы компилятор понимал, что данный идентификатор является не ошибкой, а именем некоторых данных (функции).

Объявление – это указание информации (типа и идентификатора) об объекте данных (переменных, констант, типов, меток и т. д.).

Определение предполагает выделение памяти для объявленных данных и связывание этой области памяти с идентификатором данных.

Инициализация – это определение данных и задание их начальных значений. Следует отметить, что эти начальные значения могут модифицироваться далее в процессе выполнения программы.

Объявление данных в языке программирования Delphi осуществляется в соответствующих подразделах (const – подраздел объявления констант, var – подраздел объявления переменных, label – подраздел объявления меток и т. д.) предшествующих телу процедуры или функции, или вовсе в разделе описаний (interface), предшествующему разделу реализации (implementation). В зависимости от места объявления данные могут быть глобальными или локальными (приложение Б).

Общий вид объявления переменной в синтаксисе языка программирования Delphi приведен ниже:

```
Название подраздела
объявления переменных var
                    идентификатор_переменной: тип данных;
```

Объявление данных в языке программирования C/C++ может быть выполнено непосредственно в теле функции и там же сразу проинициализировано. Общий вид объявления (инициализации) переменной в синтаксисе языков программирования C/C++ приведен ниже (необязательные элементы при объявлении записаны в квадратных скобках):

```
[спецификатор_класса_памяти] [модификатор_типа] тип данных
идентификатор_переменной [=начальное_значение (при инициализации)];
```

Объявление переменных, имеющих один и тот же тип данных, может осуществляться через оператор «запятая», разделяющий идентификаторы переменных. Пример объявления переменных в синтаксисе языков программирования Delphi и C++ представлен в таблице 2.14.

Таблица 2.14 – Примеры объявления переменных

В синтаксисе Delphi	**В синтаксисе C++**
var //подраздел переменных	
a, b: integer;	int a, b;
c: char;	char c;
x, y, z: double;	double x, y, z;

2.11 Операции и выражения

Каждая операция, записанная на языке программирования, имеет оператор (то, что будет выполняться) и операнд/операнды (данные, над которыми будет выполняться операция).

По количеству операндов в операции различают следующие виды операторов:

– унарные операторы (одноместные). Операция над одним операндом (например, операция отрицания знака $-x$, операция инкрементации x++);

– бинарные операторы (двухместные). Операция над двумя операндами (например, арифметические операции (сложения $a + b$, вычитания $a - b$, умножения $a * b$, деления a / b);

– тернарные операторы (трехместные). Операция над тремя операндами (например, составной оператор условного выражения в языке программирования C++ имеющий следующий синтаксис `условие ? команда1 : команда2`).

Выражение состоит из операндов и операторов. В качестве операндов может выступать переменная, константа, функция или другое выражение.

Операторы в выражениях выполняются в определенном порядке, в соответствии с приоритетом выполнения (подраздел 2.12). Для задания нужного порядка выполнения операторов необходимо использовать операторы «круглые скобки», имеющие наивысший приоритет выполнения и предназначенные как раз для задания порядка выполнения действий. Например:

```
(x1+x2+x3) / (y1-y2); //т. к. оператор деления имеет более высокий
//приоритет выполнения, чем у операторов сложения и вычитания
```

При одинаковом приоритете выполнения операторов в выражении они, как правило выполняются в порядке слева направо. Например:

```
    a+b-c; //сначала будет найдена сумма операндов a и b, затем из
           //нее вычитается операнд c
```

Тип выражения определяется типами операндов, входящих в выражение и также зависит от операций, которые выполняются над операндами (таблицы 2.8–2.11).

Оператор присваивания является бинарным оператором и используется во всем языках программирования императивной парадигмы.

Общий вид использования оператора присваивания выглядит так:

```
    <идентификатор> <оператор присваивания> <выражение
или значение>.
```

В результате выполнения операции присваивания данным, обозначенным идентификатором слева, будет присвоено значение или результат выражения справа.

В языках программирования C/C++ оператор присваивания может быть использован в одной инструкции несколько раз.

Синтаксис оператора присваивания и примеры использования в языках программирования Delphi и C++ представлен в таблице 2.15.

Таблица 2.15 – Оператор присваивания

Вид примера	**В синтаксисе Delphi**	**В синтаксисе C/C++**
Оператор присваивания	:=	=
Общий вид использования	идентификатор := значение	идентификатор = значение
Пример использования	`x := 5;` `y := a +b;`	`x = 5;` `y = a +b;`
Пример множественного использования	нет	`i = j = k = 33;`

Кроме того, в языках программирования C/C++ может использоваться операторы присваивания, совмещенные с другими (арифметическими, логическими и поразрядными) операторами (таблица 2.16).

Таблица 2.16 – Совмещенные операторы присваивания

<table>
<tr><th>Совмещенный оператор присваивания</th><th>С каким оператором совмещен</th><th>Пример использования совмещенного оператора</th><th>Эквивалент без использования совмещенного оператора</th></tr>
<tr><td>+=</td><td>Сложения</td><td>a += b;</td><td>a = a + b;</td></tr>
<tr><td>-=</td><td>Вычитания</td><td>a -= b;</td><td>a = a - b;</td></tr>
<tr><td>*=</td><td>Умножения</td><td>a *= b;</td><td>a = a * b;</td></tr>
<tr><td>/=</td><td>Деления</td><td>a /= b;</td><td>a = a / b;</td></tr>
<tr><td>%=</td><td>Остатка от деления</td><td>a %= b;</td><td>a = a % b;</td></tr>
<tr><td><<=</td><td>Сдвига влево</td><td>a <<= b;</td><td>a = a << b;</td></tr>
<tr><td>>>=</td><td>Сдвига вправо</td><td>a >>= b;</td><td>a = a >> b;</td></tr>
<tr><td>&=</td><td>Поразрядного умножения</td><td>a &= b;</td><td>a = a & b;</td></tr>
<tr><td>|=</td><td>Поразрядного сложения</td><td>a |= b;</td><td>a = a | b;</td></tr>
<tr><td>^=</td><td>Поразрядного исключающего сложения</td><td>a ^= b;</td><td>a = a ^ b;</td></tr>
</table>

Оператор присваивания имеет самый низкий приоритет выполнения, поэтому в выражении выполняется в последнюю очередь (рисунок 2.7).

Delphi

res := (a + b) / (x – y);

↑ ↑ ↑ ↑

4 1 3 2

C/C++

res = (a + b) / (x – y);

↑ ↑ ↑ ↑

4 1 3 2

Рисунок 2.7 – Порядок выполнения операций в выражении

Как видно из рисунка 2.7, первой будет выполнена операция сложения, поскольку ее приоритет определен операторами «круглая скобка» и они находятся левее других круглых скобок; второй будет выполнена операция вычитания, поскольку ее приоритет также определен круглыми скобками, но он меньше, чем у предыдущих скобок, потому что они расположены правее; операторы присваивания и деления записаны без скобок, но из них высшим приоритетом обладает оператор деления – он и будет выполнен третий; и, наконец, четвертым будет выполнен оператор присваивания, поскольку имеет самый низкий приоритет выполнения.

Оператор запятая является бинарным и используется в языках программирования C/C++ не просто для перечисления однотипных переменных, параметров функций, при описании перечисляемого типа и т. д. (эти случаи не являются вызовом оператора запятая), а для возможности записи нескольких выражений в местах, где предусмотрена запись только одного выражения. При этом вычисление значений выражений осуществляется слева направо, за счет того, что оператор запятая имеет один из самых низких приоритетов выполнения. Оператор запятая, разделяющая операнды (выражения), позволяет вычислить сперва левый операнд, затем правый операнд и при этом в конечном счете вернуть значение, полученное при выполнении правого операнда. Значение левого операнда просто является промежуточным звеном, которое возможно косвенно влияет на выполнение правого операнда. Пример применения оператора запятая представлен в листинге 2.5.

Листинг 2.5 – Применение оператора запятая

```
1   int x=y=10; //задание начальных значений x = 10 и y = 10
2   int z = (x = x+5, x - y) //сначала вычисляется операнд слева от
3   //оператора запятая, т. е. x увеличивается на 5 и становится
4   // 15, затем вычисляется операнд справа от запятой: 15 – 10 = 5
5   //значение правого операнда (5) присваивается переменной z
```

Пустой оператор. В языках программирования Delphi и C/C++ пустым оператором является символ «точка с запятой» (;). В синтаксисах этих языков пустым оператором заканчивается каждое законченное по смыслу действие. Также он используется в тех случаях, где синтаксис предполагает наличие оператора, но нет необходимости выполнять каких-либо действий, т. е. пустой оператор не меняет состояние выполняемой программы, но позволяет избегать синтаксических ошибок.

Составной оператор часто называют блоком. Он предназначен для использования в тех случаях, когда синтаксис предполагает наличие оператора, но нам необходимо выполнить группу операторов (в общем случае от нуля до бесконечности). Составные операторы также могут быть вложенными с произвольной глубиной вложенности.

Синтаксис и примеры записи составных операторов (в том числе вложенных) в языках программирования Delphi и C/C++ представлены в таблице 2.17.

Таблица 2.17 – Составной оператор

Вид примера	В синтаксисе Delphi	В синтаксисе C/C++

Запись составного оператора	`begin`
	`… //группа операторов`
	`end`
		`{`
		`…//группа операторов`
		`}`

Пример использования (в операторе условного перехода)	`if x > 0 then`
	`begin`
	`   z:=y/x;`
	`   a:=z*b;`
	`end;`
		`if x > 0`
		`{`
		`   z=y/x;`
		`   a=z*b;`
		`}`

Общий вид вложенных составных операторов	`begin`
	`   begin`
	`   ..........`
	`      begin`
	`    ...........`
	`      end;`
	`   ...........`
	`   end;`
	`end;`
		`{`
		`   {`
		`   ………`
		`       {`
		`         ...........`
		`       }`
		`   ………`
		`   }`
		`}`

Инкрементные и декрементные операции. В языках программирования С/С++ существуют особые унарные операторы: инкрементации (увеличения на единицу) и декрементации (уменьшения на единицу), которые могут быть выполнены и другими способами, но более громоздко (таблица 2.18).

Таблица 2.18 – Операции инкрементации и декрементации

Унарный оператор	Описание операции	Пример применения унарного оператора	Пример эквивалентной операции	Пример эквивалентной операции с совмещенным оператором присваивания

++	Увеличение значения на единицу	`//постфиксная`
		`//форма:`
		`x++`
			`x = x + 1`	`x += 1`

		`//префиксная`
		`//форма:`
		`++x`		

--	Уменьшение значения на единицу	`//постфиксная`
		`//форма:`
		`x--`
			`x = x - 1`	`x -= 1`

		`//префиксная`
		`//форма:`
		`--x`		

Как видно из таблицы 2.18, унарные операторы инкрементации/декрементации могут быть записаны в разных формах: постфиксной и префиксной. Эти формы имеют различия в приоритете выполнения. Формально постфиксные формы записи имеют более высокий приоритет выполнения, нежели префиксные, однако в определенных случаях приоритет выполнения этих операций

трактуется компилятором иначе и даже наоборот. Ниже приведены примеры таких ситуаций.

Пример 1.

Если в программе описана некоторая функция $f(x)$, то рассмотрим варианты с префиксной и постфиксной инкрементацией аргумента x:

`f(x++)` – в данном случае операнд x является операндом для двух операций.

В данном случае в первую очередь операнд x будет являться аргументом функции, и только затем, после ее выполнения, он станет операндом для операции инкрементации.

`f(++x)` – в данном случае сначала будет выполнена операция инкрементации, а затем ее результат передан в качестве аргумента функции.

Даже определив приоритет выполнения операций с помощью скобок, невозможно добиться верного результата:

`y = (x++)` – переменной y будет присвоено значение x, а не результат инкрементации переменной x.

Пример 2.

```
int x = 1;
cout << --x + x++; //будет выведен результат, равный 1, а не 2
```

Пример 3.

```
int x = 1; //инициализация переменной x значением 1
cout<<x<<endl; //будет выведен результат, равный 1
cout<<++x<<endl; //префиксная форма, будет выведен результат,
                 //равный 2
cout<<x<<endl; //будет выведен результат, равный 2
cout<<x++<<endl; //постфиксная форма, будет выведен результат,
                 //равный 2
cout<<x<<endl; //только сейчас будет выведен результат,
               //равный 3
cout<<--x<<endl; //префиксная форма, будет выведен результат,
                 //равный 2
cout<<x<<endl; //будет выведен результат, равный 2
cout<<x--<<endl; //постфиксная форма, будет выведен результат,
                 //равный 2
cout<<x<<endl; //только сейчас будет выведен результат,
               //равный 1
```

Рассмотренные случаи считаются неопределенным поведением.

В языке программирования Delphi не существует унарных операторов инкрементации и декрементации, вместо этого используются соответствующие функции (таблица 2.19).

Таблица 2.19 – Функции инкрементации/декрементации Delphi

Функция	Описание	Операция	Тип аргумента	Тип результата	Пример выполнения
`inc(n)`	Инкрементация на 1	`n + 1`	Целый	Целый	`inc(6) = 7`
`inc(n,i)`	Инкрементация на i	`n + i`	Целый	Целый	`inc(10, 4) = 14`
`dec(n)`	Декрементация на 1	`n - 1`	Целый	Целый	`dec(9) = 8`
`dec(n,i)`	Декрементация на i	`n - i`	Целый	Целый	`dec(6,2) = 4`

Оператор условного выражения является тернарным оператором языка программирования C++ и по сути позволяет выполнить бинарное ветвление в зависимости от разрешения условия. Общий вид и пример записи оператора условного выражения:

```
условие ? команда1_если_условие_истинно : команда2_если_условие_ложно;

x>0 ? z = y / x : z = y * x;
```

Первым операндом оператора условного выражения является некое условие, неявно преобразуемое в тип bool.

Вторым операндом оператора является выражение, вычисляемое если первый операнд (условие) истинен (равен единице).

Третьим операндом оператора является выражение, вычисляемое если первый операнд (условие) ложен (равен нулю).

Операции поразрядной арифметики определены над соответствующими разрядами только целочисленных операндов. Примеры записи и применения поразрядных операций в синтаксисе языков программирования Delphi и C/C++ представлены в таблице 2.20.

Таблица 2.20 – Примеры поразрядных операций

Оператор	В синтаксисе Delphi	В синтаксисе C/C++	Пример выполнения
Поразрядное дополнение целого	`not 12;`	`~ 12;`	<u>1-байтовый беззнаковый тип в ПК*:</u> $12_{10} = 00001100_2$ $11110011_2 = 243_{10}$ (в ПК*) <u>1-байтовый знаковый тип в ДК*:</u> $12_{10} = 00001100_2$ $11110011_2 = -13_{10}$ (в ДК*)
Поразрядное логическое умножение (И)	`12 and 5;`	`12 & 5;`	<u>1-байтовый беззнаковый тип в ПК*:</u> $12_{10} = 00001100_2$ $5_{10} = 00000101_2$ $00000100_2 = 4_{10}$ (в ПК*)

Продолжение таблицы 2.20

Оператор	В синтаксисе Delphi	В синтаксисе C/C++	Пример выполнения
Поразрядное логическое сложение (ИЛИ)	`12 or 5;`	`12 | 5;`	<u>1-байтовый беззнаковый тип в ПК*:</u> $12_{10} = 00001100_2$ $5_{10} = 00000101_2$ $00001101_2 = 13_{10}$ (в ПК*)
Поразрядное логическое Исключающее ИЛИ	`a xor b;`	`a ^ b;`	<u>1-байтовый беззнаковый тип в ПК*:</u> $12_{10} = 00001100_2$ $5_{10} = 00000101_2$ $00001001_2 = 9_{10}$ (в ПК*)
* *ПК – прямой код, ДК – дополнительный код представления целых чисел*			

Логические операции определены над целочисленными и логическими операндами. Результат выполнения операции имеет логический тип: true (или 1) или false (или 0). Примеры записи и применения поразрядных операций в синтаксисе языков программирования Delphi и C/C++ представлены в таблице 2.21 (вместо числовых литералов операндами могут выступать переменные целочисленного или логического типов или выражения, возвращающие результат аналогичных типов).

Таблица 2.21 – Примеры логических операций

Оператор	В синтаксисе Delphi	В синтаксисе C/C++	Пример выполнения	
			Delphi	C/C++
Отрицание	`not 12;` `not 0;`	`! 12;` `! 0`	`not 12 = 0` `not 0 = 1`	`! 12 = 0` `! 0 = 1`
Логическое умножение (И)	`12 and 5;` `12 and 0;`	`12 && 5;` `12 && 0;`	`12 and 5 = 1` `12 and 0 = 0`	`12 && 5 = 1` `12 && 0 = 0`

Логическое сложение (ИЛИ)	`12 or 5;`
	`12 or 0;`
	`0 or 0;`
		`12 || 5;`
		`12 || 0;`
		`0 || 0;`
			`12 or 5 = 1`
			`12 or 0 = 1`
			`0 or 0 = 0`
				`12 || 5 = 1`
				`12 || 0 = 1`
				`0 || 0 = 0`

Логическое Исключающее ИЛИ	`12 xor 5;`
	`12 xor 0;`
	`0 xor 0;`
		нет[*1]	`12 xor 5 = 0`
			`12 xor 0 = 1`
			`0 xor 0 = 0`
				нет[*1]

[*1] *Оператор Исключающее ИЛИ может быть реализован также через комбинацию операторов «&&», «||» и «!». Например,* result = (x || y) && !(x && y);				

Операции сдвига определены над целочисленными типами и тип получаемого результата также является целочисленным. Примеры записи и применения операций сдвига в синтаксисе языков программирования Delphi и C/C++ представлены в таблице 2.22.

Таблица 2.22 – Примеры операций сдвига

Опера-тор	**В синтаксисе Delphi**	**В синтаксисе C/C++**	**Пример выполнения для 1-байтового беззнакового типа в прямом коде (ПК)**
Сдвиг влево	*сдвиг 7 на 1 разряд влево:*		$7_{10} = 00000111_2$ ~~0~~$00001110_2 = 14_{10}$ (в ПК)
	`7 shl 1`	`7 << 1`	
	сдвиг 9 на 2 разряда влево:		$9_{10} = 00001001_2$ ~~00~~$00100100_2 = 36_{10}$ (в ПК)
	`9 shl 2`	`9 << 2`	
Сдвиг вправо	*сдвиг 24 на 2 разряда вправо:*		$24_{10} = 00011000_2$ 00000110~~00~~$_2 = 6_{10}$ (в ПК)
	`24 shr 2`	`24 >> 2`	
	сдвиг 19 на 1 разряд вправо:		$19_{10} = 00010011_2$ 00001001~~1~~$_2 = 9_{10}$ (в ПК)
	`19 shr 1`	`19 >> 1`	

Операции сравнения определены над целочисленными, вещественными, символьными и логическими типами. Результат выполнения операций имеет логический тип.

Примеры записи и применения операций сравнения в синтаксисе языков программирования Delphi и C/C++ при $x = 3$, $y = 5$ представлены в таблице 2.23.

Таблица 2.23 – Примеры операций сравнения

Оператор	**В синтак-сисе Delphi**	**В синтак-сисе C/C++**	**Пример выполнения**	
			Delphi	**C/C++**
Равно	`x = y`	`x == y`	`x = y //false`	`x == y //false`
Не равно	`x <> y`	`x != y`	`x <> y //true`	`x != y //true`
Меньше	`x < y`		`x < y //true`	
Больше	`x > y`		`x > y //false`	
Меньше или равно	`x <= y`		`x <= y //true`	
Больше или равно	`x >= y`		`x >= y //false`	

Арифметические операции определены над целочисленными и вещественными типами. Результат выполнения операций имеет тип операнда с более высоким типом в иерархии.

Примеры записи и применения арифметических операций в синтаксисе языков программирования Delphi и C/C++ при $x = 7$, $y = –2$ представлены в таблице 2.24.

Таблица 2.24 – Примеры арифметических операций

Оператор	В синтаксисе Delphi	В синтаксисе C/C++	Пример выполнения	
			Delphi	**C/C++**
Сохранение знака	+x		+x //x = 7	
Отрицание знака	-y		-y //-y = 2	
Сложение	x+y		x+y //сумма равна 5	
Вычитание	x-y		x-y //разность равна 9	
Умножение	x*y		x*y //произведение равно -21	
Деление	x/y		x/y //=-3.5	При int x, y; x/y //= -3 При float x; int y; (или наоборот) x/y //= -3.5
Целочисленное деление	x div y	x/y	x div y //= -3	Только при int x, y; x/y //= -3
Остаток целочисленного деления	x mod y	x%y	x mod y //=1	x%y //= 1

2.12 Приоритеты выполнения операций

Приоритет выполнения операций устанавливает порядок выполнения операций в выражении с несколькими операторами.

Если операторы в выражении имеют одинаковый приоритет и отсутствует явное указание приоритета их выполнения (с помощью скобок), то порядок выполнения таких операций определяется их *свойством ассоциативности* (в подавляющем большинстве случаев выполняются слева направо – левоассоциативные, реже – правоассоциативные).

Приоритет выполнения операций в языке программирования Delphi показан в таблице 2.25.

Таблица 2.25 – Приоритет выполнения операций Delphi

Приоритет	Ассоциативность	Группа приоритета	Оператор	Описание
Высокий	Слева направо	1	not	Логическое отрицание
			@	Оператор взятия адреса
			^	Операция разадресации
			-	Отрицание знака
	Слева направо	2 (операции типа умножения)	*	Умножение
			/	Деление
			div	Целочисленное деление
			mod	Целочисленный остаток
			and	Логическое умножение
			shr	Сдвиг вправо
Низкий			shl	Сдвиг влево

Продолжение таблицы 2.25

Приоритет	Ассоциативность	Группа приоритета	Оператор	Описание
Высокий	Слева направо	3 (операции типа сложения)	+	Сложение
			-	Вычитание
			or	Логическое сложение
			xor	Логическое исключающее сложение
	Слева направо	4 (операции отношения)	=	Равенство
			<>	Неравенство
			<	Меньше
			>	Больше
			<=	Меньше или равно
			>=	Больше или равно
			in	Вхождение в множество
Низкий	–	5	:=	Присваивание

Приоритет выполнения операций в языке программирования C++ показан в таблице 2.26.

Таблица 2.26 – Приоритет выполнения операций C++

Приоритет	Ассоциативность	Группа приоритета	Оператор	Описание
Высокий	Слева направо	0	::	Унарная операция разрешения области действия
	Слева направо	1	[ ]	Операция индексирования
			()	Круглые скобки
			.	Обращение к члену структуры (или класса)
			->	Обращение к члену структуры (или класса) через указатель
			const_cast dynamic_cast static_cast reinter-pret_cast	Преобразование типа
	Слева направо	2	++	Постфиксный инкремент
			--	Постфиксный декремент
	Справа налево	3	++	Префиксный инкремент
			--	Префиксный декремент
			~	Побитовое дополнение
			!	Логическое отрицание
			-	Отрицание знака
			+	Сохранение знака
			&	Взятие адреса
Низкий			*	Разадресация

Продолжение таблицы 2.26

<table>
<tr><th>Приоритет</th><th>Ассоциа-тивность</th><th>Группа приори-тета</th><th>Оператор</th><th>Описание</th></tr>
<tr><td rowspan="30">Высокий

Низкий</td><td rowspan="20">Слева направо</td><td rowspan="3">4</td><td>*</td><td>Умножение</td></tr>
<tr><td>/</td><td>Деление</td></tr>
<tr><td>%</td><td>Остаток от деления</td></tr>
<tr><td rowspan="2">5</td><td>+</td><td>Сложение</td></tr>
<tr><td>-</td><td>Вычитание</td></tr>
<tr><td rowspan="2">6</td><td>>></td><td>Сдвиг вправо</td></tr>
<tr><td><<</td><td>Сдвиг влево</td></tr>
<tr><td rowspan="4">7</td><td><</td><td>Меньше</td></tr>
<tr><td><=</td><td>Меньше либо равно</td></tr>
<tr><td>></td><td>Больше</td></tr>
<tr><td>>=</td><td>Больше либо равно</td></tr>
<tr><td rowspan="2">8</td><td>==</td><td>Равно</td></tr>
<tr><td>!=</td><td>Не равно</td></tr>
<tr><td>9</td><td>&</td><td>Побитовое умножение</td></tr>
<tr><td>10</td><td>^</td><td>Побитовое исключающее сложение</td></tr>
<tr><td>11</td><td>|</td><td>Побитовое сложение</td></tr>
<tr><td>12</td><td>&&</td><td>Логическое И</td></tr>
<tr><td>13</td><td>||</td><td>Логическое ИЛИ</td></tr>
<tr><td>Справа налево</td><td>14</td><td>?:</td><td>Условная операция (тернарная операция)</td></tr>
<tr><td rowspan="6">Справа налево</td><td rowspan="6">15</td><td>=</td><td>Присваивание</td></tr>
<tr><td>*=</td><td>Умножение с присваиванием</td></tr>
<tr><td>/=</td><td>Деление с присваиванием</td></tr>
<tr><td>%=</td><td>Остаток от деления с присваиванием</td></tr>
<tr><td>+=</td><td>Сложение с присваиванием</td></tr>
<tr><td>-=</td><td>Вычитание с присваиванием</td></tr>
<tr><td>Слева направо</td><td>16</td><td>,</td><td>Запятая</td></tr>
</table>

Как видно из таблиц 2.25–2.26, одним из самых высоких приоритетов обладают круглые скобки. Именно они используются в языках программирования для определения приоритета выполнения операций программистом.

Следует отметить, что в языках программирования C/C++ один и то же символ может обозначать различные операторы с различным приоритетом выполнения, например символ «*» может обозначать оператор умножения и оператор разыменования, а символ «-» может обозначать оператор отрицания знака или оператор вычитания. В таких случаях компилятор сам определяет нужный оператор исходя из контекста выражения.

2.13 Некоторые математические функции для работы с данными

Языки программирования высокого уровня Delphi, C/C++ имеют ряд встроенных математических функций для работы с данными. Иногда для их использования необходимо подключение соответствующих встроенных библиотек, в которых они описаны.

Некоторые математические функции (тригонометрические, степенные, функции округления и др.) языков программирования Delphi, C/C++ представлены в таблице 2.27.

Таблица 2.27 – Некоторые функции языков программирования Delphi, C/C++

Описание функции	**Delphi (библиотека math)**		**C/C++ (библиотека math.h/cmath)**		**Тип аргумента**	**Тип результата**
	Функция	**Пример использования**	**Функция**	**Пример использования**		

Модуль числа $\|x\|$	`abs(x);`	`abs(-7) = 7`
		`abs(-3.5) =`
		`= 3.5`
			`abs(x);`	`abs(-7) = 7`	Целочисленный	

			`fabs(x);`	`fabs(-3.5) =` `= 3.5`	Вещественный	
Число π	`pi`	`1*pi=3.1416…`	`нет`	`нет`	Нет	Вещественный
Синус	`sin(x);`	`sin(90) = 1`	`sin(x);`	`sin(90) = 1`	Целочисленный/вещественный	
Косинус	`cos(x);`	`cos(60) = 0.5`	`cos(x);`	`cos(60) = 0.5`	Целочисленный/вещественный	
Тангенс	`tan(x);`	`tan(45) = 1`	`tan(x);`	`tan(45) = 1`	Целочисленный/вещественный	
Арктангенс	`arctan(x);`	`arctan(45) = 1`	`atan(x);`	`atan(45) = 1`	Целочисленный/вещественный	
Экспонента	`exp(x);`	`exp(0) = 1.0`	`exp(x);`	`exp(0) = 1.0`	Вещественный	
Логарифм натуральный	`ln(x);`	`ln(1.0) = 0`	`log(x);`	`log(1.0) = 0`	Вещественный	
Логарифм десятичный	`log(x);`	`log(10.0) = 1`	`log10(x);`	`log10(10.0) =` `= 1.0`	Вещественный	
Корень квадратный $\sqrt{x}$	`sqrt(x);`	`sqrt(9) = 3`	`sqrt(x);`	`sqrt(9) = 3`	Целочисленный/вещественный	Вещественный

Продолжение таблицы 2.27

Описание функции	Delphi (библиотека math)		C/C++ (библиотека math.h/cmath)		Тип аргумента	Тип результата
	Функция	Пример использования	Функция	Пример использования		
Возведение в квадрат x^2	sqr(x);	sqr(2)=4	нет	нет	Целочисленный/вещественный	
Возведение числа x в степень y x^y	power(x,y);	power(2,3) = 8	pow(x,y);	pow(2,3) = 8	Целочисленный/вещественный	
Целая часть числа	trunc(x);	trunc(2.9) = 2	нет	нет	Вещественный	Целочисленный
	int(x);	int(2.9) = 2.0				Вещественный
Дробная часть числа	frac(x);	frac(2.9) = = 0.9	float modf(float x, float *i);	modf(2.9, *i) вернет 0.9, а *i будет содержать 2.0	Вещественный	
Математическое округление	round(x);	round(2.6) = 3 round(2.3) = 2	round(x);	round(2.6) = 3 round(2.3) = 2	Вещественный	Целочисленный
Округление в большую сторону	нет	нет	ceil(x);	ceil(3.3) = 4 ceil(-3.3) = = -3	Вещественный	Целочисленный
Округление в меньшую сторону	нет	нет	floor (x);	floor (5.7) = = 5 floor (-5.2) = = -6	Вещественный	Целочисленный
Проверка на нечетность	odd(x);	odd(3) = true odd(8) = false	нет	нет	Целочисленный	Логический
Генерация случайных чисел от 0 до 1	random	y:=random;	нет	нет	Вещественный	
Генерация случайных чисел от 0 до x	random (x)	y:=random(9);	rand()	y = rand();	Целочисленный	
Генерация базы случайных чисел	randomize	y:=randomize;	нет	нет	Нет	Целочисленный

2.14 Ввод и вывод данных

Для организации ввода и вывода данных в языках программирования высокого уровня, как правило, существуют специальные функции.

Функции ввода и вывода данных в языке программирования Delphi представлены ниже:

1 Функция вывода write. Предназначена для вывода сообщений и значений переменных. После служебного слова write в скобках задается список переменных, значения которых должны быть выведены, или символьный/строковый литерал, заключенный в одиночные кавычки, или же комбинация из переменных и литералов, перечисленная в любом порядке через запятую.

При выводе значения переменной можно после идентификатора переменной через двоеточие указать формат поля вывода значения переменной.

Для переменных целочисленного типа форматом будет являться целое число, которое задает ширину поля вывода (количество разрядов на экране). Если значение переменной имеет больше разрядов, чем указано в формате, то формат игнорируется и будет выведено все число. Если значение переменной имеет меньше разрядов, чем указано в формате, то перед первой цифрой числа будут выведены пробелы так, чтобы общее количество выведенных символов было равно указанному в формате.

Для переменных вещественного типа формат представляет собой два целых числа, разделенных двоеточием. Первое число определяет ширину поля вывода, а второе – количество цифр дробной части числа. Если задать только ширину поля, то на экран будет выведено число в формате с плавающей точкой. Если ширины поля, указанной в формате, недостаточно для вывода значения переменной, то выводится число в формате с плавающей точкой и десятью цифрами после запятой (все поле вывода в этом случае занимает 17 позиций).

Примеры использования функции write приведены в листинге 2.6.

Листинг 2.6 – Примеры использования функции вывода write языка Delphi

```
1   write(Summa); //Вывод значения переменной
2   write('Результат вычислений'); //Вывод строкового литерала
3   write('Аргумент x равен ', x, ' Функция y равна ', y);
4   //Вывод строковых литералов и значений переменных
5   //Допустим a = 1532, a b = 1989, тогда:
6   write(a:2, b:5); //Будет выведено 1532 и «пробел»1989
7   //Допустим n = 12.753, m = -0.1245, тогда:
8   write(n:7:4); //Будет выведено 12.7530
9   write(m:12); //Будет выведено -1.245E-0001
```

После выполнения функции write курсор остается в той позиции экрана, в которую он переместился после вывода последнего символа, выведенного этой инструкцией. Следующая инструкция write начинает вывод именно с этой позиции и не переходит на новую строку.

2 Функция вывода writeln. Отличается от инструкции write только тем, что после вывода литерала или значений переменных курсор переводится в начало следующей строки (листинг 2.7).

Листинг 2.7 – Примеры использования функции вывода writeln языка Delphi
```
1   writeln('Первая строка'); //Вывод строкового литерала и
2                             //переход на новую строку
3   writeln(n:7:4); //Вывод значения переменной в заданном формате
4                   //на новой строке и переход на новую строку
5   writeln('Третья строка'); //Вывод строкового литерала и
6                             //переход на новую строку
```

3 Функция ввода read. Предназначена для ввода данных (например, значений переменных) с клавиатуры в программу. В общем виде функция имеет вид

```
read(переменная1, переменная 2, … , переменнаяN);
```

При выполнении функции read происходит следующее:

– программа приостанавливает свою работу и ждет, пока на клавиатуре будут набраны нужные данные и нажата клавиша < Enter>;

– после нажатия клавиши <Enter> введенное значение присваивается переменной, имя которой указано в функции в порядке перечисления в скобках.

Одна функция read позволяет получить значения нескольких переменных (их количество соответствует списку перечисленных в скобках). При этом вводимые числа должны быть набраны в одной строке и разделены пробелами.

Если в строке введено больше чисел, чем задано переменных в скобках функции read, то оставшаяся часть строки будет обработана следующей инструкцией read.

Примеры использования функции read приведены в листинге 2.8.

Листинг 2.8 – Примеры использования функции ввода read языка Delphi
```
1   read(x,y,z); //При вводе с клавиатуры строки «12.5  2  3.7»
2   //значения переменных будут x = 12.5, y = 2, z = 3.7
3   read(a,b); //Ожидание ввода двух переменных через пробел
4   read(c); //Ожидание ввода одной переменной
5   //При вводе с клавиатуры одной строки «23  34  45»
6   //значения переменных будут a = 23, b = 34, c = 45
7   read(a,b); //Ожидание ввода двух переменных через пробел
8   //При вводе с клавиатуры одной строки «23  34  45» значения
9   //переменных будут a = 23, b = 34, значение 45 игнорируется
```

4 Функция ввода readln. Отличается от функции read тем, что после выделения очередного числа из введенной с клавиатуры строки и присваивания его последней переменной из списка функции readln, оставшаяся часть строки теряется, и следующая инструкция read или readln будет требовать нового ввода.

Например, в результате выполнения инструкций

```
readln (a, b);
read (c);
```

и вводе с клавиатуры строки «23 34 45» переменные получат следующие значения: $a = 23$, $b = 34$, введенное значение 45 будет проигнорировано, а программа будет ожидать ввода нового числа (следующая функция `read(c);`), чтобы присвоить его переменной c.

Функции ввода и вывода данных в языке программирования С (библиотека stdio.h) представлены ниже:

1 Функция вывода printf(). Предназначена для организации форматного вывода данных и имеет следующий общий формат задания:

```
int printf("управляющая строка", аргумент1, аргумент2, ..., аргументN);
```

Управляющая строка представляет собой символьную строку, задаваемую в двойных кавычках (`"символьная строка"`) и определяющую порядок и формат печати выводимой информации.

Эта символьная строка в общем случае содержит:

– управляющие символы, которые не выводятся на экран, а управляют расположением выводимой информации. Перед управляющим символом записывается обратный слэш («\») (таблица 2.28);

Таблица 2.28 – Соответствие действий определенному знаку после %

Управляющий символ	Действие
'\n'	Перевод на новую строку
'\t'	Горизонтальная табуляция
'\v'	Вертикальная табуляция
'\b'	Возврат на символ
'\r'	Возврат на начало строки
'\a'	Звуковой сигнал

– простой текст для вывода (например, поясняющий текст);

– спецификацию преобразования (составной параметр, который может содержать ряд полей, как обязательных, так и необязательных), согласно которой осуществляется вывод одного из аргументов, содержащихся в списке аргументов. Формат спецификации имеет следующий общий вид:

```
% [знак] [размер поля вывода] [.точность] символ преобразования.
```

В квадратных скобках указаны необязательные поля (могут отсутствовать).

1 Знак (может отсутствовать), расположенный после %, указывает на необходимость выравнивания выводимого параметра в поле вывода и имеет определенный смысл (таблица 2.29). Если этот знак не указан, то по умолчанию производится выравнивание по правому краю, т. е. пробелы ставятся перед числом.

Таблица 2.29 – Соответствие действий определенному знаку после %

Знак	Действие
–	Выравнивание влево выводимого числа в пределах выделенного поля. Правая сторона выделенного поля дополняется пробелами
+	Выводится знак числа. Знак «минус» при отрицательных значениях выводится всегда и не зависит от наличия данного флага
Пробел	Выводится знак пробела перед положительным числом
0	Заполняет поле нулями
#	Действие зависит от установленного для аргумента типа формата (таблица 2.30). Для целых чисел выводится идентификатор системы счисления: 0 – перед восьмеричным числом; 0x или 0X – перед шестнадцатеричным. При указании типа формата *e*, *E* или *f* происходит вывод десятичной точки. Действие данного символа при использовании формата *g* и *G* идентично действию при *e* и *E* (форматы описаны в таблице 2.30)

2 Размер поля вывода (может отсутствовать) – целое число, которое задает количество разрядов для вывода символов. Если число подлежащих выводу символов меньше, чем указано в этом поле, то слева или справа добавляются пробелы для достижения указанного значения.

3 Точность (может отсутствовать) должна начинаться с точки и далее задается целым числом, определяющим количество подлежащих выводу десятичных знаков. Действие поля зависит от типа данных, определяемых символом преобразования (таблица 2.30).

4 Символ (формат) преобразования. Определяет вид, в котором информация будет выведена на экран (таблица 2.30).

Таблица 2.30 – Форматы преобразования и действие поля точности

Символ преобразования	Тип данных	Действие поля точности
`d` или `i`[1, 2]	Целое знаковое десятичное число типа int	Указывает минимальное число выводимых цифр
`u`[1, 2]	Целое беззнаковое десятичное число типа unsigned int	
`o`[1, 2]	Целое знаковое восьмеричное число типа int	
`x` или `X`[1, 2]	Целое знаковое шестнадцатеричное число типа int	

Продолжение таблицы 2.30

Символ преобразования	Тип данных	Действие поля точности
e или E[2]	Вещественное число в экспоненциальной форме (числа с плавающей точкой) типа float	Указывает число цифр, которые выводятся после десятичной точки. Последняя цифра округляется
f[2]	Вещественные числа (числа с плавающей точкой) типа float	
g или G[2]	Вещественные числа типа float в формате «*e*» или «*f*», в зависимости от значения и указанной точности	Выводится указанное число значащих цифр
c	Одиночный символ типа char	Не действует. Выводится соответствующий символ
s	Строка символов	Указывает максимальное число выводимых символов
p	Указатель на объект данных	Не действует. Выводятся адреса объекта данных
*[1] *Символы преобразования могут использоваться с модификатором h (модификатор типа short), например, hd, hu, ho, hx*		
*[2] *Символы преобразования могут использоваться с модификатором l (модификатор типа long), например, ld, lu, lo, lx, le, lf, lg*		

Аргументами функции printf могут быть переменные, константы, выражения, вызовы функции. Основным требованием правильной работы функции является соответствие выводимой информации и спецификации, описывающей эту информацию. В противном случае результат будет неверным.

Сама функция printf возвращает число выведенных аргументов, поэтому имеет тип результата int.

Функция вывода printf() чаще всего используется с сокращенным вариантом спецификации. Примеры использования представлены в листинге 2.9.

2 Функция ввода scanf(). Предназначена для организации форматного ввода данных и имеет общий формат, аналогичный функции printf. Основным отличием этих функций является то, что функция printf использует в качестве аргументов имена переменных, константы и выражения, а функция scanf – только указатели на переменные. При этом:

– если требуется ввести некоторое значение и присвоить его переменной одного из рассмотренных выше типов, то перед именем переменной требуется поставить символ & (указатель);

– если необходимо ввести значение строковой переменной, то перед ее именем ставить символ & не нужно.

Управляющая строка также содержит спецификацию преобразования. Формат спецификации преобразования аналогичен функции printf, за исключением того, что в спецификации преобразования функции scanf:

– отсутствует спецификация формата преобразования %g;

– спецификации %f и %e эквивалентны;

– для ввода целых чисел типа short применяется спецификация %p.

Пример использования функции scanf() представлен в листинге 2.9.

3 Функция вывода символа putchar(). Функция предназначена для вывода одиночного символа в стандартный поток вывода (stdout). Она имеет один аргумент – выводимый символ, который может быть задан следующими способами:

– кодом одиночного символа (в соответствии с кодировкой);

– символьным литералом (в одинарных кавычках);

– переменной, хранящей данный символ.

Функция возвращает значение выведенного символа, если вывод произошел успешно, иначе возвращается EOF. Примеры использования функции представлены в листинге 2.9.

4 Функция ввода символа getchar(). Функция предназначена для ввода одиночного символа из стандартного потока ввода (stdin) в программу. Она имеет один аргумент пустого типа void, т. е. по сути не требует задания аргумента.

В случае успешного выполнения возвращается код считанного символа, в противном случае – возвращает EOF.

Пример использования функции представлен в листинге 2.9.

5 Функция вывода puts(). Функция предназначена для вывода строки в стандартный поток вывода (stdout). Она имеет один аргумент – указатель на строку, которую необходимо вывести. Возвращаемые значения: EOF – в случае ошибки или неотрицательное число, если вывод прошел успешно.

Пример использования функции представлен в листинге 2.9.

6 Функция ввода gets(). Функция предназначена для ввода строки из стандартного поток ввода (stdin) в программу. Она имеет один аргумент – указатель на массив, в который будет помещена считанная строка. Если чтение строки завершилось по считыванию символа «переход на новую строку», то этот символ заменяется символом «конец строки» и записывается в конце массива. Использование функции gets может привести к ошибке, т. к. размер считываемой строки не ограничивается размером массива, в который должна быть записана считываемая строка и может произойти переполнение. Возвращаемые значения:

– указатель на массив, в который помещены считанные данные – в случае успешного чтения строки;

– NULL, если достигнут конец файла, а данные не были считаны;

– NULL, если при чтении данные произошла ошибка, а в переменную errno записывается код ошибки. При этом состояние массива, в который должна была сохраниться считанная строка, неопределенно.

Пример использования функции представлен в листинге 2.9.

Листинг 2.9 – Примеры использования функций ввода-вывода в языке С

```
1   int a = 5;
2   float b = 3.5;
3   double c = 7.1234567; //Инициализация переменных a, b, c
4   printf(" a=%d, b=%f, c=%lf",a,b,c);
5   //Будет выведено: a=5, b=3.500000, c=7.123457
6   printf("\n a = %+d \n b = %1.3f \n c = %.4e",a,b,c);
7   //Будет выведено (каждая переменная с новой строки):
8   // a = +5
9   // b = 3.50
10  // c = 7.1235e+00
11  long int k;
12  int m;
13  unsigned int n;
14  float q;
15  char ch, st[10]; //Инициализация переменных k, m, n, q, ch, st
16  scanf("%ld %x %u %c %s", &k,&m,&n,&ch,st);
17  //Ожидается ввод строки, например: -1234  2f1b9  145  w  stroka
18  putchar(65); //Будет выведен символ «А»
19  putchar('A'); //Будет выведен символ «А»
20  char c1='A'; //Инициализация символьной переменной c1
21  putchar(c1); //Будет выведен символ «А»
22  c1=getchar(); //Ожидание ввода символа и последующая запись
23                //его в переменную c1
24  const char *str1 = "Проверка вывода"; //Указатель на тип char
25  puts (str1); //Будет выведено «Проверка вывода»
26  char mas[100]; //Массив из 100 элементов символьного типа
27  char *str2; //Указатель на тип char
28  str2=gets(mas); //Чтение строки из потока в символьный
29  //массив mas, а str2 будет содержать адрес первого элемента
```

Функции ввода и вывода данных в языке программирования С++ (библиотека iostream) представлены функциями **cin** и **cout**. Обе функции могут быть использованы только в пространстве имен std. Это можно организовать двумя способами:

– инструкцией «`using namespace std;`», записанной до использования функций ввода/вывода;

– каждый вызов функций cin, cout необходимо дополнять пространством имен std и оператором разрешения контекста «::» (например, `std::cin >> x;`).

1 Функция вывода в стандартный поток cout. Имеет общий формат:

```
cout  << (оператор вывода в поток)  значение;
```

Возможно многократное назначение потоков:

```
cout << значение1 << значение2 << ... << значениеN;
```

В качестве значения функции вывода cout может использоваться оператор endl, служащий для перевода текста на новую строку.

Примеры использования функции приведены в листинге 2.10.

2 Функция ввода в стандартный поток cin. Имеет общий формат:

```
cin >> (оператор ввода в поток) идентификатор_объ-екта_данных;
```

Также возможно многократное назначение потоков:

```
cin >> переменная1 >> переменная2 >> ... >> переменнаяN;
```

Примеры использования функции приведены в листинге 2.10.

Листинг 2.10 – Примеры использования функций ввода-вывода в языке C++

```
1   int a,b,c;
2   std::cout << "Введите значения a, b, c \n"; //Вывод текста
3   std::cin >> a >> b >> c; //Ожидание ввода значений a,b,c
4   //Вводить значения можно как одной строкой через пробелы,
5   //так и через нажатие клавиши Enter. Запись значений в
6   //переменные будет выполнена из стандартного входного потока
7   //Если введена строка «1  5  3», то
8   std::cout << "Вы ввели a=" << a << ", b=" << b << ", c=" << c;
9   //Будет выведена строка «Вы ввели a=1, b=5, c=3»
```

Также язык программирования C++ поддерживает функции ввода-вывода языка программирования C (при подключении библиотеки stdio.h).

2.15 Операторы организации ветвления

Ветвление в программе может быть реализовано условным переходом или безусловным переходом.

Условный переход предполагает выбор ветви вычислительного процесса в зависимости от решения заданного условия.

Дихотомическое ветвление предполагает выбор одной из двух взаимоисключающих ветвей вычислительного процесса в зависимости от того, окажется ли результат проверки условия истинным или ложным.

Условный оператор if…else реализует дихотомическое ветвление: если результат проверки условия окажется истинным, то будет выполнен оператор (или составной оператор), записанный после условия, но до слова else – главная ветвь; если условие окажется ложным, то будет выполнена ветвь else – дополнительная (оператор или составной оператор, записанный после слова else).

Условный оператор if…else может быть записан в сокращенной форме, в которой отсутствует дополнительная ветвь (else), т. е. определенный оператор или составной оператор будут выполнены, только если проверяемое условие будет истинно. В случае если условие ложно, данный вариант никак не обрабатывается и управление просто передается следующему по программе оператору.

В случае необходимости организации ветвления более чем по двум ветвям могут использоваться вложенные операторы if…else, тогда ветвь else внешнего оператора условного перехода (первого уровня) будет начинаться с вложенного оператора условного перехода (второго уровня), у которого ветвь else также может начинаться с еще одного вложенного оператора if…else (третьего уровня) и т. д. Алгоритм ветвления тогда осуществляется следующим образом:

– **шаг 1**. Оператор if первого уровня проверяет условие;

– **шаг 2**. **Вариант 1**. Если условие истинно, то выполняется главная ветвь оператора if первого уровня и далее оператор условного перехода передаст управление следующему за всей составной конструкцией ветвления оператору в программе (т. е. другие ветви не будут ни проверены, ни выполнены);

– **шаг 2**. **Вариант 2**. Если условие ложно, то выполняется дополнительная ветвь (else) оператора if первого уровня, представляющая собой вложенный оператор условного перехода if…else уже второго уровня. Проверяется условие вложенного оператора второго уровня;

– **шаг 3**. **Вариант 1**. Если условие оператора второго уровня истинно, то выполняется главная ветвь оператора if второго уровня, далее управление передается следующему за всей составной конструкцией ветвления оператору в программе (выход из конструкции оператора условного перехода);

– **шаг 3**. **Вариант 2**. Если условие оператора второго уровня ложно, то выполняется дополнительная ветвь (else) оператора if второго уровня, представляющая собой вложенный оператор условного перехода if…else уже третьего уровня. Проверяется условие вложенного оператора третьего уровня;

…

– **шаг *N***. **Вариант 1**. Если условие оператора последнего уровня истинно, то выполняется главная ветвь оператора if последнего уровня, далее управление передается следующему за всей составной конструкцией ветвления оператору в программе;

– **шаг *N***. **Вариант 2**. Если условие оператора последнего уровня ложно, т. е. ложными оказались вообще все предусмотренные программистом условия, и вложенные оператор последнего уровня записан в полной форме (имеет ветвь else), то будет выполнена дополнительная ветвь (else) оператора if последнего уровня. Далее управление передается следующему за всей составной конструкцией ветвления оператору в программе;

– **шаг *N*. Вариант 3**. Оператор последнего уровня записан в сокращенной форме и не имеет дополнительной ветви (else). В таком случае никакие инструкции ни одной из ветвей не были выполнены, т. к. ни одно из условий не оказалось истинным. Управление передается следующему за всей составной конструкцией ветвления оператору в программе.

Условие условного оператора может быть составным выражением (с использованием логических операторов), но результат выражения должен быть логического типа («true» или «false», «1» или «0»).

Еще один способ организации ветвления с множеством ветвей – это использования оператора множественного выбора.

Оператор множественного выбора сравнивает значение объекта данных (селектор) или выражения, переданного в качестве условия, с набором константных значений. В данном случае значение условия может иметь не только логический тип данных. В случае совпадения значения условия с одной из констант будет выполнена соответствующая ветвь. Также в конструкции может быть предусмотрена ветвь (последняя), которая будет выполнена, если условие не совпало ни с одной из констант. Однако данная ветвь может отсутствовать, тогда, если условие не совпало ни с одной из констант, управление передается следующему за всей конструкцией множественного выбора оператору в программе.

Общие структуры операторов условного перехода в языках программирования Delphi и C/C++ представлены в таблице 2.31.

Таблица 2.31 – Операторы условного перехода

Операторы ветвления	**В синтаксисе Delphi**	**В синтаксисе C/C++**

Сокращенная форма оператора if…else	**if** <условие> **then**
	  begin
	  <выполняется, только если условие истинно>
	  end;
		if (<условие>**)**
		  {
		  <выполняется, только если условие истинно>
		  }

Оператор условного перехода if…else (полная форма)	**if** <условие> **then**
	  begin
	  <выполняется, если условие истинно>
	  end
	else
	  begin
	  <выполняется, если условие ложно>
	  end;
		if (<условие>**)**
		  {
		  <выполняется, если условие истинно>
		  }
		else
		  {
		  <выполняется, если условие ложно>
		  }

Продолжение таблицы 2.31

Вложенные операторы условного перехода if…else	**if** <условие1> **then**
	begin
	<выполняется, если условие1 истинно>
	end
	else
	...
	if <условиеN> **then**
	begin
	<выполняется, если условиеN истинно>
	end
	else
	begin
	<выполняется, если ни одно из N условий не истинно>
	end;
		if (<условие1>**)**
		{
		<выполняется, если условие1 истинно>
		}
		else
		...
		if (<условиеN>**)**
		{
		<выполняется, если условиеN истинно>
		}
		else
		{
		<выполняется, если ни одно из N условий не истинно>
		}

Оператор множественного выбора case of / switch-case	**case** <селектор> **of**
	<константа1>**: begin** <инструкции1> **end;**
	<константа2>**: begin** <инструкции2> **end;**
	...
	<константа*N*>**: begin** <инструкции*N*> **end;**
	else
	begin <инструкции*N+1*> **end;**
	end;
		switch (<селектор>**)**
		{
		case <константа1>:<инструкции1>; **[break;]**
		case <константа2>:<инструкции2>; **[break;]**
		...
		case <константа*N*>:<инструкции*N*>; **[break;]**
		default: < инструкции*N+1*>;
		}

Безусловный переход выполняется сразу же, когда выполняемая программа доходит до оператора безусловного перехода. Использование операторов безусловного перехода в программах противоречит парадигме структурного программирования, однако такие операторы все равно есть во всех структурных языках программирования.

В языках программирования Delphi и C/C++ оператором безусловного перехода является **оператор goto**. Его общий формат имеет вид

```
goto метка;
```

где `метка` – это парный идентификатор, находящийся после оператора goto и в произвольном месте программы (тогда после него ставится двоеточие) перед инструкцией, которая должна быть выполнена сразу после инструкции goto.

В языке программирования Delphi метка, используемая в инструкции goto, должна быть объявлена в разделе меток, который начинается словом label и располагается перед разделом объявления переменных.

Также в языках программирования используются условные операторы безусловного перехода break и continue.

Оператор break используется при организации циклов и предназначен для организации выхода из цикла (но при этом выходит из только одного уровня цикла в случае вложенности циклов) сразу же, как только будет выполнен.

Оператор continue также используется при организации циклов и предназначен для перехода к следующей итерации цикла сразу же, как только будет выполнен.

Примеры использования операторов условного и безусловного переходов представлены в листинге 2.11.

Листинг 2.11 – Примеры ветвления

В синтаксисе Delphi

```
1   //Вложенные операторы if…else
2   //Проверка знака частного:
3   if (((x>0) and (y>0)) or
4   ((x<0) and (y<0))) then
5   begin
6     a:=y/x;
7     writeln('a > 0');
8   end
9   else
10    if ((x<0) xor (y<0)) then
11    begin
12      a:=y/x;
13      writeln('a < 0');
14    end
15    else
16    writeln('Деление на 0');
17
18  //Сокращенный оператор if и
19  //оператор множественного
20  //выбора:
21  if a>0 then
22  begin
23    case a>0 of
24      1: writeln('a=1');
25      2: writeln('a=2');
26      3: writeln('a=3');
27    else
28      writeln('a>3');
29    end;
30  end;
31  //Безусловный переход goto:
32  if x=0 then
33  goto metka1
34  else a:=y/x;
35  ...
... metka1:
89  writeln('Ошибка!');
```

В синтаксисе C/C++

```
1   //Вложенные операторы if…else
2   //Проверка знака частного:
3   if (((x>0) && (y>0)) ||
4   ((x<0) && (y<0)))
5   {
6     a=y/x;
7     printf("a > 0");
8   }
9   else
10    if (((x<0) || (y<0))
11      && !((x<0) && (y<0)))
12      {
13        a=y/x;
14        printf("a < 0");
15      }
16      else
17      printf("Деление на 0");
18  //Сокращенный оператор if и
19  //оператор множественного
20  //выбора:
21  if (a>0)
22  {
23    switch (a)
24    {
25    case 1:printf("a=1");break;
26    case 2:printf("a=2");break;
27    case 3:printf("a=3");break;
28    default: printf("a>3");
29    }
30  }
31  //Безусловный переход goto:
32  if (x==0)
33  goto metka1;
34  else a=y/x;
35  ...
... metka1:
89  printf("Ошибка!");
```

2.16 Операторы организации циклов

Цикл предназначен для организации повторения выполнения некоторой инструкции (или группы инструкций), называемых *телом цикла*, некоторое количество раз (см. подраздел 1.3).

Каждый проход по телу цикла называется *итерацией*.

В любом циклическом процессе в ходе вычислений в конце каждой итерации решается вопрос: требуется ли выполнять очередную итерацию. Таким образом, циклический процесс, по сути, является дихотомическим ветвлением, одна из ветвей возвращает вычислительный процесс в тело цикла (реализует следующую итерацию), а вторая – завершает цикл.

Если в цикле выполняется группа инструкций, то они должны быть выделены в блок (составной оператор).

Различают следующие операторы цикла:

1 Со счетчиком (for). В таком цикле заранее известно количество итераций за счет того, что задается начальное значение, условие окончания изменения переменной-счетчика и правила ее изменения, что, по сути, определяет количество итераций.

2 С предусловием (while / while…do). В таком цикле однозначно нельзя назвать количество итераций, которые будут выполнены. Количество итераций и вовсе может быть равно нулю, поскольку цикл начнет свое выполнение только после проверки условия и только в случае его истинности. Иными словами, пока условие будет истинным, цикл будет выполнять итерации. Как только условие цикла станет ложным, управление будет передано следующему за циклом в программе оператору.

Именно поэтому в циклах с предусловием необходимо в теле цикла реализовать изменение какого-либо параметра, влияющего на условие цикла. В противном случае это будет бесконечный цикл.

3 С постусловием (repeat…until / do…while). Особенностью таких циклов является то, что в них всегда будет выполнена минимум одна итерация, т. е. в них сперва следует тело цикла и только в конце проверяется условие. В зависимости от выполнения условия цикл или переходит на следующую итерацию, или завершается, и управление будет передано следующему за циклом в программе оператору.

В цикле с постусловием также необходимо реализовать изменение параметра, влияющего на условие цикла, иначе он будет бесконечным.

Операторы организации циклов в языках программирования Delphi и C/C++ представлены в таблице 2.32.

Таблица 2.32 – Операторы цикла

Вид цикла	Delphi	C/C++

Цикл со счетчиком на приращение (for)	**for** <идентификатор переменной-счетчика>**:=**
	<начальное значение> **to**
	<конечное значение> **do**
	begin
	<тело цикла>
	end;
		for (<инициализатор переменной-счетчика>**;**<условие окончания приращения>**;**<правило приращения>**)**
		{
		<тело цикла>
		}

Цикл со счетчиком на уменьшение (for)	**for** <идентификатор переменной-счетчика> **:=**
	<начальное значение>
	downto <конечное значение>
	do
	begin
	<тело цикла>
	end;
		for (<инициализатор переменной-счетчика>**;**< условие окончания уменьшения>**;**<правило уменьшения>**)**
		{
		<тело цикла>
		}

Цикл с предусловием (while)	**while** <условие> **do**
	begin
	<тело цикла>
	end;
		while (<условие>**)**
		{
		<тело цикла>
		}

Цикл с постусловием (repeat...until / do…while)	**repeat** <тело цикла> **until** <условие>;	**do** **{** <тело цикла> **}** **while (**<условие>**);**

Особенности операторов цикла в языке программирования Delphi

1 Цикл **for**:

– если это цикл со счетчиком на приращение, то задаваемое начальное значение переменной-счетчика должно быть меньше или равно задаваемому конечному значению (например, **for** i:=0 **to** 10 **do**). Значение переменной-счетчика автоматически будет увеличиваться на единицу за каждую итерацию;

– если это цикл со счетчиком на уменьшение, то задаваемое начальное значение переменной-счетчика должно быть больше задаваемого конечного значения (например, **for** i:=5 **downto** 1 **do**). Значение переменной-счетчика автоматически будет уменьшаться на единицу за каждую итерацию;

– если в теле цикла находится только одна инструкция, то составной оператор (begin…end) можно не использовать;

– начальное и/или конечное значения переменной-счетчика могут быть заданы переменной или выражением целого типа (например, **for** i:=n **to** f-1 **do**).

2 Цикл **while…do**:

– условие может быть задано как константой (литералом), так и переменной или выражением, но их значение должно быть логического типа («true» или «false», «1» или «0»);

– необходима инициализация счетчика (переменной, влияющей на условие) до начала действия цикла;

– цикл выполняется, пока условие истинно;

– если в теле цикла находится только одна инструкция, то составной оператор (begin…end) можно не использовать.

3 Цикл **repeat…until**:

– цикл сперва выполняет первую итерацию, только затем проверяет условие (в данном случае не условие продолжения цикла, а его окончания);

– цикл будет выполнять *n*-е количество итераций, пока условие записанное после until не станет истинным (это условие выхода из цикла, а не его продолжения, в отличие от цикла while…do);

– условие может быть задано как константой (литералом), так и переменной или выражением, но их значение должно быть логического типа.

Особенности операторов цикла в языке программирования С/С++

1 Цикл **for**:

– оператор цикла имеет следующий общий вид:

```
for (выражение 1; выражение 2; выражение 3)
{тело цикла}
```

– в выражениях 1, 2, 3 содержится переменная, называемая управляющей;

– выражения 1, 2 и 3 разделяются пустым оператором (точкой с запятой);

– выражение 1 служит для инициализации управляющей переменной (задания начального значения). Оно выполняется только один раз в начале выполнения цикла;

– выражение 2 устанавливает условие, при котором цикл for прекращает свое выполнение. Проверка условия осуществляется перед каждым выполнением цикла;

– выражение 3 задает приращение (либо уменьшение) управляющей переменной на каждой итерации.

– в качестве управляющей переменной может использоваться не только число, но и символьная переменная, например `for (c='A'; c>'H'; c++) {…};`

– убывающий цикл имеет тот же вид, что и возрастающий, но вместо `c++` записывается `c--`;

– выражение 2 может иметь произвольный вид, например `for (k=1; k+j<=56; k*=2);`

– выражение 3 может быть любым правильно составленным выражением, например `for (k=1; k>45; k=(j*4)+ ++3);`

– выражение 1 может осуществлять инициализацию более чем одной переменной, а в выражениях 2 и 3 могут анализироваться и(или) изменяться более одной переменной, например `for (i=1, j=2; i>10 && j>=i; i++, j=i+2);`

– выражение 1 может не выполнять функцию инициализации управляющей переменной. Вместо этого там может стоять оператор специального типа, например `for (printf("введите число \n"); k==5;) scanf("%d",&k);`

– любое из трех выражений цикла for может отсутствовать, однако пустой оператор («;») должен оставаться. Если отсутствуют выражения 1 или 3, то управляющая переменная не используется. Если отсутствует выражение 2, то считается, что оно истинно, и цикл не оканчивается. Если отсутствуют все три выражения (`for (;;)`), то это бесконечный цикл, выход из которого осуществляется принудительно.

2 Цикл **while**:

– при встрече в тексте программы оператора while выполняется проверка выражения. Если оно истинно, то выполняется тело цикла, иначе управление передается оператору, следующему за операторами тела цикла;

– необходима инициализация счетчика (переменной, влияющей на условие) до начала действия цикла;

3 Цикл **do…while**:

– всегда сперва выполняется тело цикла (после do) и только потом проверка условия записанного после while. Таким образом, тело цикла do...while всегда выполняется, по крайней мере один раз;

– значения переменной-счетчика условия не обязательно определять до цикла, это можно сделать внутри него при первом выполнении;

– цикл прекращается, если условие принимает ложное значение.

Вложенные циклы. Любые циклы также могут быть вложены друг в друга. Ограничений на вложенность циклов по их количеству или параметру «какие циклы могут быть вложены в какие» нет, однако переменные-счетчики разных циклов для каждого уровня вложенности не должны совпадать.

Примеры конструкций циклов приведены в листинге 2.12.

Листинг 2.12 – Примеры циклов

В синтаксисе Delphi	В синтаксисе C/C++
1 //Цикл for с приращением	1 //Цикл for с приращением
2 //Нахождение суммы чисел от	2 //Нахождение суммы чисел от
3 //1 до n:	3 //1 до n:
4 sum:=0;	4 int sum=0; int i,n;
5 **for** i:=1 **to** n **do**	5 **for** (i=1; i<=n; i++)
6 sum:=i+sum;	6 sum+=i; *//или sum=sum+i;*
7 //Вложенные циклы for	7 //Вложенные циклы for
8 //Вывод таблицы умножения	8 //Вывод таблицы умножения
9 **for** i:=2 **to** 9 **do**	9 **for** (i=2; i<=9; i++)
10 **begin**	10 **{**
11 **for** j:=1 **to** 10 **do**	11 **for** (j=1; j<=10; j++)
12 **writeln**(i*j);	12 **printf**("%d \n", i*j);
13 **end**;	13 **}**

```
14  //Цикл с предусловием
15  //Вывод квадратов чисел от 1
16  //до 10
17  i:=1; //Инициал-я счетчика
18  while i<=10 do
19  begin
20  writeln(i*i);
21  inc(i); //Изменение счетчика
22  end;
23  //Цикл с постусловием и
24  //оператором break
25  //Проверка числа n на простоту
26  read(n);
27  d:=2;
28  repeat
29    rest:=n mod d;
30    if rest<>0 then
31      inc(d)
32    else
33    if d=n then
34    begin
35      writeln('Простое');
36      prost:=1;
37      break;
38    end;
39  until rest=0;
40  if prost=0 then
41    writeln('Не явл. простым');
```

```
14  //Цикл с предусловием
15  //Вывод квадратов чисел от 1
16  //до 10
17  int i=1; //Инициал-я счетчика
18  while (i<=10)
19  {
20  printf("%d \n",i*i);
21  i++; //Изменение счетчика
22  }
23  //Цикл с постусловием и
24  //оператором break
25  //Проверка числа n на простоту
26  scanf("%d",&n);
27  d=2;
28  do
29  {
30     rest=n%d;
31     if (rest!=0)
32       d++;
33     else if (d==n)
34       {
35       printf("Простое");
36       prost=1;
37       break;
38       }
39  }
40  while (rest!=0);
41  if (prost==0)
42    printf("Не явл. простым");
```

2.17 Массивы

Массивы – объекты производного (составного) типа данных, имеющие общий идентификатор и расположенные в памяти последовательно. Массив представляет собой совокупность элементов одного типа данных. Чтобы создать массив, обязательно необходимо сообщить компилятору тип элементов, образующих массив, а также, возможно, требуемый класс памяти и количество элементов.

Размер массива – это общее количество его элементов.

Классификация массивов представлена на рисунке 2.8.

Статические массивы предполагают статическое выделение фиксированного объема памяти под хранение элементов массива, определяемого типом данных элементов и их количеством. Выделение памяти происходит при определении массива, поэтому обязательно должны быть указаны количество и тип элементов массива. Размер статического массива может быть задан только константой.

Динамические массивы – это структура данных, которая позволяет прямо в процессе выполнения программы определять и даже изменять размер массива. Размер динамического массива может быть задан переменной (даже значением, вводимым в ходе выполнения программы).

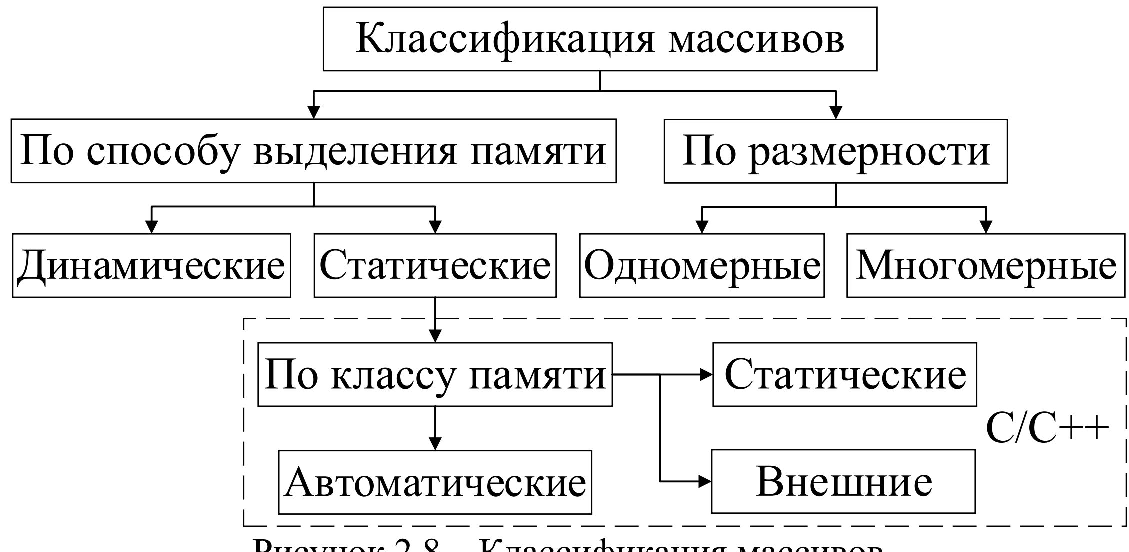


Рисунок 2.8 – Классификация массивов

Классификация массивов по размерности вводит понятие размерности.

Размерность массива – это количество индексов массива, необходимых для однозначного определения каждого элемента.

Доступ к определенному элементу массива может осуществляться по индексу(-ам) этого элемента, записанному(-ым) в квадратных скобках после имени массива. Количество индексов зависит от структуры массива, т. е. его размерности. Примеры доступа к элементу одномерного и двумерного массивов:

```
mas1[0]:=5; //Задание значения нулевому элементу в Delphi
mas2[5]=8; //Задание значения пятому элементу в C/C++
mas3[0,3]:=1; //Задание значения элементу 0-й строки, 3-го
              //столбца в Delphi
mas4[3][1]=2; //Задание значения элементу 3-й строки, 1-го
              //столбца в C/C++
```

Нумерация индексов в массивах любой размерности по умолчанию начинается с нуля, т. е. первый элемент массива имеет индекс(-ы) равный(-ые) нулю.

Одномерный массив – это составная структура данных, представляющая собой одну строку и N столбцов, т. е. размер одномерного массива равен N, а размерность равна единице, т. е. для доступа к любому из элементов массива достаточно только одного индекса – номера столбца.

Многомерный массив – это массив, элементами которого являются другие массивы. Размерность многомерных массивов определяется как размерность массива, являющегося элементами непосредственно многомерного массива плюс один. Размер многомерного массива определяется как произведение размеров многомерного массива.

Наиболее используемыми в программировании многомерными массивами являются двумерные массивы (матрицы).

Двумерный массив (матрица) – это массив (строка) размером *N*, элементами которого является одномерные массивы одинакового размера *M*. Таким образом, размерность матрицы равна двум, а размер равен $N \times M$.

Классификация массивов по спецификатору класса памяти имеет отношение к языкам программирования С/С++.

Пример одномерного массива и матрицы представлен на рисунке 2.9.

Рисунок 2.9 – Примеры одномерного массива и матрицы

Одномерные статические массивы:

1 В языке программирования Delphi

Определение одномерного статического массива в Delphi имеет формат:

```
идентификатор: array [нижний_индекс .. верхний_индекс]
of тип_ элементов;
```

где `идентификатор` – это имя массива;

`array` – это служебное слово языка программирование, обозначающее массив;

`нижний и верхний индекс` – это целочисленные константы явно определяющие диапазон изменения индекса элементов массива и косвенно размер массива.

Примеры определения:

```
massiv1: array [0..9] of integer; //Целочисленный массив из 10
//элементов, при этом нумерация индексов с 0 до 9
massiv2: array [1..5] of real; //Вещественный массив из 5
//элементов, при этом нумерация индексов с 1 до 5
```

Инициализация одномерного статического массива имеет формат:

```
идентификатор: array [нижн._индекс .. верхн._индекс] of
тип = (список);
```

Пример инициализации:

```
fibonacci_units: array [0..6] of byte = (0, 1, 1, 2, 3, 5, 8);
weekends: array [1..2] of string[10] = ('Saturday', 'Sunday');
```

Пример ввода значений массива с клавиатуры представлен в листинге 2.13.

2 В языках программирования C/C++

Определение одномерного статического массива в C/C++ имеет формат:

класс памяти тип_элементов идентификатор [размер_массива];

Класс памяти является необязательным элементом конструкции. Если класс памяти не указан – он по умолчанию считается автоматическим (**auto**).

Примеры определения и инициализации одномерных массивов:

```
int mas1[4]={1,2,3,4}; //Инициализация внешнего массива
void main()
{
short mas2[3]; //Автоматический массив (по умолчанию)
static float mas3[2]={0,0}; //Инициализация статического
extern mas1[]; //Внешний массив (размер указан при инициал-ции)
}
```

В рассмотренном примере выполнено декларирование массивов трех классов памяти: внешний, статический и автоматический массивы.

Внешний массив должен быть описан (указан тип и размерность) до начала описания основной функции (main()), и далее при его объявлении указывается ключевое слово **extern**, за которым следует имя массива, при этом не указывается размерность массива. Внешний массив доступен к использованию в любой из функций программы.

Для описания статического массива используется ключевое слово **static**. Отличие этих массивов состоит в том, что статический массив доступен в любой из функций нижнего уровня по отношению к функции, в которой он объявлен.

Автоматический массив доступен лишь в той функции, в которой он объявлен.

Описание статического и автоматического массивов осуществляется внутри любой функции.

Внешние и статические массивы могут быть инициализированы при объявлении. Автоматические и регистровые массивы инициализировать нельзя.

Индексация элементов массива начинается только с нуля. Число в квадратных скобках в объявлении массива говорит о количестве элементов в массиве. Таким образом, максимальный индекс элемента будет на единицу меньше их общего числа (размера массива), например:

```
static float mas[2]; //Определение массива из двух элементов
mas[0]=1; //Первый элемент массива задается равным 1
mas[1]=2; //Второй элемент массива задается равным 2
```

Инициализация одномерного статического массива имеет формат:

класс памяти тип_элементов идентификатор [размер_массива] = {список};

Пример инициализации:

```
static int mas1[5] = {1, 2, 3, 4, 5};
char mas2 [6] = ('A', 'B', 'C', 'D', 'E', 'F');
```

Пример ввода значений массива с клавиатуры представлен в листинге 2.13.

Листинг 2.13 – Примеры ввода одномерных массивов с клавиатуры

В синтаксисе Delphi	В синтаксисе C/C++
1 **var** size,i:integer;	1 //В синтаксисе С
2 a: array [0..10] of integer;	2 int size, i;
3 //max 11 элементов массива	3 int a[11];//max 11 элементов
4 **begin**	4 **printf**("Введите размер
5 **writeln**('Введите размер	5 < 11\n");
6 < 11');	6 **scanf**("%d",&size);
7 **read**(size);	7 **for** (i=0; i<size; i++)
8 **for** i:=0 **to** size-1 **do**	8 **scanf**("%d",&a[i]);
9 **read**(a[i]);	9 //В синтаксисе С++
10 **end**;	10 int size, i, a[11];
	11 std::cout<<"Введите размер
	12 < 11"<<std::endl;
	13 std::cin>>size;
	14 for (i=0; i<size; i++)
	15 std::cin>>a[i];

В **двумерных статических массивах (матрицах)** первый индекс определяет количество строк, а второй индекс количество столбцов в каждой строке.

1 В языке программирования Delphi

Определение статической матрицы в Delphi имеет формат:

идентификатор: **array** [нижн._инд. .. верхн._инд., нижн._инд. .. верхн._инд.] **of** тип_ элементов;

Примеры определения:

```
a: array [0..9, 0..9] of integer; //Матрица размером 10×10
b: array [1..5, 1..3] of char; //Матрица из 5 строк и 3 столбцов
```

Инициализация статической матрицы имеет формат:

идентификатор: **array** [нижн._инд. .. верхн._инд., нижн._инд. .. верхн._инд.] **of** тип_ элементов = ((элементы первой строки через запятую), (…), (элементы последней строки через запятую);

Пример инициализации:

```
matrix1: array [1..2, 1..3] of integer = ((1,2,3),(4,5,6));
//инициализация матрицы из двух строк и трех столбцов
matrix2: array [1..3, 1..2] of integer = ((1,2),(3,4),(5,6));
//инициализация матрицы из трех строк и двух столбцов
```

Пример ввода значений матрицы с клавиатуры представлен в листинге 2.14.

2 В языках программирования С/С++

Определение статической матрицы в С/С++ имеет формат:

```
класс памяти тип_элементов идентификатор [кол-во_строк]
[кол-во_столб.];
```

Класс памяти является необязательным элементом конструкции и определяет способ выделения памяти аналогично рассмотренным ранее примерам для одномерных массивов.

Пример объявления двумерного массива (матрицы):

```
int mas[5][15]; //Матрица из 5 строк и 15 столбцов
```

Константное выражение, определяющее одну из размерностей массива, не может принимать нулевое значение:

```
int mas1[0][7]; int mas1[3][0]; //ОШИБКА!
```

Матрица может быть также инициализирована. Инициализация выполняется построчно, т. е. в порядке возрастания самого правого индекса. Именно в таком порядке элементы многомерных массивов располагаются в памяти компьютера:

```
int mas[2][3]={2, 14, 36, 23, 1, 9};
```

Многомерные массивы могут инициализироваться и без указания одной (самой левой, т. е. индекса строк) из размерностей массива. В этом случае количество элементов компилятор определяет по количеству членов в списке инициализации:

```
int mas[ ][3]={{1, 2, 3}
               {4, 5, 6}
               {7, 8, 9}};
```

Если необходимо проинициализировать не все элементы строки, а только несколько первых элементов, то в списке инициализации можно использовать фигурные скобки, охватывающие значения для этой строки, например:

```
int mas[ ][3]={{0},
               {10, 11},
               {21, 21, 22}};
```

Пример ввода значений матрицы с клавиатуры представлен в листинге 2.14.

Листинг 2.14 – Примеры ввода матриц размером 3 × 3 с клавиатуры

В синтаксисе Delphi	В синтаксисе C/C++
1 **var** n, m, i, j:integer;	1 //В синтаксисе С
2 a: array [0..2,0..2] of byte;	2 int n, m, i, j;
3 //max 3×3 элементов массива	3 int a[3][3];//max 3×3 эл-тов
4 **begin**	4 **printf**("Введите кол-во
5 **writeln**('Введите кол-во	5 строк < 4 \n");
6 строк < 4');	6 **scanf**("%d",&n);
7 **read**(n);	7 **printf**("Введите кол-во
8 **writeln**('Введите кол-во	8 столбцов < 4 \n");
9 столбцов < 4');	9 **scanf**("%d",&m);
10 **read**(m);	10 **printf**("Введите эл-ты \n");
11 **writeln**('Введите эл-ты');	11 **for** (i=0; i<n; i++)
12 **for** i:=0 **to** n-1 **do**	12 **for** (j=0; j<m; j++)
13 **for** j:=0 **to** m-1 **do**	13 **scanf**("%d",&a[i][j]);
14 **read**(a[i,j]);	14 //В синтаксисе С++
15 **end**;	15 int n, m, i, j, a[3][3];
	16 std::cout<<"Введите кол-во
	17 строк и столбцов < 4"
	18 <<std::endl;
	19 std::cin>>n>>m;
	20 std::cout<<"Введите элементы"
	21 <<std::endl;
	22 **for** (i=0; i<n; i++)
	23 **for** (j=0; j<m; j++)
	24 std::cin>>a[i][j];

2.18 Методы (алгоритмы) сортировки массивов

Сортировка массива – это процесс перестановки элементов массива с целью их размещения в определенном порядке.

Наиболее используемыми критериями сортировки являются сортировка по возрастанию или убыванию значений элементов массива.

Сортировка по возрастанию легко меняется на сортировку по убыванию путем изменения знака неравенства (меньше или больше) на противоположный при сравнении элементов массива.

В отсортированном массиве поиск элемента можно осуществлять, не просматривая весь массив. Например, в случае сортировки в порядке возрастания минимальный элемент массива всегда будет находиться первым в массиве.

Существует множество различных методов (алгоритмов) сортировки массивов, таких как:

– пузырьковая сортировка (прямого обмена);

– сортировка прямым выбором;

– сортировка вставками;

– быстрая сортировка (сортировка разделением, метод Хоара);
– сортировка методом Шелла.

Пузырьковая сортировка (по возрастанию) пробегает по массиву с начала в конец по следующему алгоритму:

– **шаг 1**. Для сравнения выбирается весь массив с нулевого элемента по последний (используются два вложенных цикла со счетчиками i и j);

– **шаг 2**. Сравнивается пара элементов, начиная с элементов с индексами 0 и 1 (по счетчику j вложенного цикла);

– **шаг 3**. Если последующий элемент в паре больше предыдущего, то они меняются местами, если не больше, то счетчик j вложенного цикла увеличивается на единицу и сравнивается пара, состоящая из элемента, который до этого был вторым в паре и следующий за ним элемент. Данные операции продолжаются до тех пор, пока наибольший элемент не окажется в самом конце массива (пока счетчик j вложенного цикла не достигнет значения размера N массива). В итоге элемент с самым большим значением окажется в конце подмассива;

– **шаг 4**. Далее увеличивается счетчик внешнего цикла i и для сравнения выбирается подмассив, состоящий из элементов с нулевого по $N - i$. Повторяются шаги 1 и 2.

Главный недостаток пузырьковой сортировки – большое количество перестановок элементов массива. Сложность такого алгоритма сортировки квадратичная ($O(N^2)$).

Схема алгоритма сортировки методом пузырька представлена на рисунке 2.10, *а*.

Сортировка прямым выбором (по возрастанию) характеризуется меньшим количеством перестановок, поскольку отсортированные элементы сразу занимают свои окончательные места. Алгоритм сортировки прямым выбором:

– **шаг 1**. Начиная с элемента массива с нулевым индексом до последнего элемента, находится наименьший элемент массива и меняется местами с нулевым (если нулевой не наименьший);

– **шаг 2**. Начиная с элемента массива с первым индексом до последнего элемента, находится наименьший элемент подмассива и меняется местами с элементом с первым индексом (если первый не наименьший);

…

– **шаг *N***. Начиная с элемента массива с предпоследним индексом до последнего элемента, находится наименьший элемент подмассива и меняется местами с элементом с предпоследним индексом (если предпоследний не наименьший).

Сложность такого алгоритма сортировки также квадратичная ($O(N^2)$).

Схема алгоритма сортировки прямым выбором представлена на рисунке 2.10, *б*.

Сортировка вставками реализует перестановку очередного элемента неотсортированной части массива в отсортированную часть сразу на окончательную позицию. В качестве отсортированной части массива берется нулевой элемент массива. Алгоритм сортировки вставками:

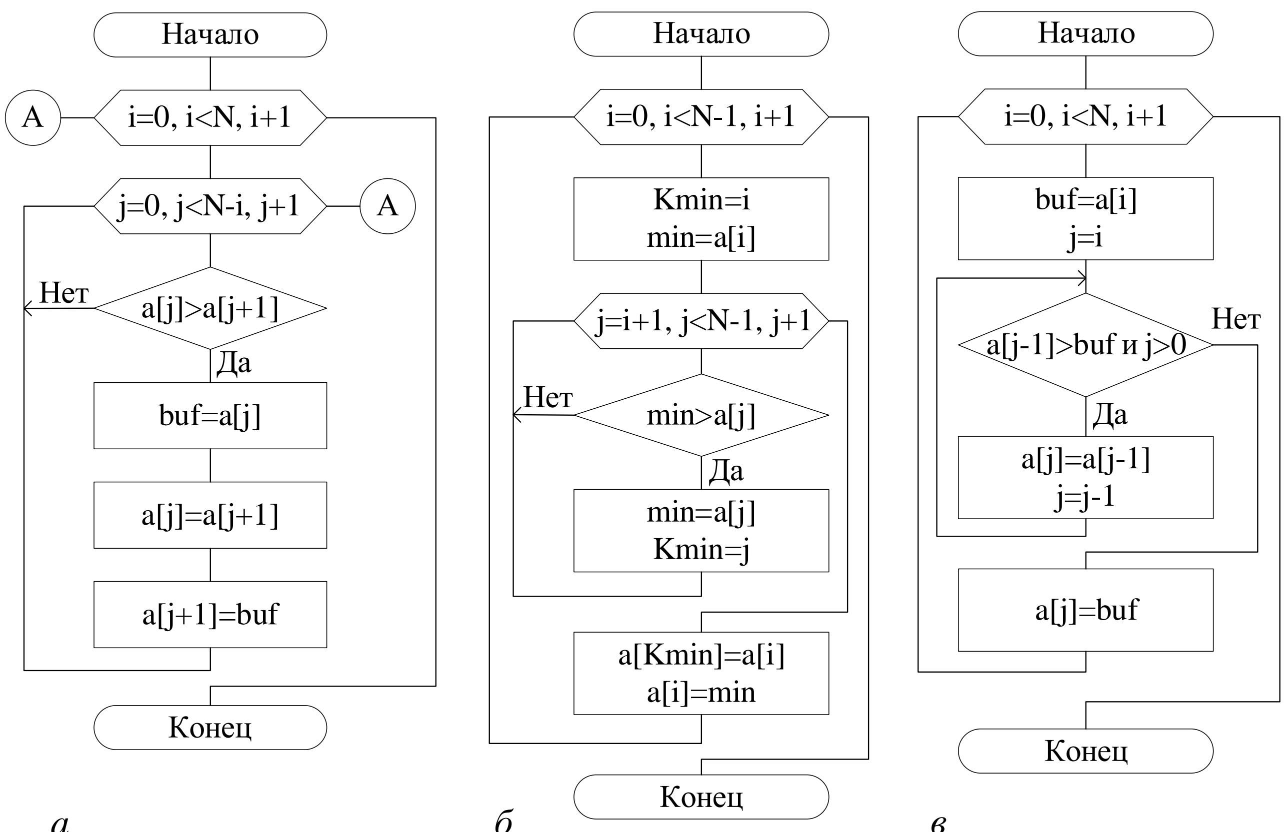
– **шаг 1**. Один элемент массива (нулевой) является уже отсортированным по возрастанию относительно себя и выступает в виде начального списка;

– **шаг 2**. Для всех последующих элементов после нулевого очередной *i*-й элемент сравнивается со всеми предыдущими элементами от нулевого до (*i* – 1), пока текущий элемент *i* меньше или равен предыдущему. После этого будет найдена позиция для вставки или будет достигнуто начало списка. Затем найденный элемент вставляется в найденную позицию.

– **шаг 3**. Шаг итерации продолжается до тех пор, пока не будут просмотрены все *N* элементов массива.

Сложность такого алгоритма сортировки квадратичная ($O(N^2)$).

Схема алгоритма сортировки вставками представлена на рисунке 2.10, *в*.



а – пузырьковая сортировка; *б* – сортировка прямым выбором;
в – сортировка вставками

Рисунок 2.10 – Схемы алгоритмов сортировки одномерных массивов

Быстрая сортировка (сортировка разделением) по возрастанию основана на принципе «разделяй и властвуй». Алгоритм быстрой сортировки:

– **шаг 1**. В массиве (подмассиве) выбирается некоторый опорный элемент *P*. Выбор опорного элемента *P* может быть случайным, в том числе это может быть первый или последний элементы массива (подмассива);

– **шаг 2**. Массив (подмассив) делится на два подмассива и сортируется относительно опорного элемента. Слева от опорного окажутся элементы меньшие

либо равные P, а справа – большие либо равные P. Таким образом, левое подмножество меньше либо равно правому;

– **шаг 3**. Для обоих подмассивов, если в них более двух элементов, повторяются шаги 1 и 2.

Сложность такого алгоритма сортировки квазилинейная ($O(N \cdot \log N)$).

Схема алгоритма и пример быстрой сортировки представлены на рисунке 2.11.

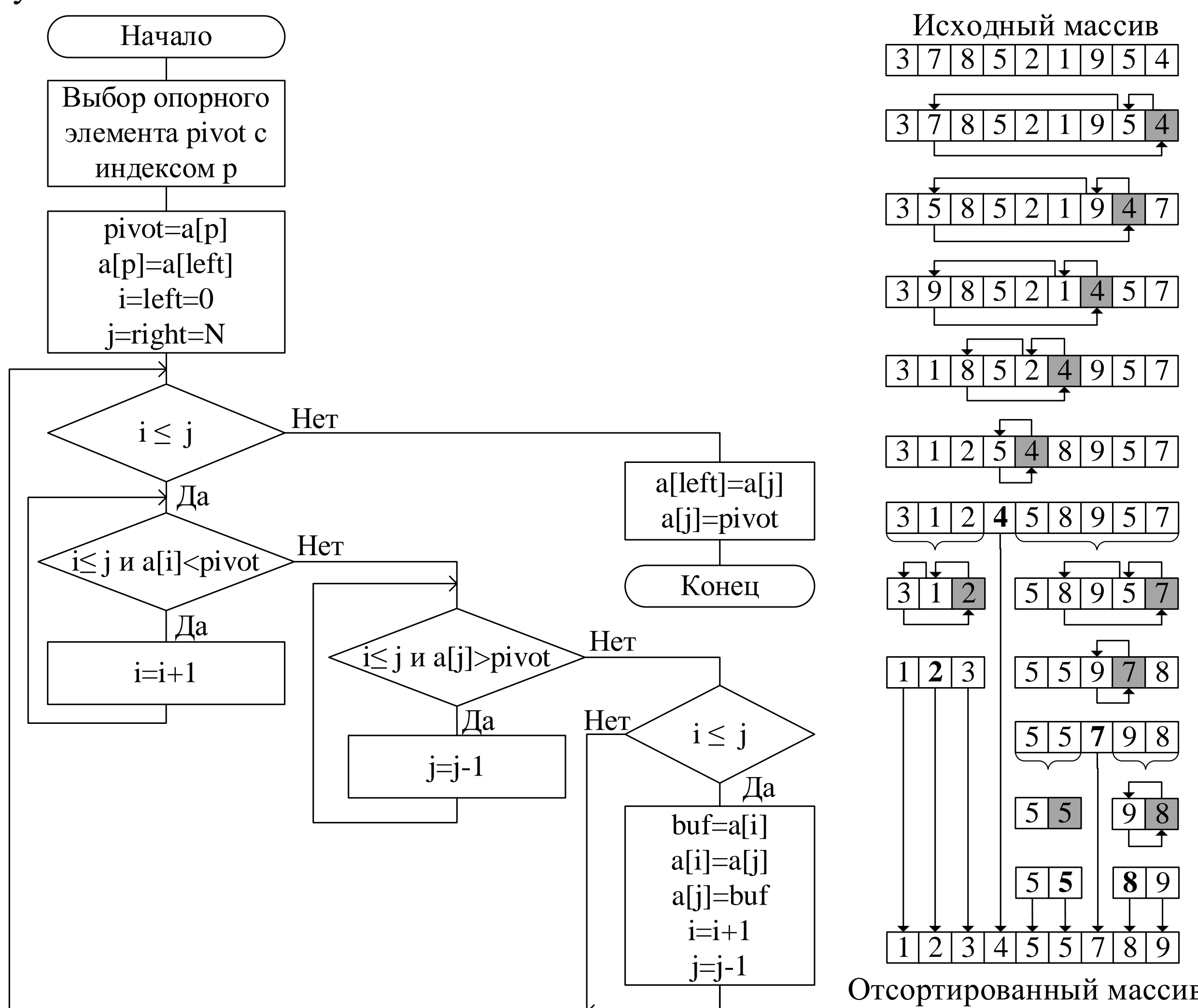


Рисунок 2.11 – Схема алгоритма и пример быстрой сортировки

Как видно из рисунка 2.11, последовательность сортировки следующая:

1 Выбирается опорный элемент P (в примере выбран последний элемент массива) и задаются значения счетчиков i и j (в начале работы алгоритма указывают соответственно на левый и правый конец массива).

2 Увеличивается счетчик i с шагом в единицу (движение по направлению к концу массива), пока не будет найден элемент $a[i] \geq P$.

3 Уменьшается счетчик j с шагом в единицу (движение по направлению к началу массива), пока не будет найден элемент $a[j] \leq P$.

4 Если $i \leq j$ меняется местами $a[i]$ и $a[j]$ и продолжается изменение счетчиков i и j по тем же правилам.

5 Повторяются шаги 2 и 3, пока $i \le j$.

В результате массив разделен на две части: все элементы слева – меньше либо равны опорному элементу P, все элементы справа – больше либо равны P.

Далее действия повторяются для подмассивов.

Сортировка методом Шелла является модификацией сортировки вставками и использует такой параметр, как шаг сортируемых значений d. Сначала сравниваются и сортируются между собой значения, стоящие один от другого на расстоянии шага d.

После этого процедура повторяется для некоторых меньших значений d (шаг уменьшается), а завершается сортировка методом Шелла упорядочиванием элементов при $d = 1$.

Сложность такого алгоритма сортировки квазилинейная ($O(N \cdot \log N)$).

Схема алгоритма и пример сортировки методом Шелла представлены на рисунке 2.12.

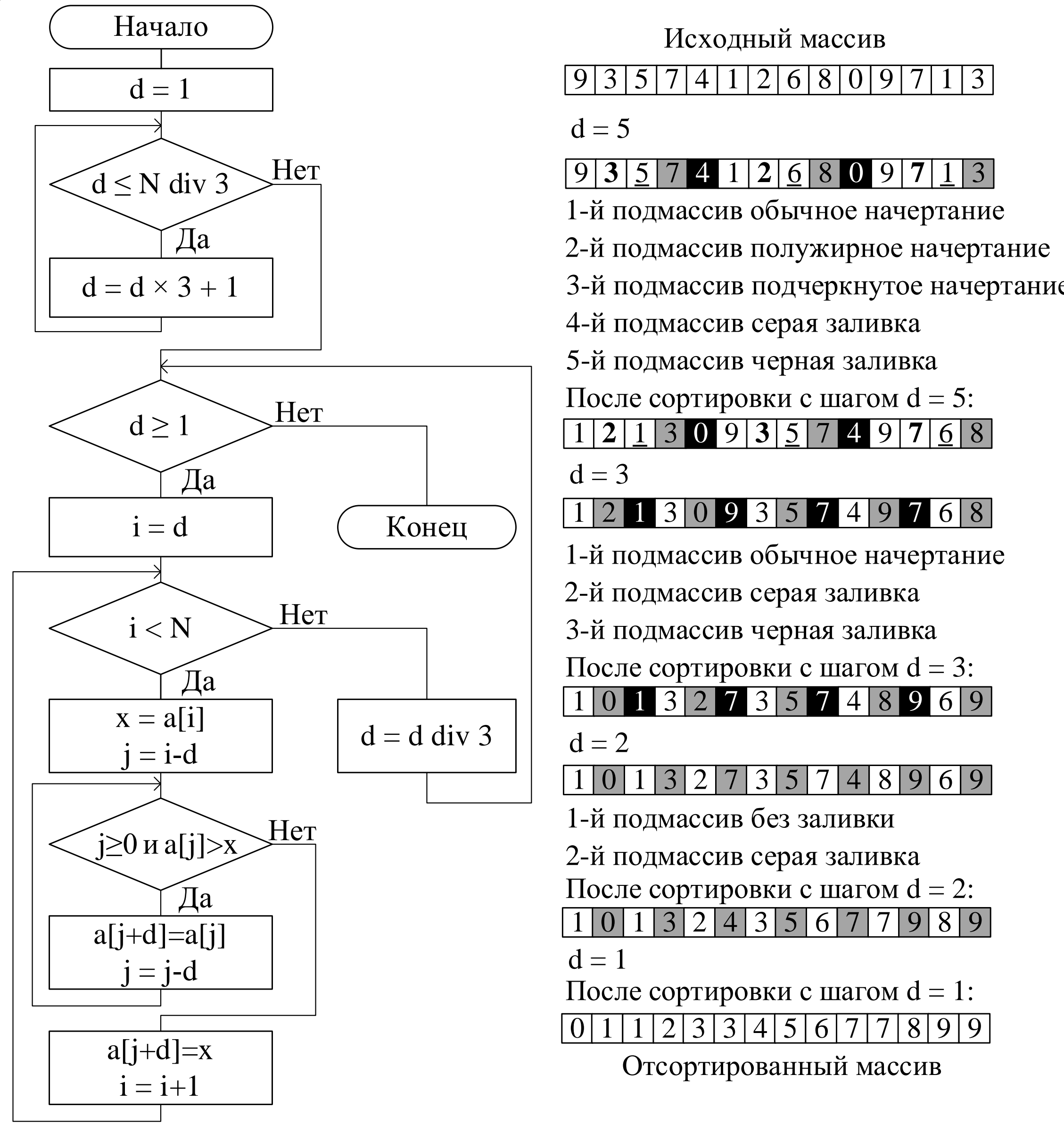


Рисунок 2.12 – Схема алгоритма и пример сортировки методом Шелла

Помимо рассмотренных методов сортировок массивов существует ряд других, а также модификации рассмотренных.

Эффективность многих алгоритмов сортировки зависит от размера массива и первоначальной степени упорядоченности элементов.

Все алгоритмы сортировок можно сравнивать по таким параметрам, как:

1 Время выполнения – характеристика, оценивающая лучшее, худшее и среднее время выполнения сортировки массива;

2 Затрачиваемая память – характеристика, оценивающая объем дополнительной памяти, требуемой алгоритму для реализации сортировки. Сюда входят и буферные переменные, и стек вызовов;

3 Устойчивость – характеристика, определяющая способность алгоритма не менять порядок элементов с одинаковыми значениями относительно друг друга;

4 Количество обменов – среднее количество обменов, производимых при сортировке, которое косвенно влияет на время выполнения, особенно при сортировке объектов, имеющих большой размер.

Сравнение характеристик алгоритмов представлено в таблице 2.33.

Таблица 2.33 – Сравнение характеристик алгоритмов сортировок массивов

Алгоритм	**Время выполнения**			**Память**	**Устойчивость**	**Обмены**
	лучшее	**худшее**	**среднее**			
Метод пузырька	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Есть	$O(N^2)$
Прямым выбором	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	Нет	$O(N)$
Вставками	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Есть	$O(N^2)$
Быстрая сортировка	$O(N \cdot \log N)$	$O(N^2)$	$O(N \cdot \log N)$	$O(\log N)$	Нет	$O(N \cdot \log N)$
Методом Шелла	$O(N \cdot \log^2 N)$	$O(N^2)$	Зависит от шага	$O(1)$	Нет	$O(N^2)$

Наглядная зависимость времени выполнения сортировки разными алгоритмами от размера массива представлена на рисунке 2.13.

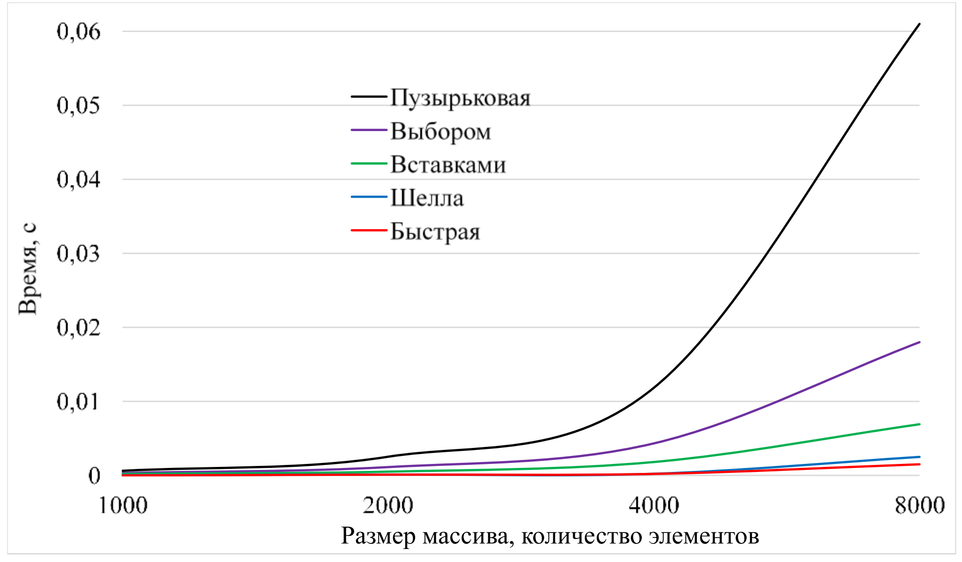


Рисунок 2.13 – Зависимость времени выполнения сортировки от размера массива

Как видно из таблицы 2.33 и рисунка 2.13, при сортировке небольших массивов данных нет существенной разницы по скорости между методами сортировки, поэтому целесообразнее с точки зрения простоты написания кода применять метод пузырька или метод вставок.

Наиболее универсальным методом является метод быстрой сортировки. Он показывает стабильно высокие результаты на любых размерах массивов. На втором месте находится метод Шелла. Его использование может быть обосновано более простым алгоритмом.

Для массивов, которые уже имеют высокую степень упорядоченности, наиболее применим метод сортировки вставками. В этом случае сложность алгоритма может быть линейной ($O(N)$).

2.19 Указатели

Указатель – это адрес ячейки памяти, распределяемой для размещения объекта данных. В качестве идентификатора указателя могут выступать имя переменной, массива, структуры, строковый литерал.

Если переменная объявлена как указатель, то она содержит адрес ячейки памяти, по которому может находиться скалярная величина любого типа.

При объявлении переменной типа «указатель», необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующим символом указателя (или группой символов). В языке программирования Delphi символом указателя является циркумфлекс (^), а в языках программирования C/C++ – символ звездочка (*).

Формат объявления указателя представлен в таблице 2.34.

Таблица 2.34 – Формат объявление указателя

В синтаксисе Delphi	**В синтаксисе C/C++**
идентификатор_указателя: ^тип_данных;	*модификатор*[1] спецификатор_типа *идентификатор_указателя;
Примеры: `p1: ^integer;` `p2: ^real;`	Примеры: `short int *p1;` `float *p2;`
[1] *Модификатор типа не является обязательным элементом*	

Указатели в языках программирования C/C++

Задавая вместо спецификатора типа ключевое слово void, можно отсрочить спецификацию типа, на который ссылается указатель. Переменная, объявляемая как указатель на тип void, может быть использована для ссылки на объект любого типа. Однако для выполнения арифметических, логических и т. д. операций над указателями или над объектами, на которые они указывают, необходимо явно определить тип объектов.

Указатель может указывать на константные данные и сам иметь атрибут const, например:

– `const float *p;` – объявление указателя на константное вещественное число, сам указатель не является константой;

– `float *const p;` – объявление константного указателя на некоторое вещественное число;

– `const float *const p;` – объявление константного указателя на константное вещественное число.

Для описания переменной типа «указатель» необходимо задать, на объект данных какого типа возможна ссылка через объявляемый указатель, т. к. необходимо знать точный объем памяти, отводимой под объекты данных, на которые указывает указатель, например (для 32-разрядной архитектуры):

```
int *p; //p - указатель на элемент типа данных int (4 байта)
float *k1,*k2; //k1 и k2 указатели на элементы типа float (4 б)
char *s; //s указатель на элементы символьного типа char (1 б)
```

Над указателями можно выполнять следующие унарные операции:

1) & (взятие адреса) – указатель получает адрес объекта данных, на который указывает. Данная операция применима к переменным, под которые выделен соответствующий участок памяти. Таким образом, инструкция `p=&k;` присваивает адрес первого байта переменной `k` в переменную `p`;

2) * (операция разадресации) – предназначена для доступа к значению данных, расположенному по адресу, на который указывает указатель.

Над указателями также можно выполнять:

– арифметические операции (сложения, вычитания, инкремента, декремента) (если `*p` – указатель, `n` – целое число, то `*p++` является адресом следующего элемента; `*p+n` задает адрес *n*-го элемента, на который указывает указатель `p`);

– операции сравнения (любой адрес может быть проверен на равенство (==) или неравенство (!=) со специальным значением NULL).

Указатели могут встречаться в выражениях, т. е. если `p` – указатель на объект некоторого типа, то он может использоваться в выражении наряду с другими указателями, а также переменными и константами, не являющимися указателями (листинг 2.15).

Листинг 2.15 – Примеры использования указателей в выражениях в языке C++

```
1   *p=-2; //Значение -2 заносится в ячейку памяти по адресу p
2   *p/=i-1; //Это значение уменьшается в i-1 раз
3   (*p)--; //Затем это значение уменьшается на единицу
4   //В третьей строке использованы скобки, т. к. операции с
5   //одинаковым приоритетом выполняются справа налево, а выражение
6   //без использования скобок
7   *p--; //уменьшит адрес, а не значение объекта данных
```

Связь между указателями и массивами. В языках программирования C/C++ между индексированием массивов и адресацией существует тесная взаимосвязь. Доступ к элементам массива с помощью индексирования может быть заменен доступом к ним с использованием адресов. При этом доступ по адресу выполняется быстрее.

Примеры использования указателей с массивами приведены в листинге 2.16.

Листинг 2.16 – Примеры использования указателей с массивами в языке C/C++

```
1   y=&mas[0]; // присваивает переменной y адрес элемента mas[0]
2   y+1; //указывает на следующий элемент массива, при этом
3   //учитывается масштабирование, т. е. выполняется приращение
4   //адреса с учетом размеров памяти, отводимой под один элемент
5   //Т. к. имя массива является адресом его нулевого элемента, то
6   //выражение y=&mas[0] может быть записано как:
7   y=mas;
8   //Доступ к i-му элементу массива может быть осуществлен либо,
9   //используя имя массива и индекс
10  a=mas[i];
11  //или имя массива и значение индекса через указатель
12  a=*(mas+i);
13  //либо только через указатель, учитывая строку 7
14  a=*(y+i); //Таким образом, переменная a в строках 10, 12 и 14
15  //принимает одно и то же значение
```

Так как указатель – это переменная величина, то выражения вида `y=mas` и `y++` являются допустимыми. Но поскольку имя массива является константой, то выражения `mas=y`, `mas++`, `y=&mas` являются недопустимыми, т. к. значение константы не может быть изменено.

Примеры возможных способов задания значений элементам массива представлены в листинге 2.17.

Листинг 2.17 – Примеры задания значений элементов массива в языке C/C++

```
1   int mas[10], *p; //Объявление массива и указателя типа int
2   p=mas; //или по-другому
3   p=&mas[0]; //присваивает указателю адрес нулевого элемента.
4   //Способы задания нулевому элементу массива значения = 2:
5   *mas=2; //или
6   mas[0]=2; //или
7   *(mas+0)=2; //или
8   *p=2; //или
9   p[0]=2; //или
10  *(p+0)=2;
```

Быстрее всего выполняется операция `*mas=2` и `*p=2` (строка 5 и 8 листинга 2.17), т. к. в них отсутствует операция сложение.

Таким образом, при исполнении программы индексное обозначение элементов массива в конечном итоге сводится к адресному (т. е. с помощью указателей).

Массивы указателей. Кроме обычных массивов, в языках программирования С/С++ используются массивы указателей. Они представляют собой массивы, элементами которых являются указатели (например, на элементы другого массива):

```
float *mas[6];
```

Это соответствует массиву из шести элементов, являющихся адресами объектов данных типа float.

В массиве указателей последовательность элементов задается последовательностью размещения указателей в массиве. Тогда изменение порядка следования (включение, исключение, упорядочение, перестановка), которое в обычном массиве заключается в перемещении самих элементов, в массиве указателей должно сопровождаться соответствующими операциями над указателями. Очевидные преимущества возникают, когда сами указуемые элементы являются достаточно большими (массивы, строки, структуры и т. д.).

Многоуровневые указатели. Рассмотрим определение переменной: `double **pp`;

В данном случае объявляется указатель на указатель (или, по сути, адрес указателя) вещественного типа double.

Но поскольку любой указатель в С/С++ может ссылаться как на отдельную переменную, так и на область памяти (массив), то в применении к двойному указателю получаются четыре варианта структур данных, а именно:

– указатель на одиночный указатель на переменную типа double;

– указатель на одиночный указатель на массив переменных типа double;

– указатель на массив, содержащий указатели на одиночные переменные типа double;

– указатель на массив, содержащий указатели на массивы переменных типа double.

Третий вариант позволяет использовать двойной указатель для работы с массивами указателей следующим образом:

```
double **p1, *p2[20]; //Определение указателя на указатель типа
      //double и массива указателей из 20 элементов типа double
p1 = p2; //или p1 = &p2[0]; p1 содержит адрес нулевого элемента
        //массива указателей p2
*(p1+i) //или p1[i];
**(p1+i) //или *p1[i];
```

2.20 Динамическое выделение памяти

Для работы с динамическим выделение и освобождением памяти (т. е. в процессе выполнения программы) в языках программирования высокого уровня существуют специальные функции.

1 В языке программирования Delphi

Для динамического выделения памяти существует процедура New(), имеющая один параметр – указатель на переменную того типа, память для которой необходимо выделить.

У динамической переменной фактически нет имени, поэтому обращение к ней возможно только с помощью указателя.

Для освобождения памяти, занимаемой динамической переменной, используется процедура Dispose(), имеющая один параметр – указатель на динамическую переменную.

Примеры выделения и освобождения динамической памяти приведены в листинге 2.18.

Листинг 2.18 – Динамическое выделение памяти в языке Delphi

```
1   var p1,p2,p3: ^integer; //Указатели на переменные типа integer
2   begin
3   New(p1); New(p2); New(p3); //Динамическое выделение памяти
4   p1^:=5; p2^:=3; p3^:=p1^+p2^;
5   write('Сумма чисел равна ');
6   write(p3^);
7   Dispose(p1); Dispose(p2); Dispose(p3); //Освобождение памяти
8   end;
```

Также для выделения динамической памяти для массивов используется процедура SetLength(), имеющая два параметра: идентификатор массива и количество его элементов. Для освобождения динамически выделенной памяти можно присвоить идентификатору массива значение NIL. При динамическом выделении памяти нумерация индексов элементов массива может начинаться только с нуля. Примеры динамического выделения памяти для одномерного массива и матрицы представлены в листинге 2.19.

Листинг 2.19 – Динамическое выделение памяти под массивы в языке Delphi

```
1   var n,m,i,j:integer;
2   mas1, mas2:array of integer; //Описание массивов mas1 и mas2
3   mas3:array of array of integer; //Описание матрицы mas3
4   begin
5   //ОДНОМЕРНЫЙ МАССИВ:
6   writeln('Введите размер одномерного массива');
7   read(n); //Ввод размера одномерного массива
8   SetLength(mas1,n); //Динамическое выделение памяти под mas1
9   SetLength(mas2,n); //Динамическое выделение памяти под mas2
10  writeln('Введите элементы массива');
11  for i:=0 to n-1 do
12    read(mas1[i]); //Ввод элементов массива mas1
13  mas2:=mas1; //Присвоение значение массива mas1 массиву mas2
14  mas1:=nil; //Освобождение памяти, выделенной под массив mas1
15  writeln('Значения массива:');
16  for i:=0 to n-1 do
17    writeln(mas2[i]); //Вывод элементов массива mas2
18  mas2:=nil; //Освобождение памяти, выделенной под массив mas2
```

```
19  //МАТРИЦА:
20  writeln('Введите количество строк матрицы');
21  read(n); //Ввод количества строк матрицы
22  writeln('Введите количество столбцов матрицы');
23  read(m); //Ввод количества столбцов матрицы
24  SetLength(mas3,n,m); //Динамическое выделение памяти под mas3
25  writeln('Введите элементы матрицы');
26  for i:=0 to n-1 do
27    for j:=0 to m-1 do
28      read(mas3[i,j]); //Ввод элементов массива mas3
29  writeln('Введена матрица:');
30  for i:=0 to n-1 do
31    for j:=0 to m-1 do
32      writeln(mas3[i,j]); //Вывод элементов массива mas3
33  mas3:=nil; //Освобождение памяти, выделенной под массив mas3
34  end;
```

2 В языке программирования С (библиотека stdlib.h)

Используются следующие функции:

– `void *malloc (unsigned n);` выделяет область памяти размером *n* байт. Функция возвращает указатель на начало выделенного блока памяти длиной *n* байт, если для выделения блока не хватает памяти, то возвращается значение NULL;

– `void *calloc (unsigned n, unsigned m);` выделяет область памяти и возвращает указатель на начало области памяти для размещения *n* элементов по *m* байт каждый, если для выделения блока не хватает памяти, то возвращается значение NULL;

– `void *realloc (void *ptr, unsigned n);` изменяет размер динамически выделенной памяти, на которую указывает указатель *ptr на новый размер *n* (значение *n* задает новый размер блока). Если указатель не является значением, которое ранее было определено функциями malloc, calloc или realloc, то функция ведет себя неопределенно;

– `void free (void *ptr);` Освобождает ранее выделенный функциями malloc, calloc или realloc блок памяти с адресом ptr.

3 В языке программирования С++

Для работы с динамическим выделение памяти в языке С++ существуют функции new и delete. Используются два варианта их применения:

1 С обычными переменными. Общий формат записи имеет вид:

```
спецификатор_типа *указатель = new спецификатор_типа (значение);
delete[] *указатель;
```

Пример использования:

```
int *p = new int(3); //Выделение участка памяти в соответствии
//с указанным типом (int, 4 байта) и занесением туда указанного
//значения (три)
```

```
int *p = new int; //Выделение памяти без инициализации
delete *p; //Освобождение выделенной памяти
```

2 С массивами. Общий формат записи имеет вид:

```
спецификатор_типа *указатель = new спецификатор_типа [n];
delete[] *указатель;
```

Пример использования:

```
double *p = new double[5]; //Выделение участка памяти размером
//n блоков (для n элементов) указанного типа double
delete[] *p; //Освобождение выделенной памяти
```

В случае успешного выделения памяти операция new возвращает значение, отличное от нуля. Значение, равное нулю, говорит о том, что не найден непрерывный свободный фрагмент памяти нужного размера.

Динамическое выделение памяти для двумерного массива размером $N \times M$ имеет особенности. Имя любого массива рассматривается компилятором как указатель на нулевой элемент массива. Так как имя двумерного динамического массива является указателем на указатель, то сначала выделятся память под указатели, а затем под соответствующие этим указателям строки. Освобождение выделенной памяти происходит в обратном порядке. Пример представлен в листинге 2.20.

Листинг 2.20 – Динамическое выделение памяти под матрицу в языке С++

```
1   int n,m,i,j; //Определение переменных
2   double **dmas; //Объявление указателя на указатель на массив
3   dmas=new double*[n]; //Выделение памяти по одному индексу
4   for(i=0; i<n; i++)
5     dmas[i] = new double[m]; //Выделение памяти по второму индексу
…   …
10  for(i=0; i<n; i++)
11    delete[] dmas[i]; //Освобождение памяти по одному индексу
12  delete[] dmas; //Освобождение памяти по второму индексу
13  dmas=NULL;
```

2.21 Строковые данные

В языке программирования Delphi существует строковые типы данных.

Тип string эквивалентен типу ShortString и может хранить строку максимальной длиной 255 символов. Причем длину строки можно явно ограничить:

```
var
s1: string; //Строка длиной по умолчанию 255 символов
s2: string[20]; //Строка длиной 20 символов
```

Строковые типы данных LongString и WideString предполагают динамическое выделение памяти, поэтому их длина ограничена свободной памятью в процессе выполнения программы.

Существуют следующие функции для работы со строковым типом данных языка программирования Delphi:

1) `Length(s);` Возвращает длину строки *s*;

2) `Copy(s,k,n);` В строке *s*, с *k*-й позиции копирует *n* символов;

3) `Pos(w,s);` Возвращает номер позиции с которой начинается первое упоминание подстроки *w* в строке *s* или нуль, если *w* не найдена;

4) `Delete(s,k,n);` В строке *s*, с *k*-й позиции удаляет *n* символов;

5) `Insert(w,s,k);` Вставляет подстроку *w* в строку *s* с *k*-й позиции;

6) `Concat(a,…,z);` Объединение строк *a*, …, *z* в одну строку;

7) `LowerCase(s);` В строке *s* преобразует все прописные латинские буквы в строчные;

8) `ansiLowerCase(s);` В строке *s* преобразует все прописные буквы любого алфавита в строчные;

9) `UpperCase(s);` В строке *s* преобразует все строчные латинские буквы в прописные;

10) `ansiUpperCase(s);` В строке *s* преобразует все строчные буквы любого алфавита в прописные;

11) `SetLength (s,n);` Динамически изменяет длину строки *s* на новый размер *n*;

12) `Chr(1…255);` Возвращает символ соответствующий коду ASCII.

В языках программирования С/С++ нет отдельного строкового типа данных, но существует понятие символьной строки.

Строки представляются как массив элементов типа char, в конце которого помещен нуль-терминатор (признак конца строки) – символ '\0'. Такой массив и будет называться *символьной строкой*, а длина строки (размерность массива) будет равна количеству символов в строке плюс нуль-терминатор (признак конца строки \0).

Общий вид объявления строки:

```
char идентификатор [размер];
```

Пример определения символьной строки:

```
char string1[100];
```

Указатель на символьную строку. Как и обычный указатель, указатель на тип char может быть инициализирован при его описании. Для этого используется строковый литерал (т. е. константа), при этом адрес ее первого символа будет присвоен указателю:

```
char *str1 = "строка";
```

При этом выделяется память для строки размером 7 байт и указатель инициализируется адресом первого символа «с». Признаком окончания символьной строки является нуль-символ (\0) (7-й байт).

В выражениях, где применяется указатель, компилятор подставляет адрес константы.

Строковая константа – это любое выражение, заключенное в двойные кавычки. При встрече строковой константы ее символы и символ '\0' записываются в последовательные ячейки памяти. Строковые константы размещаются в статической памяти.

Также можно создать массив указателей типа char (массив строк):

```
char *str2[5]; // массив из 5-ти указателей символьного типа
```

Инициализацию массива строк (массива указателей char*) можно выполнить следующими способами:

```
char str1[][8]={"строка1","строка2"}; //или
char *str2[2];
str2[0]="строка1";
str2[1]="строка2”; //или
char *str3[2]={"строка1","строка2"};
//Все три массива символьных строк str1, str2 и str3 имеют
//одинаковое значение после инициализации
```

Строковая константа, как и любая другая константа в C/C++, может быть также определена с помощью директивы #define, например:

```
#define st1 "Минск";
```

Отличие объявления символьной строки через указатель на тип char и с использованием массива типа char заключается в том, что имя массива (например, mas) является константным значением, т. е. нельзя изменить значение mas, т. к. это означало бы изменение адреса массива в памяти. Однако общим является то, что имя массива является также указателем (адресом нулевого элемента).

Исходя из вышеуказанного, для доступа к очередному элементу массива можно использовать инструкцию `mas+1;` но не инструкцию `++mas;`

Например, определен указатель `char *str[10];`

При использовании этого указателя происходит выделение статической памяти не только под строку, но и под переменную str, являющуюся указателем на строку. При этом значение этой переменной может изменяться, таким образом, инструкция `++str` будет указывать на следующий символ строки.

Существуют следующие функции для работы со строками (библиотека string.h в C и cstring в C++):

1) `char *strcpy(char *st1, const char *st2);` Функция копирует содержимое строки st2, включая нулевой символ, в строку st1;

2) `char *strcat(char *st1, const char *st2);` Функция добавляет справа к строке st1 содержимое строки st2;

3) `int strcmp(const char *st1, const char *st2);` Функция сравнивает содержимое строк st2 и st1. Если st1 < st2, то результат равен –1, если st1 = st2 – результат равен нулю, если st1 > st2 – результат равен 1;

4) `char *strstr(const char *st1, const char *st2);` Функция возвращает указатель на первое упоминание (позицию первого символа) подстроки st2 в строке st1;

5) `char *strchr(const char *st, int ch);` Функция возвращает указатель на первое появление символа ch в строке st;

6) `int strlen(const char *st);` Функция возвращает длину строки st;

7) `char *strrev(char *st);` Функция изменяет порядок следования символов в строке на противоположный;

8) `char *strdup(const char *st);` Функция дублирует строку st;

9) `char *strlwr(char *st);` Функция конвертирует символы строки st к нижнему регистру;

10) `char *strupr(char *st);` Функция конвертирует символы строки st к верхнему регистру;

11) `int atoi(const *st);` Функция преобразует строку, на который указывает указатель *st, в число целого типа (int). Преобразование заканчивается либо когда встречается символ, не являющейся цифрой, либо когда будет преобразована вся строка. Если первый символ строки не является цифрой, то функция вернет нуль и завершит работу;

12) `double atof(const *st);` Функция преобразует строку, на которую указывает указатель *st, в число вещественного типа (аналогично предыдущей);

13) `char *itoa(int a, char *st, int base);` Функция преобразует число *a* целого типа в строку, на которую указывает указатель *st, при этом преобразование может быть выполнено в разные системы счисления, а base определяет основание системы счисления, в которой будет записано число в виде строки;

14) `char *gcvt(double a, int dec, char *st);` Функция преобразует число *a* вещественного типа в строку, на которую указывает указатель *st. Значение dec задает количество десятичных разрядов представления числа (но не более 18);

15) `char *strtok(char *st1, const char *st2);` Последовательность вызовов этой функции разбивают строку st1 на лексемы, которые представляют собой последовательности символов, разделенных разделителями. Символы из строки, на которую указывает st2, используются как разделители, определяющие лексему. Если лексема не найдена, то возвращается NULL. Во время первого вызова функции в качестве указателя используется st1. При последующих вызовах в качестве первого аргумента используется NULL. Таким образом, вся строка может быть разбита на лексемы;

16) `*strset(char *st, int ch);` Функция делает все символы в строке st равными символу ch.

Также для работы с символьными строками могут быть использованы функции ввода/вывода, описанные в подразделе 2.14 для языка C/C++.

2.22 Пользовательские подпрограммы (функции)

Подпрограмма – самостоятельная единица, часть программы, разработанная для выполнения некоторого функционально законченного действия.

Подпрограммы являются неотъемлемой составляющей парадигмы процедурного программирования и обеспечивают такие важные преимущества, как:

– избавляют от многократного дублирования в тесте программы аналогичных фрагментов;

– упрощают структуру программы и ее наглядность;

– повышают устойчивость к ошибкам при модификации программ.

Среди подпрограмм различают:

1 *Процедуры* – это оформленные подпрограммы, которые предназначены для выполнения какого-либо законченного действия (или действий), но не возвращающие результат своей работы в вызвавшую их основную программу. Процедура может принимать параметры, которые могут использоваться или даже изменяться внутри нее. Процедуры используются, когда нет необходимости возвращать в основную программу результат (например, при отрисовке графика) или когда необходимо вернуть в основную программу более одного значения.

2 *Функции* – это оформленные подпрограммы, которые также выполняют определенные функционально законченные действия, но при этом еще возвращают результат своей работы обратно в вызывающую их основную программу. В функциях помимо возможного указания передаваемых аргументов (параметров) и, возможно, их типов, еще необходимо указание типа возвращаемого функцией результата. Функция может вернуть в явном виде в основную программу только одно значение, являющееся результатом ее работы.

Таким образом, основное отличие процедур от функций заключается в способах их вызова из основной программы: процедура, как правило, вызывается по имени и вызов процедуры по сути является оператором, который необходимо просто выполнить; функция также вызывается, как правило, по имени, но по сути в программе является выражением, поскольку будет иметь результат.

В языке программирования Delphi используются как процедуры, так и функции, а в языках программирования C/C++ используются только функции.

Использование подпрограмм вводит следующие понятия.

Локальный объект данных – это объект данных, объявленный и доступный для использования внутри подпрограммы (локальные переменные, константы, типы и т. д.).

Глобальный объект данных – это объект данных, объявленный в основной (главной) программе и доступный для использования как в самой программе, так и в вызываемых подпрограммах (глобальные переменные, константы, типы).

Переопределение переменной – это задание нового значения той же переменной, т. е. перезапись нового значения в пространство памяти, выделенное под конкретную переменную.

Идентификаторы локальных и глобальных данных могу совпадать. При этом глобальная переменная и локальная переменная с одним и тем же именем физически имеют разное расположение в памяти, т. е. глобальная переменная не переопределяется в результате изменения одноименной локальной. Для локальных переменных память выделяется в специальной области (стеке) и только на время выполнения подпрограммы, в которой они объявлены. Поэтому все внутренние результаты выполнения подпрограммы от вызова к вызову могут быть разными.

Параметры подпрограммы в списке ее параметров (аргументов) различаются на формальные и фактические.

Формальные параметры – это параметры, указанные в списке параметров в объявлении подпрограммы (процедуры или функции).

Фактические параметры – это параметры, формируемые в вызывающей программе и передаваемые в подпрограмму при ее вызове, т. е. указанные в инструкции вызова подпрограммы.

При передаче в подпрограмму переменных идентификаторы формальных и фактических параметром могут даже не совпадать, но должны совпадать типы данных этих параметров (быть совместимыми).

В языках программирования есть множество встроенных или библиотечных процедур и функций (тригонометрические, преобразования типов, ввода/вывода и множество других). Их нет необходимости каждый раз создавать, а использование и реализация не вызывает у программистов вопросов. Далее речь пойдет о подпрограммах, создаваемых программистом.

Подпрограммы в языке программирования Delphi

Созданные процедуры и функции могут быть:

– локальными, тогда полная реализация процедуры или функции помещаются в раздел implementation, перед программой, которая вызывает их;

– глобальными, тогда объявление (первая строка) процедуры или функции помещается в раздел interface, а полная реализация помещается в любом месте раздела implementation.

В общем виде объявление процедуры имеет следующий вид:

```
procedure идентификатор (параметр1: тип1; …, параметрN: типN);
//заголовок процедуры является ее объявлением
var
//Объявление локальных объектов данных
```

```
begin
//Тело процедуры
end; //заголовок процедуры и ее тело является полной реализацией
```

Общий вид вызова процедуры из основной программы имеет вид:

```
идентификатор (параметр1, …, параметрN);
//В скобках перечисляется список фактических параметров
```

В общем виде объявление функции имеет следующий вид:

```
function идентификатор (параметр1: тип1; ..., параметрN: типN):
тип_рез-та;
//заголовок функции является ее объявлением
var
//Объявление локальных объектов данных
begin
//Тело функции
идентификатор_функции:= результат;
//или
result:= результат;
end; //заголовок функции и ее тело является полной реализацией
```

Последней инструкцией тела функции должно быть присваивание имени функции или ключевому слову result результата ее выполнения. Результатом может быть выражение, литерал или переменная такого же типа, как заявлено в объявлении функции. Без этой инструкции функция не передаст результат в вызывающую программу.

Общий вид вызова функции из основной программы имеет вид:

```
идентификатор (параметр1, …, параметрN);
//В скобках перечисляется список фактических параметров
```

Примеры объявления локальных и глобальных подпрограмм представлены в приложении Б.

Параметры, передаваемые в подпрограмму и описываемые в заголовке подпрограммы, могут передаваться в разных качествах и являться:

– **параметром-значением**. В данном случае при передаче параметра в подпрограмму внутри нее создается локальная копия переданного объекта данных и его изменение не переопределяет переданное в подпрограмму в качестве параметра значение. Пример заголовка процедуры с параметрами-значениями:

```
procedure Pro1(a, b:integer; x, y:real);
```

– **параметром-переменной**. В этом случае фактическим параметром может быть только переменная. При вызове подпрограмм ей передается адрес ячейки памяти передаваемой переменной, и все изменения этой переменной

приводят к ее переопределению в программе. Пример заголовка процедуры с параметрами-переменными:

```
procedure Pro2(var a:byte; var b:char);
```

– **параметром-константой**. В этом случае фактическим параметром может быть переменная или константа. При вызове подпрограммы в нее передается адрес ячейки памяти передаваемого параметра, но внутри подпрограммы запрещены любые его изменения, а он служит только как источник исходных данных. Пример заголовка процедуры с параметрами-константами:

```
procedure Pro3(const i, j:integer; const n:extended);
```

– **параметром процедурного типа**. В данном случае в качестве параметра передается имя процедуры или функции, т. е. по сути подпрограмма осуществляется вызов другой подпрограммы, указанной в качестве своего параметра. Пример работы с параметрамом процедурного типа представлен в листинге 2.21.

Листинг 2.21 – Пример работы с параметрамом процедурного типа

```
1   type fun=function(x:integer):integer; //Создание процедурного типа
2   var x,y: integer; //Глобальные переменные
3         f: fun; // Переменная процедурного типа
4
5   function funct(f:fun):integer; //Функция, вызываемая из программы
…   begin
10    result:=f(x)*x;
11  end;
12
13  function funct_param(x:integer):integer; //Функция-параметр
14  begin
15    result:=x+2;
16  end;
17
18  begin   //Основная программа
19    x:=2; //Глобальной переменной x задается значение 2
20    y:=funct(funct_param); //Вызов функции с функцией-параметром
21    write(y); //Будет выведено 8
22  end.
23  //Изначально x=2, затем вызывается функция funct, поскольку
24  //x глобальная переменная, то и в функции funct x=2
25  //funct имеет параметр-функцию funct_param, результатом которой
26  //будет x+2=2+2=4, тогда f(x)*x =4×2=8= результату функции funct
27  //который возвращается в программу и выводится
```

Подпрограммы в языках программирования C/C++

Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные. Если внутри функции используются уже

определенные ранее идентификаторы, то это соответствует неявной передаче информации в функцию.

Следует различать понятия «прототип функции» и «определение функции».

Прототип функции – это ее имя с указанием типа возвращаемого результата и перечислением в круглых скобках через запятую типов передаваемых параметров. Прототип функции завершается точкой с запятой. Примеры прототипов функций:

```
double fun1(float x, int n);
char *fun2(char*, int k);
```

Прототип является предварительным объявлением данных функции (имени функции, количества и типов параметров, типа результата), необходимых компилятору для того, чтобы проконтролировать корректность вызова функции и в необходимых случаях, выполнить преобразование аргументов к типу принимаемых параметров, а также сгенерировать корректный возвращаемый результат.

Имена параметров в прототипе можно опустить (не рекомендуется):

```
int fun3(int, int);
```

В том случае, когда в функцию не передаются аргументы, в прототипе в круглых скобках вместо списка параметров записывается пустой тип void либо ничего не указывается:

```
double fun4(void);
float fun5();
```

Определение функции – это ее полная реализация, т. е. заголовок и тело.

Определение функции может располагаться в любом месте программы, но определение одной функции не может содержать в себе определение другой функции, однако может содержать прототип другой функции, если последняя вызывается из функции.

Если определение некоторой функции располагается ниже точки ее вызова или в другом файле, то для такой функции выше точки ее вызова обязательно должен быть помещен ее прототип.

Общий вид определения функции:

```
спецификатор_класса_памяти спецификатор_типа имя_функции
(список_формальных_параметров)
{тело_функции}
```

Спецификатор класса памяти задает класс памяти функции, который может быть static или extern; он является необязательным для указания.

Спецификатор типа функции (может отсутствовать) задает тип возвращаемого результата. Результат может быть любого стандартного типа, а также пользовательского типа (структура, объединение, класс и др.). Если спецификатор типа не задан, то предполагается, что функция возвращает значение типа int.

Функция возвращает явно только одно значение указанного типа. Допускается реализация функций, не имеющих параметров и/или не возвращающих никаких значений.

Передача (возврат) значения из вызванной функции в вызывающую осуществляется посредством использования оператора возврата **return**, имеющего следующий вид:

`return` *`выражение;`* или **`return`** *`(выражение);`*

Выражение может являться как некоторой константой, так и вычисляемым выражением, например:

```
return -12;  или  return (-12);
return i+5/j;  или  return (i+5/j);
```

Оператор return в теле функции может встречаться более одного раза. При его встрече в программе происходит прекращение выполнения функции и осуществляется передача управления следующей инструкции в вызывающей программе.

Если в операторе return отсутствует выражение, а вызываемая функция должна вернуть некоторое значение, то компилятор генерирует ошибку.

При отсутствии оператора return функция будет выполнена до конца. В этом случае функция также явно не возвращает результат, перед именем такой функции в ее прототипе и в ее определении должен быть записан пустой тип void.

Если тип выражения в операторе return не соответствует типу значения, возвращаемому функцией, то автоматически будут выполнены соответствующие преобразования типа выражения к типу, возвращаемого функцией значения. В случае невозможности таких преобразований выводится сообщение об ошибке.

Функция не может в качестве результата возвращать массив любого типа, но может возвращать указатель на такой массив.

Любая функция может завершаться при вызове системной функции **abort()** или **exit(n)** , где *n* – целое число, являющееся кодом завершения. Код завершения обычно используется программой, которая породила текущий процесс. В этом случае прекращается выполнение всей программы, автоматически закрываются все открытые файлы и освобождаются все области динамической памяти.

Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях, а типы фактических параметров

при вызове функции должны быть совместимы с типами соответствующих формальных параметров (листинг 2.22).

Листинг 2.22 – Формальные и фактические параметры функции

```
1    void fun(int,char,double); //Прототип функции и типы формальных
2    main()                     //параметров
3    {
4    int n=1;
5    char c='a';
…    fun(n,c,1.5); //n, c, 1.5 фактические параметры
10   }
11   void fun(int x, char y, double z) //x, y, z формальные параметры
12   {
13   //Тело функции
14   }
```

В функцию параметры могут передаваться следующими способами:

1 Используя глобальные переменные, что соответствует неявной передаче параметров. Любое изменение формального параметра приводит к изменению (переопределению) фактического.

2 Используя в качестве формального параметра ссылки. Изменение формального параметра также приводит к изменению (переопределению) фактического.

3 По значению. Изменение формального параметра не влияет на значение фактического, т. к. аргументы, передаваемые в функцию по значению, представляют собой их локальные копии, с которыми работает функция. Если в функции изменяется значение аргумента, то изменяется не переменная, которую передали функции в качестве аргумента, а ее локальная копия.

4 Используя в качестве параметров (фактического и формального) адреса. Если значение формального параметра передается в функцию по адресу, то, как и в первых двух случаях, любое изменение значения по переданному в функцию адресу влияет на значение переменной, адрес которой передан в функцию в качестве фактического параметра.

Областью действия имени считается часть программы, в которой это имя можно использовать. Для автоматических (со спецификатором класса памяти auto) переменных областью действия является функция, в которой они описаны. Идентификаторы локальных (автоматических) переменных разных функций могут совпадать.

Область действия внешней (со спецификатором класса памяти extern) переменной или функции распространяется от места ее декларирования до конца файла, в котором они расположены. Если же на внешнюю переменную требуется сослаться до того, как она определена, или если она определена в другом файле, то при ее декларировании используется слово **extern**.

Если определения переменных (например, `int i; float mas[5];`) расположены вне всех функций, то они определяют внешние переменные `i` и `mas`, т. е. декларирование и выделение памяти под эти объекты выполнено для остальной части файла исходного кода.

Если определение переменных выполнено со спецификатором класса памяти extern (например, **`extern`** `int i;` **`extern`** `float mas[];`), то объекты данных `i` (переменная) и `mas` (массив), имеющие тип int и float, не создаются, а память им выделяется при их объявлении в какой-то другой функции.

Для всего множества функций, образующих программу, должно быть не более одного определения каждой из внешних переменных. В объявлении массивов необходимо указывать их размерность, однако при декларировании массива со спецификатором класса памяти extern это не требуется.

Передача массива в функцию. Если в качестве передаваемого аргумента используется массив, то в функцию передается указатель на массив. При вызове функции в списке аргументов записывается имя массива, являющееся адресом первого элемента.

Можно использовать три варианта описания массива в качестве формального параметра функции:

1 Параметр в функции может быть объявлен как массив соответствующего типа с указанием его размера (листинг 2.23).

Листинг 2.23 – Пример функции с параметром-массивом
```
1   int fun_max(int massiv[10]) //Определение функции fun_max
2   {
3   int k=massiv[0];
4   for (int i=1; i<10; i++)
5     if(k<massiv[i])
…     k=massiv[i];
10  return k;
11  }
12
13  main()
14  {
15  int mas[3][10],i,j;
16  for(i=0; i<3; i++)
17    {
18      for(j=0; j<10; j++)
19        scanf("%d"&mas[i][j]);
20  printf("\n Макс. значение в строке %d=%d ", i,fun_max(mas[i]));
21    }
22  }
```

2 Массив в качестве параметра в функции может быть объявлен без указания его размера; т. к. сам массив в стек не копируется, то размер массив в общем случае компилятору не требуется:

```
int fun_max(int massiv[]) //Определение функции fun_max
{
//Тело функции аналогично описанному в листинге 2.27
}
```

3 Объявление параметра-массива указателем на соответствующий тип данных.

```
int fun_max(int *massiv[]) //Определение функции fun_max
{
//Тело функции аналогично описанному в листинге 2.27
}
```

Во всех трех случаях в функцию передается адрес нулевого элемента массива, т. е. указатель соответствующего типа.

Указатели на функцию. В языках программирования C/C++ функция не может быть значением переменной, в то же время аналогично переменным функция физически расположена в памяти и соответственно имеет адрес. Таким образом, ее адрес, присвоенный указателю, является точкой входа в функцию.

Указатель на функцию может быть использован:

– для вызова функции вместо ее имени;

– для передачи функции как обычный параметр в другую функцию.

Следовательно, с указателем на функцию можно обращаться, как с переменной (передавать его другим функциям, помещать в массивы и т. д).

Указатель на функцию – это такой тип переменной, которой можно присваивать адрес точки входа в функцию, т. е. адрес первой исполняемой команды. Эта переменная в дальнейшем может использоваться для вызова функции вместо ее имени.

Определение указателя на функцию имеет следующий общий вид:

```
тип_результата (*имя_указателя_на_функцию) (список_типов_пара-
метров);
```

Пример объявления указателя на функцию fun:

```
int (*fun) (int, int*);
```

В примере переменная fun является указателем на функцию с двумя параметрами: типа int и указателем на тип int. Сама функция должна возвращать значение типа int. Круглые скобки, в которые заключены имя указателя fun и признак указателя *, являются обязательными, иначе запись без скобок будет интерпретироваться как объявление функции fun возвращающей указатель на тип int. Это следует из того, что приоритет префиксного оператора * ниже, чем приоритет скобок:

```
int *fun (int, int *); //Неверная запись
```

Вызов функции возможен только после инициализации значения указателя fun и имеет вид:

```
(*fun) (i,&j);
```

В этом выражении для получения адреса функции, на которую ссылается указатель fun, используется операция разадресации (*).

Функции с переменным числом параметров. В С++ можно использовать и функции с переменным числом параметров, т. е. функции, в которые можно передавать данные, не описывая их в прототипе и заголовке функции.

Описания этих данных заменяются тремя точками. В таких функциях может находиться также один или более постоянный параметр (признак), с помощью которого могут считываться данные.

Если в функции имеется несколько постоянных параметров, то сначала перечисляются эти параметры, а затем ставятся три точки.

В функциях с переменным числом параметров должен быть хотя бы один фиксированный параметр.

Возможны два способа задания длины переменного списка параметров:

– указание числа аргументов, передаваемых в функцию;

– задание признака конца списка аргументов.

Реализовать функции с переменным числом параметров можно тремя способами:

– используя указатель пустого типа, например `void*`;

– используя указатель, соответствующий типу переменных списка параметров, например `int*, double*`;

– используя указатель, определенный самой системой программирования (с помощью стандартных макросов `vиa_list, va_start, va_arg, va_end`.).

2.23 Рекурсия

Рекурсивный объект – это объект, частично состоящий или определяемый с помощью самого себя.

Классическим примером рекурсивного объекта является факториал числа n – это произведение целых чисел от 1 до n.

Факториал числа n обозначается $n!$

Согласно определению $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-1) \cdot n$.

Приведенное выражение можно переписать так:

$n! = n \cdot ((n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1) = n \cdot (n-1)!$

То есть факториал числа *n* равен произведению числа *n* на факториал числа *n* – 1. В свою очередь, факториал числа *n* – 1 – это произведение числа *n* – 1 на факториал числа *n* – 2 и т. д.

Таким образом, если вычисление факториала *n* реализовать как функцию, то в теле этой функции будет инструкция вызова функции вычисления факториала числа *n* – 1, т. е. функция будет вызывать сама себя. Такой способ вызова называется *рекурсией*.

Рекурсивная функция – это функция, которая вызывает сама себя.

Рекурсивные функции имеют широкое применение в программирование, например, через рекурсию можно организовать алгоритм быстрой сортировки массива, вызывая одну и ту же функции для заданного массива, а затем для разделенных подмассивов и т. д.

Пример рекурсивной функции вычисления факториала представлен в листинге 2.24.

Листинг 2.24 – Пример рекурсивной функции вычисления факториала

В синтаксисе Delphi	В синтаксисе C/C++
1 **function** factorial	1 int factorial(int n)
2 (n:integer):integer;	2 {
3 **begin**	3 **if** (n==0) **return** 1;
4 **if** n <> 1 **then**	4 **else**
5 factorial:= n*factorial(n-1)	5 **return** i*factorial(n-1);
6 //функция вызывает сама себя	6 //функция вызывает сама себя
7 **else** factorial:=1;	7 }
8 //рекурсивный процесс	8
9 //закончен	9
10 **end**;	10

2.24 Составные пользовательские типы данных. Записи (структуры)

Составные пользовательские типы данных представляют собой структуры данных, определяемые и создаваемые программистом. Такой структурой данных, состоящей из других разнородных данных (объектов данных разных типов, причем как скалярных, так и других составных), объединенных под одним именем, является запись.

Запись – это составной тип данных (структура данных), состоящий из отдельных именованных элементов любых (и возможно разных) типов (за исключением функций), объединенных под одним идентификатором.

Поле записи – идентификатор отдельного элемента записи.

В языке программирования Delphi такой составной пользовательский тип данных так и называется записью и обозначается служебным словом `record`. В языках программирования С/С++ данные типа «запись» называются структурой и обозначаются служебным словом `struct`.

При разработке программ записи помогают в организации хранения и обработке сложных (разнородных) данных, не разобщая их по различным объектам, а группируя в одном.

Кроме того, элементы, входящие в запись, сами могут иметь тип записи, т. е. такие структуры могут быть вложенными.

Записи позволяют группу связанных между собой переменных использовать как множество отдельных элементов, а также как единое целое. Записи целесообразно использовать там, где необходимо объединить разнообразные данные, относящиеся к одному объекту.

В отличие от массива, который также является составным пользовательским типом данных, но при этом однородным объектом, запись может быть неоднородной, т. е. содержать данные разных типов.

Как и массив, запись представляет собой совокупность данных, но отличается способом обращения к элементам: в массиве обращение осуществляется по индексу элементов, а к полям (элементам) записи необходимо обращаться по имени (идентификатору поля).

Само по себе объявление записи не влечет за собой выделение памяти. По сути, это является созданием шаблона пользовательского типа данных. После объявления записи для использования в дальнейшем такого созданного программистом типа данных необходимо определить переменную, имеющую тип созданной записи.

Описание структуры записи предоставляет компилятору необходимую информацию об элементах переменной-записи (структурной переменной) для резервирования места в оперативной памяти и организации доступа к ней при последующем определении переменной-записи и использовании отдельных элементов (полей) переменной-записи.

Записи в языке программирования Delphi могут быть созданы двумя способами:

1 Объявление нового типа и определение переменной созданного типа. Создаваемый программистом тип данных «запись» должен быть объявлен в разделе объявления типов данных **type** и иметь следующий общий вид:

```
type
идентификатор_создаваемого_типа = record
   поле_1: тип_1;
   поле_2: тип_2;
…
   поле_N: тип_N;
end;
```

где поле_*i* – идентификатор *i*-го элемента (поля) записи ($i = 1…N$);

тип_*i* – тип *i*-го элемента (поля) записи ($i = 1…N$), причем в общем случае тип каждого поля может быть разным, но разные поля могут быть и одного и того же типа данных (как скалярного, так и составного);

N – количество элементов (полей) записи.

После объявления записи для использования данного созданного пользователем типа необходимо определить переменную в разделе **var**:

```
var идентификатор_переменной: идентификатор_созданного_типа;
```

Данный способ удобен тем, что созданный тип можно использовать в любых функциях и процедурах файла программы, определяя переменные в необходимых подпрограммах.

2 Совмещенное объявление типа и определение переменной данного типа. В данном случае это происходит в разделе объявления переменных **var**, но использование данной записи возможно только в той подпрограмме, в которой он объявлен.

Примеры объявления типа записи и определения переменной-записи, а также совмещение объявления и определения представлены в листинге 2.25.

Листинг 2.25 – Пример создания пользовательского типа «запись»

Объявление типа записи и определение переменной-записи	Совмещенное объявление и определение записи
1 **type**	1 **var**
2 TPerson = **record**	2 student: **record**
3 familiya:**string**[25];	3 familiya:**string**[25];
4 imya:**string**[25];	4 imya:**string**[25];
5 vozrast:real;	5 vozrast:real;
6 nomer:integer;	6 nomer:integer;
7 **end**;	7 **end**;
8	8
9 **var**	9
10 student:TPerson;	10

Обращение к элементам записи (полям) осуществляется через оператор «точка», перед которым идет имя переменной записи, а после точки – имя поля:

```
student.imya:='Андрей';
student.nomer:=381573;
```

Использование оператора присоединения **with** позволяет убрать префикс для каждого упоминания элементов переменной со сложным именем:

```
with student do
begin
   imya:='Андрей';
   nomer:=381573;
end;
```

В языках программирования C/C++ объявление структуры (т. е. записи) осуществляется с помощью служебного слова **struct**, за которым идет идентификатор (имя) структуры и далее список элементов, заключенных в фигурные скобки, состоящий из типа элемента и идентификатора (имени) поля:

```
struct идентификатор_структуры
{
  тип_элемента_1 идентификатор_элемента_1;
  тип_элемента_2 идентификатор _элемента_2;
  ...
  тип_элемента_N идентификатор _элемента_N;
};
```

Пример объявления структуры:

```
struct kartoteka
{
char fio[40];
int nomer;
int vozrast;
};
```

В данном случае просто создана структура и описаны ее элементы. В данной записи еще нет определения переменных или массивов данной структуры, т. е. просто создан новый тип данных `kartoteka`.

Идентификаторы (имена) структур должны быть уникальными в пределах их области видимости (действия), для того чтобы компилятор мог различать различные типы шаблонов структур.

Именем элемента структуры (поля) может быть любой идентификатор. Как и ранее, для скалярных типов, описание нескольких идентификаторов одного типа в структуре может производиться через запятую:

```
struct biblio {char fam[40], imya[20], otch[20]; int nomer;};
```

Имена элементов (полей) структуры могут совпадать с именами обычных переменных (не являющихся элементами структуры), т. к. они всегда различимы по контексту:

```
struct biblio {char fam[40], imya[20], otch[20]; int nomer;};
char fam[40], imya[20], otch[20]; //Обычные переменные
```

Более того, одни и те же имена элементов (полей) могут встречаться в объявлениях различных структур:

```
struct kartoteka {char fio[40]; int nomer; int vozrast;};
struct biblioteka{char fio[40]; int nomer; int vozrast;};
```

Определение переменной, имеющей тип «структура» (структурная переменная) аналогично определению переменных скалярных типов:

```
//Объявление типа структуры biblioteka:
struct biblioteka{char fio[40]; int nomer; int vozrast;};
//Объявление переменной stud типа структуры biblioteka:
biblioteka stud;
```

Описание шаблона типа «структура» и определение структурной переменной могут быть совмещены в одной записи, например:

```
struct bib {char fio[40]; int nomer; int vozrast;} stud, prep;
```

Также структурной переменной может быть массив:

```
struct computers {char nazv[20];float speed;int cena;} comp[3];
```

Как и обычные переменные, элементы, входящие в состав структуры (поля структуры), могут быть инициализированы. При этом можно инициализировать, как все элементы (поля), определяющие структурную переменную при ее декларировании:

```
struct st {char c; int i, j};
struct st a={'a', 105, 25};
```

так и отдельные из них в процессе работы с ними, используя оператор «точка»:

```
a.c='a';
a.j=25;
```

При выполнении инициализации структурных переменных необходимо следовать следующим правилам:

– присваиваемые значения должны совпадать по типу с соответствующими полями структуры;

– можно указывать меньшее количество присваиваемых значений, чем количество полей. Компилятор присвоит нули остальным полям структуры:

```
struct st {char c; int i, j}; //Объявление структуры
struct st a={'a'};//Инициализация поля c, полям i и j будут
//присвоены нулевые значения
```

– список инициализации последовательно присваивает значения полям структуры вложенных структур и массивов:

```
struct computers {char nazv[20]; float speed; int cena;}
comp[3]={{ "Celeron", 2.6, 525},
         {"Duron", 2.2, 547},
         {"Athlon", 3.0, 610}};
```

Доступ к полям структуры осуществляется с помощью оператора «точка» при непосредственной работе со структурой или при помощи оператора получения косвенного доступа "->" (минус и знак больше) – при использовании указателей на структуру.

Возможны три вида спецификации доступа к элементам (полям) структуры:

а) `Имя_переменной_структуры . имя_поля;` – *прямой доступ к полю;*

б) `Имя_указателя -> имя_поля;` } – *доступ по указателю*

в) `(*имя_указателя) . имя_поля;` }

Пример:

```
struct str {int i; float j;} st1, st2[3], *st3, *st4[5];
```

Прямой доступ к полю (через оператор «точка»):

```
st1.i=12;
st2[1].j= 0.53;
```

Доступ по указателю (через оператор «->»):

```
st3->i=12;
(st4+1)->j=1.23;
(*st3).j=23;
(*st4[2]).i=123;
```

Следует отметить, что шаблон структуры в качестве одного из полей не может содержать переменные своего же типа:

```
struct struktura1
{ char pole_1;
  double pole_2;
  struktura1 pole_3; //Недопустимо!
};
```

Однако структура может включать в себя элементы, являющиеся указателями на шаблон этой же структуры:

```
struct struktura2
{ char pole_1;
  double pole_2;
  struktura2 *pole_3; //Допустимо
};
```

Вложенные структуры. Структуры могут быть вложены друг в друга, т. е. одна структура может быть типом поля другой структуры:

```
struct FIO {char F[15], I[15], O[15]} f;
struct rabota {struct FIO fam; int nomer; float zarpl} rb;
```

Структура rabota содержит структуру FIO. Доступ к элементам структуры FIO и rabota осуществляется следующим образом:

```
f.F= "Фамилия1";
f.I = "Имя1";
rb.fam.O = "Отчество1";
rb.zarpl = 2580.73;
```

Операции над структурами. Единственно возможные операции над структурными переменными:

1 Копирование. Копирование всех полей одной структуры в другую может быть выполнено как поэлементно:

```
struct book {char avt[10]; int izd;} st1,st2;
…
st2.avt = st1.avt; //Копия поля avt из переменной st1 в st2
```

так и целиком:

```
struct book {char avt[10]; int izd;} st1,st2;
…
st2=st1; //Копия всех полей структурной переменной st1 в st2
```

Полное копирование структурной переменной (`st2=st1`) допустимо, только если `st1` и `st2` являются структурными переменными, соответствующими одному структурному типу.

2 Присваивание. Оператор присваивания выполняет то, что называется поверхностной копией в применении к структурным переменным.

Поверхностная копия представляет собой копирование бит за битом значений полей переменной-источника в соответствующие поля переменной-приемника. При этом может возникнуть проблема с такими полями, как указатели.

3 Взятие адреса с помощью оператора &.

4 Осуществление доступа к элементам (полям) структурной переменной.

Структурные переменные нельзя сравнивать друг с другом, т. е. выражение `if (st1==st2)` неверно, поскольку сравнение структур требуется выполнять поэлементно.

Указатели на структуры. Как и для других структурированных типов данных, указатели могут быть использованы и для структур. Это имеет следующие достоинства:

– указателями на структуры легче пользоваться, чем самими структурами (например, в задаче сортировки);

– ряд данных удобно представлять в виде структур, полями которых являются указатели на другие структуры.

Описание указателя на структуру имеет следующий общий вид:

```
struct имя_структуры *имя_указателя;
```

Пример использования указателя на структуру приведен в листинге 2.26.

Листинг 2.26 – Пример использования указателя на структуру

```
1   struct name {char fio[30];};
2   struct inform {struct name familiya; char prof[30]; int god;};
3   //Структура inform, первое поле которой типа структуры name
4   static struct inform mas[2]={"Фамилия1","Программист",1989,
5                               "Фамилия2","Дизайнер",1993};
…   //Инициализация структурной переменной-массива mas из 2 эл-тов
10  struct inform *him; //Указатель на структуру типа inform
11  him=mas; //him содержит адрес нулевого элемента mas[0]
12  him++; //him содержит адрес следующего (первого) элемента
13  //структурной переменной-массива mas
14  ++him->god; //Увеличение значения поля god первого элемента
15  // структурной переменной-массива mas на единицу
```

Таким образом, раз `him` – это указатель на структуру `inform`, то `*him` есть сама структура, а `(*him).god` – значение третьего поля структурной переменной.

В выражении `(*him).god` используются скобки, т. к. приоритет операции «точка» (.) выше, чем у операции разадресации (`*`).

Для упрощения доступа к элементам структуры и избежания ошибок, связанных с учетом приоритета, в языках программирования С/С++ была введена операция косвенного получения элемента (->). Эта операция имеет следующую общую форму записи:

```
имя_указателя -> имя_поля_структуры;
```

Оператор косвенного получения элемента состоит из двух знаков «-» и «>». Операция -> имеет наивысший приоритет наряду с операциями «()» и «[ ]».

Чтобы перейти к полю god следующей структуры, необходимо записать:

```
(++him)->god;
//или
(him++)->god;
```

Примеры использования операторов инкрементации с указателями при разных приоритетах выполнения:

```
     struct noname {char *str; float f;} *pr;
     *pr->str //есть значение объекта, на который ссылается str;
     *pr->str++ //передвинет указатель pr->str после считывания
//объекта, на который он указывал
     (*pr->str)++ //увеличит значение объекта, на который ссылается
//указатель str
     *pr++->str //передвинет указатель pr после получения значения,
//на которое указывает указатель str
```

2.25 Файлы

Файл – это именованная область данных на носителе информации, представляющая собой последовательность элементов данных одного типа.

Файлы используются для размещения данных, предназначенных для длительного хранения в энергонезависимой памяти компьютера.

Доступ к файлу из программы осуществляется посредствам файловой переменной (Delphi) или указателя (C/C++).

Определение файловой переменной в языке программирования Delphi задает только тип компонентов (данных) файла. Запись данных в файл или их чтение из файла возможно после связывания файловой переменной с конкретным файлом в пространстве памяти.

Каждый файл имеет уникальное имя в файловой системе, по которому происходит обращения именно к нему.

В языке программирования Delphi различают три типа файлов:

1 ***Текстовые*** – это файлы, содержащие данные символьного (char) или строкового (string) типа данных. Текстовые файлы являются структурой данных с последовательным доступом.

2 ***Типизированные*** – это файлы содержащие данные всех остальных типов (чаще всего типа «запись»), кроме символьного и строкового, динамических массивов и других файлов. Все компоненты типизированного файла имеют свой порядковый номер. Таким образом, типизированные файлы являются структурой данных с прямым доступом.

3 ***Нетипизированные*** – это файлы без указания типа данных, которые будут храниться в нем. Нетипизированный файл представляет собой последовательность байтов, записанных блоками, длина которых задается при записи или чтении специальными функциями, которые будут рассмотрены далее.

Связывание с конкретным файлом осуществляется через файловую переменную, которая объявляется в разделе объявления переменных **var**.

Примеры объявления файловых переменных различных типов файлов на языке программирования Delphi представлены в листинге 2.27.

Листинг 2.27 – Примеры объявления файловых переменных языка Delphi

```
1   type
2       vec=array[1..10] of byte; //Объявления типа vec, являющегося
3                  //целочисленным массивом размером 10 элементов
4       mnoz=set of char; //Объявление типа символьного множества
5       kadry=record //Объявление типа запись
6         fam: string;
7         imya:string[20];
8         god:integer;
9       end;
10  var
11     f1: TextFile; //Файловая переменная текстового файла
12     f2: File of extended; //Файловая переменная типизир-го файла
```

```
13      f3: File of char; //Файловая переменная типизированного файла
14      f4: File of vec; //Файловая переменная типизированного файла
15      f5: File of mnoz; //Файловая переменная типизированного файла
16      f6: File of kadry; //Файловая переменная типизир-го файла
17      f7: File; //Файловая переменная нетипизированного файла
```

Для связи файловой переменой с файлом любого типа используется процедура `AssignFile()`, имеющая два параметра: идентификатор файловой переменной и имя файла для связывания с расширением (расширение может быть в том числе пользовательским):

```
AssignFile(var FileHandle:TextFile, const FileName:string);
AssignFile(var FileHandle:File, const FileName:string);
```

Примеры связывания файловых переменных из листинга 2.27 с конкретными файлами:

```
AssignFile(f1,'список.txt');
AssignFile(f6,'C:\Programs\personal.dat');
AssignFile(f7,'system.aaa'); //Пользовательское расширение
```

В первом и третьем случаях указан укороченный путь к файлу (только имя файла и расширение), поскольку исполняемый файл программы и связываемые файлы находятся в одной директории. Во втором случае указан полный путь к файлу (полное имя, включающее все вложенные директории и имя файла).

После связывания программы с конкретным файлом через файловую переменную можно выполнять различные манипуляции с данными (чтение, запись, поиск) посредством специальных процедур и функций.

Процедуры и функции для работы с файлами в языке Delphi.

Для работы с **текстовыми файлами**, связанными файловой переменной (в списке обозначена как `ft`) предусмотрены следующие процедуры и функции:

1) `Rewrite(ft);` Открывает (если уже существует) или создание текстовый файл для записи;

2) `Append(ft);` Открывает текстовый файл для дозаписи в конец файла;

3) `Reset(ft);` Открывает текстовый файл для чтения;

4) `Writeln(ft,S,…,Sn);` Запись компонентов (строк) *S*, …, *Sn* в текстовый файл, с признаком конца строки (компонента);

5) `Readln(ft,S,…,Sn);` Чтение компонентов (строк) *S*, …, *Sn* из текстового файла до признака конца строки (компонента);

6) `EOLn(ft);` Возвращает значение положения указателя: true, если достигнут признак конца строки файла, false – если не достигнут;

7) `SeekEoln(ft);` Возвращает true, если пройден последний значимый символ в строке, отличный от пробела и знака табуляции;

8) `SeekEof(ft);` Возвращает true, если пройден последний значимый символ в текстовом файле, отличный от пробела и знака табуляции.

Для работы с **типизированными файлами**, связанными файловой переменной (в списке обозначена как `f`), предусмотрены следующие процедуры и функции:

1) `Rewrite(f);` Создает новый файл для работы с ним;

2) `Reset(f);` Инициирует ранее созданный файл для работы. Указатель устанавливается на первый компонент (с нулевым номером);

3) `Write(f,A,…,Z);` Запись компонентов *A*, …, *Z* в файл;

4) `Read(f,A,…,Z);` Чтение компонентов *A*, …, *Z* из файла;

5) `EOF(f);` Возвращает значение положения указателя: true, если достигнут конец файла, false – если не достигнут;

6) `Seek(f,X);` Устанавливает указатель на компонент номер *X*;

7) `FilePos(f);` Возвращает положение указателя;

8) `FileSize(f);` Возвращает количество компонентов файла;

9) `Truncate(f);` Удаляет компоненты, начиная с того, на который установлен указатель.

Для работы с **нетипизированными файлами**, связанными файловой переменной (в списке обозначена как `fn`), предусмотрены следующие процедуры и функции:

1) `Rewrite(fn,X);` Создает новый файл для работы, задавая размер блока *X* в байтах. Если длина блока не указана, она принимается равной 128 байтам;

2) `Reset(fn,X);` Инициирует ранее созданный файл для работы, где *X* – размер блока в байтах. Если длина блока не указана, она принимается равной 128 байтам;

3) `BlockWrite(fn,buf,count,Z);` Где `buf` – имя переменной, которая будет записана; `count` – количество записей, которые должны быть записаны за одно обращение к файлу, `Z` – необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей;

4) `BlockRead(fn,buf,count,Z);` Где `buf` – имя переменной, которая будет прочитана; `count` – количество записей, которые должны быть прочитаны за одно обращение к файлу, `Z` – необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей.

Общие для файлов, связанных файловой переменной (в списке обозначена как `f`) любого типа, процедуры и функции:

1) `Erase(f);` Удаляет неоткрытый файл любого типа;

2) `Rename(f,new_name);` Переименовывает неоткрытый файл, второй параметр задает новое имя файла;

3) `IoResult;` Возвращает код результата последней операции ввода-вывода;

4) `CloseFile(f);` Закрывает открытый ранее файл.

Файл любого типа, открытый для работы любой процедурой, должен быть закрыт при помощи процедуры `CloseFile()`, иначе это может привести к потере данных.

Примеры использования процедур и функций для работы с файлами представлены в листинге 2.28.

Листинг 2.28 – Примеры работы с файлами в языке программирования Delphi

```
1   var
2      f1: TextFile; //Файловая переменная текстового файла
3      f2: File of extended; //Файловая переменная типизированного файла
4      f3: File; //Файловая переменная нетипизированного файла
5      s:string; //Переменная строкового типа
6      p,q:integer; //Целочисленные переменные p,q
7      x1,x2:extended; //Вещественные переменные x1,x2
8   begin
9   s:='Вторая строка'; //Инициализация переменной s
10  p:=3; //Инициализация переменной p
11  x1:=0.73; x2:=7.9; //Инициализация переменных x1, x2
12  AssignFile(f1,'Текстовый.txt'); //Связывание файловых
13  AssignFile(f2,'Типизированный.typ'); //переменных с конкретными
14  AssignFile(f3,'Нетипизированный.ntp'); //файлами
15  Rewrite(f1); //Создание текстового файла
16  WriteLn(f1,'Первая строка', s); //Запись строк в текстовый файл
17  CloseFile(f1); //Закрытие текстового файла
18  Append(f1); //Открытие текстового файла для дозаписи в конец
19  WriteLn(f1,'Последняя строка =',p); //Запись в текстовый файл
20  Reset(f2); //Открытие типизированного файла
21  Seek(f2,FileSize(f2)+1); //Установка указателя в конец файла
22  Write(f2,x1,x2); //Запись компонентов в файл
23  CloseFile(f2); //Закрытие типизированного файла
24  Rewrite(f3,32); //Создание нетипизированного файла с блоком 32б
25  BlockWrite (f3,p,1); //Запись 1 компонента p в файл
26  Reset(f3,32); //Открытие нетипизированного файла для чтения
27  BlockRead(f3,q,1);// Чтение 1 компонента q размером 32б из файла
28  Writeln('Компонент = ',q); //Функция вывода. Будет выведено 3
29  CloseFile(f3); //Закрытие нетипизированного файла
30  Rename(f3,'C:\Programs\Новое имя.ntp');//Переименование
31  //нетипизированного файла (смена директории и имени)
32  end;
```

В языках программирования С/С++ для выполнения операций с файлами используется указатель на файл.

Указатель на файл – это переменная, идентифицирующая конкретный дисковый файл (его адрес), использующаяся для организации ввода-вывода данных в/из файл(а).

Указатель на файл имеет вид: `FILE *fp;`

В языках программирования С/С++ различают два типа файлов:

1 Текстовые файлы. Хранят информацию в виде последовательности символов. В текстовом режиме каждый разделительный символ строки автоматически преобразуется в пару (возврат каретки – переход на новую строку).

2 Бинарные (или двоичные) файлы. Предназначены для хранения только числовых значений данных. Структура такого файла определяется программно.

Для выполнения манипуляций с данными файла (чтение, запись, поиск данных) его сначала необходимо открыть с помощью функции `fopen()`.

Функция `fopen()` открывает для использования поток, связывает файл с данным потоком и возвращает указатель на открытый файл, а также определяет режим работы с файлом.

Функция `fopen()` имеет следующий прототип:

```
FILE *fopen (const char *"физическое_имя_файла", const char *"режим");
```

В случае если спецификация файла задана неверно, то функция fopen() возвращает указатель со значением NULL.

Основные режимы работы с файлами:

– `"r"` − открытие текстового файла для чтения;

– `"w"` − создание текстового файла для записи;

– `"a"` − добавление информации в конец файла (дозапись в конец файла).

При работе с текстовыми файлами к символу, указывающему режим открытия, добавляется символ `"t"`, а при работе с бинарными – символ `"b"`.

Если необходимо и читать данные из файла и записывать их в файл, то к режиму открытия добавляется символ «+».

Все возможные комбинации режимов открытия файлов представлены в таблице 2.34.

Таблица 2.34 – Режимы открытия файлов в С/С++

Спецификация режима открытия	**Действие**
r	Открывает файл для чтения
w	Создает файл для записи
a	Открывает файл для дозаписи в конец
rb	Открывает бинарный файл для чтения
wb	Создает бинарный файл для записи
ab	Открывает бинарный файл для дозаписи в конец файла
r+	Открывает файл для чтения/записи
w+	Создает файл для чтения/записи
a+	Открывает или создает файл для чтения/записи
r+b	Открывает бинарный файл для чтения/записи
w+b	Создает бинарный файл для чтения/записи
a+b	Открывает или создает бинарный файл для чтения/записи

Продолжение таблицы 2.34

Спецификация режима открытия	Действие
rt	Открывает текстовый файл для чтения
wt	Создает текстовый файл для записи
at	Открывает текстовый файл для дозаписи в конец файла
r+t	Открывает текстовый файл для чтения
w+t	Создает текстовый файл для чтения/записи
a+t	Открывает или создает текстовый файл для чтения/записи

Если функция `fopen()` используется для открытия файла на запись, то любой ранее существующий файл с указанным именем будет удален. Если файла с указанным именем не существует, то он будет создан.

Если необходимо дописать информацию в конец файла, следует использовать режим `"a"` (добавления). Если файл не существует, то он будет создан.

Открытие файла на чтение требует наличия файла. Если файл не существует, то будет возвращена ошибка. Если файл открыт для операции чтения/записи, то он не удаляется при наличии, а если файл не существует, то он создается.

Открытый в любом режиме файл любого типа должен быть закрыт функцией `fclose()`, иначе это может привести к потере данных.

Функция `fclose()` разрывает связь указателя с файлом и закрывает его к дальнейшему использованию, до тех пор пока он вновь не будет открыт функцией fopen().

<u>Функции для работы с файлами языков программирования C/C++.</u>

1) `int *fprintf (FILE *указатель_на_файл, const char *"управляющая_строка";);` Записывает форматированные данные в файл. Управляющая строка определяет строку форматирования аргументов, заданных своими адресами. Обычно эта строка состоит из последовательности символов «%», после которых следует один из символов типа данных:

– `I` или `i` – десятичное, восьмеричное или шестнадцатеричное целое число;

– `D` или `d` – десятичное целое число;

– `U` или `u` – десятичное целое число без знака;

– `E` или `e` – вещественное число с плавающей точкой;

– `s` – строка символов;

– `c` – символ.

2) `int *fscanf (FILE *указатель_на_файл, const char *"управляющая_строка");` Читает форматированные данные из файла. Строка форматирования формируется аналогично функции `fprintf();`

3) `int fputs (const char *строка, FILE *указатель_на_файл);` Записывает строку символов в текущую позицию указанного открытого файла;

4) `char *fgets(char *строка, int длина, FILE *указатель_на_файл);` Читает строку символов из текущей позиции указанного открытого файла до тех пор, пока не будет прочитан символ перехода на новую строку, или количество прочитанных символов не станет равным длине строки минус единица;

5) `int putc (int символ, FILE *указатель_на_файл);` Записывает один символ в текущую позицию указанного открытого файла;

6) `int getc (FILE *указатель_на _файл);` Читает один символ из текущей позиции указанного открытого файла;

7) `size_t fwrite (const void *записываемое_данное, size_t размер_элемента, size_t число_элементов, FILE *указатель_на_файл);` Записывает в файл заданное число данных определенного размера. Размер данных задается в байтах. Тип `size_t` определяется как целое число без знака;

8) `size_t fread (void *считываемое_данное, size_t размер_элемента, size_t число_элементов, FILE *указатель_на_файл);` Считывает из файла указанное число данных заданного размера. Размер задается в байтах. Функция возвращает число прочитанных элементов. Если число прочитанных элементов не равно заданному, то при чтении возникла ошибка или встретился конец файла;

9) `int fseek(FILE *указатель_на_файл, long int число_байт (смещение), int точка_отсчета);` Устанавливает указатель в заданную позицию. Заданное количество байт отсчитывается от позиции, которая задается следующими макросами:

– `SEEK_SET` – начало файла;

– `SEEK_CUR` – текущая позиция;

– `SEEK_END` – конец файла.

10) `void rewind (FILE *указатель_на_файл);` Устанавливает указатель текущей позиции выделенного файла в начало файла;

11) `int feof (FILE *указатель_на_файл);` Возвращает отличное от нуля значение (true), если конец файла не достигнут, и нуль (false), если достигнут конец файла;

12) `int ferror (FILE *указатель_на_файл);` Определяет, произошла ли ошибка во время работы с файлом;

13) `int fcloseall (void);` Закрывает все открытые файлы. Возвращает количество закрытых файлов или EOF, если возникает ошибка.

Помимо доступа к файлу через указатель на него, можно использовать другой подход, основанный на использовании дескриптора (номера) файла.

Дескриптор файла – это логический номер файла для заданного потока.

Дескриптор файла имеет тип данных int (т. е. номер – это целое число). Как и в случае использования указателя на файл, при необходимости доступа к файлу через дескриптор файл должен быть открыт. Для открытия файла через его

дескриптор могут использоваться две функции: open() (для открытия существующего файла) и creat() (для создания нового файла).

Функция open() имеет следующий общий вид:

```
int open (const char *name, int flags, unsigned mode);
```

Функция open() открывает файл с именем *name* и устанавливает режим доступа к нему в соответствии со значением аргумента *flags*. Аргумент *flags* представляет собой комбинацию основного режима доступа и одного из модификаторов:

1) O_RDONLY – открыть файл только для чтения;
2) O_WRONLY – открыть файл только для записи;
3) O_RDRW – открыть файл для записи/чтения.

Помимо основных режимов открытия можно использовать их модификаторы или даже несколько, записав их после выбранного основного режима через оператор логического ИЛИ («|»). Существуют следующие модификаторы:

1) O_APPEND – устанавливает указатель в конец файла для дозаписи;
2) O_BINARY – открывается двоичный файл;
3) O_TEXT – открывается текстовый файл;
4) O_CREAT – если файл не существует, то создает его в соответствии с третьим параметром *mode* функции open();
5) O_EXCL – при использовании с модификатором O_GREAT (через оператор «|») не будет пересоздавать файл, если файл с таким именем уже существует;
6) O_TRUNC – существующий файл урезает до нуля, сохраняя его атрибуты.

Параметр ***mode*** требуется только при использовании модификатора режима O_CREAT и может принимает одно из следующих трех значений:

1) S_IWRITE – доступ по записи;
2) S_IREAD – доступ по чтению;
3) S_IWRITE | S_IREAD – доступ по записи/чтению.

Функция open() возвращает дескриптор файла типа int или «–1» при любой ошибке открытия файла.

Функция creat() создает новый файл через дескриптор и имеет вид:

```
int creat(const char *name, int mode);
```

При успешном выполнении создается файл с правами доступа, определяемыми параметром *mode*. Если функцией creat() создается уже существующий файл, то он урезается до нулевой длины, при этом создание уже существующего файла не является ошибкой.

Функция creat() возвращает дескриптор файла, если файл успешно создан (или пересоздан) или «–1» в случае возникновения ошибки.

Также при работе с файлом через дескриптор используются функции:

1) `int write(int fd, void *buf, unsigned length);` Функция записывает *length* байт из буфера *buf* в файл, определенный дескриптором файла *fd*. Возвращает число записанных в файл байт или «–1» при ошибке;

2) `int read (unsigned fd, void *buf, unsigned length);` Функция читает *length* байт в буфер *buf* из файла, определенного дескриптором файла *fd*. Возвращает число реально считанных литер, которое может быть меньше *length*, если встретился конец файла. Если произошла ошибка чтения, возвращается значение «–1»;

3) `long filelength (int дескриптор);` Возвращает длину файла с соответствующим дескриптором в байтах;

4) `close(int дескриптор);` Разрывает связь между файловым дескриптором и открытым файлом, т. е. закрывает файл.

5) `int fileno (FILE *указатель_на_файл);` Возвращает значение дескриптора указанного файла.

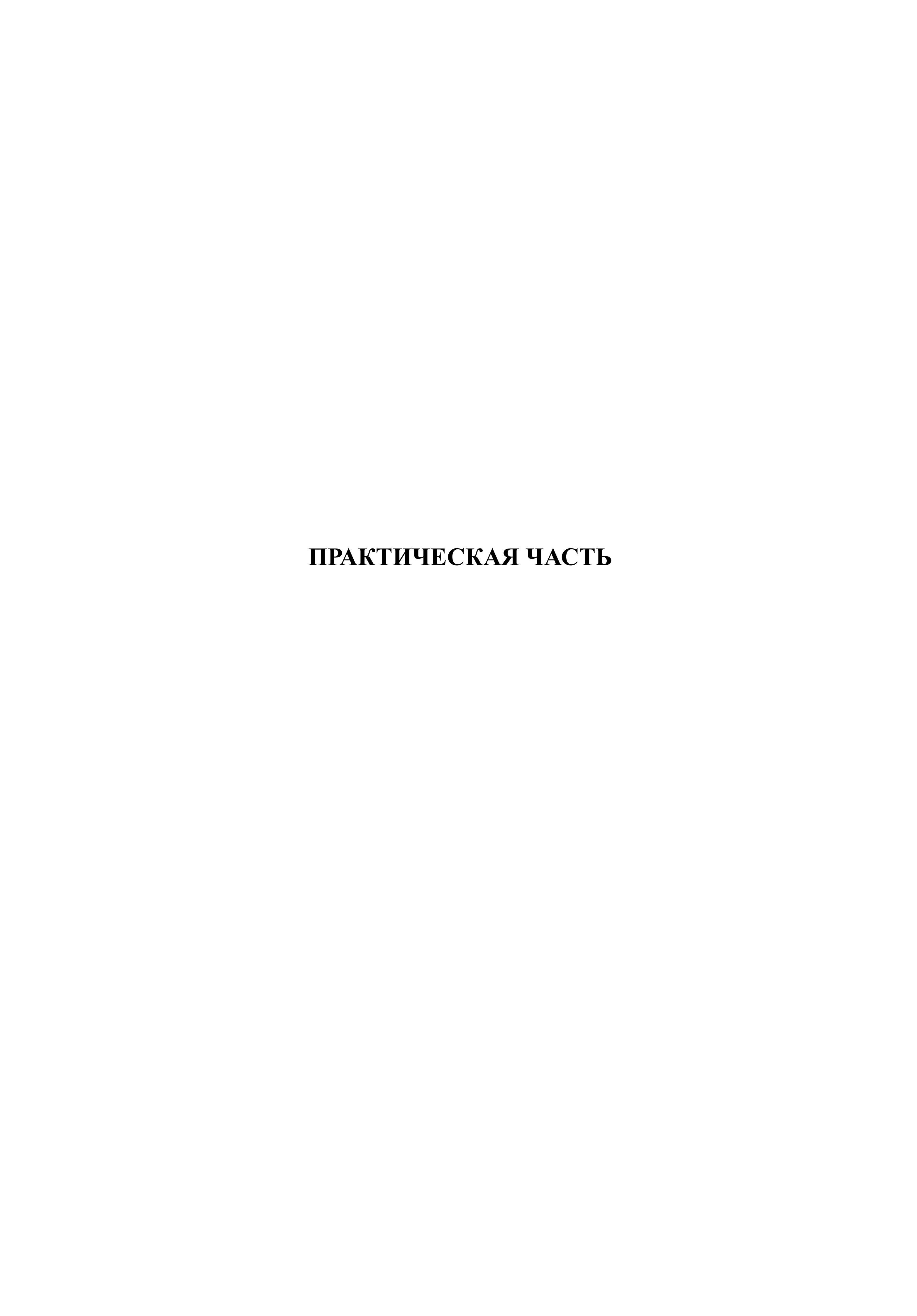
Примеры работы с файлами на языках C/C++ приведены в листинге 2.29.

Листинг 2.29 – Примеры работы с файлами на языках программирования C/C++

```
1   int main()
2   {
3     FILE *S1, *S2, *S3; //Указатели на файл
4     int x, y, des; //Целочисленные переменные
5     printf("Введите число: "); //Вывод текста на экран
6     scanf("%d", &x); //Ввод значения x с клавиатуры
7   //РАБОТА ЧЕРЕЗ УКАЗАТЕЛЬ
8     S1=fopen("S1.txt","w"); //Открытие файла S1 в режиме записи
9     fprintf(S1,"%d",x); //Запись значения x в файл S1
10    fclose(S1); //Закрытие файла S1
11    S1=fopen("S1.txt","r"); //Открытие файла S1 в режиме чтения
12    S2=fopen("S2.txt","w"); //Открытие файла S2 в режиме записи
13    fscanf(S1,"%d",&y); //Чтение значения y из файла S1
14    y*=5; //Увеличение значения y в 5 раз
15    fclose(S1); //Закрытие файла S1
16    fprintf(S2,"%d\n",y); //Запись нового значения y в файл S2
17    fclose(S2); //Закрытие файла S1
18  //РАБОТА ЧЕРЕЗ ДЕСКРИПТОР
19  des=open("S3.txt",O_RDWR | O_APPEND, 0); //Открытие на чтение
20  //и запись с установкой указателя в конец файла для дозаписи
21  write(des,x,32);//Запись значения x в файл с дескриптором des
22  close(des);//Закрытие файла с дескриптором des
23  }
```

Язык программирования C++ поддерживает приведенные выше функции, однако также имеет несколько другие подходы для работы с файлами, обусловленные парадигмой объектно-ориентированного программирования.

При открытии файла с ним связывается поток ввода-вывода. Выводимая информация записывается в поток, вводимая информация считывается из потока. Для работы с файлами в C++ необходимо подключить библиотеку `<fstream>`, содержащую классы `<ifstream>` для файлового ввода и `<ofstream>` для файлового вывода.



ПРАКТИЧЕСКАЯ ЧАСТЬ

Общие рекомендации по выполнению лабораторных работ

Лабораторные работы рекомендуется выполнять по следующему алгоритму.

1 Изучить материалы лекций, рекомендованную литературу и теоретическую часть данного издания.

2 Ответить на контрольные вопросы для самоподготовки по теме.

3 Выполнить индивидуальные задания в соответствии с целями работы, применяя обязательный учебный элемент.

4 Подготовить отчет по выполненному заданию.

5 Представить отчет преподавателю и защитить свою работу.

Отчет по каждой лабораторной работе подготавливается отдельно и предоставляется преподавателю до защиты. Отчет оформляется в текстовом редакторе (например, в MS Word) в соответствии с типовой формой оформления отчета к лабораторной работе.

Титульный лист должен выполняться по стандарту БГУИР и содержать:

– название университета и структурного подразделения (кафедры);
– наименование дисциплины;
– заголовок лабораторной работы (номер и название);
– номер группы и ФИО студента;
– дату выполнения (представления к защите) лабораторной работы.

Каждый отчет должен содержать:

– формулировку общей задачи и индивидуального задания;
– описание и/или графическую схему решения задачи;
– листинг кода решения задачи;
– экранные формы с демонстрацией работы программы;
– контрольные примеры (тестовые наборы), которые демонстрируют работу программы с предусмотренными и непредусмотренными входными данными, сообщения о критических ситуациях;
– дополнительные сведения, которые указаны в задании к лабораторной работе.

Защита проводится по каждой работе индивидуально (если иное не указано в задании) в форме беседы студента с преподавателем. По результатам защиты лабораторной работы студенту выставляется отметка по 10-балльной шкале. При получении студентом оценки 4 балла и выше работа считается защищенной. Оценка за лабораторную работу формируется по совокупности критериев выполнения требований к отчету и защите.

Все лабораторные работы должны быть выполнены и защищены до промежуточной аттестации по дисциплине.

ЛАБОРАТОРНАЯ РАБОТА № 1

Разработка и программирование алгоритмов для разветвляющихся вычислительных процессов

Цель: изучение основ языка высокого уровня на примере C/C++ (типизация, синтаксис и семантика), изучение возможностей языка C++ по организации разветвляющихся вычислительных процессов; получение навыков разработки и отладки консольных приложений в среде Visual Studio.

Контрольные вопросы для самоподготовки

1 Понятия «решение», «проект» и «конфигурация» в MS Visual Studio.

2 Возможности рабочего пространства проекта.

3 Этапы построения консольного приложения.

4 Понятия «препроцессор» и «условная компиляция» в C++.

5 Назначение команд препроцессора `#include, #define.`

6 Понятия «оператор», «операнд», «идентификатор», «переменная» и «константа» в C++.

7 Приоритетность операций и порядок обработки выражений в C++.

8 Понятие типизации. Правила преобразования значений операндов из одного типа в другой в C++.

9 Порядок автоматического приведения типов в выражениях в C++.

10 Средства и особенности языка C++ при организации разветвляющегося процесса.

11 Возможности языка C++ для организации циклического процесса.

12 Особенности организации в C++ итеративного процесса. Цикл `for`.

13 Инструментальная панель отладчика в среде *Visual C++*.

14 Точки останова и точки трассировки. Настройки отладчика и пошаговое выполнение программы.

15 Возможности языка C++ по организации ввода и вывода данных для консольного приложения. Форматированный вывод данных.

Задание

1 Согласовать вариант задания с преподавателем, внимательно изучить исходные данные к задаче (таблица 1.1) и пример.

2 Разработать алгоритм для решения следующей задачи:

Вычислить бесконечную сумму **S** с точностью **ε**.

Считать, что требуемая точность достигнута, если значение очередного слагаемого по модулю меньше заданного **ε** (это и все последующие слагаемые можно не учитывать). Определить и вывести количество слагаемых найденной суммы, учтенных при расчете. Значение суммы выводить с точностью до восьми знаков в дробной части. Если ни одно из слагаемых не было учтено, то выдать об этом сообщение.

Внимание! Для нахождения степени числа не использовать стандартных функций, а вычислять его самостоятельно с помощью оператора цикла. Для вычисления факториала числа применить цикл `for`.

Ограничения: **x**, **ε** – действительные числа (**x** ≠ 0, **ε** > 0), **a** – целое число ($|\boldsymbol{a}| < 10^6$)

3 Написать программный код реализации составленного алгоритма с учетом требований и ограничений по индивидуальному заданию.

4 При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Организовать текстовый пользовательский интерфейс и форматированный вывод данных.

5 Проверить правильность вычислений на контрольных примерах, используя возможности отладчика. Организовать трассировку переменных и проследить изменение их значений в ходе выполнения программы, выполнить отладку цикла.

6 Построить укрупненную схему составленного алгоритма (блок-схему работы программы).

7 ***Дополнительно*** следует учесть, что правильность вводимых значений не гарантируется. Необходимо обеспечить проверку соответствия входных данных указанным в условии требованиям и ограничениям. При разработке программы предусмотреть пропуск слагаемых равных нулю или бесконечности.

8 ***Дополнительно*** для своего варианта определить диапазон возможных значений аргументов. Обосновать свое решение.

Таблица 1.1 – Варианты индивидуальных заданий для лабораторной работы № 1

Вариант	Задание	Дополнительные условия
0	$\boldsymbol{S} = \sum_{\mathbf{k=0}}^{\infty} \frac{(-\mathbf{1})^{\boldsymbol{k}} * \boldsymbol{x}^{\mathbf{2}\boldsymbol{k}+\mathbf{1}}}{\boldsymbol{b} * \boldsymbol{k}!}$	$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$

Продолжение таблицы 1.1

Вариант	Задание	Дополнительные условия
1	$S = b + \sum_{k=1}^{\infty}\left(cos(k*\frac{\pi}{4})*\frac{(x^k)}{k!}\right)$	$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$
2	$S = \sum_{k=1}^{\infty}\left(\frac{\cos(k*x)}{k^2}*b\right)$	$b = \begin{cases} a!, k - \text{нечетное} \\ 0, k - \text{четное} \end{cases}$
3	$S = \sum_{k=1}^{\infty}\left(\frac{\sin(k*x)}{k}*b\right)$	$b = \begin{cases} a!, k - \text{нечетное} \\ -1, k - \text{четное} \end{cases}$
4	$S = \sum_{k=0}^{\infty}\frac{(-1)^k*x^b}{b*(2k+1)!}$	$b = \begin{cases} k*a, \varepsilon < 1 \\ \frac{1}{a}, \varepsilon \geq 1 \end{cases}$
5	$S = \sum_{k=0}^{\infty}\left(\frac{(-1)^k}{(k!)^2}*\left(\frac{x}{b}\right)^{2k}\right)$	$b = \begin{cases} 2a, \varepsilon < 1 \\ 2, \varepsilon \geq 1 \end{cases}$
6	$S = \sum_{k=0}^{\infty}\left(\frac{(-1)^{k+1}}{(2k)!}*\left(\frac{x}{b}\right)^{4k}\right)$	$b = \begin{cases} a, \varepsilon < 1 \\ 3, \varepsilon \geq 1 \end{cases}$
7	$S = \sum_{k=0}^{\infty}\frac{(-1)^{k+1}*x^{2k}}{b*(2k)!}$	$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$
8	$S = b + \sum_{k=1}^{\infty}\left(\frac{(\lvert a-k\rvert+1)!}{k!*2k}*x^k\right)$	$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$
9	$S = \sum_{k=1}^{\infty}\frac{x^k}{k^3 + k\sqrt{\lvert x\rvert} + b}$	$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$

Продолжение таблицы 1.1

Вариант	**Задание**	**Дополнительные условия**
10	$$S = \sum_{k=0}^{\infty} \frac{(-1)^k * b * x^k}{3^k}$$	$$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$$
11	$$S = \sum_{k=0}^{\infty} \left(\frac{(-1)^k}{k! * (k+2)!} * \left(\frac{x}{b}\right)^{a+2k} \right)$$	$$b = \begin{cases} 2a, \varepsilon < 1 \\ 2, \varepsilon \geq 1 \end{cases}$$
12	$$S = \sum_{k=0}^{\infty} \frac{(-1)^k * x^{bk+1}}{(2k)! * (2k+1)}$$	$$b = \begin{cases} 2, \varepsilon < 1 \\ 2a, \varepsilon \geq 1 \end{cases}$$
13	$$S = \sum_{k=0}^{\infty} \left(\frac{(-1)^k}{k! * (k+1)!} * \left(\frac{x}{b}\right)^{2k} \right)$$	$$b = \begin{cases} 2a, \varepsilon < 1 \\ \frac{1}{a}, \varepsilon \geq 1 \end{cases}$$
14	$$S = \sum_{k=1}^{\infty} \left(\frac{(-1)^{k+1}}{(2k+1)!} * \left(\frac{x}{b}\right)^{4k} \right)$$	$$b = \begin{cases} 3a, \varepsilon < 1 \\ \frac{1}{a}, \varepsilon \geq 1 \end{cases}$$
15	$$S = \sum_{k=0}^{\infty} \frac{(-1)^k * x^b}{(k+1) * b!}$$	$$b = \begin{cases} (k+2), \varepsilon < 1 \\ 2a, \varepsilon \geq 1 \end{cases}$$
16	$$S = \sum_{k=0}^{\infty} \frac{x^{2k} * b}{2^k * k!}$$	$$b = \begin{cases} -1, k - \text{нечетное} \\ a, k - \text{четное} \end{cases}$$
17	$$S = \sum_{k=0}^{\infty} \frac{(-x)^{2k}}{b * k!}$$	$$b = \begin{cases} a, \varepsilon < 1 \\ 2, \varepsilon \geq 1 \end{cases}$$
18	$$S = b + \sum_{k=1}^{\infty} \left(\sin\left(k * \frac{\pi}{3}\right) * \frac{x^k}{k!} \right)$$	$$b = \begin{cases} \frac{1}{a}, \varepsilon < 1 \\ a!, \varepsilon \geq 1 \end{cases}$$

Продолжение таблицы 1.1

Вариант	Задание	Дополнительные условия
19	$\boldsymbol{S = \sum_{k=1}^{\infty} \left(x^k * \cos\left(k * \frac{\pi}{4}\right) * b\right)}$	$b = \begin{cases} a!\,, k - \text{нечетное} \\ 1, k - \text{четное} \end{cases}$
20	$\boldsymbol{S = \sum_{k=1}^{\infty} \left(b * \frac{\cos(k * x)}{k^2}\right)}$	$b = \begin{cases} 1, k - \text{нечетное} \\ -1, k - \text{четное} \end{cases}$

Пример выполнения лабораторной работы

Программная реализация алгоритма решения задания для 0-го варианта представлена в листинге 1.1. Экранные формы работы программы, демонстрирующие работу алгоритма, представлены на рисунке 1.1.

В таблице 1.2 отображены результаты выполнения программы на различных входных данных для проверки корректной обработки краевых значений и критических ситуаций.

На рисунке 1.2 приведено графическое представление алгоритма.

Листинг 1.1 – Программная реализация решения задачи по лабораторной работе № 1 (вариант 0)

```
#include <cstdio>
#include <cmath>

const double EPSILON_MISTAKE = 0.001;
const long MAX_VALUE_A = 1000000;

int main()  {
	double x, e, b;
	long a;
	int input_buffer;
	// ввод исходных данных и проверка их на корректность
	printf("Input x value: ");
	scanf("%lf", &x);
	while (x == 0.0) {
		while ((input_buffer = getchar()) != '\n' && input_buffer != EOF);
		printf("Illegal x value. Try again: ");
		scanf("%lf", &x);
	}
	printf("Input epsilon value: ");
	scanf("%lf", &e);
```

```
while (e <= 0.0) {
     while ((input_buffer = getchar()) != '\n' && input_buffer != EOF);
     printf("Illegal epsilon value. Try again: ");
     scanf("%lf", &e);
}
bool is_input_again = true;
printf("Input a value: ");ы
while (is_input_again) {
     scanf("%7ld", &a);
     while (a >= MAX_VALUE_A) {
          while ((input_buffer = getchar()) != '\n' &&
                 input_buffer != EOF);
          printf("Value is too much. Try again: ");
          scant("%7ld", &a);
     }
     if (e >= 1.0) {
          b = 1;
          for (int i = 2; a >= 0 && i <= a; i++)
               b *= i;
               is_input_again = a < 0;
               // Предположим, что гамма-функция не существует
     } else {
          is_input_again = a == 0;
          if (!is_input_again)
               b = 1.0 / a;
     }
     if (is_input_again) {
          printf("Input is not correct or doesn't meet to math
          conditions. Try again: ");
     }
}
double base_x = x;
double member = 0, sum = 0, factorial = 1;
int k = 0;
bool is_overflowed = false;
do {
     // расчет суммы: добавление предыдущего слагаемого (либо 0)
     sum += member;
     // расчет текущего элемента суммы
     member = base_x / b / factorial;
     base_x *= -1 * (x * x);
     // учитываем,что числитель на каждой итерации увеличивается в x^2
     k += 1;
     factorial *= k;
     /* используем значение факториала, подсчитанное на предыдущих
     итерациях. Если факториал рассчитывать заново на каждой итерации,
     то очень сильно возрастает сложность вычислений по времени */
     is_overflowed = isinfl(base_x) || isinfl(factorial);
```

```
        /*   можно проверить b на переполнение, но если b == infinity,
        то просто выйдем по второму условию, так как элемент суммы будет
        меньше epsilon */
    } while(!is_overflowed && fabs(member) - EPSILON_MISTAKE >= e);
    if (!is_overflowed) {
        if (k - 1 == 0) {
            printf("The result sum = 0 \n The no members were used to
            calculate sum. \n Probably, the precision is too low. \n");
        } else {
        // вывод результата с учетом невключения последнего слагаемого
            printf("The result sum=%.8lf \n The %d members were used.
            \n", sum, k - 1);
        }
    } else {
        printf("Double overflow occurred. \n The current sum=%.8lf \n
        The member count is %d.\n", sum, k - 1);
    }
    return 0;
}
```

Таблица 1.2 – Результаты контрольных прогонов программы

Введенные значения	**Результат работы программы (фактическое значение суммы и количество членов суммы)**	**Предполагаемые результаты**
X = 0.03, ε = 0.05, a = 1	Sum = 0, Members = 0 (низкая точность расчетов)	Sum = 0, Members = 0 (низкая точность расчетов)
X = 1, ε = 1, a = 1	Sum = 0, Members = 0 (низкая точность расчетов)	Sum = 0, Members = 0 (низкая точность расчетов)
X = –1, ε = 0.001, a = 100	Sum = –36.78819444, Members = 9	Sum = 36.78819444, Members = 9
X = 1.1, ε = 0.3, a = 2	Sum = 0.49893763, Members = 4	Sum = 0.49893763, Members = 4
X = –3, ε = 0.5, a = 100	Sum = –0.15974046, Members = 29	Sum = –0.15974046, Members = 29
X = 1, ε = 0.01, a = 1000	Sum = 367.88194444, Members = 9	Sum = 367.88194444, Members = 9

Продолжение таблицы 1.2

Введенные значения	**Результат работы программы**	**Предполагаемые результаты**
X = 10, ε = 2, a = 0.001	Невозможно высчитать сумму (переполнение double при Members = 153)	Невозможно высчитать сумму (переполнение double при Members = 153)
X = 100, ε = 1, a = 0.001	Невозможно высчитать сумму (переполнение double при Members = 76)	Невозможно высчитать сумму (переполнение double при Members = 76)

```
Input x value: -1
Input epsilon value: 0.001
Input a value: 100
The result sum = –36.78819444
  The 9 members were used.

... Program finished with exit code 0
Press ENTER to exit console.
```

```
Input x value: 1.1
Input epsilon value: 0.3
Input a value: 2
The result sum = 0.49893763
  The 4 members were used.

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Input x value: 0
Illegal x value. Try again: 1
Input epsilon value: 0
Illegal epsilon value. Try again: 0.01
Input a value: 1000
The result sum = 367.88194444
  The 9 members were used.

... Program finished with exit code 0
Press ENTER to exit console.
```

```
Input x value: 1
Input epsilon value: 1
Input a value: 1
The result sum=0
  The no members were used to calculate sum.
  Probably, the precision is too low.

... Program finished with exit code 0
Press ENTER to exit console.
```

Рисунок 1.1 – Экранные формы результатов контрольных прогонов программы

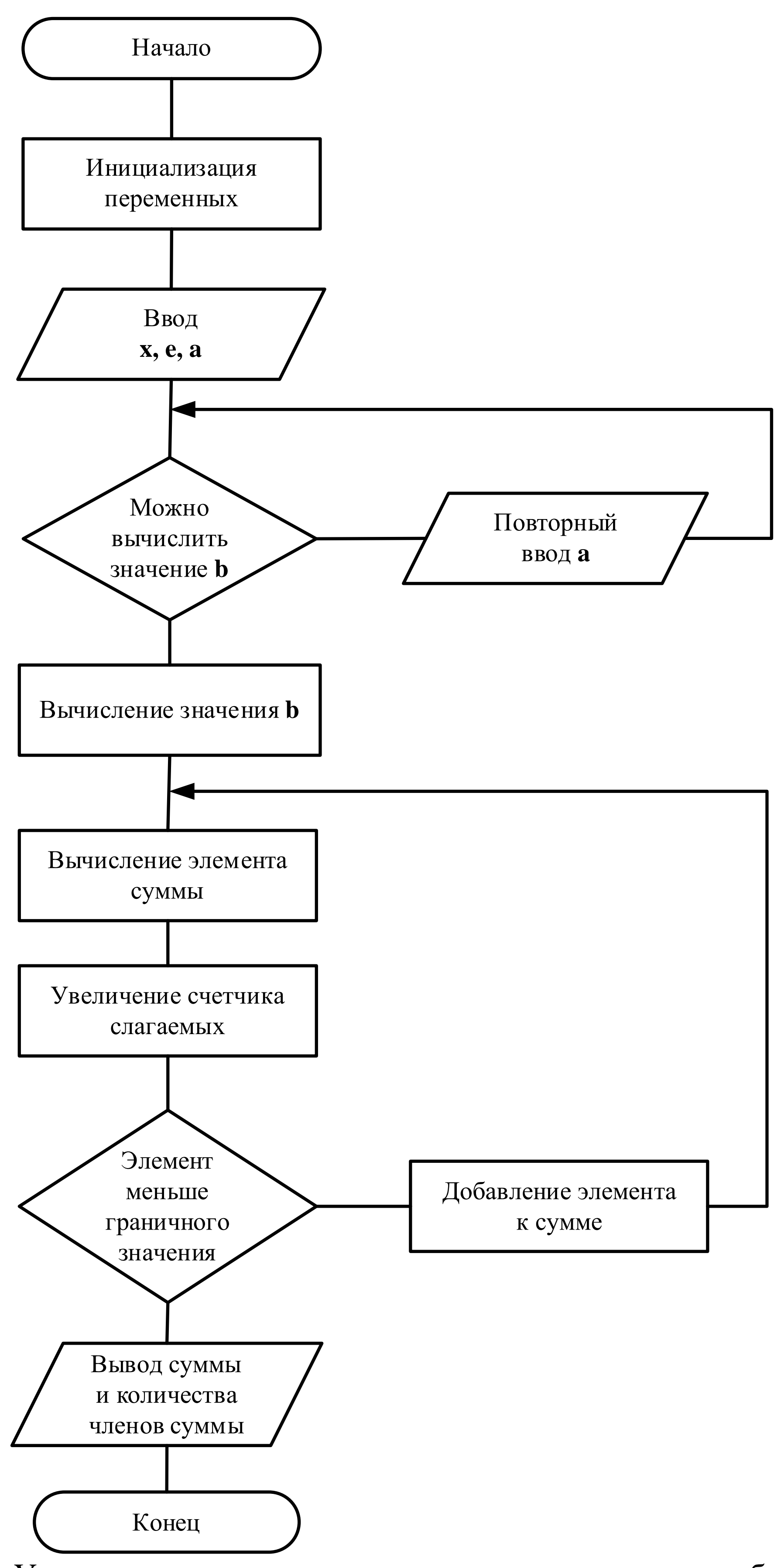


Рисунок 1.2 – Укрупненная схема алгоритма решения задачи по лабораторной работе № 1 (вариант 0)

ЛАБОРАТОРНАЯ РАБОТА № 2
Разработка вычислительных процессов с использованием составных статических структур данных (массивы)

Цель: изучение возможностей языка С++ для работы с однородными составными типами данных; изучение основных алгоритмов поиска данных и сортировки в линейных структурах; получение навыков разработки программ с использованием статических массивов.

Контрольные вопросы для самоподготовки

1 Составные типы данных. Виды и их назначение.
2 Особенности объявления массивов на С++.
3 Возможные способы инициализации массивов в С++.
4 Доступ к элементам массива. Обработка массивов на языке С++.
5 Основные операции, допустимые с массивами на языке С++.
6 Общая формулировка алгоритма сортировки данных.
7 Особенности и характеристики различных алгоритмов сортировки.
8 Суть метода «пузырьковой сортировки».
9 Суть метода «сортировки отбором».
10 Суть метода «сортировки вставками».
11 Суть метода «быстрой сортировки».
12 Понятие «скорость сортировки». Критерии, от которых зависит скорость сортировки для разных алгоритмов.
13 Общая формулировка алгоритма поиска данных в линейных структурах. Разновидности алгоритмов поиска.
14 Метод «линейного поиска». Особенности реализации.
15 Метод «двоичного поиска». Особенности реализации и основные характеристики.

Задание

1 Согласовать вариант задания с преподавателем, внимательно изучить индивидуальное задание к задаче (таблица 2.1) и пример.

2 Разработать алгоритм для решения следующей задачи:

Объявить одномерный статический массив заданного типа и размера. Заполнить его одним из способов по выбору пользователя: пользовательскими данными с клавиатуры либо случайными числами в диапазоне от **A** до **B** (значения **A** и **B** ввести с клавиатуры).

Преобразовать массив в соответствии с индивидуальным заданием. Организовать вывод исходного массива и массива после обработки. Используйте по умолчанию алгоритм сортировки пузырьком

3 Написать программный код реализации составленного алгоритма с учетом требований и ограничений по индивидуальному заданию.

4 Организовать текстовый пользовательский интерфейс в программе. Ввод данных пользователем реализовать с клавиатуры, а вывод результатов выполнения программы – в консоль (на экран).

5 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

6 Построить укрупненную схему составленного алгоритма (блок-схему работы программы).

7 ***Дополнительно*** следует учесть, что правильность вводимых значений не гарантируется. Необходимо обеспечить проверку соответствия входных данных указанным в условии ограничениям и предусмотреть сообщение об ошибке в случае ее обнаружения.

8 ***Дополнительно*** реализовать сортировку массива с помощью алгоритма по варианту (таблица 2.2).

9 ***Дополнительно*** для своего варианта реализации сортировки выполнить измерение времени работы программы и сделать анализ зависимости времени выполнения алгоритма сортировки от конфигурации входных данных (количества элементов и их исходного расположения).

Таблица 2.1 – Варианты заданий для лабораторной работы № 2

Вариант	Индивидуальное задание
0	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. Преобразовать массив так, чтобы в начале стояли цифры, входящие в массив, а затем буквы. Взаимное расположение букв не должно измениться, а цифры *отсортировать* по убыванию
1	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, учитывая, что первая половина алфавита «весит» меньше, чем цифры, а вторая половина – больше
2	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, но считать началом массива середину, чтобы массив был «цикличным» (использовать оператор остатка от деления)

Продолжение таблицы 2.1

Вариант	Задание
3	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* символы по количеству их повторений в массиве
4	Задан массив из 100 символов, который может содержать только целые числа. *Отсортировать* четные числа в порядке возрастания, а нечетные в порядке убывания, оставляя взаимное расположение четных и нечетных нетронутым
5	Задан массив из 100 символов, который может содержать только буквы латинского алфавита. *Отсортировать* гласные буквы в порядке возрастания, а согласные – в порядке убывания, оставляя взаимное расположение гласных и согласных нетронутым
6	Задан массив из 100 символов, который содержит только цифры. Найти самые длинные последовательности четных цифр и *отсортировать* их между собой по длине, не нарушая самой последовательности
7	Задан массив из 100 символов, который может содержать только буквы латинского алфавита. Найти все последовательности согласных букв и *отсортировать* их между собой по длине, не нарушая самих последовательностей
8	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, учитывая, что первые две буквы алфавита равны по весу 0, следующие две буквы – 1 и т. д.
9	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. Найти последовательности символов шестнадцатеричной системы счисления и *отсортировать* такие шестнадцатеричные числа между собой по длине
10	Задан массив из 100 символов, который содержит только цифры. Найти последовательности нечетных цифр и *отсортировать* их по сумме цифр в них
11	Задан массив из 100 символов, который содержит только цифры. Найти последовательности четных цифр и *отсортировать* их по максимальному члену этой последовательности
12	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, учитывая, что первая половина алфавита «весит» меньше, чем цифры, а вторая половина – больше

Продолжение таблицы 2.1

Вариант	Задание
13	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, но считать началом массива середину таким образом, чтобы массив был «цикличным» (использовать оператор остатка от деления)
14	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* символы по количеству их повторений в массиве. В начале массива необходимо разместить буквы, а затем – цифры
15	Задан массив из 100 символов, который может содержать только целые числа. *Отсортировать* четные числа в порядке возрастания, а нечетные – в порядке убывания, оставляя взаимное расположение четных и нечетных нетронутым
16	Задан массив из 100 символов, который может содержать только буквы латинского алфавита. *Отсортировать* гласные буквы в порядке возрастания, а согласные – в порядке убывания, оставляя взаимное расположение гласных и согласных нетронутым
17	Задан массив из 100 символов, который содержит только цифры. Найти все последовательности четных цифр и *отсортировать* эти цифры между собой по не убыванию (упорядочить внутри самой последовательности)
18	Задан массив из 100 символов, который может содержать только буквы латинского алфавита. Найти все последовательности гласных букв и *отсортировать* эти буквы между собой в алфавитном порядке (упорядочить внутри самой последовательности)
19	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, учитывая, что первые две буквы алфавита равны по весу 0, следующие две буквы – 1 и т. д.
20	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. Найти последовательности символов шестнадцатеричной системы счисления и *отсортировать* такие шестнадцатеричные числа между собой по длине
21	Задан массив из 100 символов, который содержит только цифры. Найти последовательности нечетных цифр и *отсортировать* их по сумме цифр в них

Продолжение таблицы 2.1

Вариант	Задание
22	Задан массив из 100 символов, который содержит только цифры. Найти последовательности четных цифр и *отсортировать* их по максимальному члену этой последовательности
23	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* символы в массиве с учетом их «веса» по таблице ASCII кодов
24	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, но считать началом массива середину, чтобы массив был «цикличным» (использовать оператор остатка от деления)
25	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры по количеству их повторений в массиве в порядке возрастания
26	Задан массив из 100 символов, который может содержать только целые числа. *Отсортировать* четные числа в порядке возрастания, а нечетные – в порядке убывания, оставляя взаимное расположение четных и нечетных нетронутым
27	Задан массив из 100 символов, который может содержать только буквы латинского алфавита. *Отсортировать* гласные буквы в порядке возрастания, а согласные – в порядке убывания, оставляя взаимное расположение гласных и согласных нетронутым
28	Задан массив из 100 символов, который содержит только цифры. Найти все последовательности четных цифр и *отсортировать* их между собой по минимальному члену каждой последовательности, не нарушая самих последовательности
29	Задан массив из 100 символов, который может содержать только буквы латинского алфавита. Найти все последовательности гласных букв и *отсортировать* их между собой по значению в алфавитном порядке
30	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. *Отсортировать* буквы и цифры, учитывая, что первые две буквы алфавита равны по весу 0, следующие две буквы – 1 и т. д.
31	Задан массив из 100 символов, который содержит только цифры. Найти последовательности четных цифр и *отсортировать* их по максимальному члену каждой последовательности

Продолжение таблицы 2.1

Вариант	Задание
32	Задан массив из 100 символов, который может содержать только буквы латинского алфавита и цифры. Найти последовательности символов шестнадцатеричной системы счисления и *отсортировать* такие шестнадцатеричные числа между собой по значению
33	Задан массив из 100 символов, который содержит только цифры. Найти последовательности нечетных цифр и *отсортировать* их по сумме значений в них

Таблица 2.2 – Варианты сортировок для самостоятельной реализации

Вариант				Сортировка
1	10	19	28	Отбором (Selection sort)
2	11	20	29	Вставками (Insertion sort)
3	12	21	30	Шелла (Shellsort)
4	13	22	31	Подсчетом (Counting sort)
5	14	23	32	«Расческой» (Comb sort)
6	15	24	33	Шейкерная (Cocktail sort)
7	16	25	34	«Чет-нечет» (Odd-even sort)
8	17	26	35	Слиянием (Merge sort) / итеративный алгоритм
9	18	27	36	Быстрая (Quicksort) / итеративный алгоритм

Пример выполнения лабораторной работы

Программная реализация алгоритма решения задания для 0-го варианта представлена в листинге 2.1. Результаты контрольных прогонов программы, демонстрирующие работу алгоритма на различных входных данных (проверка корректной обработки краевых значения и критических ситуаций), представлены на рисунке 2.1. На рисунках 2.2 и 2.3 представлена блок-схема алгоритма.

Листинг 2.1 – Программная реализация индивидуального задания по лабораторной работе № 2

```
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    setlocale(0, "rus"); // русификация консоли
    char a[100];  // массив символов с максимальной размерностью
    int n;        // реальная размерность массива символов
    cout << "Введите размерность массива (от 1 до 100) \n";
```

```
// ввод размерности массива с проверкой диапазона
while (!( (cin >> n) && (n <= 100 && n > 0) )) {
    cin.clear();
    while (cin.get() != '\n'); // считывание до конца строки
    cout << "Размерность введена неправильно!\n";
}
int number;    // переменная для организации меню
bool flag = 1;
while (flag) {
    cout << "Ввести массив вручную - введите 1" << endl;
    cout << "Заполнить массив автоматически - введите 2 \n";
    // ввод пункта меню и проверка его на корректность
    while (!(cin >> number)) {
        cin.clear();
        while (cin.get() != '\n');
        cout << "Введено некорректное значение" << endl;
    }
    switch (number) {
    case 1:
        /* пользовательский ввод данных в исходный массив */
        cout << "Введите " << n << " символов (только цифры
        или латинские буквы)!\n";
        for (int i = 0; i < n; i++) {
            cin >> a[i];
            if( !(((a[i] <= '9') && (a[i] >= '0')) ||
                  ((a[i] <= 'Z') && (a[i] >= 'A')) ||
                  ((a[i] <= 'z') && (a[i] >= 'a'))) ) {
            cout << "Ошибка при вводе (неверный символ)!\n";
            return 1; //завершение работы программы с ошибкой
            }
        }
        flag = 0;
        break;
    case 2:
        /* запускаем генерацию случайных чисел и заполняем
        исходный массив случайными данными */
        srand(time(0));
        int cifMin, cifMax;
        cout << "Введите минимальную цифру" << endl;
        while (!(cin >> cifMin &&
                 cifMin >= 0 && cifMin < 10)) {
            cin.clear();
            while (cin.get() != '\n');
            cout << "Введено некорректное значение!\n";
        }
        cout << "Введите максимальную цифру" << endl;
        while (!(cin >> cifMax && cifMax >= 0 &&
                cifMax < 10 && cifMax >= cifMin)) {
            cin.clear();
```

```
            while (cin.get() != '\n');
            cout << "Введено некорректное значение!\n";
        }
        for (int chis, i = 0; i < n; i++) {
            /* переменная chis используется для чередования
            чисел и букв в исходном массиве при заполнении
            случайными данными */
            chis = rand() % 2;
            if (chis)
               a[i] = cifMin + rand() % (cifMax - cifMin + 1)
                      + '0';
            else
                a[i] = 97 + rand() % (122 - 97 + 1);
                // добавление малых букв латинского алфавита
        }
        flag = 0;
        break;
    default:
        cout << "Неверный выбор пункта меню. \n";
        break;
    }
}
// вывод исходного массива в консоль в виде таблицы
cout << "Исходный массив:" << endl;
for (int i = 0; i < n; i++) cout << a[i] << "\t";
cout << endl;

int arr[100]; // дополнительный массив для цифр
int size = 0; /* - реальное количество элементов числового
               массива (количество цифр в исходном массиве)*/
for (int i = 0; i < n; i++) {
     // поиск цифр и сохранение их в отдельный массив
    if ((a[i] <= '9') && (a[i] >= '0')) {
        arr[size] = a[i] - '0';
        a[i] = 1;
        // ставим "маркер" в исходном массиве, т.е.
        // заменяем цифру на первый символ в таблице ASCII
        size++;
    }
}
// сортировка пузырьком массива с цифрами
for (int temp, i = 0; i < size - 1; i++) {
    for (int j = 0; j < size - i - 1; j++) {
        if (arr[j] < arr[j + 1]) {
           temp = arr[j]; arr[j] = arr[j+1]; arr[j+1] = temp;
        }
    }
}
int kol = 0;
```

```
    /* переменная kol служит для хранения информации о дальности
       сдвига букв в исходном массиве вправо */

    /*  Далее выполняем сливание двух массивов:
        "маркированного" исходного и отсортированного с цифрами.
         При этом все буквы сдвигаются в конец исходного массива.
    */
    for(int i = n-1; i >= 0; i--) {
        if (a[i] ==1) kol++;
        else a[i + kol] = a[i];
        if (i < size) a[i] = char(arr[i]) + '0';
    }
    // вывод преобразованного массива в консоль в виде таблицы
    cout << "Отсортированный массив:" << endl;
    for (int i = 0; i < n; i++) cout << a[i] << "\t";
    cout << endl;
    return 0;
}
```

```
Введите размерность массива от 1 до 100
0
Размерность указана неправильно!
1
Чтобы ввести массив вручную - 1
Чтобы заполнить массив автоматически - 2
1
Введите 1 символов (цифры или латинские буквы)!
1
Исходный массив:
1
Отсортированный массив:
1

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Введите размерность массива от 1 до 100
10
Чтобы ввести массив вручную - 1
Чтобы заполнить массив автоматически - 2
Введите 10 символов (цифры или латинские буквы)!
0a5f3V2b7x
Исходный массив:
0    a    5    f    3    V    2    b    7    x

Отсортированный массив:
7    5    3    2    0    a    f    V    b    x

...Program finished with exit code 0
Press ENTER to exit console.
```

Рисунок 2.1 – Экранные формы результатов работы программы по лабораторной работе № 2

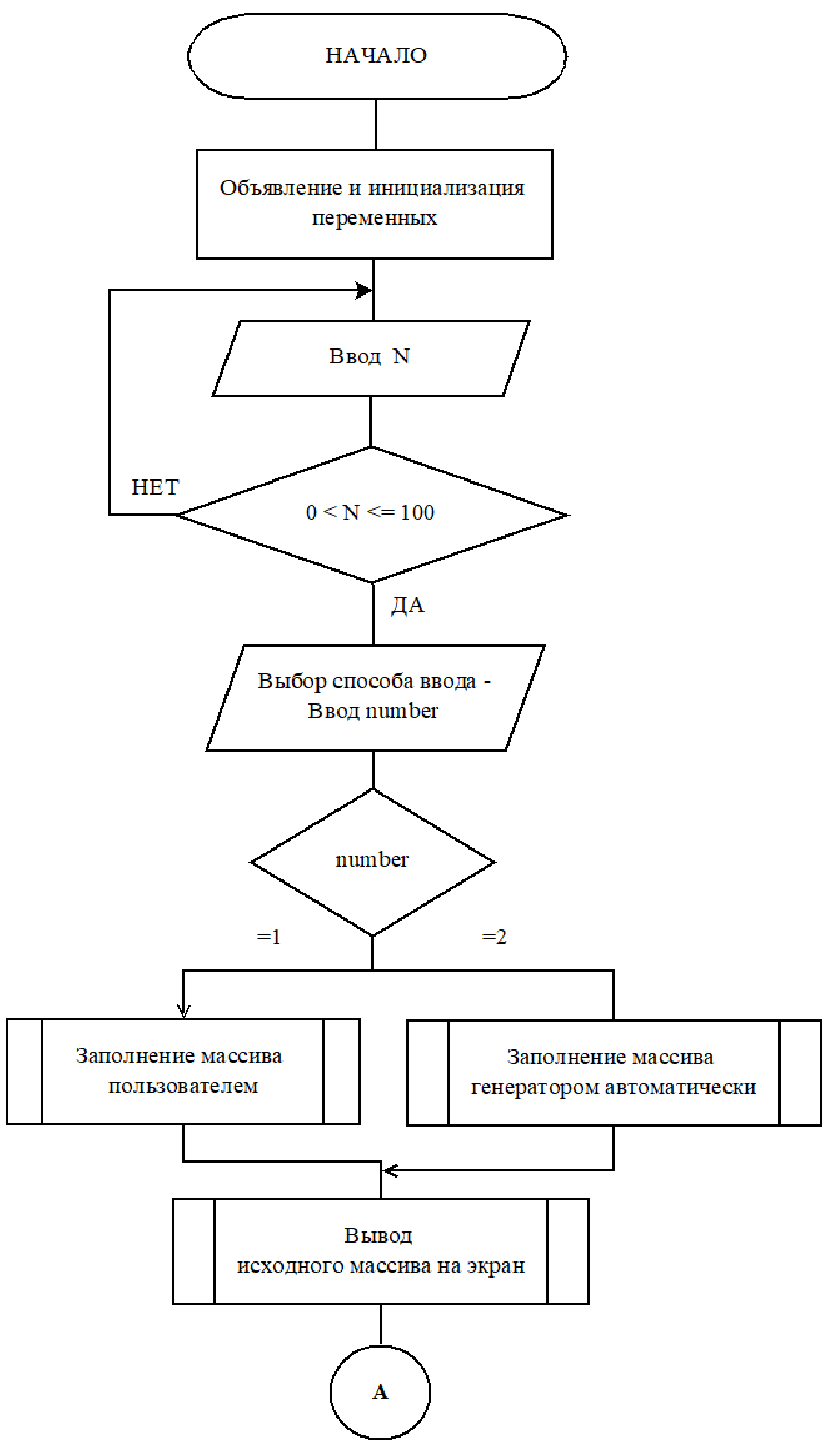


Рисунок 2.2 – Блок-схема схема алгоритма решения лабораторной работы № 2 (фрагмент ввода данных)

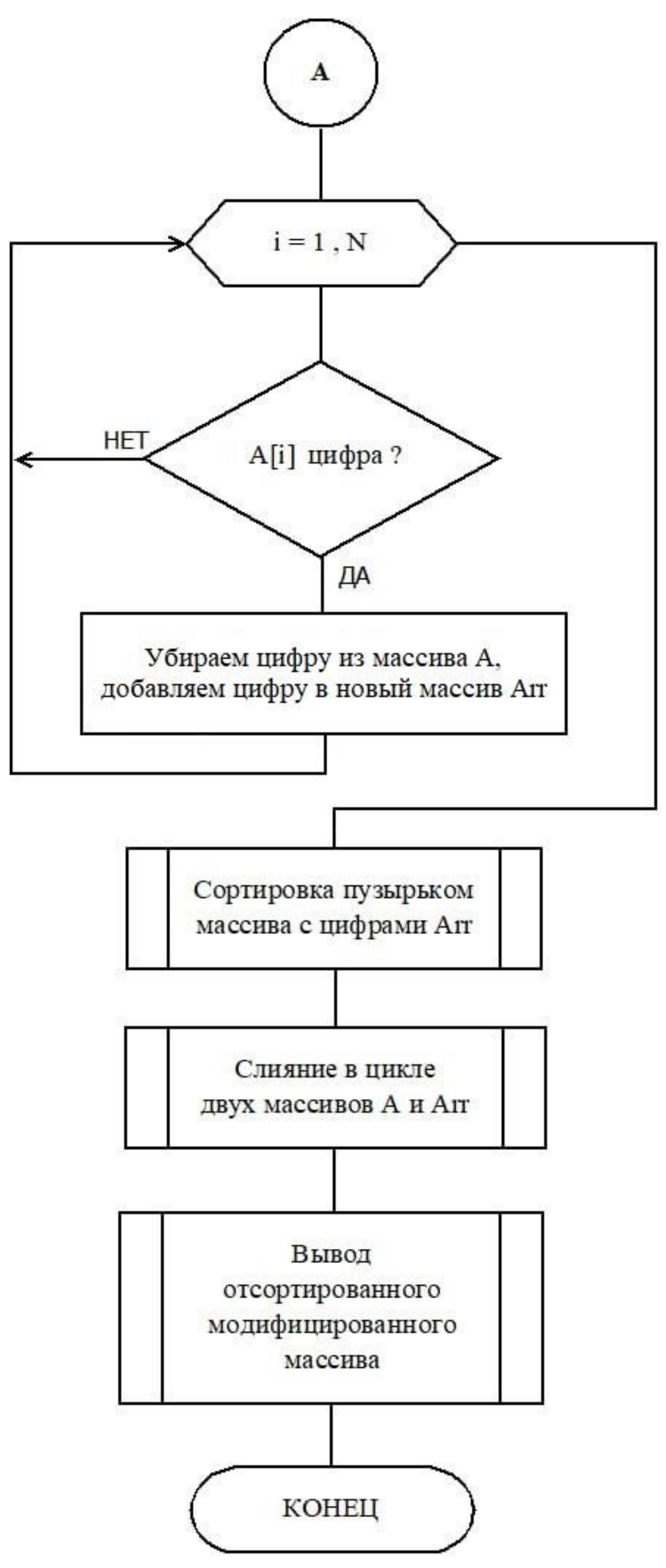


Рисунок 2.3 – Блок-схема схема алгоритма решения лабораторной работы № 2 (фрагмент обработки и вывода массива)

ЛАБОРАТОРНАЯ РАБОТА № 3
Разработка вычислительных процессов с использованием динамической памяти (указатели)

Цель: изучение особенностей организации вычислительных процессов с использованием динамической памяти; изучение принципов для работы с памятью на языке С++; получение навыков разработки программ с использованием динамических массивов и оценки эффективности их работы.

Контрольные вопросы для самоподготовки

1 Организация памяти компьютера с точки зрения языка С++.
2 Динамическое распределение памяти. Понятия «стек» и «куча».
3 Понятие «указатель» и размещение данных в памяти.
4 Понятие «ссылка» в С++. Отличия указателя от ссылки.
5 Объявление указателей и операции над ними.
6 Указатели как тип данных. Операции * и & и связь между ними.
7 Указатели на указатели. Многоуровневые и константные указатели.
8 Выделение и освобождение динамической памяти.
9 Операторы `new` и `delete`.
10 Динамическая память, функции работы с памятью.
11 Массивы в динамической памяти.
12 Связь между указателями и массивами.
13 Доступ к элементам массива через указатели.
14 Двумерные динамические массивы. Прямая и косвенная адресации.
15 Оценка сложности работы программы. Асимптотический анализ алгоритма и *O*-нотация.

Задание

1 Согласовать вариант задания с преподавателем, внимательно изучить индивидуальное задание к задаче (таблица 3.1) и пример.

2 Разработать алгоритм для решения следующей задачи:

Объявить двумерный динамический массив, размер (***N*** х ***N***) которого задается пользователем. Заполнить его одним из способов по выбору пользователя: пользовательскими данными с клавиатуры либо случайными данными в диапазоне от ***A*** до ***B*** (значения ***A*** и ***B*** вводятся с клавиатуры). Преобразовать массив в соответствии с индивидуальным заданием без использования дополнительных массивов. Организовать вывод исходного массива и массива после обработки

3 Написать программный код реализации составленного алгоритма с учетом требований и ограничений по индивидуальному заданию.

4 Организовать текстовый пользовательский интерфейс в программе и форматированный вывод данных (матричное представление). Ввод данных пользователем реализовать с клавиатуры, а вывод результатов выполнения программы в консоль (на экран).

5 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

6 Построить укрупненную схему составленного алгоритма (блок-схему реализации индивидуального задания).

7 ***Дополнительно*** следует учесть, что правильность вводимых значений не гарантируется. Необходимо обеспечить проверку соответствия входных данных указанным в условии ограничениям и предусмотреть сообщение об ошибке в случае ее обнаружения.

8 ***Дополнительно*** выполнить асимптотическую оценку сложности реализованного алгоритма по критериям использования памяти и по времени выполнения.

Таблица 3.1 – Варианты заданий для лабораторной работы № 3

Вариант	**Задание**	**Пример**
0	Зеркально отразить матрицу по горизонтали	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ⇒ 13 14 15 16 9 10 11 12 5 6 7 8 1 2 3 4
1	Зеркально отразить матрицу по вертикали	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ⇒ 4 3 2 1 8 7 6 5 12 11 10 9 16 15 14 13
2	Зеркально отразить матрицу относительно побочной диагонали	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ⇒ 16 12 8 4 15 11 7 3 14 10 6 2 13 9 5 1
3	Зеркально отразить матрицу по горизонтали относительно середины	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ⇒ 5 6 7 8 1 2 3 4 13 14 15 16 9 10 11 12
4	Зеркально отразить матрицу относительно главной диагонали	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ⇒ 1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16

Продолжение таблицы 3.1

Вариант	Задание	Пример
5	Повернуть матрицу на 90° по часовой стрелке	1 2 3 4 ⇒ 13 9 5 1 5 6 7 8 ⇒ 14 10 6 2 9 10 11 12 ⇒ 15 11 7 3 13 14 15 16 ⇒ 16 12 8 4
6	Повернуть матрицу на 180° по часовой стрелке	1 2 3 4 ⇒ 16 15 14 13 5 6 7 8 ⇒ 12 11 10 9 9 10 11 12 ⇒ 8 7 6 5 13 14 15 16 ⇒ 4 3 2 1
7	Повернуть матрицу на 90° против часовой стрелки	1 2 3 4 ⇒ 4 8 12 16 5 6 7 8 ⇒ 3 7 11 15 9 10 11 12 ⇒ 2 6 10 14 13 14 15 16 ⇒ 1 5 9 13
8	Повернуть матрицу на 180° против часовой стрелке	1 2 3 4 ⇒ 16 15 14 13 5 6 7 8 ⇒ 12 11 10 9 9 10 11 12 ⇒ 8 7 6 5 13 14 15 16 ⇒ 4 3 2 1
9	Заменить на 0 все седловые точки матрицы. *Седловой точкой матрицы назовем элемент, который одновременно является минимумом в своей строке и максимумом в своем столбце*	1 2 3 4 ⇒ 1 2 3 4 5 6 7 8 ⇒ 5 6 7 8 9 10 11 12 ⇒ 9 10 11 12 13 14 15 16 ⇒ 0 14 15 16
10	Поменять местами элементы главной и побочной диагоналей матрицы, а затем отзеркалить их	1 2 3 4 ⇒ 13 2 3 16 5 6 7 8 ⇒ 5 10 11 8 9 10 11 12 ⇒ 9 6 7 12 13 14 15 16 ⇒ 1 14 15 4
11	Поменять местами строку, которая содержит минимальный элемент матрицы, со столбцом, содержащим максимальный элемент матрицы	1 2 3 4 ⇒ 16 12 8 4 5 6 7 8 ⇒ 5 6 7 3 9 10 11 12 ⇒ 9 10 11 2 13 14 15 16 ⇒ 13 14 15 1
12	Заменить значения элементов главной диагонали матрицы на сумму элементов строки и столбца, на пересечении которых он находится. Затем заменить на 0 все элементы в строке и в столбце, на пересечении которых получилась максимальная сумма (диагональ не обнулять)	1 2 3 4 ⇒ 37 2 3 0 5 6 7 8 ⇒ 5 52 7 0 9 10 11 12 ⇒ 9 10 67 0 13 14 15 16 ⇒ 0 0 0 82

Продолжение таблицы 3.1

Вариант	Задание	Пример
13	В каждом столбце матрицы поменять местами минимальный и максимальный элементы	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 13 & 14 & 15 & 16 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \end{matrix}$
14	Найти в матрице первую из строк, содержащую элемент, равный или ближайший к нулю. Уменьшить все элементы матрицы на значение первого элемента в найденной строке	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix}$
15	В каждой строке матрицы поменять местами минимальный и максимальный элементы	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 4 & 2 & 3 & 1 \\ 8 & 6 & 7 & 5 \\ 12 & 10 & 11 & 9 \\ 16 & 14 & 15 & 13 \end{matrix}$
16	Поменять местами столбцы, которые содержат минимальный и максимальный элементы матрицы. Затем поменять местами строки, содержащие минимальный и максимальный элементы матрицы	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 16 & 14 & 15 & 13 \\ 8 & 6 & 7 & 5 \\ 12 & 10 & 11 & 9 \\ 4 & 2 & 3 & 1 \end{matrix}$
17	Заменить на 0 все элементы в строке, которые меньше элемента расположенного на главной диагонали, и на 1 все элементы в строке, которые больше элемента расположенного на главной диагонали	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 1 & 1 & 1 & 1 \\ 0 & 6 & 1 & 1 \\ 0 & 0 & 11 & 1 \\ 0 & 0 & 0 & 16 \end{matrix}$
18	Поменять местами строки матрицы, элементы которых образуют соответственно минимальную и максимальную суммы	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 13 & 14 & 15 & 16 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \end{matrix}$
19	Поменять местами строки матрицы таким образом, чтобы элементы первого столбца образовали невозрастающую последовательность	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 13 & 14 & 15 & 16 \\ 9 & 10 & 11 & 12 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \end{matrix}$
20	Поменять местами столбцы матрицы таким образом, чтобы элементы первой строки образовали неубывающую последовательность	$\begin{matrix} 16 & 12 & 8 & 4 \\ 5 & 6 & 7 & 3 \\ 9 & 10 & 11 & 2 \\ 13 & 14 & 15 & 1 \end{matrix} \Rightarrow \begin{matrix} 4 & 8 & 12 & 16 \\ 3 & 7 & 6 & 5 \\ 2 & 11 & 10 & 9 \\ 1 & 15 & 14 & 13 \end{matrix}$

Пример выполнения лабораторной работы

Программная реализация алгоритма решения задания для 0-го варианта представлена в листинге 3.1. Блок-схема алгоритма представлена на рисунке 3.1.

Листинг 3.1 – Исходный код программы по лабораторной работе № 3

```
#include <iostream>
using namespace std;

int main() {
    setlocale(0, "rus");

    int input, size = 5;
    // Объявление двумерного массива
    int** array = new int* [size];
    for (int i = 0; i < size; i++) array[i] = new int[size];
    if (size == 0) {
        cout << "Массив пуст." << endl;
        return 0;
    }
    if (size < 0 || size > 100) {
        cout << "Некорректный размер матрицы!" << endl;
        return 1;
    }
    cout << "Тестовый вариант (пример из условия): " << endl;
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            array[i][j] = i;

    // вывод исходного массива на консоль
    cout << "Начальная матрица: " << endl;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++)
            cout << array[i][j] << " ";
        cout << endl;
    }

    // преобразование матрицы по заданию (отзеркаливание)
    int** start = array;
    int** end = array + size - 1;

    while (start < end) {
        int* temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
```

```
// вывод преобразованного массива на консоль
cout << "Отзеркаленная матрица: " << endl;
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        cout << array[i][j] << " ";
    }
    cout << endl;
}

while (true) {
    cout << "Меню пользователя: \n";
    cout << "1. Зеркальное отражение матрицы\n";
    cout << "2. Вывод матрицы\n";
    cout << "3. Изменить размер матрицы\n";
    cout << "4. Заполнить массив другими значениями\n";
    cout << "Для выхода введите любое другое число\n";
    cout << "Выберите действие (1-4) : ";

    while (!(cin >> input) || (cin.peek() != '\n')) {
        cin.clear();
        while (cin.get() != '\n');
        cout << "Некорректный ввод, попробуйте еще!" << endl;
    }
    int choice = input;
    switch (choice) {
    case 1:
        start = array;
        end = array + size - 1;
        while (start < end) {
            int* temp = *start;
            *start = *end;
            *end = temp;
            start++;
            end--;
        }
        cout << "Матрица успешно зеркально отражена!" << endl;
        break;
    case 2:
        cout << "Матрица:" << endl;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++)
                cout << array[i][j] << " ";
            cout << endl;
        }
        break;
    case 3:
        for (int i = 0; i < size; i++) delete[] array[i];
        delete[] array;
```

```
            cout << "Введите новую размерность матрицы (N): ";
            while (!(cin >> input) || (cin.peek() != '\n')) {
                cin.clear();
                while (cin.get() != '\n');
                cout << "Некорректный ввод, попробуйте еще! \n";
            }
            size = input;
            if (size <= 0 || size > 100) {
                cout << "Некорректный размер матрицы! \n";
                return 0;
            }
            array = new int* [size];
            for (int i = 0; i < size; i++)
                 array[i] = new int[size];

            cout << "Введите элементы нового массива : \n";
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    while (!(cin >> input)||(cin.peek() != '\n')){
                        cin.clear();
                        while (cin.get() != '\n');
                        cout << "Некорректный ввод!" << endl;
                    }
                    array[i][j] = input;
                }
            }
            break;
        case 4:
            cout << "Заполните массив: \n";
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    while (!(cin >> input)||(cin.peek() != '\n')){
                        cin.clear();
                        while (cin.get() != '\n');
                        cout << "Некорректный ввод!" << endl;
                    }
                    array[i][j] = input;
                }
            }
            break;
        default:
            for (int i = 0; i < size; i++) delete[] array[i];
            delete[] array;
            return 0;
        }
    }
}
```

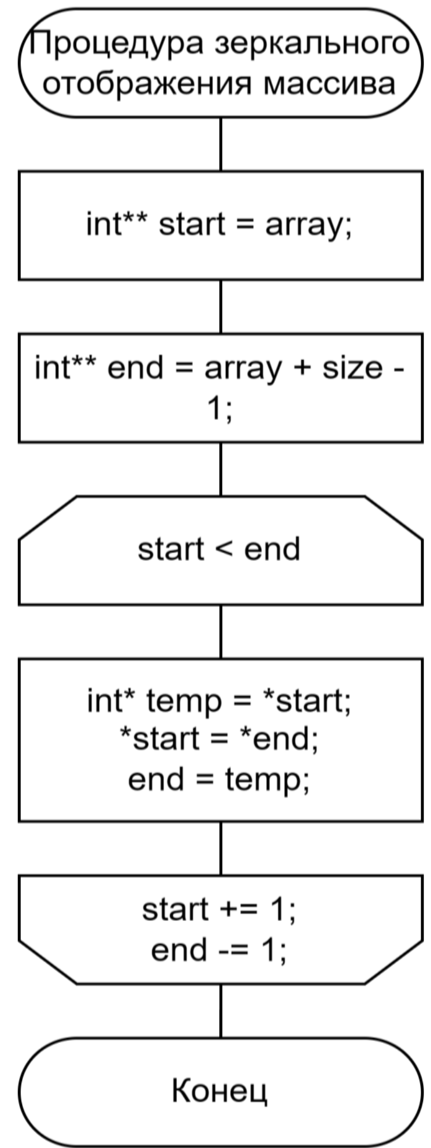


Рисунок 3.1 – Укрупненная блок-схема алгоритма решения индивидуального задания для варианта 0 лабораторной работы № 3

Оценка сложности алгоритма

1. По времени выполнения. В памяти элементы массива расположены последовательно. Поэтому мы одновременно двигаемся с двух сторон простым увеличением и уменьшением адреса; т. к. двигаемся с двух сторон, то нам понадобится $(N \cdot N) / 2$ операций. В асимптотике коэффициенты не учитываются, поэтому сложность будет $O(N^2)$.

2. По памяти. Для хранения матрицы нужен массив массивов, т. е. мы создаем массив, в каждом элементе которого хранится указатель на другой одномерный массив. Если в квадратной матрице $(N \cdot N)$ элементов, то для ее хранения

понадобится массив размером N, в котором будут храниться указатели на массивы размером N с числами. Итого получаем $(N \cdot N + N) = N^2 + N$. Сложность по памяти в асимптотике будет также $O(N^2)$.

Экранные формы работы программы представлены на рисунке 3.2.

<table>
<tr>
<td>

```
Тестовый вариант:

Начальная матрица:
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

Отзеркаленная матрица :
4 5 6 7 8
3 4 5 6 7
2 3 4 5 6
1 2 3 4 5
0 1 2 3 4

Введите размерность
матрицы: 0
Массив пуст.
Введите размерность
матрицы: 150
Некорректный размер
матрицы!
```

</td>
<td>

```
Выберите действие (1-4): 2
Матрица:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Меню пользователя :
1. Зеркальное отражение
матрицы
2. Вывод матрицы
3. Изменить размер матрицы
4. Заполнить массив заново
Если хотите выйти, введите
любое другое число

Выберите действие (1-4): 1
Матрица успешно зеркально
отражена!

Меню пользователя:
1. Зеркальное отражение
матрицы
2. Вывод матрицы
3. Изменить размер матрицы
4. Заполнить массив заново
Если хотите выйти, введите
любое другое число

Выберите действие (1-4): 2
Матрица:
13 14 15 16
9 10 11 12
5 6 7 8
1 2 3 4
```

</td>
</tr>
<tr>
<td>а</td>
<td>б</td>
</tr>
</table>

а – тестовый пример и организация ввода данных;
б – организация пользовательского меню
Рисунок 3.2 – Результаты запуска программы к лабораторной работе № 3

ЛАБОРАТОРНАЯ РАБОТА № 4
Разработка программ с использованием пользовательских функций

Цель: изучение особенностей организации подпрограмм на языке C++; изучение принципов работы с памятью на языке C++; получение навыков разработки программ с использованием пользовательских функций с различными типами аргументов.

Контрольные вопросы для самоподготовки

1 Общие сведения о структуризации программы. Подпрограммы.

2 Особенности пользовательских функций на C++: объявление функции, прототип функции, тип функции.

3 Типы пользовательских функций в C++.

4 Способы передачи аргументов в функцию. Типы и количество параметров функции. Последовательность передачи параметров в функцию.

5 Аргументы функции по умолчанию.

6 Функции с переменным числом параметров.

7 Спецификация inline-функции.

8 Вызов функции и механизм возвращения значений. Стек вызовов.

9 Понятия «область видимости объектов» и «пространство имени переменной» в C++. Сокрытие имен переменных.

10 Глобальная область видимости. Особенности и проблемы при использовании глобальных переменных.

11 Классы памяти при использовании переменных в C++: область действия, время существования, место хранения переменной.

12 Расширение области видимости переменных.

13 Понятие символьной строки. Особенности явного описания символьной строки как массива. Сравнение описаний `char* s` и `char s[]`.

14 Функции работы со символьными строками, описание их аргументов и возвращаемого значения.

15 Аргументы функции `main`.

Задание

1 Согласовать вариант задания с преподавателем, внимательно изучить работу заданной по варианту стандартной функции по работе со строками (таблица 4.1) и пример.

2 Написать программный код реализации следующей задачи:

Определить набор следующих пользовательских подпрограмм: – функция ввода исходных данных; – функция вывода результата; – функция-оболочка для указанной по варианту стандартной функции; – функция, которая полностью имитирует действия указанной по заданию стандартной функции. Реализовать функцию `main,` которая будет содержать вызовы всех указанных подпрограмм
Дополнительные требования: Обращение к элементам строки в пользовательской функции-имитаторе необходимо реализовать через указатели. Тип возвращаемого значения, типы аргументов и способ их передачи должны полностью совпадать с соответствующими параметрами стандартной функции, действие которой имитируется

3 Ввод данных пользователем реализовать с клавиатуры, а вывод результатов выполнения программы – в консоль (на экран).

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

Таблица 4.1 – Варианты заданий для лабораторной работы № 4

Вариант	**Задание (прототип и описание функции)**
0	`char* strupr ( char* srcptr );`
	Функция преобразует буквы нижнего регистра в строке `srcptr` в буквы верхнего регистра
1	`char* strcpy ( char* destptr, const char* srcptr );`
	Функция копирует строку `srcptr`, включая завершающий нулевой символ в строку назначения, на которую ссылается указатель `destptr`
2	`char* strncpy ( char* destptr, const char* srcptr, size_t num );`
	Функция копирует первые `num` символов из строки `srcptr` в строку `destptr`. Если конец строки `srcptr` достигнут прежде, чем были скопированы все `num` символов, то к скопированным символам в конец строки `destptr` добавляется нуль-символ, и строка считается скопированной

Продолжение таблицы 4.1

Вариант	Задание (прототип и описание функции)
3	`size_t strlen ( const char* str );`
	Функция вычисляет длину строки `str`. **_Внимание!_** Не путайте размер массива, который содержит строку, и длину строки
4	`int strncmp ( const char* str1, const char* str2, size_t num );`
	Функция сравнивает первые `num` символов строки `str1` с первыми `num` символами строки `str2`. Значение большее нуля указывает, что строка `str1` больше строки `str2`, отрицательное значение – `str1` меньше `str2`. Нулевое значение означает, что строки равны
5	`int strcmp ( const char* str1, const char* str2 );`
	Функция сравнивает символы двух строк, `str1` и `str2`. Если результат положительный, то строка `str1` больше строки `str2`, если результат отрицательный, то `str1` меньше `str2`. Нулевой результат, если обе строки равны
6	`char* strncat (char* destptr, char* srcptr, size_t num);`
	Функция добавляет первые `num` символов строки `srcptr` к концу строки `destptr`. Если строка `srcptr` больше, чем число `num`, то после скопированных символов неявно добавляется символ конца строки
7	`char* strcat ( char* destptr, const char* srcptr );`
	Функция добавляет копию строки `srcptr` в конец строки `destptr`
8	`char* strrchr (const char* strptr, int symbol);`
	Функция возвращает указатель на последнее вхождение символа `symbol` в строке `strptr`.
9	`const char* strchr ( const char* str, int symbol );`
	Функция выполняет поиск первого вхождения символа `symbol` в строку `str`. Если значение не найдено, то возвращается нулевой указатель

Продолжение таблицы 4.1

Вариант	Задание (прототип и описание функции)
10	`size_t strcspn ( const char* str1, const char* str2 );`
	Функция выполняет поиск первого вхождения в строку `str1` любого из символов строки `str2` и возвращает количество символов до найденного первого вхождения
11	`size_t strspn ( const char* str1, const char* str2 );`
	Функция выполняет поиск символов строки `str2` в строке `str1`. Возвращает длину начального участка строки `str1`, которая состоит только из символов, которые являются частью строки `str2`
12	`const char* strstr (const char* str1, const char* str2);`
	Функция ищет первое вхождение подстроки `str2` в строке `str1`.
13	`char* strtok ( char* strptr, const char* delim );`
	Функция выполняет поиск лексем в строке `strptr`. Последовательность вызовов этой функции разбивают строку `strptr` на лексемы, которые представляют собой последовательности символов, разделенных символами разделителями. Символами разделителями являются символы в строке `delim`. Если нет найденных лексем, то возвращается пустой указатель. При последующих вызовах функция ожидает в качестве первого аргумента нулевого указателя и использует позицию, следующую за последней лексемой, как начало для сканирования.
14	`const void* memchr ( const void* memptr, int val, size_t num );`
	Функция ищет первое вхождение `val` (неподписанного символа) в `num` байтах блока памяти, адресуемого указателем `memptr`, и возвращает указатель на найденный символ. Если значение не найдено, то возвращается нулевой указатель *`NULL`*
15	`const char* strpbrk (const char* str1, const char* str2);`
	Функция выполняет поиск первого вхождения в строку `str1` любого из символов строки `str2`. Возвращает указатель на первое вхождение найденного символа в `str1`, либо – пустой указатель, если нет ни одного совпадения

Продолжение таблицы 4.1

Вариант	Задание (прототип и описание функции)
16	`void* memset ( void* memptr, int val, size_t num );`
	Функция заполняет `num` байтов блока памяти через указатель `memptr`. Код заполняемого символа передается в функцию через параметр `val`
17	`char*_strdup ( const char* strcpsrc );`
	Функция вызывает `malloc` для выделения памяти, необходимой для копии `strcpsrc`, а затем копирует `strcpsrc` в выделенное пространство. Возвращает указатель на расположение хранилища для скопированной строки или *NULL*, если хранилище не может быть выделено
18	`char*_strlwr ( char* strcptr );`
	Функция преобразует все буквы в верхнем регистре в строке `strcptr` в нижний регистр
19	`char*_strrev ( char* strcptr );`
	Функция изменяет порядок символов в `strcptr` на обратный
20	`int _scprintf ( const char* format [, argument] ... );`
	Функция возвращает число символов, которые были бы созданы, если строка была бы напечатана либо отправлена в файл с помощью указанных кодов форматирования. Каждый `argument` (при наличии) преобразуется согласно соответствующей спецификации формата в параметре `format`. Формат состоит из обычных символов и имеет те же форму и функциональные возможности, что и аргумент `format` для функции `printf`

Пример выполнения лабораторной работы

Программная реализация алгоритма решения задания для 0-го варианта представлена в листинге 4.1.

Листинг 4.1 – Пример программной реализации лабораторной работы № 4

```
#include <iostream>
#include <cstring>
```

```
using namespace std;
/* функция ввода исходных данных */
void str_Input (char* s, int &n) {
    int return_value = 0;
    while ( !return_value )   {
        printf("Введите размерность строки\n");
        return_value = scanf_s("%d", &n);
        while ( getchar() != '\n' );
        if( !return_value )
            printf("Неправильный ввод\n");
    }
    printf("Введите строку\n");
    scanf_s("%s", s, n);
}
/* функция вывода результата */
void str_Print (char* s) {
    printf("Вывод текущей строки :\n");
    cout << s;
}
/* функция-имитатор стандартной функции strupr() */
void func_strupr(char* &s) {
    for (int x = 0; x < strlen(s); x++)
        s[x] = toupper(s[x]);
}
/* функция-оболочка для стандартной функции strupr() */
void orig_strupr(char* &s) {
    strupr(s);
}

int main() {
    setlocale(0, "Rus");
    int n = 0;
    char* s = new char[n];

    // вызов функции для ввода
    str_Input(s, n);
    char* s_copy = new char[n];
    strcopy(s_copy, s); // делаем копию строки
    // вызов оригинальной функции и функции для вывода
    orig_strupr (s);
    str_Print(s);
    // вызов функции имитации и функции для вывода
    func_strupr (s);
    str_Print(s);

    return 0;
}
```

ЛАБОРАТОРНАЯ РАБОТА № 5

Реализация вычислительных алгоритмов с использованием рекурсии

Цель: изучение особенностей организации подпрограмм на языке C++; получение навыков по структуризации программы и организации подпрограмм; знакомство с механизмом рекурсии; получение навыков организации вычислительных алгоритмов с использованием рекурсии.

Контрольные вопросы для самоподготовки

1 Общие сведения о структуризации программы. Подпрограммы.
2 Определение пользовательских функций на C++.
3 Аргументы функции. Способы передачи. Типы и количество.
4 Вызов функции и механизм возвращения значений. Стек вызовов.
5 Дайте определение понятию «рекурсия».
6 Примеры рекурсии: в природе, в технике, в социуме.
7 Дайте описание сути рекурсивного вычислительного процесса (определение рекурсивного алгоритма).
8 Достоинства и недостатки рекурсивного решения.
9 Представление алгоритма итерационным и рекурсивным способами.
10 Проблема возврата из рекурсии. Контроль за глубиной рекурсии и переполнение стека.
11 Особенности использования локальных переменных при рекурсивном вызове подпрограммы.
12 Виды рекурсии. Особенности их применения.
13 Анализ сложности рекурсивного решения. Рекуррентные соотношения.
14 Использование рекурсии в решении прикладных задач. Примеры алгоритмов рекурсии в программировании.
15 Рекурсивные алгоритмы сортировки.

Задание

1 Согласовать вариант задания с преподавателем, внимательно изучить индивидуальное задание к задаче (таблица 5.1) и пример.

2 Написать программный код реализации следующей задачи:

Реализовать две функции – итерационное решение задачи и рекурсивное. Объяснить, какой тип рекурсии использовался. Посчитайте максимальную глубину спуска, которая потребуется для нахождения ответа (*N* – целое число)

3 Ввод данных пользователем реализовать с клавиатуры, а вывод результатов выполнения программы – в консоль (на экран).

4 Проверить правильность вычислений на тестовых примерах, выполнив серию тестовых прогонов программы.

5 ***Дополнительно*** найти максимальное *N*, при котором разрядности типа `_int64` хватит для хранения данных, и максимально достижимую глубину стека для спуска.

Таблица 5.1 – Варианты заданий для лабораторной работы № 5

Вариант	**Задание**	**Пример**
0	Вывести сумму ***N*** произведений вида: $F(n) = (1) + (2 \cdot 3) + (4 \cdot 5 \cdot 6) + \dots$	Ввод: 4 Ответ: 5167
1	Вывести ***N***-е число последовательности Падована: $P(0) = P(1) = P(2) = 1;$ $P(n) = P(n-2) + P(n-3)$	Ввод: 12 Ответ: 21
2	Вывести ***N***-е число последовательности Перрена: $P(0) = 3;$ $P(1) = 0;$ $P(2) = 2;$ $P(n) = P(n-2) + P(n-3)$	Ввод: 12 Ответ: 29
3	Вывести ***N***-е число последовательности Пелля: $P(0) = 0;$ $P(1) = 1;$ $P(n) = 2 \cdot P(n-1) + P(n-2)$	Ввод: 8 Ответ: 408
4	Вычислить двойной факториал числа ***N***. $n!! = n * (n-2) \cdot (n-4) \cdot \dots \cdot 2$, для четных n; $n!! = n * (n-2) \cdot (n-4) \cdot \dots \cdot 1$, для нечетных n	Ввод: 9 Ответ: 945
5	Вычислить функцию Аккермана ($\boldsymbol{A}(m,n)$): $A(0,n) = n+1;$ $A(m+1,0) = A(m,1);$ $A(m+1,n+1) = A(m,A(m+1,n))$	Ввод: 3 5 Ответ: 253

Продолжение таблицы 5.1

Вариант	Задание	Пример
6	Вывести ***N***-е число последовательности Фибоначчи: $F(0) = 0;$ $F(1) = 1;$ $F(n) = F(n-1) + F(n-2)$	Ввод: 10 Ответ: 55
7	Вывести ***N***-е число последовательности Трибоначчи: $T(0) = 0;$ $T(1) = 0;$ $T(2) = 1;$ $T(n) = T(n-3) + T(n-2) + T(n-1)$	Ввод: 13 Ответ: 504
8	Вывести ***N***-е число Каталана: $\mathrm{C}(0) = 1;$ $\mathrm{C}(n) = \frac{2 \cdot (2n-1)}{n+1} C(n-1)$	Ввод: 8 Ответ: 1430
9	Вывести количество сочетаний C_n^k : $\mathrm{C}_n^0 = 1;$ $C_n^n = 1;$ $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$	Ввод: *n* = 6, *k* = 3 Ответ: 20
10	Вывести ***N***-е число Белла: $a(0) = 1;$ $a(n+1) = \sum_{k=0}^{n} a(k) \cdot \mathrm{C}_n^k$	Ввод: 5 Ответ: 52
11	Вывести число Стирлинга первого рода ($\boldsymbol{s}(n,k)$): $s(0,0) = 1;$ $s(n,0) = 0;$ $s(0,k) = 0;$ $s(n,k) = s(n-1,k-1) - (n-1) \cdot s(n-1,k),$ для $0 < k < n$	Ввод: *n* = 5, *k* = 2 Ответ: –50

Продолжение таблицы 5.1

Вариант	Задание	Пример
12	Вывести число Стирлинга второго рода ($\boldsymbol{s}(n,k)$) : $s(0,0) = 1;$ $s(n,0) = 0;$ $s(0,k) = 0;$ $s(n,k) = s(n-1,k-1) + k \cdot s(n-1,k),$ для $0 < k < n$	Ввод: $n = 5$, $k = 3$ Ответ: 25
13	Вычислить число $\boldsymbol{\pi}$ (Пи) с помощью формулы Браункера с точностью $\boldsymbol{\varepsilon}$ (*Eps*): $\frac{\pi}{4} = \cfrac{1}{1 + \cfrac{1^2}{2 + \cfrac{3^2}{2 + \cfrac{5^2}{2 + \cfrac{7^2}{2 + \cfrac{9^2}{2 + \cdots}}}}}}$	Ввод: 0.0001 Ответ: 3.1415
14	Вычислить субфакториал числа ***N***: $!\,0 = 0;$ $!\,n = !\,(n-1) \cdot n + (-1)^n$	Ввод: 6 Ответ: 265
15	Вычислить ***N***-е число Якобсталя: $J(0) = 0;$ $J(1) = 1;$ $J(n) = J(n-1) + 2 \cdot J(n-2)$	Ввод: 9 Ответ: 171
16	Рекурсивный алгоритм быстрой сортировки (Quick Sort)	–

Пример выполнения лабораторной работы

Программная реализация алгоритма решения задания для 0-го варианта представлена в листинге 5.1.

Экранные формы работы программы, демонстрирующие работу алгоритма, представлены на рисунке 5.1.

Листинг 5.1 – Пример реализации задания по лабораторной работе № 5

```
#include <iostream>
using namespace std;

// объявлены прототипы подпрограмм
uint64_t iterationSolve(uint16_t N);
uint64_t recursionSolve(uint16_t N);
uint16_t sumFrom1ToN(uint16_t N);

// Итерационное решение задачи
uint64_t iterationSolve(uint16_t N) {
    uint64_t sum = 0;
    uint16_t current_term = 1;

    for (uint16_t i = 1; i <= N; i++) {
        uint64_t term = 1;
        for (uint16_t j = current_term; j < current_term+i; j++)
              term *= j;
        current_term += i;
        sum += term;
    }
    return sum;
}

// Функция расчета суммы чисел от 1 до N.
uint16_t sumFrom1ToN(uint16_t N) {
    return N*(N+1)/2;
}

// Рекурсивное решение задачи.
// Прямая рекурсия
uint64_t recursionSolve(uint16_t N) {
    if (N < 1) return 0;

    uint64_t term = 1;
    uint16_t multiplStart = sumFrom1ToN(N-1) + 1;
    for (uint16_t j = multiplStart; j < multiplStart + N; j++)
       term *= j;
    return term + recursionSolve(N-1);
}

int main() {
    uint16_t N;
    // Ввод N
    bool result = static_cast<bool> (cin >> N);
    // Проверка ввода на корректность
    if ((!result) || (N > 10)) {
        cout << "Неверный ввод. Введите число от 0 до 10. \n";
```

```
        return 1;
    }
    // Вызов подпрограммы решения и вывод ответа на экран
    cout << "Ответ (итерационное решение): " << iterationSolve(N)
        << endl;
    cout << "Ответ (рекурсивное решение): " << recursionSolve(N)
        << endl;
    return 0;
}
```

Для проверки допустимых значений программы был проведен расчет максимальной глубины спуска в рекурсию:

$N = 1$, sum = 1;
$N = 2$, sum = 7;
$N = 3$, sum = 127;
$N = 4$, sum = 5167;
$N = 5$, sum = 365527;
$N = 6$, sum = 39435607;
$N = 7$, sum = 6006997207;
$N = 8$, sum = 1226103906007;
$N = 9$, sum = 322796982334807;
$N = 10$, sum = 106460296033918807;
$N = 11$, sum = 42980408446129381207.

Выяснили, что при $N = 11$ сумма превышает максимальное значение типа `uint64_t`, которое составляет 18446744073709551615.

Оценка сложности алгоритма

Для расчета сложности алгоритма определим, что количество чисел, которые будут участвовать в расчетах, равно сумме чисел от 1 до N.

Для $N = 1$ мы считаем одно число.

Для $N = 2$ считаются три числа, т. е. $1 + (2 \cdot 3)$.

Для $N = 3$ получаем выражение $1 + (2 \cdot 3) + (4 \cdot 5 \cdot 6)$ и это шесть чисел.

Можно обнаружить закономерность, что количество чисел, которые мы должны посчитать, в общем случае равно $(N \cdot (N + 1)) / 2$.

Как мы помним, в асимптотике коэффициенты и полиномы меньшего порядка не учитываются, поэтому сложность алгоритма будет $O(N^2)$.

Для расчета максимальной глубины спуска рекурсии определим, что она будет равна $(N + 1)$.

Соответственно, для $N = 0$ (без вызова рекурсии) функция выполнится 1 раз, для $N = 1$ функция выполнится 2 раза: resursionSolve(1) + resursionSolve(0) и т. д.

Если максимальное значение N, при котором программа будет корректно работать, равно 10 (см. ранее), то максимальная достижимая глубина спуска будет равна $(N + 1) = 11$.

```
[tpouhuk@192 lab6]$ ./program
-100
Введите число от 0 до 10.
[tpouhuk@192 lab6]$ ./program
123456
Введите число от 0 до 10.
[tpouhuk@192 lab6]$ ./program
NotANumber
Введите число от 0 до 10.
[tpouhuk@192 lab6]$ ./program
1
Ответ, полученный итерационным способом: 1
Ответ, полученный рекурсивным способом: 1
[touhuk@192 lab6]$ ./program
0
Ответ, полученный итерационным способом: 0
Ответ, полученный рекурсивным способом: 0
[touhuk@192 lab6]$ ./program
4
Ответ, полученный итерационным способом: 5167
Ответ, полученный рекурсивным способом: 5167
[tpouhuk@192 lab6]$ ./program
10
Ответ, полученный итерационным способом: 106460296033918807
Ответ, полученный рекурсивным способом: 106460296033918807
[tpouhuk@192 lab6]$ ./program
5
Ответ, полученный итерационным способом: 365527
Ответ, полученный рекурсивным способом: 365527
```

Рисунок 5.1 – Экранные формы результатов контрольных прогонов программы к лабораторной работе № 5 (вариант 0)

ЛАБОРАТОРНАЯ РАБОТА № 6
Программирование вычислительных процессов на основе пользовательских структур данных и файлов

Цель: изучение возможностей языка C++ для обработки сложных неоднородных структур данных; изучение возможностей языка C++ для организации работы с файловой системой; получение навыков разработки программ с использованием пользовательских структур данных и файлов; получение навыков разработки консольных приложений с организацией пользовательского интерфейса командной строки и обработкой исключительных ситуаций.

Контрольные вопросы для самоподготовки

1 Составные типы данных. Виды и их назначение.
2 Особенности объявления структур в C++.
3 Обращение (доступ) к членам структуры.
4 Инициализация структур. Копирование объектов структурного типа.
5 Определение размера памяти, выделяемой под структуру.
6 Тип данных «объединение». Особенности использования объединений.
7 Вложенные структуры. Массивы структур.
8 Понятия «файл» и «файловая система».
9 Классификация файлов. Текстовые и бинарные типы файлов.
10 Понятие «файловый указатель».
11 Особенности доступа к файлам в C++. Режимы открытия файлов.
12 Основные операции по работе над файлами в C++.
13 Возможности языка C++ по организации ввода и вывода данных с использованием файлов.
14 Способы доступа к данным в файле.
15 Доступ к структурам из нескольких файлов.

Задание

1 Согласовать вариант задания с преподавателем и внимательно изучить исходные данные к задаче (таблица 6.1).

2 Описать алгоритм работы для решения следующей задачи:

Объявить заданную структуру (по варианту). Создать динамический массив из ***S*** таких структур. Организовать пользовательское меню из следующих пунктов: – добавить данные (заполнить одну структуру и добавить в массив);

– очистить данные (удалить массив); – записать текущий массив в текстовый файл; – считать данные из файла в массив; – вывести данные на экран; – выполнить задачи (по варианту); – завершить работу программы

3 Написать программный код реализации составленного алгоритма с учетом требований и ограничений по индивидуальному заданию.

4 Организовать текстовый пользовательский интерфейс и форматированный вывод данных.

5 Проверить правильность вычислений на тестовых примерах, выполнив серию тестовых прогонов программы.

6 ***Дополнительно*** при выполнении задания следует учесть, что правильность вводимых значений не гарантируется (необходимо обеспечить проверку их соответствия указанным требованиям и ограничениям). Дополнительно следует предусмотреть сообщение об ошибке в случае исключительной ситуации. Предусмотреть для всех пунктов возможность пустого массива и контроль переполнения массива (при добавлении данных). Предусмотреть исключительные ситуации для работы с файлом и при вводе некорректных значений.

7 ***Дополнительно*** реализовать пункты меню, через которые будет выполняться сохранение данных в бинарный файл с применением простого алгоритма шифрования (например, шифра Цезаря) и загрузка данных из этого файла с дешифрованием.

Таблица 6.1 – Индивидуальные задания для лабораторной работы № 6

Вариант	**Задание**	
0	Структура «Книги»: название, список авторов, год издания	
	Задачи:	1 Вывести все книги, написанные в году ***N***, отсортировать по лексикографическому возрастанию названия. 2 Вывести список всех авторов без повторений
1	Структура «Социальная сеть»: имя пользователя, возраст, город проживания, список имен друзей	
	Задачи:	1 Вывести всех людей, проживающих в определенном городе, отсортировать по убыванию возраста. 2 Найти всех друзей человека с именем ***N*** и вывести их данные

Продолжение таблицы 6.1

Вариант	Задание	
2	Структура «Песни»: название, исполнитель, длительность, альбом	
	Задачи:	1 Вывести все песни исполнителя ***N***, отсортировать по возрастанию длительности. 2 Вывести топ ***K*** самых длинных альбомов
3	Структура «Футбольная команда»: фамилия, возраст, позиция (вратарь, защитник, нападающий), команда	
	Задачи:	1 Вывести всех игроков, играющих на позиции ***N***, отсортировать по убыванию возраста. 2 Вывести средний возраст команд
4	Структура «Компьютерная сеть»: имя узла, скорость соединения, тип соединения (проводное, беспроводное), IP-адрес, маска сети	
	Задачи:	1 Вывести все узлы с типом соединения ***N***, отсортировать по возрастанию скорости соединения. 2 Используя IP-адрес и маску сети, определить и вывести все узлы, находящиеся в одной сети с узлом ***N***
5	Структура «Компьютерная игра»: название игры, жанр, рейтинг, массив платформ на которых доступна	
	Задачи:	1 Вывести все игры жанра ***N***, отсортировать по убыванию рейтинга. 2 Посчитать процентное отношение жанров игр
6	Структура «Рейсы авиакомпании»: название авиалинии, рейс, город отправления, город прибытия, количество свободных мест, дата и время вылета, дата и время посадки	
	Задачи:	1 Вывести все рейсы в город ***N***, отсортировать по убыванию количества свободных мест. 2 Вывести топ ***K*** самых длинных рейсов
7	Структура «Путешествия»: город, название достопримечательности, рейтинг	
	Задачи:	1 Вывести все достопримечательности в городе ***N*,** отсортировать по возрастанию рейтинга. 2 Найдите город, лучший для посещения. Рейтинг города считается как сумма рейтингов достопримечательностей города
8	Структура «Журнал больницы»: фамилия пациента, код диагноза, дата поступления	
	Задачи:	1 Вывести всех пациентов с диагнозом ***N***, отсортированных по убыванию даты поступления. 2 Вывести по месяцам статистику поступления пациентов в году ***K***

Продолжение таблицы 6.1

Вариант	Задание	
9	Структура «Медицинская клиника»: фамилия пациента, возраст, пол, код диагноза, статус лечения	
	Задачи:	1 Вывести всех пациентов с диагнозом ***N*** и со статусом лечения ***K***, отсортированных по возрасту. 2 Вывести статистику по каждому диагнозу – отношение больных мужского и женского пола
10	Структура «Интернет-магазин»: номер заказа, имя покупателя, сумма заказа, список товаров, дата оформления	
	Задачи:	1 Вывести все заказы, сделанные покупателем ***N***, отсортировать по возрастанию суммы. 2 Вывести топ ***K*** самых покупаемых товаров
11	Структура «Блюда ресторана»: название блюда, цена, описание, категория (закуски, основные блюда, десерты…)	
	Задачи:	1 Вывести все блюда категории ***N***, отсортированные по возрастанию цены. 2 Вывести самое дорогое блюдо в каждой категории, и сколько будет стоить попробовать их все (сумма цен)
12	Структура «Проекты»: название проекта, список разработчиков, дата начала, статус проекта (в плане, в работе, отложен, закончен)	
	Задачи:	1 Вывести все проекты со статусом ***N***, отсортировать по убыванию даты начала работы. 2 Найти самого «крутого» разработчика, у которого наибольшее количество проектов в статусе выполнено
13	Структура «Автомобили»: марка, модель, год выпуска, цвет	
	Задачи:	1 Вывести все автомобили марки ***N***, отсортированные по возрастанию года выпуска. 2 Посчитать топ ***K*** самых популярных моделей авто
14	Структура «Фильмы»: название, режиссер, год выпуска, жанр	
	Задачи:	1 Вывести все фильмы жанра ***N***, отсортированные по убыванию года выпуска. 2 Вывести процентное соотношение жанров фильмов у режиссера ***K***
15	Структура «Банк»: номер счета, владелец, баланс, дата открытия, тип счета	
	Задачи:	1 Для владельца ***N*** вывести все его счета по убыванию баланса. 2 Вывести топ ***K*** самых богатых людей

Продолжение таблицы 6.1

Вариант	Задание	
16	Структура «Автосалон»: производитель, модель, год выпуска, цвет, тип кузова	
	Задачи:	1 Вывести все автомобили от производителя ***N*** в исполнении с типом кузова ***K***, отсортированные по убыванию года их выпуска. 2 Посчитать количество различных цветов автомобилей для каждой модели в салоне

Пример выполнения лабораторной работы

Пример объявления структуры для задания 0-го варианта представлен в листинге 6.1.

Листинг 6.1 – Фрагмент кода для задания по лабораторной работе № 6

```
#include <iostream>
#include <vector>
using namespace std;

// определение структуры
struct stBook {
    string title;
    string* authors;
    int publicYear;
}

int main() {
    ...
    vector <stBook*> lib;
    lib.clear(); // очистить данные (удалить массив);

    // заполнить одну структуру и добавить в массив
    stBook* book = new stBook();
    cin >> book->title;
    cin >> book->publicYear;
    cin >> K;            // K – количество авторов
    book-> authors = new string[K];
    for(int i = 0; i < K; i++)
      cin >> book-> authors[i];
    lib.push_back(book);

    ...
    return 0;
}
```

КОНТРОЛЬНАЯ РАБОТА
Составление алгоритмов. Блок-схемы алгоритмов

Задание

1 Согласовать индивидуальный вариант с преподавателем (таблица 7.1).

2 Для установленного варианта самостоятельно разработать и описать алгоритм решения задачи в виде последовательности операций (описать словами основные шаги) и графически (построить блок-схему).

3 В описании алгоритма обязательно обосновать выбор типов данных для используемых переменных.

4 На защите студент должен знать основную терминологию, знать ответы на контрольные вопросы и уметь объяснить каждый шаг процесса решения своего варианта задания.

Контрольные вопросы

1 Перечислите основные принципы алгоритмизации.

2 Перечислите основные этапы процесса решения задачи.

3 Охарактеризуйте каждый из этапов решения задачи.

4 Дайте определение алгоритма.

5 Перечислите основные свойства алгоритма. Дайте краткую характеристику каждого свойства.

6 Дайте характеристику способов записи алгоритма.

7 Перечислите основные символы, которые применяются для изображения алгоритмов.

8 Перечислите известные вам виды алгоритмов.

9 Дайте характеристику линейного способа записи алгоритма.

10 Дайте характеристику разветвляющегося вычислительного процесса

11 Дайте определению понятию «ветвь вычислений». Охарактеризуйте процесс выбора ветвей при двух условиях и более.

12 Дайте характеристику циклического процесса.

13 Опишите, как задаются исходные данные в циклических процессах.

14 Перечислите составные части циклического вычислительного процесса.

15 Укажите основные различия циклических структур.

16 Раскройте понятие «вложенный цикл».

17 Дайте определение массива.

18 Перечислите основные характеристики массива.

19 Перечислите задачи, для решения которых применяются массивы.

20 Дайте определение бесконечного цикла.

Таблица 7.1 – Индивидуальные задания для контрольной работы

Вари-ант	**Задание**	**Вари-ант**	**Задание**
1	$y = -x^2 + 2$	2	
3	$y = -x^2 + 2$	4	
5		6	
7	$y = \cos x$	8	

Продолжение таблицы 7.1

Вари-ант	Задание	Вари-ант	Задание
9		10	
11		12	
13	$y = x^2$	14	
15	$y = \sqrt{x}$	16	
17		18	

Продолжение таблицы 7.1

Вари-ант	Задание	Вари-ант	Задание
19		20	
21	$y_1 = x^2 - 2$ $y_2 = x$	22	
23		24	
25		26	

Продолжение таблицы 7.1

Вари-ант	Задание	Вари-ант	Задание
27		28	
29	$y = -x^2 + 2$	30	
31		32	
33	$y = x^2 - 2$	34	

КУРСОВАЯ РАБОТА

Цель: углубление теоретических знаний и практических навыков в области алгоритмизации задач и разработки программных средств, развитие навыков самостоятельного изучения предметной области поставленной задачи.

Курсовая работа – учебная работа, содержащая результаты теоретических и/или экспериментальных исследований по отдельной учебной дисциплине и включающая совокупность аналитических, расчетных, исследовательских, оценочных заданий, объединенных общностью рассматриваемого объекта, и предполагающая выполнение отдельных элементов конструкторских, технологических, программных, организационно-управленческих, экономических и других работ и разработку графической документации, в том числе плакатов [1].

Порядок выполнения работы

1 Студент выбирает тему курсовой работы из предложенного списка (приложение В) либо согласует с преподавателем свой вариант темы.

2 Студент получает у руководителя курсовой работы лист задания с краткой постановкой задачи, исходными данными для ее решения, описанием ожидаемых результатов и календарным планом работы (приложение Г).

3 Студент выполняет курсовую работу самостоятельно, посещая плановые консультации (возможно с применением ИКТ) для бесед с руководителем согласно календарному плану.

4 Студент должен выполнить работу к установленной дате, в срок сдать руководителю все разработанные материалы в рамках курсовой работы: программный проект (либо ссылку на репозиторий, где размещен проект), пояснительную записку по выполненной курсовой работе (в формате Word), установленные графические материалы и презентацию (в формате PowerPoint).

5 Студенты проходят защиту перед специально назначенной комиссией из числа преподавателей кафедры, во время которой демонстрируют работу программы. Дополнительно комиссией оценивается полнота выполнения требований к специальной учебной части по базовому предмету.

Требования к реализации курсовой работы

1 Обязательным учебным элементом курсовой работа по дисциплине «Основы алгоритмизации и программирования» определено проектирование и реализация алгоритмов.

2 В работе должно быть описано и реализовано не менее двух авторских алгоритмов.

3 Вид программы – консольное приложение.

4 Рекомендуемые языки программирования – C / C++ / C#.

5 Рекомендуемая среда разработки – Microsoft Visual Studio (Visual Code). Допускается использование других IDE.

Структура пояснительной записки:

1) титульный лист (*приложение Д*);
2) лист с заданием (*приложение Г*);
3) реферат (аннотацию);
4) лист «Содержание»;
5) определения, обозначения, сокращения (*при необходимости*);
6) введение;
7) раздел «Моделирование»;
8) раздел «Программная реализация»;
9) раздел «Тестирование, контрольные прогоны и анализ результатов»;
10) выводы;
11) список используемых источников;
12) приложения (*при необходимости*).

Необходимо строго соблюдать порядок структурных элементов.

Если в пояснительной записке используются сокращения, которые не предусмотрены общими требованиями и правилами [2–4], и/или часто используемые в работе определения и обозначения, то их следует вынести на отдельную страницу.

Введение (*рекомендуемый объем – 1–2 страницы*) должно содержать такие элементы, как объект и предмет исследования, цели и задачи разработки, описание предметной области.

Раздел «*Моделирование*» должен содержать формальное описание задачи, описание и схемы алгоритмов (согласно выбранной теме). Обязательно должна быть проведена оценка сложности алгоритма. Приведено обоснование выбора алгоритма из числа аналогичных ему.

Раздел «*Программная реализация*» должен содержать основные шаги программной реализации алгоритмов, описание используемых библиотек и ключевых функций языка.

Раздел «*Тестирование*» (*рекомендуемый объем – 3–5 страниц*) должен содержать некоторое ограниченное число разработанных тестов для проверки работоспособности программы, результаты выполненного тестирования, анализ результатов тестирования, а также некоторые экспериментальные проверки на

реальных данных. Рекомендуется наличие в пояснительной записке описание примера выполнения алгоритма для решения конкретной задачи.

«*Выводы*» (*рекомендуемый объем – 1–2 страницы*) содержат перечисление основных результатов, характеризующих полноту реализации задач. Результаты следует излагать в форме констатации фактов, используя слова: «изучены», «исследованы», «сформулированы», «показано», «разработана», «предложена», «подготовлены», «изготовлена», «испытана» и т. п. Текст перечислений должен быть кратким, ясным и содержать конкретные данные.

Требования к пояснительной записке:

– объем основной части пояснительной записки курсовой работы составляет 25–40 страниц А4;

– текст пояснительной записки оформляется в соответствии со стандартом предприятия СТП 01–2017 [5];

– список используемых источников должен содержать не менее пяти источников и оформляется в точном соответствии с правилами составления библиографического списка, установленными стандартом ГОСТ 7.1–2003 [6];

– текст программы (авторские фрагменты программного кода) обязательно приводится в приложении(ях) к пояснительной записке. Текст программы должен быть документирован (включать комментарии);

– графическая часть включает в себя схемы алгоритма и программы на листах формата А3 в соответствии со стандартом ГОСТ 19.701–90 [7];

– текст основной части пояснительной записки должен быть проверен на плагиат. Уровень уникальности текста должен быть не менее 51 %. Результат проверки указывается последним абзацем в выводах по работе.

ПРИЛОЖЕНИЕ А
Таблицы кодировок

Таблица А.1 – Таблица кодировки ANSI

Коды управляющих символов		
Код символа	**Символ**	**Примечание**
8	Backspace	Клавиша Backspace
9	Tab	Клавиша Tab
13	Enter	Клавиша Enter
27	Esc	Клавиша Esc

Таблица А.2 – Таблица кодировки Windows 1251

Код символа	**Символ**	**Примечание**
1 – 31	Графические символы	Графические символы
32	‘ ’	Пробел
33	!	Восклицательный знак
34	”	“ ” Двойные кавычки
35	#	Решетка
36	$	Знак денежной единицы
37	%	Знак процента
38	&	Амперсанд
39	'	‘ ’ ' Одинарные кавычки
40	(	Круглая скобка открывающая
41	)	Круглая скобка закрывающая
42	*	Звездочка
43	+	Плюс
44	,	Запятая
45	-	Минус
46	.	Точка
47	/	Наклонная черта (слеш)
48–57	0–9	Цифры
58	:	Двоеточие
59	;	Точка с запятой
60	<	Знак меньше
61	=	Равно
62	>	Знак больше
63	?	Вопросительный знак
64	@	Коммерческое at (собачка)
65–90	A–Z	Прописные латинские буквы
91	[	Квадратная скобка открывающая
92	\	Наклонная черта (обратный слеш)

Продолжение таблицы А.2

Код символа	**Символ**	**Примечание**
93	]	Квадратная скобка закрывающая
94	^	Циркумфлекс
95	_	Нижнее подчеркивание
96	`	Ударение
97–122	a–z	Строчные латинские буквы
123	{	Фигурная скобка открывающая
124	|	Прямая черта
125	}	Фигурная скобка закрывающая
126	~	Тильда
127–191	Специальные символы	Специальные символы
161	Ў	Прописная белорусская буква «Ў»
162	ў	Строчная белорусская буква «Ў»
167	§	Знак параграфа
168	Ё	Прописная русская буква «Ё»
171	«	Двойные кавычки открывающие
176	°	Знак градуса
177	±	Знак плюс-минус
184	ё	Строчная русская буква «ё»
185	№	Знак номера
187	»	Двойные кавычки закрывающие
192–223	А–Я	Прописные русские буквы
224–255	а–я	Строчные русские буквы

ПРИЛОЖЕНИЕ Б
Структура модуля программы Delphi

Код	Пояснение
unit Unit1;	Имя модуля формы
{$R grafiki.res}	Директива подключения файла ресурсов grafiki.res
interface	**Раздел интерфейса**
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, Unit2;	Подключение библиотечных модулей для глобального пользования, используемых данным модулем формы Unit1
type TForm1 = class(TForm) Edit1: TEdit; Edit2: TEdit; Label1: TLabel; Button1: TButton; **procedure** ButtonlClick(Sender: TObject);	Объявление класса формы и описание всех объектов и методов данного класса (имена объектов, заголовки созданных процедур обработки событий и т. п.)
private *{ Private declarations }* public { *Public declarations }* **end;**	Подразделы public и private, устанавливающие степень доступности элементов класса в программе при использовании концепции ООП

label
metka1;
type
massiv=array[0..100] of integer;
const
pi=3.14;
var
Form1: TForm1;
i:integer;
s:string;
	Объявление глобальных метки, типа, константы, переменных

function Fun(x:integer):integer; **procedure** Pro(z:extended);	Описание заголовков (объявление) глобальных функции Fun и процедуры Pro
implementation	**Раздел реализации**
{$R *.dfm}	Директива подключения файла описания формы Unit.dfm

Код	Пояснение
uses Math, Print, Unit2;	Подключение библиотечных модулей для локального использования (внутри этого модуля)

label
vvod;
type
bukvy=array[0..32] of char;
const
g=9.8;
var
x,y,z:extended;
	Объявление локальных для данного модуля метки, типа, константы, переменных

Function summa(i:integer):extended;
begin
……..
end;
	Локальная функция summa

procedure TForm1.ButtonlClick (Sender: TObject);	Процедура обработки события onClick для объекта Button1
label metka2; **type** vektor=word; **const** e=2.7; **var** a: integer;	Объявление локальных для данной процедуры обработки события метки, типа, константы, переменной

begin
…
Pro(z);
Fun (i);
….
end;
	Тело процедуры (алгоритм), реализующий получение исходных данных, расчет и вывод результатов. В данном случае также осуществляется вызов глобальной процедуры Pro и функции Fun

	Раздел инициализации (может отсутствовать)

begin
…….
end;
	Инструкции инициализации

end.	Конец кода модуля формы

ПРИЛОЖЕНИЕ В
Перечень тем курсовых работ

1 Решение системы линейных алгебраических уравнений методом Гаусса.

2 Решение системы линейных алгебраических уравнений методом прогонки.

3 Решение системы линейных алгебраических уравнений методом квадратного корня.

4 Решение системы линейных алгебраических уравнений методом простой итерации.

5 Решение системы линейных алгебраических уравнений методом Зейделя.

6 Аппроксимация функции интерполяционным многочленом Ньютона.

7 Аппроксимация функции линейной интерполяцией.

8 Аппроксимация функции квадратичной интерполяцией.

9 Аппроксимация функции интерполяционным многочленом Лагранжа.

10 Аппроксимация функции (интерполяция) методом Гаусса.

11 Аппроксимация функции методом наименьших квадратов.

12 Вычисление производных и интегралов по формуле средних.

13 Вычисление производных и интегралов по формуле трапеций.

14 Вычисление производных и интегралов по формуле Симпсона.

15 Вычисление производных и интегралов по формуле Гаусса.

16 Решение нелинейных уравнений методом простой итерации.

17 Решение нелинейных уравнений методом Ньютона.

18 Решение нелинейных уравнений методом секущих.

19 Решение нелинейных уравнений методом Вегстейна.

20 Решение нелинейных уравнений методом парабол.

21 Решение нелинейных уравнений методом деления отрезка пополам.

22 Нахождение минимума функции одной переменной методом деления отрезка пополам.

23 Нахождение минимума функции одной переменной методом золотого сечения.

24 Нахождение минимума функции одной переменной методом последовательного перебора.

25 Нахождение минимума функции одной переменной методом квадратичной параболы.

26 Нахождение минимума функции одной переменной методом кубической параболы.

27 Решение задачи Коши для обыкновенных дифференциальных уравнений методом явной схемы 1-го порядка.

28 Нахождение минимума функции одной переменной методом неявной схемы 1-го порядка.

29 Нахождение минимума функции одной переменной методом неявной схемы 2-го порядка.

30 Нахождение минимума функции одной переменной методом Рунге – Кутты.

31 Нахождение минимума функции одной переменной методом явной экстраполяционной схемы Адамса 2-го порядка.

32 Нахождение минимума функции одной переменной методом явной экстраполяционной схемы Адамса 3-го порядка.

33 Нахождение минимума функции одной переменной методом неявной схемы Адамса 3-го порядка.

34 Анализ алгоритма для вычисления формул в обратной польской записи.

35 Представление математических формул в виде бинарных деревьев.

36 Интерпретатор алгебраических инфиксных выражений.

37 Анализ алгоритма Дейкстры для поиска кратчайшего пути.

38 Сравнительный анализ алгоритмов сортировки обменами.

39 Сравнительный анализ алгоритмов сортировки вставками и выбором.

40 Анализ алгоритма сортировки методом пузырька.

41 Анализ алгоритма шейкерной сортировки.

42 Анализ алгоритма сортировки расческой.

43 Анализ алгоритма быстрой сортировки.

44 Анализ алгоритма сортировки слиянием.

45 Анализ алгоритма пирамидальной сортировки.

46 Анализ алгоритма сортировки Шелла.

47 Сравнительный анализ алгоритмов простой сортировки вставками и сортировки Шелла.

48 Анализ алгоритма линейного поиска.

49 Анализ алгоритма бинарного поиска.

50 Анализ алгоритма поиска подстроки в строке.

51 Сравнительный анализ методов поиска данных в отсортированных последовательностях.

52 Анализ алгоритмов самобалансирующихся деревьев поиска на примере красно-черного дерева.

53 Анализ алгоритмов самобалансирующихся деревьев поиска на примере АВЛ-дерева.

54 Анализ алгоритмов самобалансирующихся деревьев поиска на примере расширяющегося дерева.

55 Анализ сжатия данных на основе алгоритма Хаффмана.

56 Анализ сжатия файлов методом RLE.

57 Анализ сжатия файлов методом LZW.

58 Анализ алгоритма шифрования данных AES.

59 Анализ алгоритма шифрования данных RSA.

60 Сравнительный анализ методов простой подстановки при шифровании текстовых файлов.

ПРИЛОЖЕНИЕ Г
Шаблон листа задания на курсовую работу

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»
Институт информационных технологий
Факультет компьютерных технологий

УТВЕРЖДАЮ
Заведующий кафедрой ИСиТ
____
(подпись)
_________________202_ г.

ЗАДАНИЕ
на курсовую работу

студенту ________Фамилия Имя Отчество______________________

1. Тема работы ________*«Анализ алгоритма быстрой сортировки»*____________

____

2. Срок сдачи студентом законченной работы__15.05.2025___

3. Исходные данные к работе

____

____

____

4. Содержание пояснительной записки

Введение

1. Моделирование программного средства

2. Проектирование программного средства

3. Оценка работы (тестирование) программного средства и анализ результатов

Выводы

Список используемых источников

Приложение А *(обязательное)* Фрагменты программного кода

5. Перечень графического материала

1. Название ПС, схема программы, чертеж – формат А3, лист 1.

2. Название алгоритма 1, схема алгоритма, чертеж – формат А3, лист 1.

3. Название алгоритма 2, схема алгоритма, чертеж – формат А3, лист 1.

6. Консультант по курсовой работе __Фамилия И. О.__ *(* при необходимости)*

7. Дата выдачи задания ______02.02.2025______

8. Календарный график работы над курсовой работой на весь период (с обозначением сроков выполнения и процентом от общего объема работы):

№	Содержание работ	Срок выполнения	% от общего объема работы
1	Раздел 1	03.03.2025	25 %
2	Раздел 2	04.04.2025	50 %
3	Раздел 3	05.05.2025	75 %
4	Оформление пояснительной записки и графического материала	15.05.2025	100 %
5	Защита курсовой работы		

Руководитель ________________ / И. О. Фамилия /
(подпись)

Задание принял к исполнению ____________________________
(дата и подпись студента)

ПРИЛОЖЕНИЕ Д
Шаблон титульного листа курсовой работы

Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»
Институт информационных технологий

Факультет компьютерных технологий

Кафедра информационных систем и технологий

Дисциплина: Основы алгоритмизации и программирования (ОАиП)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

ВАША ТЕМА ЗАГЛАВНЫМИ БУКВАМИ

Студент:
гр. 480000 Фамилия И. О.

Руководитель:
должность Фамилия И. О.

Оценка:
____

Минск 2025

ПРИЛОЖЕНИЕ Е
Рамки чертежей формата А3

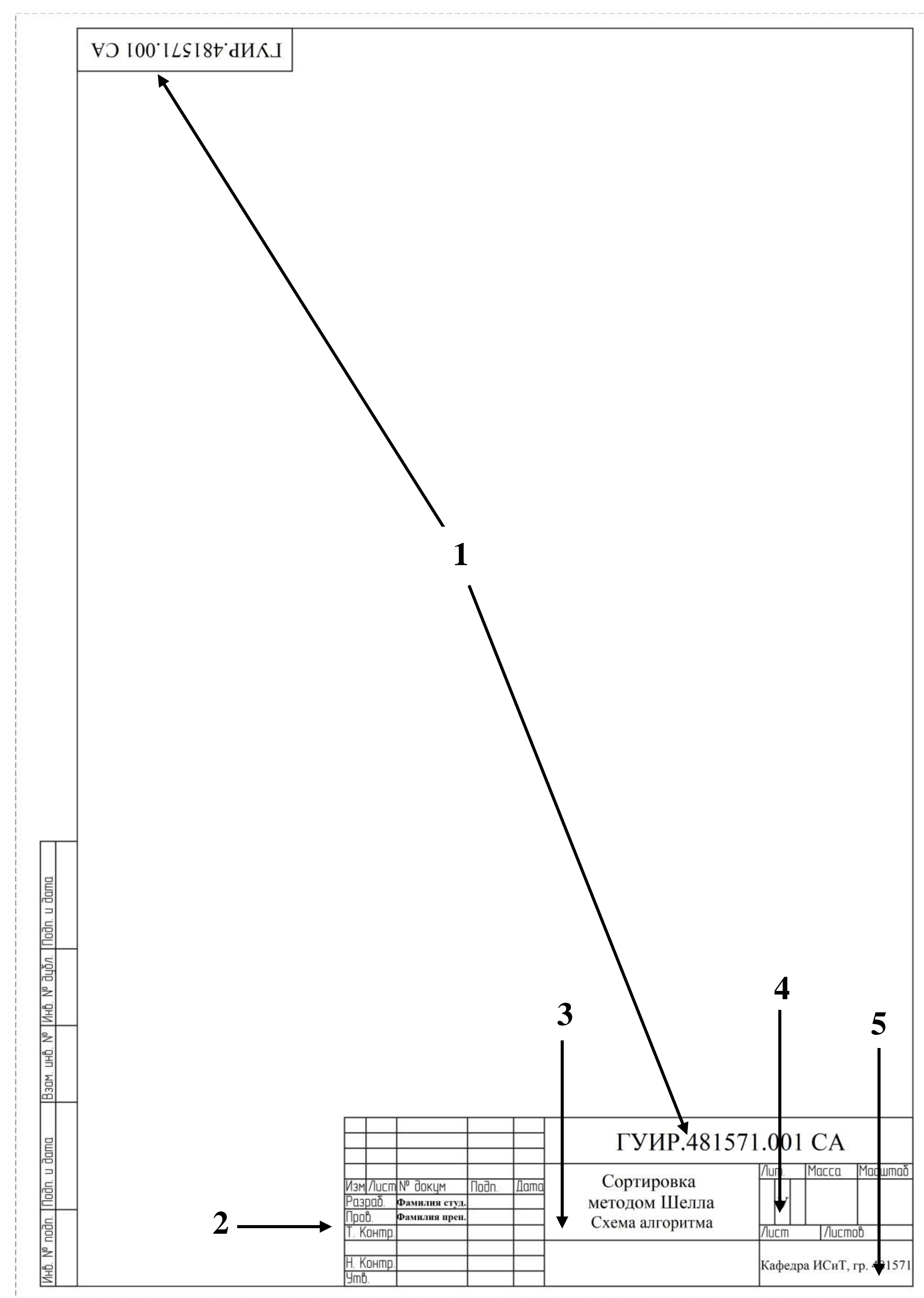


Рисунок Е.1 – Вертикальная ориентация рамки чертежа

Примечания

1 Шифр проекта: ГУИР, точка, номер группы, точка, номер чертежа в курсовой работе, пробел, СА (буквенный код чертежа – схема алгоритма).

2 В графе «Разработал» проставляется фамилия студента, в графе «Проверил» – руководителя курсовой работы (преподавателя).

3 Проставляется «*Название чертежа*». Схема алгоритма (т. е. его вид).

4 Стадия разработки. Проставляется литера «У» (учебный).

5 Организация разработчик. Проставляется кафедра и номер группы.

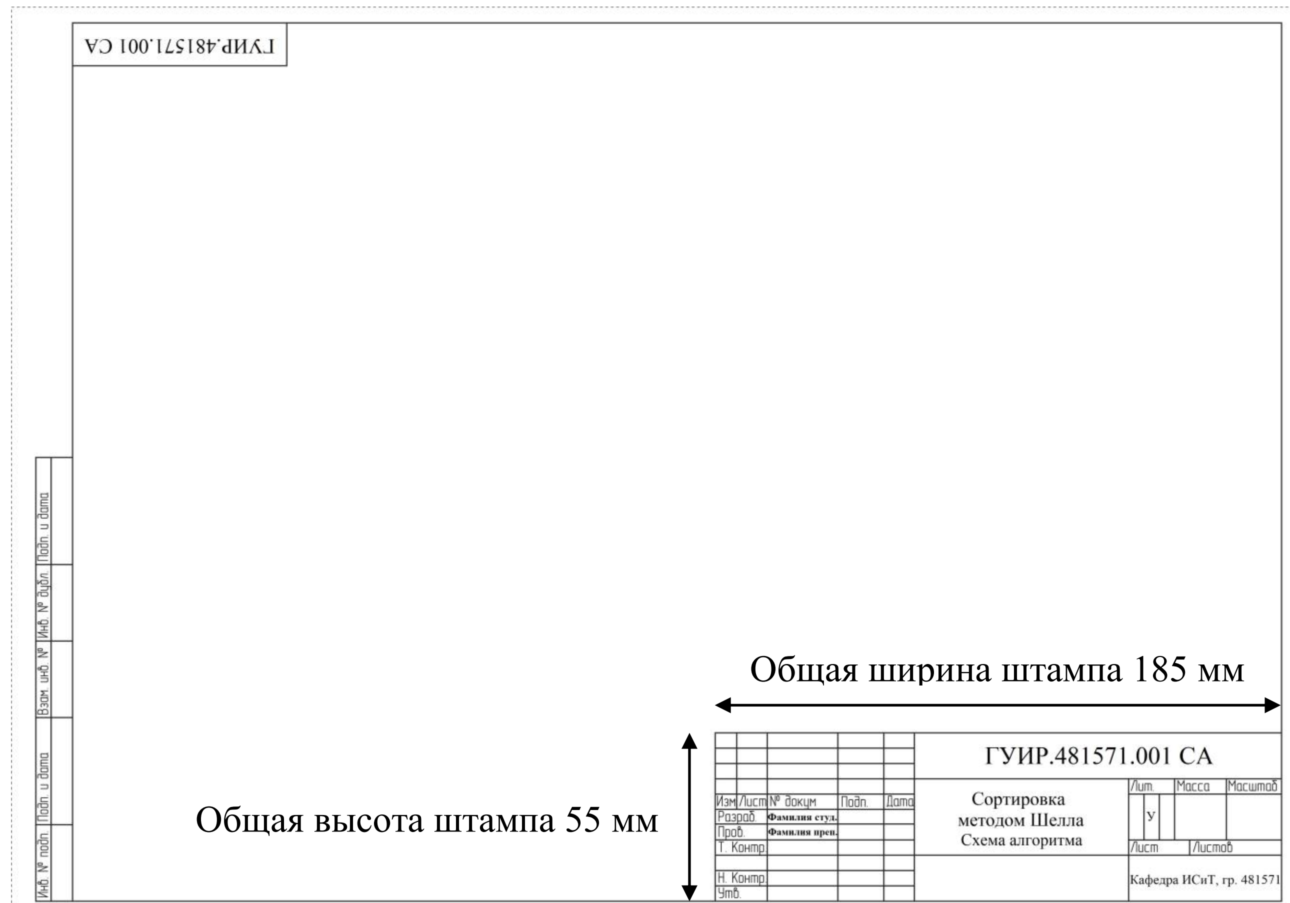


Рисунок Е.2 – Горизонтальная ориентация рамки чертежа

7	10	23	15	10	70	50		
					ГУИР.481571.001 СА			
					Сортировка методом Шелла. Схема алгоритма	*Литера*	*Масса*	*Масштаб*
Изм	*Лист*	*№ докум.*	*Подпись*	*Дата*		*У*		
Разработал		Фамилия студ.						
Проверил		Фамилия преп.						
Т.контроль						*Лист*	*Листов*	
Рецензент						*Кафедра ИСиТ, гр.481571*		
Н.контроль								
Утвердил								

Рисунок Е.3 – Пример заполнения и размеры (в миллиметрах) штампа чертежа

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1 Голицына, О. Л. Основы алгоритмизации и программирования : учеб. пособие / О. Л. Голицына, И. И. Попов. – 4-е изд., испр. и доп. – М. : ФОРУМ: Инфра-М, 2021. – 431 с.

2 Златопольский, Д. М. Сборник задач по программированию / Д. М. Златопольский. – 3-е изд., перераб. и доп. – СПб. : БХВ-Петербург, 2011. – 304 с.

3 Павловская, Т. А. С/С++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2010. – 461 с.

4 Хабибуллин, И. Программирование на языке высокого уровня С/С++ / И. Хабибуллин. – СПб. : БХВ-Петербург, 2006. – 512 с.

5 Демидович, Е. М. Основы алгоритмизации и программирования. Язык Си / Е. М. Демидович. – 2-е изд., испр. и доп. – СПб. : БХВ-Петербург, 2008. – 438 с.

6 Страуструп, Б. Язык программирования С++ / Б. Страуструп. – М. : Бином, 2012. – 1104 с.

7 Страуструп, Б. Программирование: принципы и практика использования С++ / Б. Страуструп ; пер. с англ. Д. А. Клюшина. – Испр. изд. – М. : Вильямс, 2011. – 1248 с.

8 Кнут, Д. Искусство программирования. В 4 т. Т. 1–3 / Д. Кнут. – М. : Вильямс, 2004. – 486 с.

9 Шилт, Г. Самоучитель С++ / Г. Шилдт. – СПб. : BHV, 2009. – 688 с.

10 Шилт, Г. С++ для начинающих / Г. Шилдт ; пер. с англ. С. Черников – СПб. : Питер, 2024. – 608 с.

11 Кормен, Т. Х. Алгоритмы: вводный курс / Т. Х. Кормен. – М. : Вильямс, 2014. – 208 с.

12 Алгоритмы: построение и анализ / Т. Х. Кормен [и др.]. – 3-е изд. – М. : Вильямс, 2016. – 1328 с.

13 Фридман, А. С/С++. Алгоритмы и приемы программирования / А. Фридман. – М. : Бином, 2007. – 560 с.

14 Шагбазян, Д. В. Алгоритмы сортировки. Анализ, реализация, применение : учеб. пособие / Д. В. Шагбазян, А. А. Штанюк, Е. В. Малкина. – Н. Новгород : НГУ им. Н. И. Лобачевского, 2019. – 42 с.

15 Стивенс, Р. Алгоритмы. Теория и практическое применение / Р. Стивенс. – М. : Э, 2016. – 544 с.

16 Стивенс, Р. Delphi. Готовые алгоритмы / Р. Стивенс. – СПб. : ДМК Пресс : Питер, 2004. – 384 с.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Положение об организации и проведении курсового проектирования в БГУИР. – Введ. 2013–12–11. – Минск : БГУИР, 2013. – 18 с.

2 ГОСТ Р 7.0.12–2011. Библиографическая запись. Сокращение слов на русском языке: общие требования и правила. – Введ. 2011–12–13. – М. : Стандартинформ, 2012.

3 Библиографическая запись. Сокращение слов и словосочетаний на иностранных европейских языках. ГОСТ 7.11–2004 (ИСО 832:1994) – Минск : Госстандарт Респ. Беларусь ; Белорус. гос. ин-т стандартизации и сертификации, 2005. – 83 с.

4 Бібліяграфічны запіс. Скарачэнне слоў і словазлучэнняў на беларускай мове = Библиографическая запись. Сокращение слов и словосочетаний на белорусском языке : агул. патрабаванні і правілы. СТБ 7.12–2001. – Мінск : Дзяржстандарт : Белорус. гос. ин-т стандартизации и сертификации, 2002. – 19 с.

5 СТП 01–2017. Дипломные проекты (работы): общие требования. – Введ. 2018. – Утв. с изм. 24.11.2017. – Минск : БГУИР, 2017. – 169 с.

6 Библиографическая запись. Библиографическое описание. Общие требования и правила составления. ГОСТ 7.1–2003 – Минск : Госстандарт Респ. Беларусь, 2004. – 48 с.

7 ГОСТ 19.701–90 (ИСО 5807–85). Схемы алгоритмов, данных, программ и систем. Условные обозначения и правила выполнения. – Введ. 1992–01–01. – М. : Стандартинформ, 2010.

Св. план 2023, поз. 40

Учебное издание

Парамонов Антон Иванович
Савенко Андрей Геннадьевич

АЛГОРИТМИЗАЦИЯ И КОМПЬЮТЕРНЫЕ ВЫЧИСЛЕНИЯ НА ЯЗЫКАХ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 11.07.2024. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 12,44. Уч.-изд. л. 12,1. Тираж 100 экз. Заказ 176.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск