Учреждение образования
«Брестский государственный университет имени А.С. Пушкина»

Методы алгоритмизации

электронный курс лекций
для студентов физико-математического факультета
специальностей 1-02 05 01 «Математика и информатика»,
1-02 05 02 «Физика и информатика»

Брест
БрГУ имени А.С. Пушкина
2022

кафедра прикладной математики и информатики

Автор-составитель:

Ткач Светлана Николаевна – старший преподаватель кафедры прикладной математики и информатики УО «Брестский государственный университет имени А.С. Пушкина»

Рецензенты:

Парфомук С.И. – заведующий кафедрой информатики и прикладной математики Учреждения образования «Брестский государственный технический университет», кандидат технических наук, доцент

Зубей Е.В. – доцент кафедры алгебры, геометрии и математического моделирования учреждения образования «Брестский государственный университет имени А.С. Пушкина», кандидат физико-математических наук

Электронный курс лекций содержит материалы, направленные на формирование профессиональных компетенций преподавателя информатики в области технологий программирования и методов алгоритмизации у будущих педагогов. Содержит материалы с разветвленной системой гиперссылок на необходимый теоретический и демонстрационный материал, с пояснениями для самостоятельного усвоения темы. Курс лекций также содержит вопросы к зачету со ссылками на развёрнутые ответы, тесты для определения усвоения студентами основных тем курса, основные литературные источники и др.

Данные учебно-методические материалы ориентированы на их практическое использование обучающимися специальностей 1-02 05 01 Математика и информатика, 1-02 05 02 Физика и информатика, студентами других специальностей педагогического профиля, преподавателями, педагогическими работниками учреждений образования.

СОДЕРЖАНИЕ

Предисловие 6
Примерный тематический план 7
Содержание учебного материала 8

Тема 1 **Синтаксис языка программирования** **11**
1.1 Интуитивное понятие алгоритма 11
1.1.1 Понятие алгоритма 16
1.1.2 Виды алгоритмов 18
1.2 Методология и этапы решения задач с помощью компьютера 19
1.2.1 Способы представления алгоритмов 23
1.2.2 Блок-схемы алгоритмов, основные элементы 25
1.2.3 Базовые структуры 28
1.2.4 Вложенные циклы 33
1.3 Парадигма структурного программирования 35
1.3.1 Методы разработки алгоритма 39
1.3.2 Технологии (парадигмы) программирования 40
1.3.3 Процедурное программирование 40
1.3.4 Функциональное программирование 41
1.3.5 Логическое программирование 42
1.3.6 Объектно-ориентированное программирование 43
1.3.7 Визуальное программирование 45
1.4 Системы и среды программирования 46
1.5 Основные элементы языка программирования 49
1.5.1 Компиляция 49
1.5.2 Синтаксис языка программирования Pascal (Delphi) 50
1.5.3 Тип данных 53
1.5.4 Скалярные (простые) типы 54
1.5.5 Целые типы данных 56

1.5.6 Вещественные типы данных . . . 58
1.5.7 Логический тип данных . . . 59
1.5.8 Структура программы . . . 60
1.5.9 Переменная . . . 61
1.5.10 Оператор присваивания . . . 63
1.5.11 Операции и выражения . . . 65
1.5.12 Приоритет операций . . . 70
1.5.13 Основные арифметические функции . . . 71
1.5.14 Функции преобразования типов . . . 73

Тема 2 **Операторы языка программирования** **74**
2.1 Условный оператор If (ветвления) . . . 75
2.2 Оператор выбора Case . . . 81
2.3 Оператор цикла For . . . 87
2.4 Оператор цикла While . . . 93
2.5 Оператор цикла Repeat . . . 97

Тема 3 **Подпрограммы** **100**
3.1 Структура подпрограммы . . . 100
3.2 Типы параметров . . . 103
3.3 Рекурсия . . . 105

Тема 4 **Структурированные типы данных. Символьный и строковый типы данных** **106**
4.1 Символьный тип . . . 107
4.2 Строковый тип . . . 111
4.2.1 Операции над строками . . . 113

Тема 5 **Массивы** **116**
5.1 Операции с массивами . . . 118

5.2 Динамические массивы 119
5.3 Типовые операции с массивами 122
5.3.1 Поиск элементов массива 124
5.4 Методы упорядочивания элементов массива 129

Тема 6 **Множества** **136**
6.1 Операции над множествами 137

Тема 7 **Основы работы с файлами** **142**
7.1 Текстовые файлы 147
7.2 Типизированные файлы 150
7.3 Нетипизированные файлы 152

Тема 8 **Тип данных Запись** **154**
Приложения 162
Вопросы к зачету 162
Основные определения 165
Список рекомендуемой литературы 168

Предисловие

Предлагаемый вниманию читателей электронный курс лекций (далее – ЭКЛ) является первой частью серии учебно-методических материалов для студентов специальностей 1-02 05 01 «Математика и информатика», 1-02 05 02 «Физика и информатика» по учебной дисциплине «Методы алгоритмизации».

ЭКЛ «Методы алгоритмизации» разработан в соответствии с образовательными стандартами высшего образования специальностей 1-02 05 01 «Математика и информатика», 1-02 05 02 «Физика и информатика»; учебных планов учреждения высшего образования по специальностям 1-02 05 01 «Математика и информатика», рег. № ФМ-34-21/уч., 1-02 05 02 «Физика и информатика», рег. № ФМ-33-21/уч., утвержденных 05.07.2021.

ЭКЛ содержит теоретический материал по темам учебной программы. Раздел контроля знаний включает тестовые задания, вопросы для проведения промежуточного контроля сформированности соответствующих компетенций студентов. Вспомогательный раздел включает элементы учебной программы по дисциплине «Методы алгоритмизации» (пояснительная записка, примерный тематический план, содержание учебного материала), содержит список рекомендуемой литературы.

ПРИМЕРНЫЙ ТЕМАТИЧЕСКИЙ ПЛАН

№	Название темы	УСР	ЛК	ЛБ
1	Синтаксис языка Pascal		2	
2	Операторы языка Pascal		4	8
3	Алгоритмы решения задач целочисленной арифметики		2	2
4	Линейные массивы		4	4
5	Двумерные массивы			4
6	Структурированные типы данных. Символьный и строковый типы данных		4	4
7	Подпрограммы		2	
8	Основы работы с файлами		2	2
	ИТОГО		**20**	**24**

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

Тема 1. Синтаксис языка Pascal

Интуитивное понятие алгоритма. Свойства алгоритмов. Способы представления алгоритмов. Стандартизация графического представления алгоритмов. Правила оформления блок-схем. Методы разработки и анализа алгоритмов. Реализация алгоритма в виде программы.

Методология и этапы решения задач с помощью компьютера. Роль и характеристики языков программирования. Проблема универсального языка программирования и универсальной вычислительной машины. Классификации языков программирования. Компилируемые и интерпретируемые языки. Состав и назначение систем программирования. Компиляторы и интерпретаторы. Языки школьной информатики.

Алфавит языка, синтаксис, структура программы, переменная, константы. Типы данных языка Pascal: классификация и описания. Простые типы: порядковые (целые, логический, символьный, тип-диапазон, перечислимый тип) и вещественные типы.

Арифметические выражения: функции, операции и порядок действий. Тип выражения. Унарные и бинарные операции. Совместимость и преобразования типов данных. Операция присваивания. Арифметические операции. Операции сравнения. Приоритет операций. Стандартные функции. Математические функции. Комментарии в программе. Процедуры ввода и вывода. Форматы вывода числовых данных.

Тема 2. Операторы языка Pascal

Концепция структурного программирования. Базовые алгоритмические конструкции структурного программирования: следование, ветвление, цикл.

Условный оператор if: синтаксис, семантика. Полная и сокращенная формы условного оператора. Простое и сложное условие. Тип условия.

Оператор выбора case: синтаксис, семантика. Полная и сокращенная формы оператора выбора. Понятие и тип селектора.

Понятия «цикл», «тело цикла», «итерация», «условие цикла», «зацикливание».

Оператор цикла с заранее известным количеством повторений For, его синтаксис и семантика. Понятие счетчика цикла, тип счетчика.

Оператор цикла с предусловием While, его синтаксис и семантика.

Оператор цикла с постусловием Repeat, его синтаксис и семантика.

Тема 3. Алгоритмы решения задач целочисленной арифметики

Алгоритмы целочисленной арифметики. Нахождение наибольшего общего делителя, наименьшего общего кратного натуральных чисел. Поиск чисел с заданными свойствами (простых, палиндромов, и др.) Разложение чисел на простые множители. Перевод чисел из одной системы счисления в другую. Делимость чисел. Действия с многозначными числами. Действия с дробями. Задачи комбинаторики. Анализ алгоритмов.

Тема 4. Линейные массивы

Структурированные типы данных. Понятие массива. Одномерные, двумерные и многомерные массивы. Способы задания одномерных массивов. Динамические массивы.

Операции над элементами массива: доступ к элементам массива, удаление элементов из одномерного массива, вставка элементов в одномерный массив, перестановка элементов массива.

Методы алгоритмизации: поиск в массиве, упорядочивание (сортировка) элементов массива, циклические перестановки элементов массива. Поиск в массивах элементов с заданными свойствами (свойства элементов, лежащих на (под, над) главной и побочной диагоналями квадратной матрицы и др.).

Тема 5. Двумерные массивы

Понятие двумерного массива. Описание типа массива. Формирование значений элементов массива случайным образом.

Работа с элементами массива. Вставка и удаление строк и столбцов. Перестановка групп элементов массива.

Тема 6. Структурированные типы данных. Символьный и строковый

типы данных

Тип данных char. Операции над символами.

Понятие строковой величины: определение, компоненты строковой величины, доступ к компонентам, особенность кодировки. Тип данных string. Строковые переменные, их описание. Длина строки. Операции над строками.

Стандартные функции для работы со строками (concat, copy, length, pos, upcase). Стандартные процедуры для работы со строками (delete, insert, str, val).

Понятия «множество», «базовый тип множества». Описание множеств. Операции над множествами.

Понятия «запись», «поле записи». Описание записей. Массивы записей. Оператор with.

Тема 7. Подпрограммы

Понятие подпрограммы. Виды подпрограмм в Pascal. Локальные и глобальные переменные. Процедуры пользователя. Структура процедуры пользователя. Организация вызова процедуры пользователя. Типы параметров (фактические и формальные, параметры-значения и параметры-переменные). Виды процедур пользователя: процедуры без параметров, процедуры с параметрами-значениями, процедуры с параметрами-значениями и параметрами-переменными.

Функции пользователя. Структура функции пользователя. Организация вызова функции пользователя. Отличия функции пользователя от процедуры пользователя.

Понятие итерации. Понятие рекурсии. Рекурсивные подпрограммы.

Тема 8. Основы работы с файлами

Понятие файлов. Виды файлов. Типизированные файлы. Работа с типизированными файлами: создание файла, использование данных из файла, дополнение файла новыми данными.

РАЗДЕЛ 1
Синтаксис языка программирования

1.1 Интуитивное понятие алгоритма

Человек ежедневно встречается с множеством задач, возникающих в различных областях деятельности общества, например:

а) подготовиться к уроку по математике;

б) приготовить раствор для проявления фотопленки;

в) избавиться от лишнего веса.

Для решения задач надо знать, что дано, и что следует получить. Другими словами, задача представляет собой **совокупность двух объектов:** исходных данных и искомых результатов. Чтобы получить результаты, необходимо знать **метод решения задачи**, то есть располагать предписанием (инструкцией, правилом), в котором указано, какие действия и в каком порядке следует выполнить, чтобы решить задачу (получить искомые результаты). Предписание, определяющее порядок выполнения действий над данными с целью получения искомых результатов, называется алгоритмом.

Алгоритм - это точная конечная система правил, определяющая содержание и порядок действий исполнителя над некоторыми объектами (исходными и промежуточными данными) для получения (после конечного числа шагов) искомого результата.

Следует иметь в виду, что это – не определение в математическом смысле слова, но довольно подробное описание понятия алгоритма, раскрывающее его сущность. Описание может быть другим. Так, в школьном учебнике по информатике понятие алгоритма дается в следующей форме: *«Под алгоритмом понимают понятное и точное предписание исполнителю совершить последовательность действий, направленных на решение поставленной задачи».*

Понятие алгоритма является одним из основных понятий современной математики и является объектом исследования специального раздела математики - **теории**

алгоритмов.

История термина

Современное формальное определение алгоритма было дано в 30—50-х годы XX века в работах Тьюринга, Поста, Чёрча (тезис Чёрча — Тьюринга), Н. Винера, А. А. Маркова.

Само слово «алгоритм» происходит от имени учёного Абу Абдуллах Мухаммеда ибн Муса аль-Хорезми. Около 825 года он написал сочинение, в котором впервые дал описание придуманной в Индии позиционной десятичной системы счисления. К сожалению, арабский оригинал книги не сохранился. Аль-Хорезми сформулировал правила вычислений в новой системе и, вероятно, впервые использовал цифру 0 для обозначения пропущенной позиции в записи числа (её индийское название арабы перевели как as-sifr или просто sifr, отсюда такие слова, как «цифра» и «шифр»). Приблизительно в это же время индийские цифры начали применять и другие арабские учёные. В первой половине XII века книга аль-Хорезми в латинском переводе проникла в Европу. Переводчик, имя которого до нас не дошло, дал ей название Algoritmi de numero Indorum («Алгоритмы о счёте индийском»). По-арабски же книга именовалась Китаб аль-джебр валь-мукабала («Книга о сложении и вычитании»). Из оригинального названия книги происходит слово Алгебра.

Таким образом, мы видим, что латинизированное имя среднеазиатского ученого было вынесено в заглавие книги, и сегодня ни у кого нет сомнений, что слово «алгоритм» попало в европейские языки именно благодаря этому сочинению. Однако вопрос о его смысле длительное время вызывал ожесточённые споры. На протяжении многих веков происхождению слова давались самые разные объяснения.

Одни выводили algorism из греческих algiros (больной) и arithmos (число).

Из такого объяснения не очень ясно, почему числа именно «больные». Или же лингвистам больными казались люди, имеющие несчастье заниматься вычислениями? Своё объяснение предлагал и энциклопедический словарь Брокгауза и Ефрона. В нём алгорифм (кстати, до революции использовалось написание алгоритм, через фиту) производится «от арабского слова Аль-Горетм, то есть корень». Разумеется, эти объяснения вряд ли можно счесть убедительными.

Упомянутый выше перевод сочинения аль-Хорезми стал первой ласточкой, и в течение нескольких следующих столетий появилось множество других трудов, посвящённых всё тому же вопросу — обучению искусству счёта с помощью цифр. И все они в названии имели слово algoritmi или algorismi.

Однако со временем такие объяснения всё менее занимали математиков, и слово algorism (или algorismus), неизменно присутствовавшее в названиях математических сочинений, обрело значение способа выполнения арифметических действий посредством арабских цифр, то есть на бумаге, без использования абака. Именно в таком значении оно вошло во многие европейские языки. Например, с пометкой «устар.» оно присутствует в представительном словаре английского языка Webster's New World Dictionary, изданном в 1957 г.

Алгоритм — это искусство счёта с помощью цифр, но поначалу слово «цифра» относилось только к нулю.

Постепенно значение слова расширялось. Учёные начинали применять его не только к сугубо вычислительным, но и к другим математическим процедурам.

Можно обратить внимание на то, что первоначальная форма algorismi спустя какое-то время потеряла последнюю букву, и слово приобрело более удобное для европейского произношения вид algorism. Позднее и оно, в свою очередь, подверглось искажению, скорее всего, связанному со словом

arithmetic.

Однако потребовалось ещё почти два столетия, чтобы все старинные значения слова вышли из употребления. Этот процесс можно проследить на примере проникновения слова «алгоритм» в русский язык.

Историки датируют 1691 годом один из списков древнерусского учебника арифметики, известного как «Счётная мудрость». Это сочинение известно во многих вариантах (самые ранние из них почти на сто лет старше) и восходит к ещё более древним рукописям XVI в. По ним можно проследить, как знание арабских цифр и правил действий с ними постепенно распространялось на Руси. Полное название этого учебника — «Сия книга, глаголемая по еллински и по гречески арифметика, а по немецки алгоризма, а по русски цифирная счётная мудрость».

Таким образом, слово «алгоритм» понималось первыми русскими математиками так же, как и в Западной Европе. Однако его не было ни в знаменитом словаре В. И. Даля, ни спустя сто лет в «Толковом словаре русского языка» под редакцией Д. Н. Ушакова (1935 г.). Зато слово «алгорифм» можно найти и в популярном дореволюционном Энциклопедическом словаре братьев Гранат, и в первом издании Большой Советской Энциклопедии (БСЭ), изданном в 1926 г. И там, и там оно трактуется одинаково: как правило, по которому выполняется то или иное из четырёх арифметических действий в десятичной системе счисления. Однако к началу XX в. для математиков слово «алгоритм» уже означало любой арифметический или алгебраический процесс, выполняемый по строго определённым правилам, и это объяснение также даётся в БСЭ.

Алгоритмы становились предметом все более пристального внимания учёных, и постепенно это понятие заняло одно из центральных мест в современной математике. Что же касается людей, от математики далёких, то к началу сороковых годов это слово они могли услышать разве что во время

учебы в школе, в сочетании «алгоритм Евклида». Несмотря на это, алгоритм все ещё воспринимался как термин сугубо специальный, что подтверждается отсутствием соответствующих статей в менее объёмных изданиях. В частности, его нет даже в десятитомной Малой Советской Энциклопедии (1957 г.), не говоря уже об однотомных энциклопедических словарях. Но зато спустя десять лет, в третьем издании Большой советской энциклопедии (1969 г.) алгоритм уже характеризуется как одна из основных категорий математики, «не обладающих формальным определением в терминах более простых понятий, и абстрагируемых непосредственно из опыта». Как мы видим, отличие даже от трактовки первым изданием БСЭ разительное! За сорок лет алгоритм превратился в одно из ключевых понятий математики, и признанием этого стало включение слова уже не в энциклопедии, а в словари. Например, оно присутствует в академическом «Словаре русского языка» (1981 г.) именно как термин из области математики.

Одновременно с развитием понятия алгоритма постепенно происходила и его экспансия из чистой математики в другие сферы. И начало ей положило появление компьютеров, благодаря которому слово «алгоритм» вошло в 1985 г. во все школьные учебники информатики и обрело новую жизнь. Вообще можно сказать, что его сегодняшняя известность напрямую связана со степенью распространения компьютеров. Например, в третьем томе «Детской энциклопедии» (1959 г.) о вычислительных машинах говорится немало, но они ещё не стали чем-то привычным и воспринимаются скорее как некий атрибут светлого, но достаточно далёкого будущего. Соответственно и алгоритмы ни разу не упоминаются на её страницах. Но уже в начале 70-х гг. прошлого столетия, когда компьютеры перестали быть экзотической диковинкой, слово «алгоритм» стремительно входит в обиход. Это чутко фиксируют энциклопедические издания. В «Энциклопедии кибернетики» (1974 г.) в статье «Алгоритм» он уже связывается с реализацией на вычислительных машинах, а в «Советской военной энциклопедии»

(1976 г.) даже появляется отдельная статья «Алгоритм решения задачи на ЭВМ». За последние полтора-два десятилетия компьютер стал неотъемлемым атрибутом нашей жизни, компьютерная лексика становится все более привычной. Слово «алгоритм» в наши дни известно, вероятно, каждому. Оно уверенно шагнуло даже в разговорную речь, и сегодня мы нередко встречаем в газетах и слышим в выступлениях политиков выражения вроде «алгоритм поведения», «алгоритм успеха» или даже «алгоритм предательства». Академик Н. Н. Моисеев назвал свою книгу «Алгоритмы развития», а известный врач Н. М. Амосов — «Алгоритм здоровья» и «Алгоритмы разума». А это означает, что слово живёт, обогащаясь все новыми значениями и смысловыми оттенками.

1.1.1 Понятие алгоритма

a. Описание последовательности действий для решения задачи или достижения поставленной цели;

b. правила выполнения основных операций обработки данных;

c. описание вычислений по математическим формулам.

Перед началом разработки алгоритма необходимо четко уяснить **задачу**: что требуется получить в качестве результата, какие исходные данные необходимы и какие имеются в наличии, какие существуют ограничения на эти данные. Далее требуется записать, какие действия необходимо предпринять для получения из исходных данных требуемого результата.

Хотя в определении алгоритма требуется лишь конечность числа шагов, требуемых для достижения результата, на практике выполнение даже хотя бы миллиарда шагов является слишком медленным. Также обычно есть другие ограничения (на размер программы, на допустимые действия). В связи с этим вводят такие понятия как **сложность алгоритма** (временная, по размеру программы, вычислительная и др.).

Для каждой задачи может существовать **множество** алгоритмов, приводящих

к цели. Увеличение эффективности алгоритмов составляет одну из задач современной информатики. В 50-х гг. XX века появилась даже отдельная её область — быстрые алгоритмы. В частности, в известной всем с детства задаче об умножении десятичных чисел обнаружился ряд алгоритмов, позволяющих существенно (в асимптотическом смысле) ускорить нахождение произведения.

Профессор Дейкстра первым показал в своих статьях и книге «Дисциплина программирования», как составляются алгоритмы и программы без ошибок с доказательствами их правильности.

Свойства алгоритма

Различные определения алгоритма в явной или неявной форме содержат следующий ряд общих требований:

Детерминированность — определённость. В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных.

Понятность — алгоритм для исполнителя должен включать только те команды, которые ему (исполнителю) доступны, которые входят в его систему команд.

Завершаемость (конечность) — при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов. этапы алгоритмизации.

Массовость — алгоритм должен быть применим к разным наборам исходных данных.

Результативность — завершение алгоритма определенными результатами.

Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не даёт результатов вовсе. Алгоритм не содержит

ошибок, если он даёт правильные результаты для любых допустимых исходных данных.

1.1.2 Виды алгоритмов

Особую роль выполняют **прикладные алгоритмы**, предназначенные для решения определённых прикладных задач. Алгоритм считается правильным, если он отвечает требованиям задачи (например, даёт физически правдоподобный результат). Алгоритм (программа) содержит ошибки, если для некоторых исходных данных он даёт неправильные результаты, сбои, отказы или не даёт никаких результатов вообще. Последний тезис используется в олимпиадах по алгоритмическому программированию, чтобы оценить составленные участниками программы.

Важную роль играют **рекурсивные** алгоритмы (алгоритмы, вызывающие сами себя до тех пор, пока не будет достигнуто некоторое условие возвращения). Начиная с конца XX - начала XXI века активно разрабатываются параллельные алгоритмы, предназначенные для вычислительных машин, способных выполнять несколько операций одновременно.

Наличие исходных данных и результата

Алгоритм — это точно определённая инструкция, последовательно применяя которую к исходным данным, можно получить решение задачи. Для каждого алгоритма есть некоторое множество объектов, допустимых в качестве исходных данных. Например, в алгоритме деления вещественных чисел делимое может быть любым, а делитель не может быть равен нулю.

Алгоритм служит, как правило, для решения не одной конкретной задачи, а некоторого **класса задач**. Так, алгоритм сложения применим к любой паре натуральных чисел. В этом выражается его свойство массовости, то есть возможности применять многократно один и тот же алгоритм для любой задачи одного класса.

Для разработки алгоритмов и программ используется **алгоритмизация** — процесс систематического составления алгоритмов для решения поставленных приклад-

ных задач. Алгоритмизация считается обязательным этапом в процессе разработки программ и решении задач на ЭВМ. Именно для прикладных алгоритмов и программ принципиально важны детерминированность, результативность и массовость, а также правильность результатов решения поставленных задач.

1.2 Методология и этапы решения задач с помощью компьютера

Процесс автоматизированного решения задачи с помощью компьютера включает следующие этапы:

- постановка задачи;
- формализация решения задачи;
- алгоритмизация;
- программирование;
- тестирование и отладка;
- опытная эксплуатация и доработка программы;
- эксплуатация, сопровождение и модернизация программы;
- утилизация (уничтожение) программы.

Постановка задачи. Целью этапа постановки задачи является точное (формальное, а не описательное) определение решаемой задачи. На этом этапе необходимо:

– четко уяснить решаемую задачу;

– определить назначение, ограничения и наиболее важные внутренние закономерности решаемой задачи;

– определить состав и структуру исходных данных и результата решения задачи;

– определить требования к организации пользовательского **интерфейса** (входные и выходные данные, порядок взаимодействия пользователя с программой, возможные ошибочные ситуации и действия по их устранению) на основе анализа деятельности пользователей, которые будут решать задачу с помощью создаваемой программы.

> **Интерфейс пользователя**, он же пользовательский интерфейс (UI — англ. user interface) — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы. Под совокупностью средств и методов интерфейса пользователя подразумеваются: **средства вывода информации** из устройства к пользователю — весь доступный диапазон воздействий на организм человека (зрительных, слуховых, тактильных, обонятельных и т. д.) — экраны (дисплеи, проекторы) и лампочки, динамики, зуммеры и сирены, вибромоторы и т. д.; **средства ввода информации**/команд пользователем в устройство — множество всевозможных устройств для контроля состояния человека — кнопки, переключатели, потенциометры, датчики положения и движения, сервоприводы, жесты лицом и руками, даже съём мозговой активности пользователя.

– указать математические методы для решения отдельных подзадач или всей задачи в целом, которые должны быть использованы при решении поставленной задачи.

Результатом этапа разработки должна стать точная (формальная) постановка решаемой задачи.

Формализация решения задачи. Целью этапа формализации является выбор существующих и/или разработка новых формальных (математических) методов решения отдельных подзадач и всей задачи в целом.

На этапе формализации решаются следующие задачи:

– декомпозиция задачи на подзадачи и определение взаимодействия между подзадачами по данным и управлению;

– выбор и обоснование методов решения задачи и всех подзадач;

– в случае известных методов решения необходимо оценить возможность их применения для условий решаемой задачи.

Декомпозицию задачи на подзадачи целесообразно осуществлять по функциональному принципу с построением дерева функций системы. Это позволит строить функционально законченные программные модули с минимумом связей между ними. Для решения отдельных подзадач и всей задачи в целом целесообразно использовать известные методы решения, так как для них доказана правильность решения задачи и обычно известны оценки сложности по трудоемкости и объемам памяти.

Результатом этого этапа являются доказательство возможности формального решения поставленной задачи и выбор конкретных математических методов решения задачи и всех подзадач.

Алгоритмизация. Целью этапа алгоритмизации является разработка алгоритма решения поставленной задачи, т.е. разработка формальной последовательности действий, обеспечивающей получение требуемого результата для заданных исходных данных.

На этапе алгоритмизации решаются следующие задачи:

– разработка укрупненного алгоритма и схемы работы системы;

– разработка детального алгоритма решения задачи;

– оценка алгоритма.

Программирование. Целью этапа программирования является написание программы, которая может быть выполнена на компьютере.

На этом этапе разрабатывается текст программы на одном из языков программирования в соответствии с детальным алгоритмом.

Тестирование и отладка. Целью этапа тестирования и отладки является проверка правильности решения поставленной задачи с использованием компьютера.

При этом программа, готовая к исполнению, проходит проверку на соответствие решаемой задаче путём выполнения её для определённых комбинаций исходных данных, результат решения задачи для которых уже известен. В случае обнаружения ошибок вносятся изменения в алгоритм и программу. Результатом этого этапа должна быть рабочая программа, правильно решающая поставленную задачу.

Полную гарантию правильности алгоритма может дать описание работы и результатов алгоритма с помощью системы аксиом и правил вывода или верификация алгоритма.

Для несложных алгоритмов грамотный подбор тестов и полное тестирование может дать полную картину работоспособности (неработоспособности).

Трассировка – это метод пошаговой фиксации динамического состояния алгоритма на некотором тесте. Часто осуществляется с помощью трассировочных таблиц, в которых каждая строка соответствует определённому состоянию алгоритма, а столбец – определённому состоянию параметров алгоритма (входных, выходных и промежуточных). Трассировка облегчает отладку и понимание алгоритма.

Процесс поиска и исправления (явных или неявных) ошибок в алгоритме называется **отладкой алгоритма**.

Некоторые (скрытые, труднообнаруживаемые) ошибки в сложных программных комплексах могут выявиться только в процессе их эксплуатации, на последнем этапе поиска и исправления ошибок – этапе сопровождения. На этом этапе также уточняют и улучшают документацию, обучают персонал использованию алгоритма (программы).

Опытная эксплуатация и доработка программы. Программа, прошедшая предварительное тестирование, устанавливается у заказчика на окончательное тестирование в реальных условиях. По окончании опытной эксплуатации проводится доработка программы или принимается окончательное решение о несоответствии её

поставленной задаче.

Эксплуатация, сопровождение и доработка программы. Программа, окончательно принятая заказчиком, запускается в промышленную эксплуатацию. Желательно, чтобы этот этап был как можно длиннее, т.к. именно здесь обеспечивается возврат затрат, вложенных в разработку программы, и получение прибыли. Однако в процессе эксплуатации требуется **сопровождение** программы (архивация данных, восстановление системы после сбоев и др.), а также в случае несущественных изменений решаемых задач – доработка программы.

1.2.1 Способы представления алгоритмов

Основным стимулом, приведшим к выработке понятия алгоритма и созданию теории алгоритмов, явилась потребность доказательства неразрешимости многих проблем, возникших в различных областях математики. Специалист, решая задачу, всегда должен считаться с возможностью того, что она может оказаться неразрешимой.

На практике наиболее распространены следующие формы представления алгоритмов:

- **словесная** (записи на естественном языке);
- **графическая** (изображения из графических символов);
- **псевдокоды** (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
- **программная** (тексты на языках программирования).

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке.

Пример. *Записать алгоритм нахождения наибольшего общего делителя (НОД)*

двух натуральных чисел.

Алгоритм может быть следующим:

1. задать два числа;
2. если числа равны, то взять любое из них в качестве ответа и остановиться, в противном случае продолжить выполнение алгоритма;
3. определить большее из чисел;
4. заменить большее из чисел разностью большего и меньшего из чисел;
5. повторить алгоритм с шага 2.

Описанный алгоритм применим к любым натуральным числам и должен приводить к решению поставленной задачи. Убедитесь в этом самостоятельно, определив с помощью этого алгоритма наибольший общий делитель чисел 125 и 75.

Словесный способ не имеет широкого распространения по следующим причинам:

– такие описания строго не формализуемы;
– страдают многословностью записей;
– допускают неоднозначность толкования отдельных предписаний.

Графический способ представления алгоритмов является более компактным и наглядным по сравнению со словесным.

При графическом представлении алгоритм изображается в виде последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий.

Такое графическое представление называется **схемой алгоритма** или **блок-схемой**.

Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов.

Он занимает промежуточное место между естественным и формальным языками.

С одной стороны, он близок к обычному естественному языку, поэтому алгорит-

мы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В **псевдокоде** не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя. Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке. В частности, в псевдокоде, так же, как и в формальных языках, есть служебные слова, смысл которых определен раз и навсегда. Единого или формального определения псевдокода не существует, поэтому возможны различные псевдокоды, отличающиеся набором служебных слов и основных (базовых) конструкций.

1.2.2 Блок-схемы алгоритмов, основные элементы

Блок-схемой называют графическое представление алгоритма, в котором он изображается в виде последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий.

Рис. 1.1: Блок начала, конца алгоритма, вход и выход в подпрограмму

В блок-схеме каждому типу действий (вводу исходных данных – рисунок 1.2, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий.

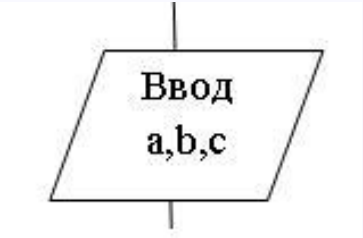


Рис. 1.2: Блок «ввод-вывод» – отображение ввода-вывода в общем виде

Блок «**процесс**» применяется для обозначения действия или последовательности действий, изменяющих значение, форму представления или размещения данных. Для улучшения наглядности схемы несколько отдельных блоков обработки можно объединять в один блок. Представление отдельных операций достаточно свободно.

Блок «**решение**» используется для обозначения переходов управления по условию (рисунок 1.3). В каждом блоке «решение» должны быть указаны вопрос, условие или сравнение, которые он определяет.

Блок «**модификация**» используется для организации циклических конструкций (рисунок 1.4). Слово «модификация» означает видоизменение, преобразование. Внутри блока записывается параметр цикла, для которого указываются его начальное значение, граничное условие и шаг изменения значения параметра для каждого повторения.

Блок «**предопределенный процесс**» (рисунок 1.5) используется для указания обращений к вспомогательным алгоритмам, существующим автономно в виде некоторых самостоятельных модулей, и для обращений к библиотечным подпрограммам.

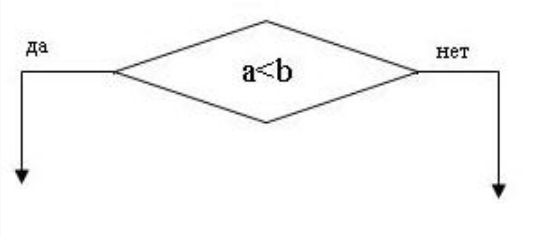


Рис. 1.3: Блок «решение»

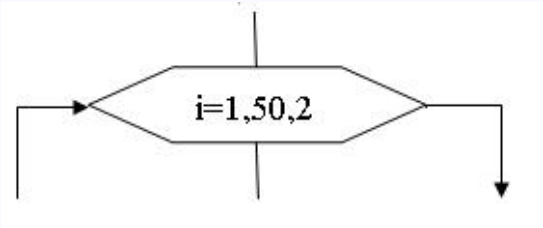


Рис. 1.4: Блок «модификация»

Рис. 1.5: Блок «предопределенный процесс»

Пример. *Составить блок-схему алгоритма определения высот* h_a, h_b, h_c *тре-*

Рис. 1.6: Отображение вывода результатов на печать

угольника со сторонами a, b, c, если

$$h_a = \left(\frac{2}{a}\right)\sqrt{p\,(p-a)\,(p-b)\,(p-c)},$$

$$h_b = \left(\frac{2}{a}\right)\sqrt{p\,(p-a)\,(p-b)\,(p-c)},$$

$$h_c = \left(\frac{2}{a}\right)\sqrt{p\,(p-a)\,(p-b)\,(p-c)},$$

где p - полупериметр

Пример составления алгоритма в виде блок схемы – на рисунке 1.7.

1.2.3 Базовые структуры

Алгоритмы можно представлять как некоторые структуры, состоящие из отдельных базовых (т.е. основных) элементов. Естественно, что при таком подходе к алгоритмам изучение основных принципов их конструирования должно начинаться с изучения этих базовых элементов.

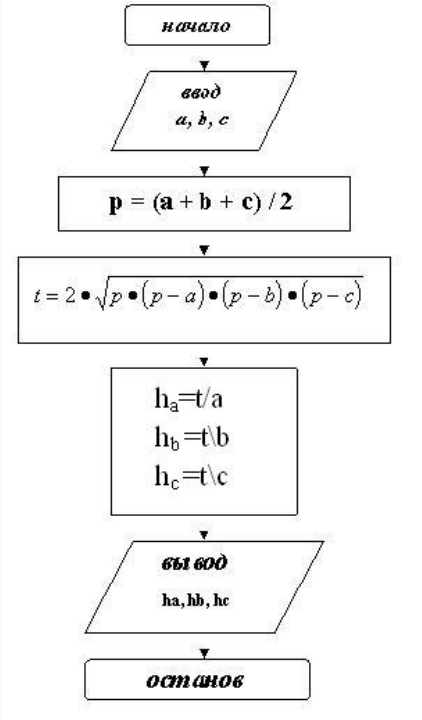


Рис. 1.7: Пример алгоритма в виде блок-схемы

Логическая структура любого алгоритма может быть представлена комбинацией трёх базовых структур: **следование, ветвление, цикл**.

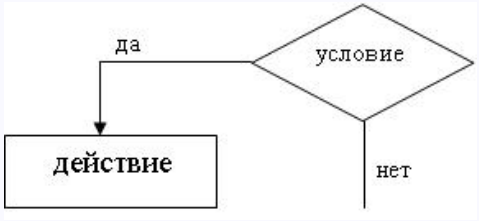
Характерной особенностью базовых структур является наличие в них одного входа и одного выхода.

1. Базовая структура «следование». Образуется из последовательности действий, следующих одно за другим.

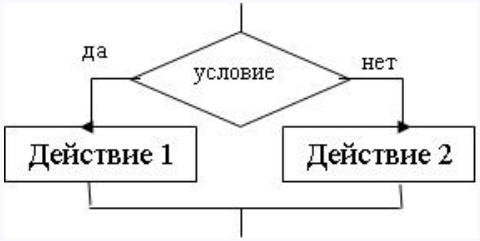
2. Базовая структура «ветвление». Обеспечивает в зависимости от результа-

та проверки условия (*да* или *нет*) выбор одного из альтернативных путей работы алгоритма. Каждый из путей ведет к общему выходу, так что работа алгоритма будет продолжаться независимо от того, какой путь будет выбран.

Структура «ветвление» существует в четырёх основных вариантах: если-то, если-то-иначе (рисунок 1.8), выбор, выбор-иначе (рисунок 1.9).



а) если-то



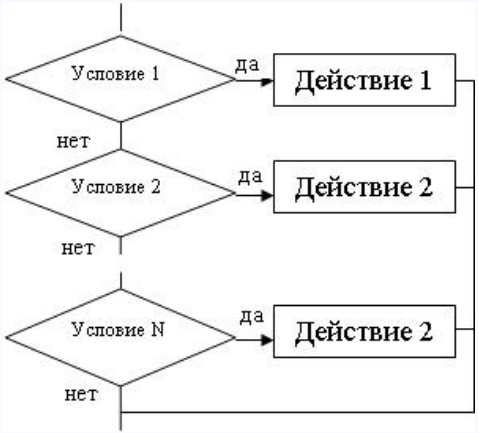
б) если-то-иначе

Рис. 1.8: Базовая структура «ветвление»

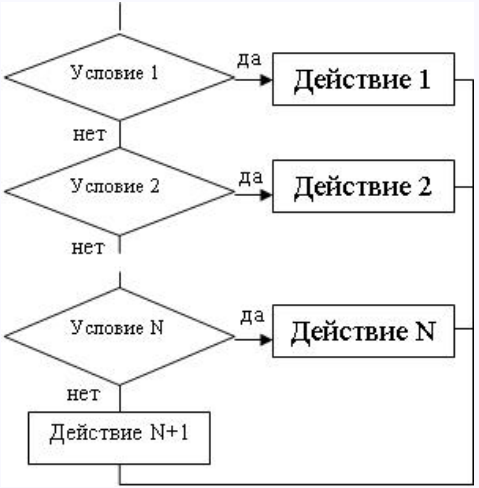
3. Базовая структура «цикл». Обеспечивает многократное выполнение некоторой совокупности действий, которая называется телом цикла. Структура цикл существует в трёх основных вариантах:

- Цикл типа «для». Предписывает выполнять тело цикла для всех значений некоторой переменной (параметра цикла) в заданном диапазоне (рисунок 1.10, а).
- Цикл типа «пока». Предписывает выполнять тело цикла до тех пор, пока выполняется условие, записанное после слова «пока» (рисунок 1.10, б).
- Цикл типа «делать - пока». Предписывает выполнять тело цикла до тех пор, пока выполняется условие, записанное после слова «пока». Условие проверяется после выполнения тела цикла (рисунок 1.10, в).

Заметим, что циклы «для» и «пока» называют также циклами с **предпроверкой условия** а циклы «делать - пока» – циклами с **постпроверкой условия**. Иными



а) выбор



б) выбор-иначе

Рис. 1.9: Базовая структура «ветвление»

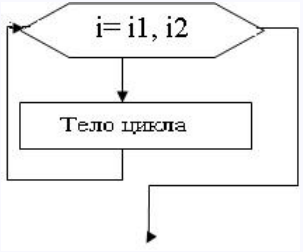
словами, тела циклов «для» и «пока» могут не выполниться ни разу, если условие окончания цикла изначально не верно. Тело цикла «делать - пока» выполнится как минимум один раз, даже если условие окончания цикла изначально не верно.

Итерационные циклы. Особенностью итерационного цикла является то, что число повторений операторов тела цикла заранее неизвестно. Для его организации используется цикл типа «пока». Выход из итерационного цикла осуществляется в случае выполнения заданного условия.

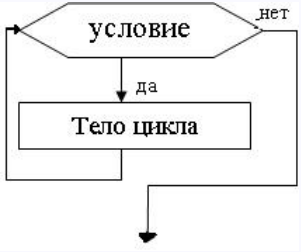
На каждом шаге вычислений происходит последовательное приближение и проверка условия достижения искомого результата.

Пример. Составить алгоритм вычисления суммы ряда

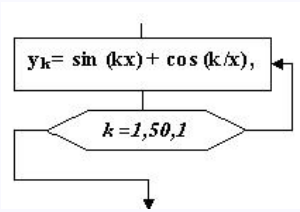
$$S = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + (-1)^{i-1}\frac{x^i}{i}$$



а) Цикл типа «для»



б) Цикл типа «пока»



в) Цикл типа «делать - пока»

Рис. 1.10: Базовая структура «цикл»

с заданной точностью (для данного знакочередующегося степенного ряда требуемая точность будет достигнута, когда очередное слагаемое станет по абсолютной величине меньше точности).

Вычисление сумм – типичная циклическая задача. Особенностью этой конкретной задачи является то, что число слагаемых (а, следовательно, и число повторений тела цикла) заранее неизвестно. Поэтому выполнение цикла должно завершиться в момент достижения требуемой точности.

При составлении алгоритма нужно учесть, что знаки слагаемых чередуются и степень числа x в числителях слагаемых возрастает.

Решая эту задачу «в лоб» путем вычисления на каждом i-ом шаге частичной суммы $S := S + (-1)^{i-1}\frac{x^i}{i}$ мы получим очень неэффективный алгоритм, требующий выполнения большого числа операций. Гораздо лучше организовать вычисления следующим образом: если обозначить числитель какого-либо слагаемого буквой p, то у следующего слагаемого числитель будет равен $-p*x$ (знак минус обеспечивает чередование знаков слагаемых), а само слагаемое m будет равно p/i, где i - номер слагаемого (рисунок 1.11).

Алгоритм, в состав которого входит итерационный цикл, называется **итерационным алгоритмом**. Итерационные алгоритмы используются при реализации итерационных численных методов. В итерационных алгоритмах необходимо обеспечить

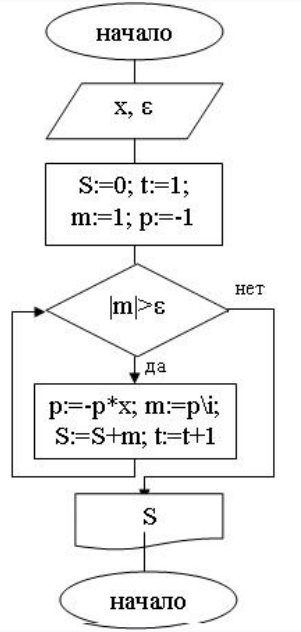


Рис. 1.11: Пример блок-схемы итерационного алгоритма

обязательное достижение условия выхода из цикла (сходимость итерационного процесса). В противном случае произойдет **зацикливание** алгоритма, т.е. не будет выполняться основное свойство алгоритма – результативность.

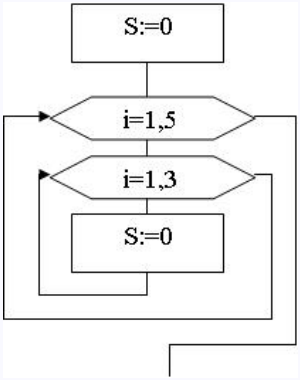
1.2.4 Вложенные циклы

Возможны случаи, когда внутри тела цикла необходимо повторять некоторую последовательность операторов, т. е. организовать внутренний цикл. Такая структура получила название цикла в цикле или вложенных циклов. Глубина вложения циклов (то есть количество вложенных друг в друга циклов) может быть различной.

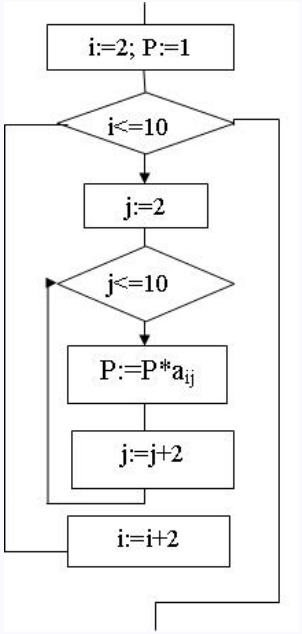
При использовании такой структуры для экономии машинного времени необходимо выносить из внутреннего цикла во внешний все операторы, которые не зависят от параметра внутреннего цикла.

Пример вложенных циклов «для» (рисунок 1.12, а): вычислить сумму элементов заданной матрицы A[5,3].

Пример вложенных циклов «пока» (рисунок 1.12, б): вычислить произведение тех элементов заданной матрицы A[10,10], которые расположены на пересечении чётных строк и чётных столбцов.



а) циклы «для»



б) циклы «пока»

Рис. 1.12: Пример блок-схемы вложенных циклов

1.3 Парадигма структурного программирования

Структурное программирование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков.

В соответствии с парадигмой, любая программа, которая строится **без** использования оператора **goto**, состоит из трёх базовых управляющих конструкций: **последовательность, ветвление, цикл** (рисунок 1.13); кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Теорема Бёма — Якопини: Любая программа, заданная в виде блок-схемы, может быть представлена с помощью трёх управляющих структур:

- последовательность;
- ветвление;
- цикл.

Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование.

Принципы структурного программирования:

Принцип 1. Следует отказаться от использования оператора безусловного перехода goto.

Принцип 2. Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.

Принцип 3. В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом. Никаких других средств управления последовательностью выполнения операций не предусматривается.

Принцип 4. Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций). Таким же образом (в виде подпрограмм) можно оформить логически целостные фрагменты программы, даже если они не повторяются.

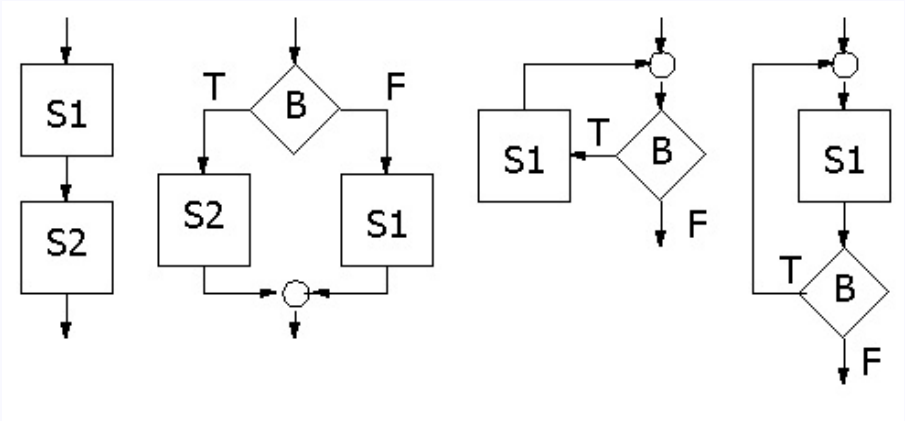


Рис. 1.13: Элементарные компоненты структурной блок-схемы

Принцип 5. Каждую логически законченную группу инструкций следует оформить как блок. Блоки являются основой структурного программирования.

> **Блок** — это логически сгруппированная часть исходного кода, например, набор инструкций, записанных подряд в исходном коде программы. Понятие блок означает, что к блоку инструкций следует обращаться как к единой инструкции. Блоки служат для ограничения области видимости переменных и функций. Блоки могут быть пустыми или вложенными один в другой. Границы блока строго определены. Например, в if-операторе блок ограничен кодом BEGIN..END (в языке Паскаль) или фигурными скобками ... (в языке C) или отступами (в языке Питон).

Принцип 6. Все перечисленные конструкции должны иметь один вход и один выход.

> Произвольные управляющие конструкции (такие, как в блюде спагетти) могут иметь произвольное число входов и выходов. Ограничив себя управ-

ляющими конструкциями с одним входом и одним выходом, мы получаем возможность построения произвольных алгоритмов любой сложности с помощью простых и надежных механизмов.

Принцип 7. Разработка программы ведётся пошагово, методом «сверху вниз» (top-down method).

При создании средних по размеру приложений (несколько тысяч строк исходного кода) используется структурное программирование, идея которого заключается в том, что структура программ должна отражать структуру решаемой задачи, чтобы алгоритм решения был ясно виден из исходного текста. Для этого надо иметь средства для создания программы не только с помощью трёх простых операторов, но и с помощью средств, более точно отражающих конкретную структуру алгоритма. С этой целью в программирование введено понятие подпрограммы – набора операторов, выполняющих нужное действие и не зависящих от других частей исходного кода. Программа разбивается на множество мелких подпрограмм (занимающих до 50 операторов – критический порог для быстрого понимания цели подпрограммы), каждая из которых выполняет одно из действий, предусмотренных исходным заданием. Комбинируя эти подпрограммы, удается формировать итоговый алгоритм уже не из простых операторов, а из законченных блоков кода, имеющих определенную смысловую нагрузку, причем обращаться к таким блокам можно по названиям.

Структурированный алгоритм – это алгоритм, представленный как следования и вложения базовых алгоритмических структур. У структурированного алгоритма статическое состояние (до актуализации алгоритма) и динамическое состояние (после актуализации) имеют одинаковую логическую структуру, которая прослеживается сверху вниз («как читается, так и исполняется»). При структурированной разработке алгоритмов правильность алгоритма можно проследить на каждом этапе его построения и выполнения.

Теорема (о структурировании). Любой алгоритм может быть эквивалентно представлен структурированным алгоритмом, состоящим из базовых алгоритмиче-

ских структур.

Структурное программирование сверху-вниз - это процесс пошагового разбиения алгоритма на все более мелкие части с целью получить такие элементы, для которых можно легко написать конкретные команды.

Структурное программирование сверху-вниз включает достаточно много других аспектов, один из которых можно выразить лозунгом «Упрощай структуры управления». Другие аспекты - это требование соответствующей документации, наглядной записи программных операторов с использованием отступов в начале строк, разбиения программы на обозримые части.

Рекомендуется, например, составлять программные модули так, чтобы их распечатка не превышала одной - двух страниц. Страница – это смысловая единица, которую можно представить себе в любой момент как единое целое и которая сводит к минимуму необходимость переворачивать страницы при прослеживании передач управления в программе.

Для разработки структурированной сверху-вниз программы потребуется затратить больше усилий, чем для получения неструктурированной программы. Структурированные программы, как правило, легче читать, и в них легче разбираться, так как их можно читать сверху-вниз, не прыгая по тексту из конца в начало. Главным образом это достигается благодаря тому, что общая структура управления в структурированной программе является деревом, а не сетью со многими циклами.

Метод последовательного уточнения состоит в следующем:

- исходная задача разбивается на ряд крупных частей (подзадач);
- сначала полностью составляется основной алгоритм (план решения задачи);
- при этом для решения подзадач (пунктов плана) используются команды вызова ещё не написанных вспомогательных алгоритмов;

- затем составляются вспомогательные алгоритмы, которые в свою очередь можно разбить на подзадачи для выполнения более мелких действий.

Восходящий метод **(снизу-вверх)**, наоборот, опираясь на некоторый, заранее определяемый корректный набор подалгоритмов, строит функционально завершенные подзадачи более общего назначения, от них переходит к более общим, и так далее, до тех пор, пока не дойдем до уровня, на котором можно записать решение поставленной задачи. Этот метод известен как метод проектирования «снизу вверх».

1.3.1 Методы разработки алгоритма

При решении задач на компьютере в первую очередь необходимо знание методов решения задач, а уже во вторую очередь умение составлять алгоритмы, пользуясь приведенными выше понятиями и конструкциями.

Сущность алгоритмизации не в том, что решение задачи представляется в виде набора элементарных операций, а в том, что процесс решения задачи разбивается на два этапа: составление алгоритма плюс кодирование программы и её выполнение. Первый этап осуществляет человек, второй – компьютер.

Существует весьма большое количество всевозможных приемов и методов разработки алгоритмов. Однако среди имеющегося разнообразия этих методов можно выделить небольшой набор основных, в том смысле, что методы из такого набора применяются часто и лежат в основе многих процедур и алгоритмов.

К основным методам можно отнести:

1. Метод частных целей – сложная задача сводится к последовательности более простых задач.

2. Метод подъема – начинается с принятия начального предположения или построения начального решения задачи. Затем начинается (насколько возможно) быстрое движение «вверх» от начального уровня по направлению к лучшим решениям. Когда алгоритм достигает точки, из которой больше невозможно двигаться «наверх», он останавливается.

3. Программирование с отходом назад – Основой программ искусственного интеллекта, независимо от того, к чему он прилагается – программирование игр, выбору решений, распознавание образов и т.п., - является программирование перебора вариантов. Программирование перебора вариантов – это сложная задача, т.к. алгоритмы перебора ищут решения не по заданным правилам вычислений, а путем проб и ошибок, и схема не укладывается в схемы циклов, имеющихся в языках программирования. Ситуация зачастую осложняется тем, что прямыми методами перебор всех возможных вариантов невозможно осуществить из-за их огромного количества. Метод программирования с отходом назад позволяет осуществить организованный исчерпывающий поиск требуемого решения задачи. При этом часто удается избежать перебора всех возможных вариантов.

4. Алгоритмы ветвей и границ – применяются для решения переборных задач и ориентированы на поиск оптимального решения из конечного множества возможных решений – вариантов.

1.3.2 Технологии (парадигмы) программирования

Для решения огромного множества совершенно различных задач были предложены и развиты определенные стили (парадигмы) программирования, каждому из которых соответствует своя уникальная модель вычислений.

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию).

В настоящее время основными парадигмами программирования являются *процедурное*, *функциональное*, *логическое*, *объектно-ориентированное* и *визуальное программирование*.

1.3.3 Процедурное программирование

Процедурное или императивное программирование соответствует архитектуре ЭВМ, предложенной фон Нейманом, а его теоретической моделью является алгоритмиче-

ская система под названием «машина Тьюринга».

Программа на процедурном языке программирования **состоит** из последовательности операторов (инструкций), задающих те или иные действия. Основным является оператор присваивания, служащий для изменения содержимого областей памяти. Вообще концепция памяти как хранилища значений, содержимое которого может обновляться операторами программы, является фундаментальной в императивном программировании.

Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти (значений переменных) в заключительное. Таким образом, с точки зрения программиста имеется программа и память, причём первая последовательно обновляет содержимое последней.

Процедурный подход характеризуется:

- значительной сложностью;
- отсутствием строгой математической основы;
- необходимостью явного управления памятью, в частности обязательное описание переменных;
- малой пригодностью для символьных вычислений;
- высокой эффективностью реализации на традиционных ЭВМ.

Типичными представителями процедурных языков программирования являются языки **Алгол, Фортран, Бейсик, Паскаль, Си, Си++.**

1.3.4 Функциональное программирование

В отличие от процедурного, функциональное (аппликативное) программирование не использует концепцию памяти как хранилища значений переменных. **Операторы присваивания отсутствуют, вследствие чего переменные обозначают**

не области памяти, а объекты программы, что полностью соответствует понятию переменной в математике. Роль основной конструкции в функциональных языках играет выражение. К выражениям относятся скалярные константы, структурированные объекты, функции, тела функций и вызовы функций. **Программа представляет собой совокупность описаний функций (возможно вложенных) и выражения, которые необходимо вычислить посредством редукции (серии упрощений).**

Функциональные языки отличаются своей простотой, легкостью реализации, компактностью и пригодностью для символьных вычислений. Основными структурированными объектами в аппликативных языках являются списки, удобные для символьной обработки. Самым первым функциональным языком явился **Лисп**.

Сейчас существует множество функциональных языков, значительно более мощных, чем Лисп, а разработка аппаратных и программных средств функционального программирования вошла составной частью в проекты ЭВМ пятого поколения.

1.3.5 Логическое программирование

Логическое (реляционное) программирование основано на результатах, полученных в области исчисления предикатов. Центральным понятием в логическом программировании является отношение. **Программа представляет собой совокупность определений, отношений между объектами и цели, а процесс её выполнения состоит в установлении значимости логической формулы, построенной из программы по определённым правилам.** При этом результат вычисления является побочным продуктом этого процесса.

В реляционном программировании нужно только специфицировать факты, на которых основывается алгоритм, а не определять последовательность шагов, которые требуется выполнить. Поэтому языки логического программирования являются декларативными языками. Логические программы отличаются принципиально низким быстродействием, так как вычисления осуществляются методом проб и ошибок (поиск с возвратом).

Таким образом, языки логического программирования являются достаточно мощными, но неэффективными с точки зрения реализации языками. Тем не менее, языки логического программирования играют центральную роль в проектах ЭВМ пятого поколения, и уже разработан ряд архитектур с целью аппаратной поддержки реляционного программирования.

Основным из языков логического программирования является язык **Пролог**, имеющий в настоящее время около пятнадцати реализаций для персональных компьютеров. Режим компиляции предусмотрен только в трёх системах, остальные же обеспечивают только интерпретацию программ.

1.3.6 Объектно-ориентированное программирование

Понятие «объектно-ориентированный» возникло в программировании сравнительно недавно. Когда вычислительная мощность машин была невысока, о создании объектно-ориентированных систем не могло быть и речи. Основой всего был программный код. Программисты записывали последовательности команд для выполнения тех или иных действий над данными, которые оформлялись в модули и процедуры. Для работы с каждым объектом создавалась своя процедура. Постепенно с увеличением производительности вычислительных систем процедурный подход начал заменяться объектным. На первое место выдвинулся объект, а не код, который его обрабатывает. Объектно-ориентированное программирование даёт более короткие, проще понимаемые и легче контролируемые программы.

Реальные объекты окружающего мира обладают тремя базовыми характеристиками:

1)они имеют набор свойств;

2)способны разными методами изменять эти свойства;

3)способны реагировать на события, возникающие как в окружающем мире, так и внутри самого объекта.

Именно в таком виде в языках программирования (например, C++) и реализовано понятие **объекта**, как совокупности:

- **свойств** (структур данных, характерных для этого объекта);
- **методов** их обработки (подпрограмм изменения свойств);
- **событий**, на которые данный объект может реагировать и которые приводят, как правило, к изменению свойств объекта.

Ключевым понятием объектно-ориентированного программирования является **класс**. **Класс** – это тип, определяемый пользователем. Классы обеспечивают скрытие данных, гарантированную инициализацию данных, неявное преобразование типов для типов, определённых пользователем, динамическое задание типа, контролируемое пользователем управление памятью и механизмы перегрузки операций.

В основе объектно-ориентированного программирования лежит понятие **объекта**, который представляет собой объединенные вместе *данные* и *методы* их обработки. Такое объединение называется **инкапсуляцией** и позволяет реализовать качественно новый уровень совместной структуризации данных и процедур их обработки.

На уровне пользователя объектный подход выражается в том, что интерфейс представляет собой подобие реального мира, а работа с машиной сводится к действиям с привычными объектами. Так, папки можно открыть, убрать в портфель, документы - просмотреть, исправить, переложить с одного места на другое, выбросить в корзину, факс или письмо - отправить адресату и т. д. Понятие объекта оказалось настолько широким, что до сих пор не получило строгого определения.

Объект, как и в реальном мире, обладает различными свойствами. Программист или пользователь может изменять не все свойства объектов, а только некоторые из них. Можно изменить имя объекта, но нельзя изменить объем свободного места на диске, который также является его свойством.

Метод – это способ воздействия на объект. Методы позволяют создавать и удалять объекты, а также изменять их свойства. Например, для того чтобы нарисовать на экране точку, линию или плоскую фигуру, составляются разные последовательности кодов или программы. Пользователь, однако, применяет для отображения

этих объектов один метод Draw(), который содержит коды для отображения всех объектов, с которыми он работает. За такое удобство приходится платить тем, что объектно-ориентированные системы могут работать только на достаточно мощных вычислительных машинах.

Второй особенностью объектов является наличие механизма **наследования** свойств одного объекта другими объектами. Это позволяет создавать сложные иерархические структуры, в которых свойства родительского объекта наследуются объектами потомками.

Третьей особенностью объектов является **полиморфизм**, то есть возможность использования методов с одинаковыми именами для работы с разными объектами.

Парадигма объектно-ориентированного программирования реализуется в ОС Windows 95/98 через модель рабочего стола. Пользователь работает с задачами и приложениями так же, как с документами на своем письменном столе. Основной упор в операционной системе делается на документ, а программа, задача, приложение или программный код вообще рассматриваются только как инструмент для работы с документом.

Парадигма поддерживается такими системами программирования, как **Турбо Паскаль, Турбо Си++, Борланд Паскаль, Борланд Си++.**

1.3.7 Визуальное программирование

Следующим шагом в развитии инструментальных средств на базе объектно-ориентированного программирования является разработка средств визуального проектирования программ для Windows-приложений.Особенно полезное свойство систем визуального программирования - возможность создания **пользовательского интерфейса** путём размещения готовых элементов на экране и связывания определенных **событий** (действий пользователя) с действиями программы. При этом для настройки тех или иных элементов интерфейса писать программы не требуется - достаточно изменить значения соответствующих свойств этих элементов с помощью встроенного редактора.

Типичным представителем систем визуального программирования является среда программирования **Visual Basic**, которая используется в качестве встроенного языка макроопределений во всех приложениях пакета Microsoft Office для Windows.

Основная особенность использования среды визуального программирования состоит в создании программного проекта, а не написании программы. Важнейшими понятиями визуального программирования являются **экранная форма**, программный **модуль** и программный **проект**.

Экранная форма – графическое представление окна Windows-приложения вместе с содержанием этого окна. Содержание включает в себя:

- перечень свойств окна с их значениями;
- перечень объектов, находящихся в этом окне;
- перечни свойств этих объектов также с их значениями.

Экранная форма хранится в отдельном файле.

Программный модуль – хранящийся в отдельном файле программный код. Он может использоваться при решении не одной, а нескольких задач. Как правило, программный код относится к отдельно взятой экранной форме.

Программный проект – совокупность элементов (программные, объектный и исполняемый модули, экранные формы и т. п.) Windows-приложения.

1.4 Системы и среды программирования

Бурное развитие вычислительной техники, потребность в эффективных средствах разработки программного обеспечения привели к появлению систем программирования, ориентированных на так называемую «быструю разработку», среди которых можно выделить Borland Delphi и Microsoft Visual Basic. В основе **систем быстрой разработки** (RAD-систем, Rapid Application Development — среда быстрой разработки приложений) лежит технология визуального проектирования и со-

бытийного программирования, суть которой заключается в том, что среда разработки берет на себя большую часть рутинной работы, оставляя программисту работу по конструированию диалоговых окон и функций обработки событий.

Delphi — это среда быстрой разработки, в которой в качестве языка программирования используется язык Delphi. Язык Delphi — строго типизированный объектно-ориентированный язык, в основе которого лежит хорошо знакомый программистам Object Pascal.

К настоящему времени программистам доступны большое количество версий пакета Delphi.

Delphi 7, выпущенная в августе 2002 года, стала стандартом для многих разработчиков Delphi. Это один из самых успешных продуктов Borland из-за стабильности, скорости и низких требований к аппаратному обеспечению. Как и предыдущие версии, Borland Delphi 7 Studio позволяет создавать самые различные программы: от простейших однооконных приложений до программ управления распределенными базами. В состав пакета включены разнообразные утилиты, обеспечивающие работу с базами данных, XML-документами, создание справочной системы, решение других задач. Отличительной особенностью седьмой версии является поддержка технологии .NET.

Borland Delphi 7 Studio может работать в среде операционных систем от Windows 98 до Windows XP. Особых требований, по современным меркам, к ресурсам компьютера пакет не предъявляет: процессор должен быть типа Pentium или Celeron с тактовой частотой не ниже 166 МГц (рекомендуется Pentium II 400 МГц), оперативной памяти - 128 Мбайт (рекомендуется 256 Мбайт), достаточное количество свободного дискового пространства (для полной установки версии Enterprise необходимо приблизительно 475 Мбайт).

Lazarus — открытая среда разработки программного обеспечения на языке Object Pascal для компилятора Free Pascal. Основная цель — предоставление кроссплатформенных и свободных средств разработки в Delphi-подобном окружении (по аналогии с Harbour для Clipper).

Позволяет переносить Delphi-программы с графическим интерфейсом в различные операционные системы: Linux, FreeBSD, Mac OS X, Microsoft Windows, Android.

1.5.1 Компиляция

Программа, представленная в виде инструкций языка программирования, называется исходной программой. Она состоит из инструкций, понятных человеку, но не понятных процессору компьютера. Чтобы процессор смог выполнить работу в соответствии с инструкциями исходной программы, исходная программа должна быть переведена на машинный язык — язык команд процессора. Задачу преобразования исходной программы в машинный код выполняет специальная программа — компилятор.

Компилятор, схема работы которого приведена на рисунке 1.14, выполняет последовательно две задачи:

1. Проверяет текст исходной программы на отсутствие синтаксических ошибок.
2. Создает (генерирует) исполняемую программу — машинный код.

Рис. 1.14: Схема работы компилятора

Следует отметить, что генерация исполняемой программы происходит только в том случае, если в тексте исходной программы нет синтаксических ошибок.

Генерация машинного кода компилятором свидетельствует лишь о том, что в тексте программы нет синтаксических ошибок. Убедиться, что программа работает пра-

вильно, можно только в процессе её тестирования — пробных запусках программы и анализе полученных результатов. Например, если в программе вычисления корней квадратного уравнения допущена ошибка в выражении (формуле) вычисления дискриминанта, то, даже если это выражение будет синтаксически верно, программа выдаст неверные значения корней.

1.5.2 Синтаксис языка программирования Pascal (Delphi)

В среде программирования Delphi для записи программ используется язык программирования Delphi. Delphi — это среда разработки программ, ориентированных на работу в Windows. В качестве языка программирования в Delphi используется объектно-ориентированный язык Object Pascal, который можно рассматривать как дальнейшее развитие Turbo Pascal 7.0.

В основе идеологии Delphi лежат технологии визуального проектирования и событийного программирования, применение которых позволяет существенно сократить время разработки и облегчить процесс создания приложений — программ, работающих в среде Windows.

Программа на Delphi представляет собой последовательность инструкций, которые называют операторами. Один оператор от другого отделяется точкой с запятой.

Алфавитом языка называют совокупность всех допустимых символов, которые можно использовать в этом языке:

- прописные и строчные буквы латинского алфавита от **A** до **z**, а также символ подчеркивания (_), который тоже считается буквой;
- арабские цифры **0 1 2 3 4 5 6 7 8 9**;
- специальные одиночные знаки: + - * / = < > . , : ; $ # @;
- специальные парные знаки: [ ] { } ( );
- составные знаки: <= >= <> .. (* *) (..).

Идентификатор, ID (англ. data name, identifier — опознаватель), имя — уникальный признак объекта, позволяющий отличать его от других объектов.

Имена переменных, констант, меток, типов, модулей, процедур и функций, используемых в программе, называются **идентификаторами**. Идентификаторы задаёт разработчик программы. На идентификаторы накладываются некоторые **ограничения**:

- в качестве идентификатора нельзя использовать ключевое (служебное) слово;
- идентификатор может содержать только символы латинского алфавита, цифры и знаки подчеркивания;
- идентификатор не может начинаться с цифры.

Например: summa3, a1, _slovo - идентификаторы, 345summa, ИТОГО, new% - не идентификаторы.

Использование осмысленных имен (идентификаторов) предпочтительнее, так как это делает программу более простой для понимания.

Основные синтаксические правила записи программ на языке Delphi сводятся к следующему:

- Все используемые типы, константы, переменные, функции, процедуры должны быть объявлены или описаны до их первого использования.
- Прописные и строчные буквы идентичны. Например, идентификаторы LABEL1, Label1 и label1 идентичны. При записи идентификаторов могут использоваться латинские буквы, цифры, символ подчеркивания « _ ». Идентификатор не может начинаться с цифры и не может содержать пробелов. Длина идентификатора не ограничена, но воспринимается не более 255 первых символов идентификатора. Впрочем, реально лучше использовать короткие, но осмысленные идентификаторы.

- При ссылках на идентичные идентификаторы, описанные в разных местах, например, в разных модулях или в разных объектах, используется нотация с точкой, в которой сначала перечисляются идентификаторы объектов, разделенные символами точки. Например, Unit2.A, или Form2.Labell.Caption.

- Каждое предложение языка заканчивается символом точка с запятой (;). Немногие исключения из этого правила будут оговорены особо. В частности, точку с запятой можно не ставить (а можно ставить) перед ключевым словом end.

- В строке может размещаться несколько операторов. Однако с точки зрения простоты чтения текста этим не надо злоупотреблять. Вообще надо писать программу так, чтобы её было легко читать и Вам, и постороннему человеку, которому, может быть, придется её сопровождать. Надо выделять объединённые смыслом операторы в группы, широко используя для этого отступы и комментарии.

- Комментарии в тексте заключаются в фигурные скобки: { текст комментария } – блочный комментарий. Вместо фигурных скобок можно использовать символы круглых скобок с символами звездочки «*»: (*текст комментария*). Комментарии, заключенные в фигурные скобки или в круглые скобки со звездочками, могут вводиться в любом месте текста, в частности, внутри операторов, и занимать любое количество строк. Текст комментария в фигурных скобках не может начинаться с символа доллара, поскольку сочетание символов { $ воспринимается как начало директивы компилятора. Еще один способ введения комментария — размещение его после двух символов «слэш» («//») – строчный комментарий. Этот комментарий должен занимать конец строки, в котором он введен, и не может переходить на следующую строку. Любой текст в строке, помещённый после символов «//» воспринимается как комментарий.

- Операторные скобки **begin...end** выделяют составной оператор. Все операторы,

помещённые между ключевыми словами begin и end, воспринимаются синтаксически как один оператор.

- Программа или отдельный модуль завершаются оператором «end.» (служебное слово end с символом точки).

1.5.3 Тип данных

Любые данные, то есть константы, переменные, свойства, значения функций или выражений в Delphi характеризуются своими типами. Тип определяет множество допустимых значений, которые может иметь тот или иной объект, а также множество допустимых операций, которые применимы к нему. Кроме того, тип определяет также и формат внутреннего представления данных в памяти компьютера. Программа может оперировать данными различных типов: целыми и дробными числами, символами, строками символов, логическими величинами.

Типы данных - специальные конструкции языка, которые рассматриваются компилятором как образцы для создания других элементов программы, таких как переменные, константы и функции. Любой тип определяет для объекта:

- множество допустимых значений, которые может иметь тот или иной объект;
- множество допустимых операций, которые применимы к объекту;
- формат внутреннего представления данных в памяти компьютера.

Первоначально типы как раз и предназначались для того, чтобы программист явно указывал, какого размера память нужна ему и что он с ней собирается делать.

Delphi характеризуется разветвленной структурой типов данных (рисунок 1.15). В языке предусмотрен механизм создания новых типов.

Рис. 1.15: Типы данных в Delphi

1.5.4 Скалярные (простые) типы

Простые типы данных, это такие типы, которые обеспечивают хранение только одного значения. К простым типам относятся порядковые и вещественные типы, а также тип дата-время.

Порядковые типы, это типы, в которых все значения образуют упорядочен-

ную последовательность, и значение переменной порядкового типа определяется его местом в этой последовательности. В Delphi определены три группы порядковых типов (**целые, символьные, логические**) и два типа, определяемых пользователем (**перечислимый и тип-диапазон**).

Порядковые типы отличаются тем, что каждый из них имеет конечное количество возможных значений. Эти значения можно определённым образом упорядочить и, следовательно, с каждым из них можно сопоставить некоторое целое число – порядковый номер значения. К любому из них применима функция Ord(x), которая возвращает порядковый номер значения выражения x.

Для целых типов **Ord(x)** возвращает само число x. Применение Ord(x) к логическому, символьному и перечислимому типам даёт положительное число в диапазоне 0..1, 0..255, 0..65535.

К порядковым типам можно также применять функции:

Pred(x) – возвращает предыдущее значение порядкового типа;

Succ(x) – возвращает следующее значение порядкового типа.

Если переменная c:=5, то значения функций равны: Pred(c) = 4; Succ(c) = 6.

Если представить себе любой порядковый тип, как упорядоченное множество значений, возрастающих слева направо и занимающих на числовой оси некоторый отрезок, то функция Pred(x) не определена для левого, Succ(x) – для правого конца этого отрезка.

Тип-дапазон (Subbrange) - это специальная конструкция языка Delphi, позволяющая присваивать переменным значения, лежащие в заданном диапазоне. Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме другого типа-диапазона.

Данный тип задаётся границами своих значений внутри базового типа, на котором он основан. Тип-диапазон может задаваться двумя способами - в специальном разделе **type** или непосредственно при объявлении переменной.

```
type
  basic_digits = 0..9;
```

l_char = 'a'.. 'z';

Можно также объявлять тип-диапазон и в разделе var:

var

basic_digits : 0..9;

l_char : 'a'.. ' z';

При объявлении диапазона важно, что две точки (..) рассматриваются как один символ, поэтому их нельзя разделять пробелами. Кроме того, левая граница диапазона не должна превышать правую.

Тип-диапазон наследует все свойства своего базового типа, но с ограничениями, связанными с его меньшей мощностью.

High(x) – функция возвращает максимальное значение типа-диапазона, к которому принадлежит переменная x.

Low(x) – функция возвращает минимальное значение типа-диапазона.

Вещественные типы, строго говоря, тоже имеют конечное количество значений, которое определяется форматом внутреннего представления вещественного числа. Однако это количество настолько велико, что сопоставить с каждым из них целое число (его номер) на представляется возможным.

Тип *дата-время* предназначен для хранения даты и времени. Фактически для этих целей он использует вещественный формат.

1.5.5 Целые типы данных

Язык Delphi поддерживает семь целых типов данных: **Shortint, Smailint, Longint, Int64, Byte, Word и Longword**, описание которых приведено в табл. 1.1.

Таблица 1.1. Целые типы

Тип	**Диапазон**	**Формат**
Shortint	-128 – 127	8 битов
Smallint	-32 768 – 32 767	16 битов
Longint = Integer	-2 147 483 648 – 2 147 483 647	32 битов
Int64	$-2^{63} - 2^{63}-1$	64 битов
Byte	0 – 255	8 битов, беззнаковый
Word	0 – 65 535	16 битов, беззнаковый
Longword	0 – 4 294 967 295	32 битов, беззнаковый

Delphi поддерживает и наиболее универсальный целый тип - Integer, который эквивалентен Longint.

При использовании процедур и функций с целочисленными параметрами следует руководствоваться «вложенностью» типов, то есть везде, где может использоваться тип Word, допускается использование типа Byte (но не наоборот), в Longint «входит» Smallint, который, в свою очередь, включает в себя Shortint.

Пример:

```
k := 65535; // максимальное значение типа word
k : word;
k := k+1; // по правилам математики k = 65536, на самом деле k = 0
edit1.text := IntToStr(k); // в компонент Edit1 будет выведено значение 0
Можно изменить программу:
k : word;
k := 65535;
edit1.text := IntToStr(k+1); // в компонент Edit1 будет выведено значение 65536
```

1.5.6 Вещественные типы данных

Язык Delphi поддерживает шесть вещественных типов: **Real, Single, Double, Extended, Comp, Currency**. Типы различаются между собой диапазоном допустимых значений, количеством значащих цифр и количеством байтов, необходимых для хранения данных в памяти компьютера (табл. 1.2).

Таблица 1.2. Вещественные (дробные) типы

Тип	**Диапазон**	**Значащих цифр**	**Байтов**
Single	$1.5 \times 10^{-45} - 3.4 \times 10^{38}$	6-8	04
Real	$5.0 \times 10 - ^{324} - 1.7 \times 10^{308}$	15-16	08
Double	$5.0 \times 10^{-324} - 1.7 \times 10^{308}$	15-16	08
Comp	$2^{63} + 1 - 2^{63} - 1$	19-20	08
Currency	-922 337 203 685 477.5808 -922 337 203 685 477.5807	19-20	08
Extended	$3.6 \times 10^{-4951} - 1.1 \times 10^{493}$	19-20	10

Язык Delphi поддерживает наиболее универсальный вещественный тип Real, который эквивалентен Double.

В тексте программы числовые константы записываются обычным образом, т. е. так же, как числа, например, при решении математических задач. При записи дробных чисел для разделения целой и дробных частей используется **точка**. Если константа отрицательная, то непосредственно перед первой цифрой ставится знак «минус».

Ниже приведены примеры числовых констант:

1230.0

-524.030

Дробные константы могут изображаться в виде числа с плавающей точкой (экспоненциальная форма записи вещественного числа). Представление в виде числа с

плавающей точкой основано на том, что любое число может быть записано в экспоненциальной форме как произведение числа, меньшего 10, которое называется мантиссой, и степени десятки, именуемой порядком.

В табл. 1.3 приведены примеры чисел, записанных в обычной форме, в алгебраической форме и форме с плавающей точкой.

Таблица 1.3. Примеры записи дробных чисел

Число	**Экспоненциальная форма (в математике)**	**Экспоненциальная форма (в программировании)**
1 000 000	1×10^6	1.0000000000E+06
-123,452	$-1,23452 \times 10^2$	-1.2345200000E+02
0,0056712	$5,6712 \times 10^{-3}$	5.6712000000E-03

Нормальной формой числа с плавающей точкой называется такая форма, в которой мантисса (без учёта знака) находится на полуинтервале $0 \leq a < 1$.

Нормализованная форма записи (распространена в информатике), в которой мантисса десятичного числа принимает значения от 1 (включительно) до 10 (исключительно), то есть $1 \leq a < 10$.

1.5.7 Логический тип данных

Логическая величина может принимать одно из двух значений True (истина) или False (ложь). В языке Delphi логические величины относят к типу Boolean. При этом служебные слова True и False можно использовать в программе в качестве логической константы. В языке Delphi существует два вида констант: обычные и именованные.

Обычная константа — это целое или дробное число, строка символов или отдельный символ, логическое значение

Именованная константа — это имя (идентификатор), которое в программе используется вместо самой константы.

Именованная константа, как и переменная, перед использованием должна быть объявлена в блоке **Const**. В общем виде инструкция объявления именованной константы выглядит следующим образом:

Const константа = значение;

где:

константа — идентификатор (имя) константы;

значение — значение константы.

Например:

```
const
Bound = 10;
Title = 'Скорость бега';
π = 3.1415926;
```

После объявления именованной константы в программе вместо самой константы можно использовать её имя.

В отличие от переменной, при объявлении константы тип явно не указывают. Тип константы определяется её видом, например:

125 — константа целого типа;

0.0 — константа вещественного типа;

' выполнить ' — строковая константа;

' \ ' — символьная константа.

1.5.8 Структура программы

Все объекты, не являющиеся зарезервированными в Delphi, наличие которых обусловлено инициативой программиста, перед первым использованием в программе должны быть описаны. Это производится для того, чтобы компьютер перед выполнением программы зарезервировал память под соответствующие объекты и поставил в соответствие этим участкам памяти идентификаторы. Раздел описаний может состоять из пяти подразделов. Правила языка Delphi предусматривают единую для всех программ форму основной структуры:

Program <Имя программы>;
Раздел описаний:
1. Описание меток (Label)
2. Описание типов (Type)
3. Описание констант (Const)
4. Описание переменных (Var)
5. Описание процедур и функций (Procedure, Function)

Begin
<тело программы>
End.

Слова **Program, Begin, End, Label, Type, Const, Var, Procedure, Function** являются служебными. Правильное и уместное употребление этих слов является обязательным. Имя программы выбирается программистом самостоятельно в соответствии с правилами построения идентификаторов. При отсутствии необходимости в каком-либо виде объектов, соответствующий подраздел может быть опущен.

1.5.9 Переменная

Переменная – именованный участок памяти для хранения данных определённого типа. Когда программа манипулирует с данными, она, фактически, оперирует содержимым ячеек памяти, т. е. переменными. Переменная характеризуется параметрами:

- имя переменной;
- тип переменной;
- значение переменной;
- вид переменной (глобальная, локальная).

Чтобы программа могла обратиться к переменной (области памяти), например, для того, чтобы получить исходные данные для расчета по формуле или сохранить результат, переменная должна иметь имя (идентификатор). Имя переменной придумывает программист.

В качестве имени (идентификатора) переменной можно использовать последовательность из букв латинского алфавита, цифр и некоторых специальных символов. Первым символом в имени переменной должна быть буква. Пробел в имени переменной использовать нельзя.

Следует обратить внимание на то, что компилятор языка Delphi не различает прописные и строчные буквы в именах переменных, поэтому имена SUMMA, Summa и summa обозначают одну и ту же переменную.

Желательно, чтобы имя переменной было логически связано с ее назначением. Например, переменным, предназначенным для хранения коэффициентов и корней квадратного уравнения, которое в общем виде традиционно записывают ax2 + bx + c = 0, вполне логично присвоить имена a, b, c, x1 и x2. Другой пример: если в программе есть переменные, предназначенные для хранения суммы покупки и величины скидки, то этим переменным можно присвоить имена TotalSumm и Discount или ObSumma и Skidka.

В языке Delphi каждая переменная перед использованием должна быть **объявлена**. Раздел описания переменных начинается служебным словом **Var**, после которого следует имя переменной и её тип. С помощью объявления устанавливается не только факт существования переменной, но и задается ее тип, чем указывается и диапазон допустимых значений.

После ключевого слова **var** может следовать не одно, а множество объявлений переменных. Каждое объявление может содержать список идентификаторов переменных, разделяемых запятыми, или один идентификатор. Указываемый в объявлении тип может быть или одним из встроенных типов, или типом, определенным ранее пользователем, или непосредственно описанием вводимого пользователем типа.

В общем виде инструкция объявления переменной выглядит так:

Var Имя : тип;

где: **Имя** — имя переменной;

тип — тип данных, для хранения которых предназначена переменная.

Пример:

var a, b : Real; i : Integer;

Если в программе имеется несколько переменных, относящихся к одному типу, то имена этих переменных можно перечислить в одной строке через запятую, а тип переменных указать после имени последней переменной через двоеточие. В приведенном примере объявлены две переменные типа real и одна переменная типа integer.

Переменные можно разделить на два вида: **локальные и глобальные**. Переменные, объявляемые в процедурах и функциях, являются **локальными**. Они существуют только во время выполнения соответствующей процедуры или функции. Т.е. память для них выделяется только при вызове соответствующей процедуры или функции и освобождается при возврате в вызвавшую процедуру. Переменные, объявленные вне процедур или функций, являются **глобальными**.

Локальные переменные инициализировать при объявлении нельзя. Глобальные переменные можно инициализировать в их объявлениях, т.е. задавать им начальные значения:

<идентификатор переменной> : <тип> = <константное выражение>;

Приведем примеры инициализации переменных:

var I : **integer** = 51;

Deg : **double** = 35*180/Pi;

1.5.10 Оператор присваивания

Самым простым действием над переменной является занесение в нее величины соответствующего типа. Иногда говорят об этом, как о *присвоении переменной конкретного значения*. **Оператор присваивания** является основной вычислительной инструкцией. Если в программе надо выполнить вычисление, то нужно использовать оператор присваивания.

В результате выполнения оператора присваивания значение переменной меняется, ей присваивается значение.

В общем виде оператор присваивания выглядит так:

Имя : = Выражение;

где: **Имя** — переменная, значение которой изменяется в результате выполнения инструкции присваивания;

: = — символ оператора присваивания;

Выражение — выражение, значение которого присваивается переменной, имя которой указано слева от символа оператора присваивания.

Пример:

Summa := Cena ∗ Kol; Skidka := 10; Found := False;

Оператор присваивания выполняется следующим образом (**семантика работы оператора присваивания**):

1. Вычисляется значение выражения справа от оператора присваивания.
2. Выполняется проверка соответствия типа выражения типу переменной.
3. Вычисленное значение выражения записывается в переменную, имя которой стоит слева от оператора присваивания.

Например, в результате выполнения последовательности операторов:

i := 0; // *значение переменной i становится равным нулю*

a := b+c; // *значением переменной a будет число, равное сумме значений переменных b и c*

j := j+1; // *значение переменной j увеличивается на единицу*

Выражение, указанное справа от знака «:=», должно приводить к значению **того же типа**, что и сама переменная, или типа, **совместимого** с типом переменной относительно операции присваивания. Операция присваивания считается верной, если тип выражения соответствует или может быть приведен к типу переменной, получающей значение. Например, переменной типа real можно присвоить значение выражения, тип которого real или integer, а переменной типа integer можно присвоить значение выражения только типа integer или любых других целых типов, не превосходящих integer.

Так, например, если переменные i и n имеют тип integer, а переменная d — тип real, то операторы i := n/10; i := 1.0; – неправильные, а оператор d := i+1 правильный.

Во время компиляции выполняется проверка соответствия типа выражения типу переменной. Если тип выражения не соответствует типу переменной, то компилятор выводит сообщение об ошибке:

Incompatible types ... and ...

где вместо многоточий указывается тип выражения и переменной. Например, если переменная n целого типа, то оператор n := m/2 неверен, поэтому во время компиляции будет выведено сообщение:

Incompatible types 'Integer' and 'Extended'.

1.5.11 Операции и выражения

Основными элементами, из которых конструируется исполняемая часть программы, являются константы, переменные и обращения к функциям. Каждый из этих элементов характеризуется своим значением и принадлежит к какому-либо типу данных. С помощью знаков операций и скобок из них можно составлять **выражения**. В простейшем случае выражение может представлять собой константу или переменную:

Y	21	(a+b)*c	Sin(t)
a>2	[1..5, 7]*set1	i+1	
A + B/C	Summa∗0.75	(B1+B3+B3)/3	Cena MOD 100

Выражение состоит из **операндов** и знаков операций. Знаки операций находятся между операндами и обозначают действия, которые выполняются над операндами. Например, в выражении **A + B** A и B – операнды, «+» – знак операции. В качестве операндов выражения можно использовать переменную, константу, функцию или другое выражение.

Выделяют **унарные операции**, для которых требуется один операнд (например, -x), и **бинарные операции**, для которых требуются два операнда (например, s+2).

Для каждого типа данных существует свой набор операций, поэтому операнды должны быть одного типа или совместимых типов. Нельзя складывать числа со строками или сравнивать их между собой!

Тип выражения определяется типом операндов, входящих в выражение, и зависит от операций, выполняемых над ними. Например, если оба операнда, над которыми выполняется операция сложения, целые, то очевидно, что результат тоже является целым. А если хотя бы один из операндов дробный, то тип результата дробный, даже в том случае, если дробная часть значения выражения равна нулю.

Важно уметь определять тип выражения. При определении типа выражения следует иметь в виду, что тип константы определяется её видом, а тип переменной задаётся в инструкции объявления. Например, константы 0, 1 и -512 — целого типа (Integer), а константы 1.0, 0.0 и 3.2E-05 — вещественного типа (Real).

В табл. 1.4 приведены правила определения типа выражения в зависимости от типов входящих операндов и используемой операции.

Таблица 1.4. Правила определения типа выражения

Операция	**Тип операндов**	**Тип выражения**
*, +, -	хотя бы один из операндов Real	Real
*, +, -	оба операнда Integer	Integer
/	Real или Integer	всегда Real
DIV, MOD	всегда Integer	всегда Integer

По характеру выполняемых действий операции разделяются на следующие группы:

- арифметические операции;
- операции сравнения (отношения);
- булевы (логические) операции;
- строковая операция (конкатенация);
- операции над множествами;
- операция взятия адреса.

С числовыми величинами можно выполнять **арифметические операции** (табл. 1.5.)

Таблица 1.5. Арифметические операции

Операция	**Действие**
+	сложение (тип результата зависит от типов входящих операндов)
-	вычитание (тип результата зависит от типов входящих операндов)
*	умножение (тип результата зависит от типов входящих операндов)
/	деление (результат – всегда число вещественного типа)
DIV	деление. Результат – целое число, обозначающее целую часть от деления нацело (операнды - только целые числа)
MOD	деление. Результат – целое число, обозначающее остаток от деления нацело (операнды - только целые числа)

При записи выражений между операндом и операцией, за исключением операции DIV и MOD, пробел можно не ставить.

Результат применения операций +, -, * и / очевиден.

Операция DIV позволяет получить целую часть результата деления одного целого числа на другое целое число. Например, значение выражения 15 div 7 равно 2. Знак результата зависит от обоих операндов («умножается»):

13 div 5 = 2; -13 div 5 = -2; 13 div (-5) = -2; -13 div (-5) = 2

Операция MOD, деление по модулю, позволяет получить остаток от деления одного целого числа на другое целое число. Например, значение выражения 15 MOD 7 равно 1. Знак результата зависит от делимого (левого операнда):

13 mod 5 = 3; -13 mod 5 = -3; 13 mod (-5) = 3; -13 mod (-5) = 3

Операции сравнения используются при сравнении двух операндов. Операции сравнения называют ещё операциями **отношения**.

Определены следующие операции сравнения: < > <= >= = <>.

Результат операций сравнения – величина логического типа, т.е. операции сравнения возвращают **True (Истина)**, если указанное соотношение операндов выполняется, и **False (Ложь)**, если соотношение не выполняется.

Операции сравнения - бинарные (используют 2 операнда). Под операндом понимается то выражение, над которым выполняется операция.

Операции сравнения применяются только к операндам одинаковых или совместимых типов. При записи условий следует обратить особое внимание на то, что операнды условия должны быть одного типа или, если тип операндов разный, то тип одного из операндов может быть приведен к типу другого операнда. Например, если переменная Key объявлена как integer, то условие Key = Chr(13) синтаксически неверное, т. к. значение, возвращаемое функцией Chr, имеет тип char (символьный). Во время компиляции программы при обнаружении неверного условия компилятор выводит сообщение: *Incompatible types* (несовместимые типы).

Приоритет операций сравнения ниже, чем у арифметических операций, поэтому в выражении a + b > 6 * x сначала будут выполняться арифметические операции (+ и *), а затем операция сравнения.

С помощью операций сравнения формируются **простые условия**. Например:
Summa < 1000 // *результат True, если значение переменной Summa меньше 1000*
Score >= HBound // *результат True, если значение переменной Score больше или равно значения переменной HBound*
Sim = Ord('1') // *результат True, если значение переменной Sim равно коду символа '1' в кодировочной таблице ASCII*

Булевы (логические) операции принимают операнды логического типа и возвращают результат также логического типа.

Операция	Обозначение
NOT	отрицание НЕ
AND	логическое И
OR	логическое ИЛИ
XOR	логическое исключающее ИЛИ

Результаты выполнения логических операций зависят от значений входящих в них операндов (False или True) и образуют **таблицу истинности логических операций**, представленую в табл. 1.6.

Таблица 1.6. Выполнение логических операций для операндов A и B

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Из простых условий при помощи логических операций можно строить **сложные условия**. При записи сложных условий важно учитывать то, что **логические операции имеют более высокий приоритет, чем операции сравнения**, и поэтому простые условия следует заключать в скобки. Например:

(ch >= '0') and (ch <= '9')

(day = 7) or (day = 6)

(Forml.Editl.Text <> ' ') or (Forml.Edit2.Text <> '')

Forml.CheckBoxl.Checked and (Forml.Editl.Text <> '')

Например, пусть условие предоставления скидки сформулировано следующим образом: «Скидка предоставляется, если сумма покупки превышает 100 руб. и день покупки — воскресенье», Если день недели обозначен, как переменная Day целого типа, и равенство её значения 7 соответствует воскресенью, то условие предоставления скидки можно записать так:

(Summa > 100) and (Day = 7)

Если условие предоставления скидки дополнить тем, что скидка предоставляется в любой день, если сумма покупки превышает 500 руб., то условие можно записать:

((Summa > 100) and (Day =7)) or (Summa > 500)

1.5.12 Приоритет операций

Последовательность выполнения операций в выражении определяется тремя факторами: приоритетом операций, порядком расположения операций в выражении, использованием скобок.

Приоритет, ранг или старшинство операции или оператора — формальное свойство оператора/операции, влияющее на очередность его выполнения в выражении с несколькими различными операторами при отсутствии явного (с помощью скобок) указания на порядок их вычисления.

При вычислении значений выражений следует учитывать, что операции имеют разный **приоритет**. Так у операций *, /, DIV, MOD более высокий приоритет, чем у операций + и -.

Приоритет операций влияет на порядок их выполнения. При вычислении значения выражения в первую очередь выполняются операции с более высоким приоритетом. Если приоритет операций в выражении одинаковый, то сначала выполняется та операция, которая находится **левее**. Операции в порядке понижения приоритета:

1. стандартные функции;
2. унарные операции (+ - not);
3. мультипликативные операции (умножения) (∗ / div mod and);
4. аддитивные операции (сложения) (+ - or xor);
5. операции (сравнения) отношения (= <> < > ⩽ ⩾ in).

Для задания нужного порядка выполнения операций в выражении можно использовать скобки, например: (r1+r2+r3)/(r1∗r2∗r3)

Выражение, заключенное в скобки, трактуется как один операнд. Это означает, что операции над операндами в скобках будут выполняться в обычном порядке, но раньше, чем операции над операндами, находящимися за скобками. При записи выражений, содержащих скобки, должна соблюдаться парность скобок, т. е. число открывающих скобок должно быть равно числу закрывающих скобок. Нарушение парности скобок — наиболее распространённая ошибка при записи выражений.

1.5.13 Основные арифметические функции

Для выполнения часто встречающихся вычислений и преобразований язык Delphi предоставляет программисту ряд стандартных математических функций.

Значение функции связано с её именем. Поэтому функцию можно использовать в качестве операнда выражения, например, в операторе присваивания.

Функция характеризуется типом её значения, а также типом и количеством входящих в неё параметров. Тип переменной, которой присваивается значение функции, должен соответствовать типу функции. Точно так же тип **фактического** параметра функции, т. е. параметра, который указывается при обращении к функции, должен соответствовать типу **формального** параметра. Если это не так, компилятор выводит сообщение об ошибке.

Математические функции (табл. 1.7) позволяют выполнять различные вычисления.

Таблица 1.7. Математические функции

Функция	**Значение**
Abs(n)	Абсолютное значение n (модуль числа n)
Sqrt(n)	Квадратный корень из n
Sqr(n)	Квадрат n
Sin(n)	Синус n
Cos(n)	Косинус n
Arctan(n)	Арктангенс n
Exp(n)	Экспонента n (число e в степени n, число e=2,72...)
Ln(n)	Натуральный логарифм n
Random(n)	Псевдослучайное целое число в интервале [0,n) (от 0 до n-1)
Random	Псевдослучайное вещественное число (0..1)
Randomize	Процедура для инициализации датчика случайных чисел
Frac(x)	Дробная часть числа
Pi	Число ПИ (число P=3,17...)
Round(n)	Целое, полученное путём округления n по известным правилам
Trunc(n)	Целое, полученное путём отбрасывания дробной части n
Power(X,Y)	Возведение X в степень Y

Величина угла в тригонометрических функциях (*sin, cos*) должна быть выражена в радианах. Для преобразования величины угла A из градусов в радианы используется формула $(A * Pi)/180$.

Обычно функции используют в качестве операндов выражений. Параметром функции может быть константа, переменная или выражение соответствующего типа. Ниже приведены примеры использования стандартных функций и функций преобразования типов.

```
n := Round((x2-x1)/dx);
x1 := (-b + Sqrt(d)) / (2*a);
m := Random(10);
cena := StrToInt(Edit1.Text);
Edit2.Text := IntToStr(100);
mes := 'x1=' + FloatToStr(xl);
```

В Delphi включен модуль **Math**, который существенно расширяет перечисленный набор встроенных математических функций. Чтобы воспользоваться этими функциями, нужно добавить к перечисленным в программе в разделе **uses** модулям название модуля **Math**.

1.5.14 Функции преобразования типов

Функции преобразования типов предназначены для перевода чисел в строковый формат и наоборот (табл. 1.8) и наиболее часто используются в выражениях, обеспечивающих ввод и вывод информации. Например, пусть необходимо вывести в поле вывода (компонент Label) диалогового окна значение переменной X типа Real, которая в процессе работы программы стала равна 2,538. Для этого необходимо преобразовать вещественное число в строковую величину, изображающую данное число, при помощи функции **FloatToStr(X)**, которая возвращает строковое представление значения переменной, указанной в качестве параметра функции: '2.538'.

Таблица 1.8. Функции преобразования типов

Функция	**Значение функции**
IntToStr(n)	строка, являющаяся изображением целого числа N
FloatToStr(x)	строка, являющаяся изображением вещественного числа X
FloatToStrF(x, f, k, m)	строка, являющаяся изображением вещественного числа X. При вызове функции указывают: f — формат (способ изображения); k — точность (нужное общее количество цифр); m — количество цифр после десятичной точки
StrToInt(s)	число целого типа, изображением которого является строка S
StrToFloat(s)	число вещественного типа, изображением которого является строка S

Ошибки

Компилятор генерирует исполняемую программу лишь в том случае, если исходный текст не содержит *синтаксических* ошибок.

Чтобы перейти к фрагменту кода, который содержит ошибку, надо установить

курсор в строку с сообщением об ошибке и из контекстного меню выбрать команду **Edit source**.

Процесс устранения ошибок носит итерационный характер. Обычно сначала устраняются наиболее очевидные ошибки, например, декларируются необъявленные переменные. После очередного внесения изменений в текст программы выполняется повторная компиляция. Следует учитывать тот факт, что компилятор не всегда может точно *локализовать* ошибку. Поэтому, анализируя фрагмент программы, который, по мнению компилятора, содержит ошибку, нужно обращать внимание не только на тот фрагмент кода, на который компилятор установил курсор, но и на тот, который находится в предыдущей строке.

Если в программе нет синтаксических ошибок, компилятор создаёт исполняемый файл программы. Имя исполняемого файла такое же, как и у файла проекта, а расширение — **.exe**. Delphi помещает исполняемый файл в тот же каталог, где находится файл проекта.

При обнаружении в программе неточностей, которые не являются ошибками, компилятор выводит подсказки (**Hints**) и предупреждения (**Warnings**).

Например, наиболее часто выводимой подсказкой является сообщение об объявленной, но не используемой переменной:

Variable ... is declared but never used in ... – Переменная описана, но не используется

Variable ... might not have been initialized. – Используется не инициализированная переменная. В программе нет инструкции, которая присваивает переменной начальное значение

2.1 Условный оператор If (ветвления)

На практике редко встречаются задачи, алгоритм решения которых является линейным. Часто оказывается, что алгоритм решения даже элементарной задачи предполагает выполнение разных действий в зависимости от выполнения / невыполнения какого-нибудь условия.

Точки алгоритма, в которых выполняется выбор дальнейшего хода программы, называются точками выбора. Выбор очередного шага решения задачи осуществляется в зависимости от выполнения некоторого условия.

В программе **условие** — это выражение логического типа (**Boolean**), которое может принимать одно из двух значений: True (Истина) или False (Ложь).

Выбор в точке разветвления алгоритма очередного шага программы может быть реализован при помощи одного из двух условных операторов **If** и **Case**. Логический оператор If позволяет выбрать один из двух возможных вариантов (оператор альтернативы), оператор Case — один из нескольких предопределённых вариантов (оператор выбора). Выбор осуществляется в зависимости от выполнения условия.

В общем виде оператор If записывается так (**синтаксис оператора If**):

```
if условие
  then
   begin
     // операторы, которые надо выполнить, если условие True
   end
  else
   begin
    // операторы, которые надо выполнить, если условие False
   end;
```

В условном операторе слова **If, then, else** – служебные слова. Перед **else** (после **end**) точка с запятой не ставится!

Выполняется оператор If следующим образом (**семантика оператора If**):

1. Вычисляется значение условия (условие — выражение логического типа, значение которого может быть равно **True** или **False**).

2. Если условие истинно (значение выражения *условие* равно **True**), то выполняется оператор, следующий за словом **then** (между begin и end). На этом выполнение оператора If заканчивается, то есть оператор, следующий за **else**, не будет выполнен.

   Если условие ложно (значение выражения *условие* равно **False**), то выполняется оператор, следующий за словом **else** (между begin и end).

На рисунке 2.1 представлена блок-схема полной формы записи логического оператора **If-then-else**

Если в операторе If между begin и end находится только один оператор, то логические операторные скобки begin ... end можно не писать.

Рис. 2.1: Полная форма записи логического оператора **If-then-else**

Например,

```
if otv=3
  then
    begin
     prav:=prav+1 ;
    end
  else
    begin
     WriteLn('Ошибка!');
    end;
```

Фрагмент кода синтаксически написан верно, но наличие логических операторных скобок begin...end в обоих ветках оператора If не оправдано, поэтому оператор *можно переписать так:*

```
if otv=3
  then prav:=prav+l
  else WriteLn('Ошибка!') ;
```

При вложенных операторах if могут возникнуть неоднозначности в понимании того, к какому из вложенных операторов if относится элемент else. Компилятор всегда считает, что else относится к последнему из операторов if, в котором не было раздела else.

Например, в конструкции

```
if <условие1>
  then
    if <условие2>
      then <оператор1>
      else <оператор2>;
```

else будет отнесен компилятором ко второму оператору if, т.е. <оператор2> будет выполняться в случае, если первое условие истинно, а второе ложно. Иначе говоря,

вся конструкция будет прочитана так:

if <условие1>
then
begin
if <условие2> **then** <оператор1> **else** <оператор2>
end;

Если же необходимо отнести else к первому if, то это надо записать в явном виде с помощью логических операторных скобок begin...end:

if <условие1>
then
begin
if <условие2> **then** <оператор1>
end
else <оператор2>;

Если какое-либо действие должно быть выполнено только при выполнении определённого условия и пропущено, если это условие не выполняется, то оператор if может быть записан в сокращённой форме без ветки else:

if условие
then
begin
{ операторы, которые надо выполнить, если условие True }
end

На рисунке 2.2 представлена блок-схема сокращённой формы записи логического оператора **If-then**

if n=m
then c:=c+l;

Здесь значение переменной с будет увеличиваться только в том случае, если значения переменных n и m оказались равны.

Рис. 2.2: Сокращённая форма записи логического оператора **If-then**

Часто в программе необходимо реализовать выбор более чем из двух вариантов. Например, составим алгоритм для оценки веса человека: известно, что для каждого человека существует оптимальное значение веса, которое может быть вычислено по формуле: Рост(см) минус 100.

Реальный вес может отличаться от оптимального: вес может быть меньше оптимального, равняться ему или превышать оптимальное значение – это три варианта развития событий. В этом случае алгоритм может быть реализован с помощью вложенных друг в друга логических операторов: если реальный вес равен оптимальному, то все хорошо (можно похвалить, например, пользователя), иначе возможны два варианта: если реальный вес меньше оптимального, то пользователю нужно поправиться, иначе ему нужно похудеть. Блок-схема алгоритма приведена на рисунке 2.3.

Создадим программу, которая запрашивает вес и рост, вычисляет оптимальное значение, сравнивает его с реальным весом и выводит соответствующее сообщение.

program Vichislenie;

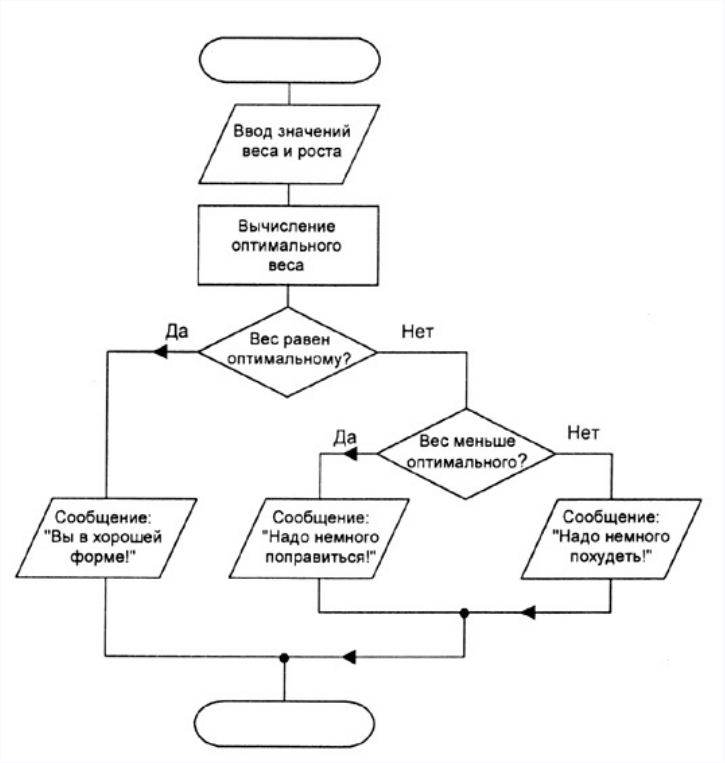


Рис. 2.3: Блок-схема алгоритма «Контроль веса человека»

```
var
   Ves, Rost, OptimVes : real;
begin
  ReaLn(Ves);
  ReadLn(Rost);
  OptimVes := Rost - 100;
```

```
  if Ves=OptimVes
    then WriteLn('Вы в хорошей форме!')
    else
      if Ves < OptimVes
        then WriteLn('Надо поправиться на ' + FloatToStr(OptimVes - Ves) + ' кг');
        else WriteLn('Надо похудеть на ' + FloatToStr(Ves - OptimVes) + ' кг');
end;
```

В приведённом примере множественный выбор реализован при помощи двух операторов if, один из которых «вложен» в другой.

2.2 Оператор выбора Case

В языке Delphi есть оператор Case, который позволяет эффективно реализовать множественный выбор. В общем виде он записывается следующим образом (**синтаксис оператора Case**):

```
Case Селектор of
  список1: begin  инструкции 1  end;
  список2: begin  инструкции 2  end;
  списокN: begin  инструкции N  end;
  else begin  инструкции  end;
end;
```

где:

Селектор — выражение, значение которого определяет дальнейший ход выполнения программы (т. е. последовательность инструкций, которая будет выполнена). В этом операторе селектор должен иметь порядковый тип. Поэтому, например, нельзя использовать выражения, возвращающие действительные числа или строки.

Список N — список констант. Если константы представляют собой диапазон чисел, то вместо списка можно указать первую и последнюю константу диапазона,

разделив их двумя точками. Например, список 1, 2, 3, 4, 5, 6 может быть заменен диапазоном 1..6. Списки могут содержать константы и константные выражения, которые совместимы по типу с объявленным селектором и которые компилятор может вычислить заранее, до выполнения программы. Допустимо использование ограниченных типов. Недопустимо использование переменных и многих функций. В списках не допускается повторение одних и тех же значений, поскольку в этом случае выбор был бы неоднозначным.

Выполняется оператор case следующим образом (**семантика оператора Case**):

1. Сначала вычисляется значение выражения-селектора.
2. Значение выражения-селектора последовательно сравнивается с константами из списков констант.
3. Если значение выражения совпадает с константой из списка, то выполняется соответствующая этому списку группа инструкций. На этом выполнение оператора Case завершается.
4. Если значение выражения-селектора не совпадает ни с одной константой из всех списков, то выполняется последовательность инструкций, следующая за Else.

Синтаксис оператора Case позволяет не писать Else и соответствующую последовательность инструкций. В этом случае, если значение выражения не совпадает ни с одной константой из всех списков, то выполняется следующая за Case инструкция программы. На рисунке 2.4 представлена блок-схема оператора Case.

Примеры различной обработки данных с помощью оператора выбора Case:

Пример 1:

case n_day of
1,2,3,4,5 : day := 'Рабочий день.' ;
6 : day := 'Суббота!';
7 : day := 'Воскресенье!';

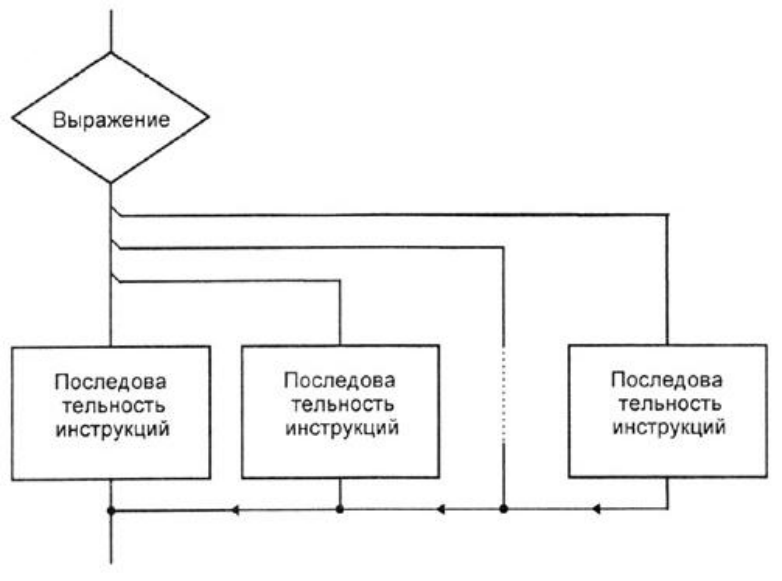


Рис. 2.4: Блок-схема логического оператора выбора Case

end;

Пример 2:

case n_day of
1..5 : day := 'Рабочий день.';
6 : day := 'Суббота!';
7 : day := 'Воскресенье!';
end;

Пример 3:

case n_day of
6 : day := 'Суббота!';

7 : day := 'Воскресенье!';
else day:='Рабочий день.';
end;

Пример: Рассмотрим пример использования оператора Case. При выводе числовой информации с поясняющим текстом возникает проблема согласования выводимого значения и окончания поясняющего текста.

Например, в зависимости от числового значения, поясняющий текст к денежной величине может быть: *«рубль», «рублей» или «рубля»* (123 рубля, 120 рублей, 121 рубль). Очевидно, что окончание поясняющего слова определяется последней цифрой числа: 0, 5, 6, 7, 8, 9 – *«рублей»*; 1 – *«рубль»*; 2, 3, 4 – *«рубля»*.

Приведённое правило имеет исключение для чисел, оканчивающихся на 11, 12, 13, 14. Для них поясняющий текст должен быть *«рублей»*.

Диалоговое окно программы реализовано для визуальной среды Delphi (рисунок 2.5) и включает компоненты Label1 («Введите целое число и нажмите Enter»), Edit1 (для введения пользователем целого числа), Label2 (для отображения результатов формирования поясняющего текста). Поясняющий текст формируется в процедуре обработки события onKeyPress (щелчка на кнопке Enter клавиатуры) компонента Edit1. Обработчик события TForm1.Edit1KeyPress выполнится после щелчка по кнопке клавиатуры в компоненте Edit1. При этом расчеты будут производиться только в случае нажатия на клавишу Enter. После нажатия на клавишу клавиатуры её код (в виде символьной величины) будет считан в параметр Key обработчика события. Условие **if** Key = chr(VK_ RETURN) - проверка на нажатие именно клавиши Enter.

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char)
var n : integer; // введенное число
 r : integer; //остаток от деления n на 10
 text: string[10]; // формируемый поясняющий текст
begin
  if Key = chr(VK_ RETURN) then
```

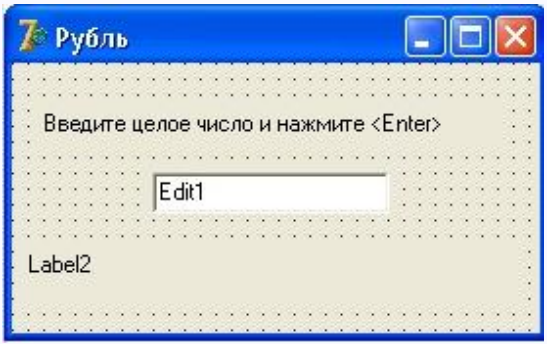


Рис. 2.5: Пример формы в Delphi для реализации алгоритма «Формирование поясняющего текста»

```
  begin
    n := StrToInt(Editl.Text);
    if n > 100 then n:=n mod 100;
    if (n >= 11) and (n <= 14) then text := ' рублей'
       else
         begin
         r:= n mod 10;
         case r of
           1 : text:=' рубль';
           2 .. 4 : text:=' рубля';
           else text := ' рублей';
         end;
         end;
    Label2.Caption := IntToStr(n)+ text;
  end;
  end;
end;
```

Рассмотрим фрагмент программы, которая вычисляет дату следующего дня, используя сегодняшнюю дату, представленную тремя целыми числами в переменных: day (день), month (месяц) и year (год).

```
// вычисление даты следующего дня
var day: integer; // день
  month: integer; // месяц
  year: integer; // год
  last:boolean; //  если день — последний день месяца, то last = True
  r:integer; // если год не високосный, то остаток от деления year на 4 не равен нулю
   begin
{ переменные day, month и year содержат сегодняшнюю дату }
    last := False; // пусть day — не последний день месяца
    case month of
     4,6,9,11 : if day = 30 then last:= True;
     2 : if day = 28 then
          begin
         r := year mod 4;
         if r <> 0 then last:= True;
         end
      else: if day=31 then last:= True;
    end;
    if last then
      begin // последний день месяца
         day:= 1;
         if month =12 then
            begin // последний месяц
              month:= 1;
              year:= year + 1;
```

```
            end
        else month:= month + 1;
    end
else day:= day + 1;// переменные day, month и year содержат завтрашнюю дату
end;
```

Сначала с помощью оператора Case проверяется, является ли текущий день последним днём месяца. Если текущий месяц — февраль и если текущее число — 28, то дополнительно выполняется проверка, является ли год високосным. Для этого вычисляется остаток от деления года на 4. Если остаток равен нулю, то год високосный, и число 28 не является последним днём месяца. В результате выполнения оператора Case переменная last логического типа (boolean) останется равна False, если сегодняшний день - не последний день в месяце, или станет равной True, если он последний (например, 31 января любого года, 28 февраля 2021 года, 29 февраля 2020 года и т.д.).

Если выясняется, что текущий день — последний день месяца, то следующее число — первое. Затем проверяется, не является ли текущий месяц декабрем. Если нет, то увеличивается номер месяца, а если да, то увеличивается номер года, а номеру месяца присваивается значение 1.

2.3 Оператор цикла For

Алгоритмы решения многих задач являются циклическими, т. е. для достижения результата определённая последовательность действий должна быть выполнена несколько раз.

Например, программа контроля знаний выводит вопрос, принимает ответ, добавляет оценку за ответ к сумме баллов, затем повторяет это действие ещё и ещё раз, и так до тех пор, пока испытуемый не ответит на все вопросы.

Другой пример. Для того, чтобы найти фамилию человека в списке, надо про-

верить первую фамилию списка, затем вторую, третью и т. д. до тех пор, пока не будет найдена нужная фамилия или не будет достигнут конец списка.

Алгоритм, в котором есть последовательность операций (группа инструкций), которая должна быть выполнена несколько раз, называется циклическим, а сама последовательность операций именуется циклом.

Циклом называется многократное повторение однотипных действий.

Телом цикла будем называть тот набор действий, которые нужно многократно повторять.

Однократное повторение тела цикла называется **итерацией**.

Бесконечное количество итераций цикла называется **зацикливание**.

В Delphi цикл может быть реализован при помощи операторов цикла **For, While и Repeat**.

Оператор For используется в том случае, если некоторую последовательность операторов (инструкций программы) надо выполнить несколько раз, причем число повторений заранее известно. Оператор For обеспечивает циклическое повторение некоторого (одного!!!) оператора (в частности, составного оператора) заданное число раз. Повторяемый оператор называется телом цикла. Если необходимо повторять в теле цикла несколько операторов, то их необходимо объединить в один составной с помощью логических операторных скобок Begin ... End. Повторение цикла контролируется некоторой управляющей переменной (**счётчиком цикла**), которая увеличивается или уменьшается на единицу при каждом выполнении тела цикла. Повторение завершается, когда счётчик достигает заданного конечного значения.

В общем виде оператор For записывается в одной из двух форм следующим образом (**синтаксис оператора цикла For**):

For счетчик **:=** *нач_знач* **to** *кон_знач* **do**

или

For счетчик **:=** *нач_знач* **downto** *кон_знач* **do**

Begin

// тело цикла: операторы, которые надо выполнить несколько раз

End

где:

счетчик - переменная-счётчик числа повторений (итераций) цикла;

нач_знач - выражение, определяющее начальное значение счётчика цикла;

кон_знач - выражение, определяющее конечное значение счётчика цикла.

Переменная *счётчик*, выражения *нач_знач* и *кон_знач* должны быть порядкового типа (целого, символьного или логического). Переменная *счётчик* внутри тела цикла не может быть изменена.

Последовательность действий при выполнении оператора цикла For...to (**семантика цикла For**):

1. вычисляются *начальное_значение* и *конечное_значение*;
2. счётчику присваивается *начальное_значение*;
3. проверяется, не стал ли счётчик больше, чем *конечное_значение*;
4. выполняется тело цикла;
5. счётчик увеличивается на 1;
6. выполняется пункт 3.

Для цикла с ключевым словом **to** значение счётчика последовательно увеличивается на единицу при каждой итерации цикла. Для цикла с ключевым словом **downto** значение счётчика при каждой итерации цикла уменьшается на 1. Очевидно, что для корректной работы цикла For...downto начальное_значение должно быть больше либо равно конечного_значения.

Количество повторений оператора цикла For можно вычислить по формуле

конечное_значение - начальное_значение + 1

Примеры:

for i := 1 to 10 do // *цикл повторится 10 раз*

for i :=1 to n do s := s+i; // *цикл повторится n раз*
for i := 1 to 1 do // *цикл повторится 1 раз*
for i := 1 to -10 do // *цикл не повторится ни разу*
for i := 'a' to 'z' do // *цикл повторится 26 раз*
for i := 1 downto -10 do // *цикл повторится 12 раз*

Примечание

Если между begin и end находится только один оператор, то логические операторные скобки begin и end можно не писать.

Блок-схема, соответствующая оператору For...to, представлена на рисунке 2.6. Если начальное значение счётчика цикла For...to больше конечного значения, то последовательность операторов между begin и end не будет выполнена ни разу. Кроме того, после каждого выполнения операторов тела цикла счётчик цикла увеличивается автоматически.

Переменную-счётчик можно использовать внутри цикла (но ни в коем случае не изменять). Однако после окончания выполнения оператора for значение управляющей переменной не определено. Например, в результате выполнения следующих инструкций:

tab1: = '' ;
for i := l to 5 do
tab1 := tab1 + IntToStr(i)+' '+IntToStr(i*i)+chr(13);

строковая переменная *tab1* будет содержать изображения таблицы квадратов чисел, а переменная *i* НЕ будет равна 5 - она станет «не определена».

Задача 1: Вычислить сумму первых N элементов ряда:

$$1 + \frac{1}{2} + \frac{1}{3} + ...$$

(значение *i*-го элемента ряда связано с его номером формулой *1/i*).

Program Summa;

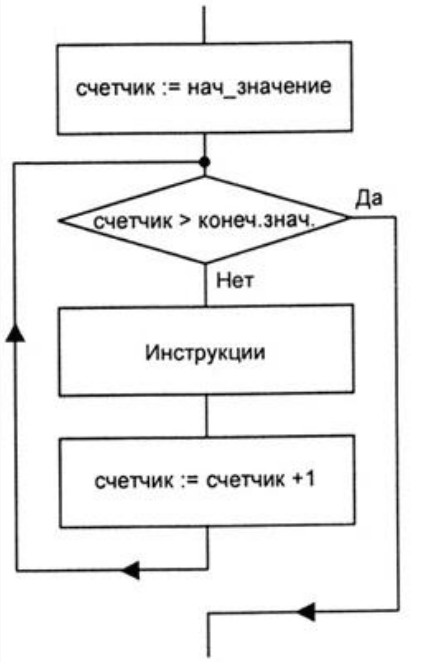


Рис. 2.6: Блок-схема оператора For...to

```
var
   n,i : integer;
   a,s : real;
begin
   ReadLn(n);
   S := 0;
   for i := 1 to n do
     begin
```

```
        a := 1/i;
        s := s + a;
        WriteLn(i, ': ', a);
      end;
      WriteLn( 'Сумма равна', s);
end;
```

Задача 2: Вычислить значение суммы ряда

$$S = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

если x изменяется в диапазоне $[0.1; 1]$, количество слагаемых задаёт пользователь. Для проверки правильности вычисления: точное значение суммы ряда: $y = \sin x$.

Заметим, что каждое последующее слагаемое каким-то образом зависит от предыдущего: $a_{n+1} = a_n * R$. Выясним, на какое значение нужно умножить слагаемое, чтобы получить следующее за ним: $R = a_{n+1}/a_n$. Это выражение называется **рекуррентным**, а формула $a_{n+1} = a_n * R$ – **рекуррентной формулой**.

```
Program Summa1;
var
  n, i : integer;
  a, s, x : real;
begin
  ReadLn(n);
  ReadLn(x);
  a := x;
  s := 0;
  for i := 0 to n do
    begin
      s := s + a;
      a := - a*x*x/(2*i+2)/(2*i+3);
```

```
    end;
    WriteLn( 'Сумма равна', s);
    WriteLn('Сумма равна', sin(x));
end;
```

Циклы, в которых выполняются арифметические вычисления некоторое заранее известное число раз, называют арифметическими циклами или A-циклами.

2.4 Оператор цикла While

Оператор (цикл) **While** используется в том случае, если некоторую последовательность операторов (инструкций программы) надо выполнить несколько раз, причём необходимое число повторений во время разработки программы неизвестно и может изменяться во время работы программы.

Типичными примерами использования цикла **While** являются вычисления с заданной точностью, поиск в массиве или в файле.

В общем виде оператор **While** записывается следующим образом (**синтаксис оператора While**):

```
While условие do
begin
    // операторы: тело цикла
end
```

где *условие* — выражение логического типа, определяющее условие выполнения оператора цикла.

Оператор **While** выполняется следующим образом (**семантика операторaWhile**):

1. Сначала вычисляется значение выражения *условие*.

2. Если значение выражения *условие* равно False (условие не выполняется), то на этом выполнение оператора While завершается.

3. Если значение выражения *условие* равно True (условие выполняется), то выполняются расположенные между begin и end операторы тела цикла. После этого снова проверяется выполнение *условия*. Если *условие* выполняется, то операторы тела цикла выполняются ещё раз. И так до тех пор, пока *условие* не станет ложным (False).

Блок-схема, соответствующая оператору While, представлена на рисунке 2.7. Цикл While может не выполниться **ни разу**, если *условие* изначально равно False. Цикл While является циклом **с заранее неизвестным числом повторений**, потому что количество повторений может зависеть от условий, изменяющихся непосредственно в теле цикла. Также он является циклом **с предусловием**, так как условие продолжения цикла проверяется перед телом цикла.

Для того чтобы тело цикла while, которые находятся между begin и end, было выполнено **хотя бы один раз**, необходимо, чтобы перед выполнением цикла while значение выражения условие было истинно.

Для того, чтобы цикл завершился, нужно, чтобы последовательность операторов между begin и end влияла на значение выражения *условие* (изменяла значения переменных, входящих в выражение *условие*).

Задача: Рассмотрим программу, которая вычисляет значение числа π с точностью, задаваемой пользователем во время работы программы (рисунок 2.8). В основе алгоритма вычисления лежит тот факт, что сумма ряда $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \ldots$ приближается к значению $\pi/4$ при достаточно большом количестве членов ряда.

Каждый член ряда с номером n вычисляется по формуле: $\frac{1}{2n-1}$ и умножается на минус один, если i чётное (определить, является ли i чётным, можно проверкой остатка от деления i на 2). Вычисление заканчивается тогда, когда значение очередного члена ряда становится меньше, чем заданная точность вычислений.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  pi : real; // вычисляемое значение π
```

Рис. 2.7: Блок-схема оператора While

```
  exp : real; // точность вычисления
  i : integer; // номер члена ряда
  elem : real; // значение члена ряда
begin
  pi := 0;
  i := 1;
  exp := StrToFloat(editl.text) ;
  elem := 1; // первое слагаемое равно 1
  While elem >= exp do // пока слагаемое больше точности
    begin
      elem := 1 / (2*i - 1) ;
      if i mod 2 = 0 // если слагаемое чётное
```

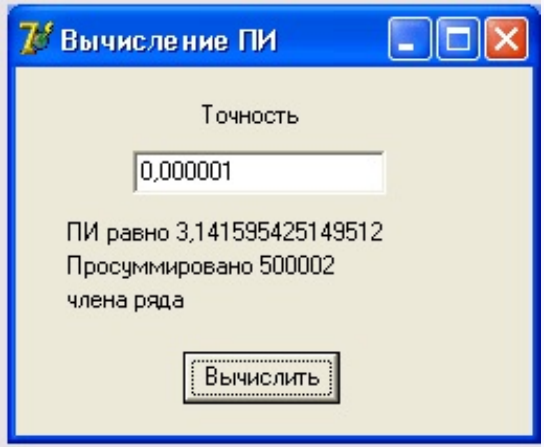


Рис. 2.8: Пример формы в Delphi для вычисления числа π

```
      then pi := pi - elem // то слагаемое берётся со знаком минус
      else pi := pi + elem; // иначе слагаемое берётся со знаком плюс
    i := i + 1;
  end;
pi: = pi * 4;
Label1.Caption:= 'ПИ равно '+ FloatToStr(pi) + # 13+ 'Просуммировано '
+IntTostr(i)+' членов ряда';
end;
```

В приведённой программе вычисления осуществляются до тех пор, пока значение *elem* не станет меньше заданной точности. При этом *elem* изменяется в теле цикла монотонно (в данном случае – монотонно уменьшается). Такие циклы называются циклами с Контролем за Монотонной Величиной или КМВ-циклами. Очевидно, что реализовать КМВ-циклы можно только с помощью с помощью операторов циклов

с заранее неизвестным числом повторений.

2.5 Оператор цикла Repeat

Оператор цикла **Repeat**, как и оператор While, используется в программе в том случае, если необходимо выполнить повторные вычисления (организовать цикл), но число повторений во время разработки программы неизвестно и может быть определено только во время работы программы, т. е. определяется ходом вычислений.

В общем виде оператор **Repeat** записывается следующим образом (**синтаксис оператора цикла Repeat**):

Repeat

// операторы: тело цикла

until условие

где *условие* — выражение логического типа, определяющее условие завершения цикла.

Цикл **Repeat** выполняется следующим образом (**семантика оператора цикла Repeat**):

1. Сначала выполняются находящиеся между repeat и until операторы тела цикла.

2. Затем вычисляется значение выражения *условие*. Если условие ложно (значение выражения условие равно False), то инструкции тела цикла выполняются ещё раз.

3. Если *условие* истинно (значение выражения условие равно True), то выполнение цикла прекращается (блок-схему см. на рисунке 2.9).

Таким образом, тело цикла, находящееся между Repeat и unti1, выполняется до тех пор, пока *условие* ложно (значение выражения условие равно False).

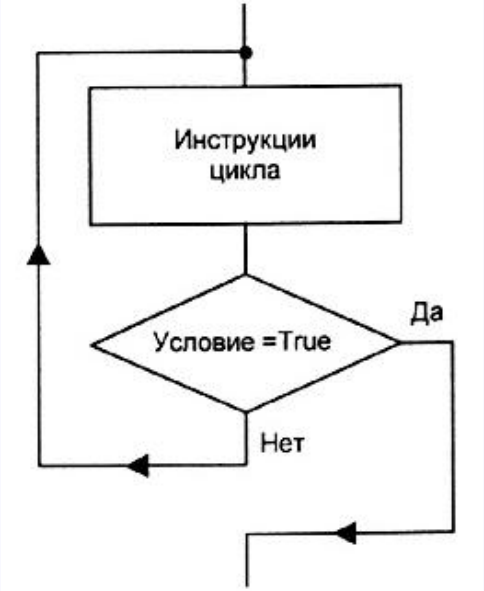


Рис. 2.9: Блок-схема оператора цикла Repeat...until

Операторы тела цикла, находящиеся между Repeat и until, выполняются **как минимум один раз**. Для того чтобы цикл завершился, необходимо, чтобы операторы тела цикла, располагающиеся между repeat и until, изменяли значения переменных, входящих в выражение *условие*. Цикл Repeat...until является циклом **с заранее неизвестным числом повторений**, потому что количество повторений может зависеть от условий, изменяющихся непосредственно в теле цикла. Также он является циклом **с постусловием**, так как условие продолжения цикла проверяется после выполнения тела цикла.

Задача 1: В качестве примера использования оператора Repeat рассмотрим программу, которая проверяет, является ли введённое пользователем число простым

(как известно, число называется простым, если оно делится только на единицу и само на себя). Например, число 21 — обычное (делится на 3), а число 17 — простое (делится только на 1 и на 17). Пример формы - на рисунке 2.10.

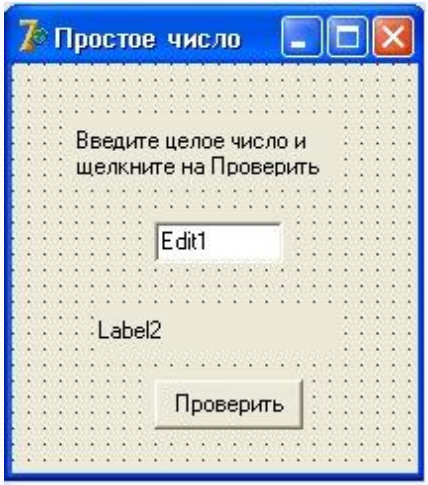


Рис. 2.10: Форма в Delphi для проверки, является ли число простым

Проверить, является ли число N простым, можно делением числа N на два, на три и т. д. до N и проверкой остатка после каждого деления. Если после очередного деления остаток равен нулю, то это означает, что найдено число, на которое N делится без остатка. Сравнив N и число, на которое N разделилось без остатка, можно определить, является ли N простым числом.

```
procedure TForm1.ButtonlClick(Sender: TObject) ;
var
  N: integer; // проверяемое число
  d: integer; // делитель
  r: integer; // остаток от деления n на d
```

```
begin
  N := StrToInt(Editl.text);
  d := 2; // сначала будем делить на два
  repeat
    r := N mod d;
    if r <> 0 // n не разделилось нацело на d
      then d := d + 1;
  until r = 0; // найдено число, на которое n разделилось без остатка
  if d = N
    then label2.caption := Edit1.text + ' — простое число.'
    else label2.caption := Edit1.text + ' — составное число.';
end;
```

Пройти тест для проверки своих знаний по пройденному материалу.

РАЗДЕЛ 3
Подпрограммы

3.1 Структура подпрограммы

В языке Delphi основной программной единицей является подпрограмма. Подпрограммы делятся на **процедуры** и **функции**.

Функциями называют такие подпрограммы, которые при своем выполнении производят какие-либо вычисления и соответственно *возвращают* какое-то значение (вычисляет и возвращает синус, длину строки, модуль и проч.).

Процедурами называют подпрограммы, которые выполняют некоторые *действия* (считывает данные с клавиатуры, удаляет символ из строки и проч.).

Описание подпрограммы состоит из трёх частей: *заголовка подпрограммы, локального описания и тела подпрограммы*. Описание подпрограммы — это только

описание, которое никогда не выполняется само по себе и может располагаться в любом месте исходного текста (но обязательно до первого вызова подпрограммы).

Заголовок используется, чтобы явно ввести в программу новую подпрограмму и обозначить начало её описания.

Локальные описания представляют собой набор описаний типов, переменных, констант и других подпрограмм, которые действуют только в рамках данной подпрограммы.

Тело подпрограммы —- это логический блок begin…end, содержащий операторы и команды языка программирования и реализующий нужную логику работы.

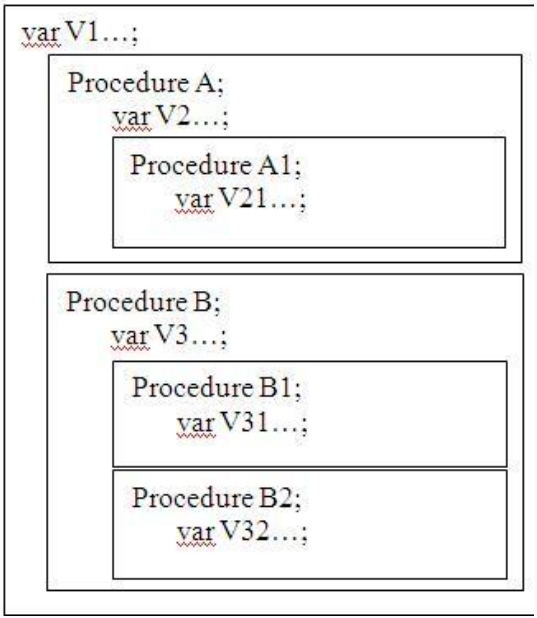


Рис. 3.1: Схема взаимодействия подпрограмм

Вызов подпрограммы осуществляется с помощью указания **имени** подпрограммы и конкретных значений для каждого из её параметров.

Когда в тексте программы указывается имя ранее описанной подпрограммы с фактическими параметрами, то выполнение основной части программы останавливается и управление передаётся подпрограмме, до тех пор, пока в ходе работы не будет достигнут её конец (зарезервированное слово end). После этого управление передается обратно в программу (или другую подпрограмму), вызывавшую данную подпрограмму.

При описании нескольких подпрограмм из более «низкой» по расположению подпрограммы можно вызвать более «высокую», а также обратиться к её переменным.

Возможно **опережающее** описание. Оно заключается в том, что объявляется лишь заголовок подпрограммы B, а её тело заменяется стандартной директивой **Forward**. Тогда в более «высокой» подпрограмме A можно использовать обращение к подпрограмме B – ведь она уже описана, точнее, известны её формальные параметры, и компилятор может правильным образом организовать её вызов. При этом тело подпрограммы B начинается заголовком, в котором уже не указываются описанные ранее формальные параметры.

```
Procedure B (j ; byte); forward;
Procedure A (i ; byte);
begin
B(i);
end;
Procedure B;
begin
A(j);
end;
```

Заголовок процедуры состоит из слова **procedure**, за которым следует имя процедуры, которое используется для вызова процедуры, активизации её выполнения. Если у процедуры есть параметры, то они указываются после имени процедуры, в

скобках. Завершается заголовок процедуры символом «точка с запятой».

Procedure ИмяПроцедуры (СписокПараметров);

За разделом объявления переменных расположен раздел инструкций begin. . . end, за которым следует символ «точка с запятой». В разделе инструкций находятся исполняемые инструкции подпрограммы.

Заголовок функции состоит из слова **Function**, имени функции, которое также используется для её вызова, списка параметров. В конце указывается тип результата, который возвращает функция. Этот тип называют **типом функции**.

Function ИмяФункции (СписокПараметров): ТипФункции;

Отличие процедуры от функции заключается в том, что результатом исполнения операторов, образующих тело функции, всегда является некоторое *значение*, поэтому обращение к функции можно использовать в соответствующих выражениях наряду с переменными и константами. Тип – это тип возвращаемого функцией результата.

Кроме того, любая функция имеет внутреннюю переменную **Result** того же типа, что и тип функции. Присваивание этой переменной значения трактуется как указание результата, возвращаемого функцией. Также для возвращения результата функции в левой части оператора присваивания может использоваться её **имя**. Использование имени функции в правой части оператора присваивания внутри самой функции означает **рекурсивный** вызов функции.

3.2 Типы параметров

Переменные можно разделить на **локальные** и **глобальные**. Переменные, объявляемые в процедурах и функциях, являются локальными. Они существуют только во время выполнения соответствующей процедуры или функции. Т.е. память для них выделяется только при вызове соответствующей процедуры или функции и освобождается при возврате в вызвавшую процедуру. Переменные, объявленные вне процедур или функций, являются глобальными, и доступны как из тела программы,

так и из любой её подпрограммы.

При написании программы используются параметры, воспринимаемые как **формальные** параметры. При вызове подпрограмм в них передаются некоторые переменные, константы, выражения. Они являются **фактическими** параметрами. При вызове подпрограммы формальные параметры заменяются фактическими. Их количество, типы, а также последовательность должны совпадать.

Задача: Создать функцию с двумя параметрами A и B, которая будет возвращать результат возведения A в степень B (рисунок 3.2).

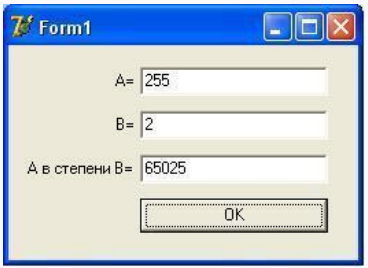


Рис. 3.2: Пример формы для вычисления A в степени B

```
function power(A,B:real):real;
begin
    result:=exp(B*ln(A)); //присваивание функции вычисленного значения
end;

procedure TForm1.Button1Click(Sender: TObject);
var X, Y : real;
begin
     X := StrToFloat(edit1.text);
```

```
    Y := StrToFloat(edit2.text);
    edit3.Text := FloatToStr(power(X,Y)); //вызов функции power
end;
```

В примере A и B - формальные параметры, X и Y - фактические параметры.

3.3 Рекурсия

Рекурсия – это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих её операторов обращается сама к себе.

Рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (**простая рекурсия**) или через другие функции (**сложная** или **косвенная рекурсия**). Например, функция A вызывает функцию B, а функция B — функцию A. Количество вложенных вызовов функции или процедуры называется **глубиной рекурсии**. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

При выполнении правильно организованной рекурсивной подпрограммы осуществляется многократный переход от некоторого текущего уровня организации алгоритма к нижнему уровню последовательно до тех пор, пока, наконец, не будет получено **тривиальное решение** поставленной задачи.

Структурно рекурсивная функция на верхнем уровне всегда представляет собой команду **ветвления** (выбор одной из двух или более альтернатив в зависимости от условия (условий), которое в данном случае уместно назвать «условием прекращения рекурсии»), имеющую две или более альтернативные ветви, из которых хотя бы одна является **рекурсивной** и хотя бы одна — **тривиальной**. Рекурсивная ветвь выполняется, когда условие прекращения рекурсии ложно, и содержит хотя бы один рекурсивный вызов — прямой или опосредованный вызов функцией самой себя. Тривиальная ветвь выполняется, когда условие прекращения рекурсии истинно;

она возвращает некоторое значение, не выполняя рекурсивного вызова. Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

Рекурсивная форма организации алгоритма обычно выглядит изящнее итерационной и даёт более компактный текст программы, но при выполнении медленнее и может вызвать **переполнение стека** (при каждом входе в подпрограмму её локальные переменные размещаются в организованной особым образом области памяти, называемой программным стеком). Теоретически, любую рекурсивную функцию можно заменить циклом и стеком.

Задача: *Вычислить факториал числа N, используя рекурсию.*

```
function factorial ( i : word ) : extended;
begin
    if i = 0
      then result := 1
      else result := i * factorial(i-1); //рекурсивный вызов функцией самой себя
end;

procedure TForm1.Button1Click(Sender: TObject);
      var N : word;
begin
      N := StrToInt(edit1.Text);
      edit2.Text := FloatToStr(factotial(N));
end;
```

В нашем случае решение при i = 0 **тривиально** (факториал 0 равен 1) и используется для остановки рекурсии.

РАЗДЕЛ 4
Структурированные типы данных. Символьный и строковый типы данных

4.1 Символьный тип

Значениями символьного типа является множество всех символов клавиатуры. Каждому символу приписывается целое число в диапазоне 0..255. Это число служит кодом внутреннего представления символа. Все 256 символов размещены в кодировочной таблице.

В Windows в основном используется кодировка, которая называется ANSI. Разновидность набора ANSI, содержащая символы русского алфавита, называется Windows-1251.

Краткая историческая справка:

ASCII (читается аски) - это первая кодировка, применявшаяся ещё в пору, когда 99% юзеров SO ещё даже не родились (1963 год). Кодировка 7-битная, то есть определено 128 символов, 8-й бит полного байта использовался для проверки чётности, поскольку в то время каналы были ненадежные, то предполагалось, что будет проверяться каждый полученный байт.

Далее со временем стало понятно, что для других языков можно использовать 8-й бит для отображения национальных символов - то есть использовать 256 символов. Эту расширенную 8-битовую кодировку условно называют ANSI (читается анси) по названию американского института стандартов, в рамках которого и была предложена 8-битовая кодировка. Соответственно, для каждого национального языка была предложена своя раскладка второй половины таблицы (от 128 до 255 символа), а первая половина таблицы от 0 до 127 - изначальные символы ASCII. KOI-8, CP-1251, 1252 и проч. - это различные инкарнации ANSI.

Далее, когда дело дошло до иероглифов, стало понятно, что в 256 символов не уместиться и появилась UNICODE (читается юникод) - где на 1 символ отводится 2 байта, то есть 65536 символов, где таблица была жёстко поделена между национальными символами, например таблица ASCII осталась в интервале U+0000 до U+007F, а кириллица в интервале U+A640 до U+A69F и т..д.

С нарастанием угара стало ясно что 65536 символов также не хватает, потому что появились эмодзи, стали поднимать голову другие национальные символы, справедливо указывавшие на нехватку места в таблице UNICODE, тогда был предложен UTF-8 (читается ютиэф 8), где количество байтов в символе имеет разную длину и может быть от 1 до 4 байт, что даёт 1 112 064 символов.

Dec	Символ	Dec	Символ	Dec	Символ	Dec	Символ	Dec	Символ	Dec	Символ	Dec	Символ	Dec	Символ
0	спец. NOP	32	спец. SP (Пробел)	64	@	96	`	128	Ђ	160		192	А	224	а
1	спец. SOH	33	!	65	A	97	a	129	Ѓ	161	Ў	193	Б	225	б
2	спец. STX	34	"	66	B	98	b	130	‚	162	ў	194	В	226	в
3	спец. ETX	35	#	67	C	99	c	131	ѓ	163	Ј	195	Г	227	г
4	спец. EOT	36	$	68	D	100	d	132	„	164	¤	196	Д	228	д
5	спец. ENQ	37	%	69	E	101	e	133	…	165	Ґ	197	Е	229	е
6	спец. ACK	38	&	70	F	102	f	134	†	166	¦	198	Ж	230	ж
7	спец. BEL	39	'	71	G	103	g	135	‡	167	§	199	З	231	з
8	спец. BS	40	(	72	H	104	h	136	€	168	Ё	200	И	232	и
9	спец. Tab	41	)	73	I	105	i	137	‰	169	©	201	Й	233	й
10	спец. LF	42	*	74	J	106	j	138	Љ	170	Є	202	К	234	к
11	спец. VT	43	+	75	K	107	k	139	‹	171	«	203	Л	235	л
12	спец. FF	44	,	76	L	108	l	140	Њ	172	¬	204	М	236	м
13	спец. CR	45	-	77	M	109	m	141	Ќ	173		205	Н	237	н
14	спец. SO	46	.	78	N	110	n	142	Ћ	174	®	206	О	238	о
15	спец. SI	47	/	79	O	111	o	143	Џ	175	Ї	207	П	239	п
16	спец. DLE	48	0	80	P	112	p	144	ђ	176	°	208	Р	240	р
17	спец. DC1	49	1	81	Q	113	q	145	‘	177	±	209	С	241	с
18	спец. DC2	50	2	82	R	114	r	146	’	178	І	210	Т	242	т
19	спец. DC3	51	3	83	S	115	s	147	“	179	і	211	У	243	у
20	спец. DC4	52	4	84	T	116	t	148	”	180	ґ	212	Ф	244	ф
21	спец. NAK	53	5	85	U	117	u	149	•	181	µ	213	Х	245	х
22	спец. SYN	54	6	86	V	118	v	150	–	182	¶	214	Ц	246	ц
23	спец. ETB	55	7	87	W	119	w	151	—	183	·	215	Ч	247	ч
24	спец. CAN	56	8	88	X	120	x	152	[illegible]	184	ё	216	Ш	248	ш
25	спец. EM	57	9	89	Y	121	y	153	™	185	№	217	Щ	249	щ
26	спец. SUB	58	:	90	Z	122	z	154	љ	186	є	218	Ъ	250	ъ
27	спец. ESC	59	;	91	[	123	{	155	›	187	»	219	Ы	251	ы
28	спец. FS	60	<	92	\	124	|	156	њ	188	ј	220	Ь	252	ь
29	спец. GS	61	=	93	]	125	}	157	ќ	189	S	221	Э	253	э
30	спец. RS	62	>	94	^	126	~	158	ћ	190	s	222	Ю	254	ю
31	спец. US	63	?	95	_	127		159	џ	191	ї	223	Я	255	я

Рис. 4.1: Сводная таблица кодов ASCII (Win-1251)

Чтобы хранить и обрабатывать символы, используют следующие типы данных:

- **AnsiChar** представляется в виде некоторого набора ANSI-символов, который содержит в себе символы, кодирующиеся одним байтом (байт – восьмиразрядное двоичное число).

- **WideChar** соответствует набор символов с кодировкой Unicode, который включает в себя символы, кодирующиеся двумя байтами.

- **Char** - эквивалентен типу AnsiChar. Введен, чтобы обеспечить совместимость с предыдущими версиями.

Символьные типы относятся к порядковым типам, т.к. значения в типе расположены в определённом порядке, пронумерованы, а кроме того, можно однозначно определить последующий и предыдущий элемент типа (в отличие от, например, вещественных типов, где это сделать однозначно невозможно). Символьные константы заключаются в одинарные кавычки: ‘Д’, ‘g’, ‘7’.

Все символы упорядочены, т.к. имеют свой личный номер. Важно, что соблюдаются следующие отношения:

’A’ < ’B’ < ’C’ < ... < ’X’ < ’Y’ < ’Z’

’0’ < ’1’ < ’2’ < ... < ’7’ < ’8’ < ’9’

К типу Char применимы операции сравнения (сравниваются не сами символы, а их коды в кодировочной таблице: меньше тот символ, код которого в кодировочной таблице меньше, т.е. который расположен левее), а также встроенные функции:

Имя функции	Возвращаемое значение
Ord	Порядковый номер элемента для перечислимых типов, код ASCII для типа Char
Chr	Символ (тип Char), преобразованный из числового аргумента chr(83) = 'S', chr(116) = 't'
Pred	Предыдущее по порядку значение данного типа. Например, значение Pred('5') = '4'
Succ	Следующее по порядку значение данного типа. Например, значение Succ('5') = '6'

4.2 Строковый тип

Строка, она же текст – это набор символов, любая их последовательность. Соответственно, один символ – это тоже строка, тоже текст. Текстовая строка имеет определённую длину. Длина строки – это количество символов, которые она содержит. Для обработки строк используются следующие типы:

1. **ShortString** (короткая строка) представляет собой статически размещаемые в памяти компьютера строки длиной от 0 до 255 символов. Эквивалент: String[N], где N <= 255.

2. Длинная строка **String** = **AnsiString**. При работе с этим типом память выделяется по мере надобности (динамически) и ограничена имеющейся в распоряжении программы доступной памятью.

3. **WideString** (широкая строка) представляет собой динамически размещаемые в памяти строки, длина которых ограничена только объёмом свободной памяти. Каждый символ строки типа WideString занимает не один, а два байта. Введён для совместимости с компонентами, основывающимися на OLE-технологии..

4. Заканчивающаяся нулём строка **PChar**.

WideString используется для представления строк в кодировке UNICODE использующей 16-ти битное представление каждого символа (WideChar). Эта кодировка используется в тех случаях, когда необходима возможность одновременного присутствия в одной строке символов из двух и более языков (помимо английского).

При описании короткой строки String[N] в памяти под неё выделяется N+1 байт, первый байт содержит текущую длину строки, а сами символы располагаются, начиная со второго.

Примеры объявления переменных строковых типов:

```
var
  S: String[250];          // короткая строка длиной до 250 символов
  ssMax: ShortString;      // короткая строка длиной до 255 символов
  stS: String;             // длинная строка
  swS: WideString;         // широкая строка
  psS: PChar;              // ссылка на заканчивающуюся нулём строку
  acS: array [0..1000] of Char;  // заканчивающаяся нулём строка длиной до
                                 1000 символов
```

Строковые константы заключаются в одинарные кавычки: ‘строка символов’ ‘Hello!’ ‘2,4’ ‘5a’. Здесь следует обратить внимание на строковую константу ’2.4’. Это именно строковая константа, т. е. строка символов, которая изображает строку «две целые четыре десятых», а не число 2,4.

В значение строковой константы можно включать как печатаемые, так и **непечатаемые** символы, указывая после символа «#» десятичное или шестнадцатеричное число от 0 до 255, соответствующее коду ASCII нужного символа. Например, обозначение #13 соответствует символу перевода строки. В записи константы можно чередовать записи в кавычках и записи с символом #. Например, константа ’строка 1’ + #13 + ’строка 2’ соответствует двум строкам:

```
строка 1
строка 2
```

Строковый тип определяет участок памяти переменной длины, каждый байт которого содержит один символ. Таким образом, тип **String** – это цепочка элементов типа Char. Каждый символ типа **String** пронумерован, начиная с единицы. Соответственно, программист может обращаться к любому элементу строки (символу) по его номеру в строке:

```
var
S1, S2: String;
...
S1 := 'Hello'; // S1[3] = 'l'
S2 := 'students'; // S2[5] = 'e'
S2[1] := 'S'; // S2 = 'Students'
```

4.2.1 Операции над строками

- Строки Delphi поддерживают операцию **конкатенации** «+» (сцепления, соединения, присоединения). Несмотря на «научное» название, операция конкатенации выполняет очень простую процедуру. С помощью операции конкатенации одна строка присоединяется к другой: S1 + ' ' + S2 = 'Hello Students'. Если длина строки превысит максимально допустимую длину N короткой строки, то «лишние символы» отбрасываются.

- Над строками допустимо использование **операций отношения**: = <> > < >= <=.

Строки сравниваются посимвольно, слева направо с учётом внутренней кодировки символов. Операции отношения (=, <>, >, <, >=, <=) проводят сравнение двух строк слева направо до первого несовпадающего символа, и та строка считается больше, в которой первый несовпадающий символ имеет больший номер в кодировочной таблице. Результат выполнения операций отношения над строками всегда имеет логический тип.

Например, выражение 'MS-DOS' < 'MS-Dos' имеет значение **True**, т.к. первые несовпадающие символы 'O' и 'o' при сравнении операцией «<» дают результат

True.

'MS-DOS' < 'Dos' имеет значение **False**, т.к. первые несовпадающие символы 'M' и 'D' при сравнении операцией «<» дают результат **False**: код символа 'M' не меньше кода символа 'D'.

Строки считаются **равными**, если они совпадают по длине и содержат одни и те же символы на соответствующих местах в строке:

'MS-DOS' = 'MS-DOS' - результат **True**;

'MS-DOS' <> 'MS-DOS' - результат **False**;

'MS-DOS' = 'MS-Dos' - результат **False**;

'MS-DOS' = 'MS-D' - результат **False**.

Стандартные процедуры и функции для работы со строками

AnsiLowerCase(S) : String	Возвращает исходную строку S, в которой все прописные буквы заменены строчными в соответствии с национальной кодировкой
AnsiUpperCase(S) : String	Возвращает исходную строку S, в которой все строчные буквы заменены прописными в соответствии с национальной кодировкой
LowerCase(S) : String	Возвращает исходную строку S, в которой все латинские прописные буквы заменены строчными
UpperCase(S) : String	Возвращает исходную строку S, в которой все латинские строчные буквы заменены прописными
Length(S) : Integer	Возвращает длину строки S: length(S2) = 8, length('привет') = 6
Concat(S1, S2 . . .) : String	Возвращает строку, представляющую собой сцепление строк-параметров S1, S2, . . .
Copy(S, I, N) : String	Из строки S копирует N символов, начиная с символа с номером I
Delete(S, I, N)	Из строки S удаляет N символов, начиная с символа с номером I
Insert(St, S, I)	Вставляет подстроку St в строку S, начиная с символа с номером I
Pos(St, S) : Integer	Отыскивает в строке S первое вхождение подстроки St и возвращает номер позиции, с которой она начинается. Если подстрока не найдена, возвращает нуль
StringOfChar(C, N) : String	Создаёт и возвращает строку, состоящую из N раз повторенного символа C

В модуле **StrUtils.pas** содержатся полезные функции для обработки строковых переменных. Чтобы подключить этот модуль к программе, нужно добавить его имя (**StrUtils**) в раздел **Uses**.

Задача: Дана строковая величина S. Подсчитать количество слов в S, начинающихся с символа X.

```
Program Slova;
var
   S : String;
   X : Char;
   i, kol : integer;
begin
   ReadLn(S);
   S := ' ' + S; //добавляем в начале строки пробел
   ReadLn(X); //забираем в переменную X только первый символ
   kol := 0;
   for i := 1 to length(S)-1 do //просматриваем до предпоследнего символа
       if (S[i] = ' ') and (S[i+1] = X) //если текущий символ равен пробелу, а следующий равен X
       then kol := kol +1; //то количество нужных слов увеличиваем на 1
     WriteLn(kol); //вывод результата
end;
```

Пройти тест для проверки своих знаний по пройденному материалу.

РАЗДЕЛ 5
Массивы

Рассмотренные выше простые типы данных позволяют использовать в программе одиночные объекты – различные числа, строки, символы. Язык **Delphi** также позволяет использовать объекты, которые содержат множество **однотипных** объектов

(чисел, символов, строк). Такие объекты называются массивами (**Array**).

Массив - это упорядоченная последовательность данных одного типа, рассматриваемых как единое целое.

Доступ к элементам массива осуществляется по индексу (порядковому номеру). В качестве данных в массивах могут храниться элементы числового, строкового и других типов, кроме файлового.

При описании массива необходимо указать его имя, общее количество входящих в массив элементов и тип элементов. Описание массива задаётся следующим образом:

<Имя массива> : **Array**[*<Список индексных типов>*] **of** *<Тип элементов массива>*;

<Список индексных типов> – список одного или нескольких индексных типов, которые определяют индексы элементов. При описании одного индексного типа получается линейный массив, при описании нескольких – многомерный. В качестве индексных типов можно использовать любые порядковые типы, имеющие мощность не более 2 Гбайт, т.е. кроме **LongWord** и **Int64**. Обычно в качестве индексного типа используется тип-диапазон. При обращении к элементу массива его индекс не должен выходить за границы индексного типа.

<Тип элементов массива> - любой тип **Delphi**, кроме файлового, в том числе и другой массив:

Mat : **Array** [0..5] **of Array** [-2..2] **of Array [Char] of Byte**;

Такую запись можно заменить более компактной:

Mat : **Array** [0..5, -2..2, **Char**] **of Byte**;

Например, следующий фрагмент кода создаёт три массива соответствующих типов.

```
var
  massiv_a : array [0..100] of string;
  massiv_b : array [1..5] of char;
  massiv_c : array [-10..10] of integer;
```

Можно сначала создать тип-массив, а затем описать переменные этого типа:

```
type
  Massiv = array [0..100] of string;
var
  A, B, C : Massiv;
```

5.1 Операции с массивами

Чтобы получить доступ к массиву (элементам массива), необходимо в программе указать идентификатор (имя) массива и затем в квадратных скобках номер элемента, к которому производится обращение.

```
var
  massiv_a : array [0..100] of string;
  massiv_b : array [1..5] of char;
  massiv_c : array [-10..10] of integer;
begin
  massiv_a[15]:='Пример использования массивов';
  massiv_b[1]:='S';
  massive_c[-5]:=24;
end;
```

При использовании массивов помните, что использование элементов массива, индекс которых выходит за пределы объявленного диапазона, недопустимо.

В **Delphi** можно одним оператором присваивания передать все элементы одного массива другому массиву того же типа:

```
  a, b : array [0..10] of integer;
                 ...
  a := b;
```

Однако следующее объявление создаст разные типы массивов:

```
  a : array [0..10] of integer;
  b : array [0..10] of integer;
```

Поэтому использование оператора присваивания a := b в данном случае приведет к ошибке. Передать элементы одного массива другому массиву в таком случае можно поэлементно:

```
for i := 0 to 10 do a[i] := b[i];
```

Обращение к элементам многомерных массивов осуществляется аналогично обращению к элементам в линейном массиве: индексы перечисляются через запятую. Например, A2[4,3] — значение элемента массива A2, лежащего на пересечении четвертой строки и третьего столбца.

Delphi имеет три основных **типа** массивов:

1. **Статические** массивы. Они определены установленными, неизменяемыми размерами. Они могут быть одномерными или многомерными - последний является массивом массивов (массивов и т.д). Размер и диапазон такого многомерного массива всегда даются для самого высокого, крайнего левого массива - родительского массива.

2. **Динамические** массивы. Динамические массивы не имеют никакой предраспределенной памяти. Определяется только когда создан указатель. Размеры таких массивов должны быть установлены прежде, чем они будут использоваться.

3. **Открытые** массивы.

```
// описание прямоугольного массива с 5 столбцами и 10 строчками
  a : array [1..10, 1..5] of integer;
// заполнение массива с 5 столбцами и 10 строчками построчно
  for i := 1 to 10 do
    for j := 1 to 5 do a[i,j] := Random(100);
```

5.2 Динамические массивы

Есть ещё один тип массива – **динамический массив** (**Dynamic array**). Его описание очень похоже на описание обычного, статического массива, однако в нём отсутствует указание границ индексных типов массива:

var

dyn_massiv1 : **array of integer**; //*одномерный массив целых чисел*

dyn_massiv2 : **array of array of integer**; //*двумерный массив целых чисел*

Динамические массивы не имеют никакой предраспределенной памяти. Определяется только когда создан указатель. Размеры таких массивов должны быть установлены прежде, чем они будут использоваться. Например :

SetLength(dynArray, 5);

устанавливает размер одномерного массива dynArray в 5 элементов. При этом будет распределена память.

Все динамические массивы начинаются с индекса = 0.

Индивидуальные подмассивы многомерного динамического массива могут иметь различные измерения – они, конечно, являются отдельными массивами. После одной такой операции **SetLength**, на элементы набора массива уже можно ссылаться, даже при том, что остальная часть массива неопределена.

Правила работы с ним точно такие же, как и с обычным массивом, то есть обращение к его элементам производится по индексу, а выход за границу массива приводит к возникновению ошибки. При этом все индексы начинают нумерацию с нуля.

Распределение памяти и указание границ индексов по каждому измерению динамических массивов (если задаётся многомерный массив) осуществляется в ходе выполнения программы путём инициализации массива с помощью процедуры **SetLength. SetLength(A, 3)** – инициализация одномерного динамического массива, количество элементов равно 3. Для инициализации многомерных массивов сначала устанавливается длина его первого измерения, затем второго и т.д.

B : **array of array of integer**;

SetLength(B, 3); //длина I измерения – количество столбцов

SetLength(B[0], 3); //длина I столбца

SetLength(B[1], 3); //длина II столбца

SetLength(B[2], 3); //длина III столбца

Можно использовать более компактную запись:

SetLength(B, 3, 3);

Как видно из приведённого примера, в Delphi можно создавать динамические массивы с разной длиной по второму и следующим измерениям (т.е. массив может быть не только квадратным, прямоугольным, но и треугольным).

Нижняя граница индексов по любому измерению динамического массива всегда равна 0. Поэтому верхней границей индексов для массива B станет 2.

Процедуры и функции для работы с массивами:

Length(A) : Integer;	Возвращает количество элементов в массиве A
Low(A) : **Integer**;	Возвращает индекс нижней границы массива A
High(A) : **Integer**	Возвращает индекс верхней границы массива A
SetLength (A, N)	Устанавливает новую длину массива A равной N, при этом сохраняет те элементы, которые были в массиве до его изменения

Надо только помнить, что при описании динамический массив всегда имеет нулевую длину. Кстати, хотя функция **Low** для любого массива вернёт индекс первого элемента, но все динамические массивы имеют индекс первого элемента, равный нулю. Рассмотрим пример использования динамического массива и функций работы с ним:

```
Program PrimerMassiva;
var
  i : integer;
  A : array of integer;
  sum : integer;
  avg : double;
  msg : string;
begin
  SetLength(A,10); // назначаем массиву A длину 10 элементов
  for i := Low(A) to High(A) do
    A[i] := Random(100)+1; // заполняем массив случ. числами от 1 до 100
```

```
  sum := 0;
  for i := Low(A) to High(A) do
    sum := sum + A[i]; // считаем сумму элементов массива
  avg := sum / Length(A); // вычисляем среднее значение элементов массива
  // формируем и выводим сообщение с результатами:
  msg := 'Массив: ';
  for i := Low(A) to High(A)-1 do
    msg := msg + IntToStr(A[i]) + ',';
  msg := msg + IntToStr(A[High(A)]) + #13;
  msg := msg + 'Сумма элементов =' + IntToStr(sum) + #13;
  msg := msg + 'Среднее значение = '+ FloatToStr(avg);
  WriteLn(msg);
end;
```

Пройти тест для проверки своих знаний по пройденному материалу.

5.3 Типовые операции с массивами

Типовыми операциями с массивами являются:

- ввод массива;
- вывод массива;
- циклические перестановки элементов массива;
- поиск максимального или минимального элемента массива;
- поиск заданного элемента массива;
- сортировка массива.

Ввод массива

Для того чтобы работать с массивом, его надо сначала заполнить. Это можно осуществить разными путями: получить от пользователя, заполнить автоматически по какому-нибудь правилу или случайным образом. В любом случае заполнение осуществляется поэлементо:

```
// описание прямоугольного массива с 10 строчками и 5 столбцами
a : array [1..10, 1..5] of integer;
// заполнение массива с 10 строчками и 5 столбцами построчно
for i := 1 to 10 do
  for j := 1 to 5 do
    a[i,j] := Random(100);
// заполнение массива с 10 строчками и 5 столбцами по столбцам
for j := 1 to 5 do
  for i := 1 to 10 do
    a[i,j] := Random(100);
```

Вывод массива

Операция вывода массива заключается в том, что на консоль осуществляется вывод всех элементов массива. Для консольного вывода элементов массива используются процедуры Write и WriteLn.

Для вывода массива поэлементно используется оператор For, параметр которого пробегает весь индексный тип, тем самым обращаясь поочередно к каждому элементу массива. Если массив многомерный, то будут использоваться соответствующее количество вложенных друг в друга циклов с разными параметрами, каждый из которых отвечает за свой индексный тип.

5.3.1 Поиск элементов массива

При любом поиске в массиве искать следует **не значение** искомого элемента, максимума, минимума и т.п., а его **индекс**. Тогда решая одну задачу, мы по сути дела **решаем сразу две**: определяем не только наличие искомого элемента, но и его местоположение в массиве.

Поиск номера минимального и номера максимального элементов:

```
imax := 1; //номер максимального элемента
imin := 1; //номер минимального элемента
for i := 2 to N do
  if a[i] < a[imin] then imin := i else
    if a[i] > a[imax] then imax := i;
```

Поиск элементов массива по заданным критериям

Примерами подобного рода задач могут служить поиск первого отрицательного, первого положительного и любого первого элемента, отвечающего некоторому условию, а также поиск единственного или определённого количества элементов, равных некоторому конкретному значению. Особенность задач этого класса в том, что нет необходимости просматривать весь массив. Просмотр можно закончить сразу, как только требуемый элемент будет найден. Но в худшем случае для поиска элемента требуется просмотреть весь массив, причём нужного элемента в нём может не оказаться.

Существует несколько методов поиска. Самый простой заключается в последовательном просмотре элементов массива. Если массив не очень большой, затраты времени линейного поиска не столь заметны. Но при солидных объёмах информации время поиска становится критичным. Поэтому существуют методы, позволяющие уменьшить время поиска, например двоичный поиск, который применяется только, если элементы массива отсортированы по возрастанию или убыванию.

Линейный поиск

Чаще всего при программировании поисковых задач используют циклы, в которых условие выхода формируется из двух условий: *первое условие* – пока искомый элемент не найден, а *второе* – пока есть элементы массива. После выхода из цикла осуществляют проверку, по какому из условий произошел выход. Такой простой перебор целесообразно использовать для неотсортированных массивов:

```
a : array[1..N] of byte;
i := 0;
repeat
   i := i + 1
until (i = N) or (a[i] = K); //K – искомый элемент
if a[i] = K then write(i)
   else write(0)
```

Поиск с барьером

Идея поиска с барьером состоит в том, чтобы не проверять каждый раз в цикле условие, связанное с границами массива. Это можно обеспечить, установив в массив так называемый *барьер*: любой элемент, который удовлетворяет условию поиска. Тем самым будет *ограничено изменение индекса*. При поиске с барьером массив также может быть неотсортирован.

Выход из цикла, в котором теперь остаётся только условие поиска, может произойти *либо* на найденном элементе, *либо* на барьере. Таким образом, после выхода из цикла проверяется, не барьер ли мы нашли? Вычислительная сложность поиска с барьером меньше, чем у линейного поиска, но также является величиной того же порядка, что и N - количество элементов массива. Существует два способа установки барьера: дополнительным элементом или вместо крайнего элемента массива.

```
a : array[0..N] of byte;
i := N;
a[0] := K;
```

```
while (a[i] <> K) do
    i := i - 1;
write(i)
```

Бинарный поиск (дихотомический поиск, поиск по бинарному дереву, метод половинного деления)

На практике довольно часто производится поиск в массиве, элементы которого *упорядочены* по некоторому критерию (такие массивы называются упорядоченными). Например, массив фамилий, как правило, упорядочен по алфавиту. Если массив упорядочен, то применяют метод бинарного поиска.

Метод (алгоритм) бинарного поиска реализуется следующим образом:

- Сначала образец сравнивается со средним (по номеру) элементом массива.
- Если образец равен среднему элементу, то задача решена.
- Если образец *больше* среднего элемента, то это значит, что искомый элемент расположен *правее* среднего элемента (при отсортированности массива по возрастанию). Отбрасываем левую часть массива, далее работаем с правой частью, как с отдельным массивом, т.е. переходим к пункту 1.
- Если образец *меньше* среднего элемента, то это значит, что искомый элемент расположен *левее* среднего элемента. Отбрасываем правую часть массива, далее работаем с левой частью, как с отдельным массивом, т.е. переходим к пункту 1.

После того, как определена часть массива, в которой может находится искомый элемент, по формуле *(НомерПервого + НомерПоследнего) /2* вычисляется новое значение номера среднего элемента и поиск продолжается.

Алгоритм бинарного поиска заканчивает свою работу, если **искомый элемент найден** или если перед выполнением очередного цикла поиска обнаруживается, что

значение *НомерПервого* **больше**, чем *НомерПоследнего* (означает, что искомого элемента в массиве нет).

```
L := 1; //L – номер «левого» элемента
R := N + 1; //R – номер «правого» элемента
while L < R do
  begin
    m := (L+R) div 2; // поиск номера среднего элемента
    if K > a[m] //K – искомый элемент
       then L := m + 1 //изменяем «левую» границу
       else R := m //изменяем «правую» границу
  end;
if a[m] = K
  then write(m) //искомый элемент на позиции m
  else write(0) //искомый элемент не найден
```

Поиск среди элементов квадратной матрицы

Двумерный массив является разновидностью многомерных. Визуально двумерный массив можно представить в виде таблицы. Положение элемента задаётся двумя индексами:

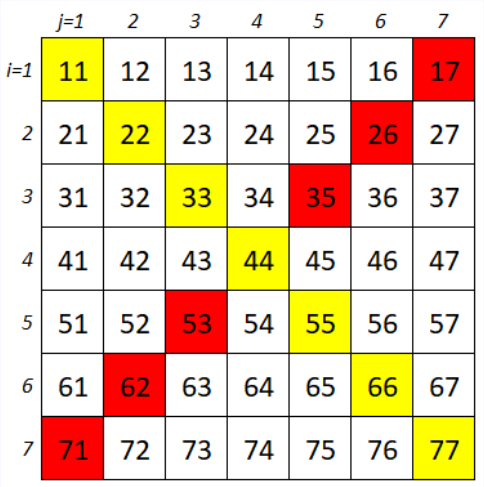
i — порядковый номер *строки*;

j — порядковый номер *столбца*.

Если в массиве *равное* количество строк и столбцов, его называют **квадратным** или **матрицей**. Выделяют *главную* и *побочную* диагонали матрицы. Они обладают особыми свойствами.

На рисунке 5.1 отражены индексы элементов матрицы с 7 строками и 7 столбцами.

Элементы **главной диагонали** располагаются в ячейках желтого цвета. Главный признак, по которому можно их идентифицировать — значения индексов строки и столбца одного элемента одинаковы, т.е. для элемента A[i,j] верно условие **i = j**.



	j=1	2	3	4	5	6	7
i=1	11	12	13	14	15	16	17
2	21	22	23	24	25	26	27
3	31	32	33	34	35	36	37
4	41	42	43	44	45	46	47
5	51	52	53	54	55	56	57
6	61	62	63	64	65	66	67
7	71	72	73	74	75	76	77

Рис. 5.1: Индексы элементов массива с 7 строками и 7 столбцами

Элементы **над главной диагональю** удовлетворяют условию: номер строки элемента меньше номера его столбца ($\mathbf{i} < \mathbf{j}$).

И наоборот: если номер строки элемента больше номера его столбца, то элемент находится **под главной диагональю** ($\mathbf{i} > \mathbf{j}$).

В **побочной диагонали** (ячейки красного цвета) расположены элементы, у которых сумма индексов равна количеству строк (столбцов) плюс один: $\mathbf{i} + \mathbf{j} = \mathbf{N} + \mathbf{1}$.

Элементы **над побочной диагональю** удовлетворяют условию: сумма индексов этих элементов меньше количества строк плюс 1 ($\mathbf{i} + \mathbf{j} < \mathbf{N} + \mathbf{1}$).

И наоборот: сумма индексов элементов **под побочной диагональю** больше количества строк плюс 1 ($\mathbf{i} + \mathbf{j} > \mathbf{N} + \mathbf{1}$).

Зная эти свойства, легко найти, например, сумму всех элементов побочной диагонали:

```
S := 0;
for i:=1 to N do
    for j:=1 to N do
        if i + j = N + 1 then S := S + A[i,j];
```

5.4 Методы упорядочивания элементов массива

Сортировка - это процесс упорядочивания наборов данных одного типа по *возрастанию* или *убыванию* значения какого-либо признака.

При конструировании эффективных программ (подпрограмм), осуществляющих сортировку массивов, требуется использование *быстродействующих* алгоритмов и *минимальное использование* дополнительной памяти.

В качестве **показателя быстродействия** алгоритма используют оценку

- количества операций присваивания;
- количества операций сравнения.

Методы сортировки делятся на:

- Прямые методы сортировки.
- Улучшенные методы сортировки.

Прямые методы сортировки:

- Сортировка обменом (метод «пузырька»).
- Сортировка выбором.
- Сортировка вставками.

Улучшенные методы сортировки используют те же принципы, что и прямые методы, но улучшают быстродействие сортировки с помощью *оригинальных идей*.

По **числу сравнений** все методы имеют квадратичную зависимость от длины массива.

По **числу присваиваний**: методы обмена и вставки имеют квадратичную зависимость от длины массива; метод выбора имеет число присваиваний порядка n*ln(n).

В среднем методы вставки и выбора оказываются приблизительно эквивалентными и в несколько раз (в зависимости от длины массива) лучше, чем метод обмена.

Сортировка обменом (пузырьковая)

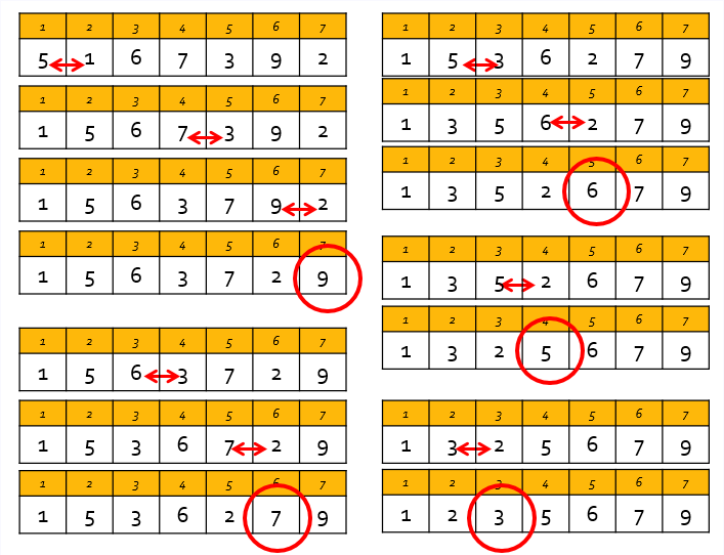


Рис. 5.2: Сортировка обменом (пузырьковая)

Рассмотрим **принцип сортировки обменом** (пузырьковой сортировки):

- Последовательно просматриваются все пары рядом стоящих элементов массива и, если они расположены в неправильном порядке, они меняются местами (рисунок 5.2).

- Это правило действует до тех пор, пока все пары не будут стоять в правильном порядке (то есть, по возрастанию или убыванию).

Алгоритм сортировки обменом может быть реализован следующим образом:

```
const N=10; //количество элементов массива
var a: array[1..N] of integer; //массив
i: integer; //счётчик для цикла
f: boolean; //признак наличия неупорядоченных пар
c: integer; //переменная для промежуточного хранения
...
repeat
   f := false; //пока неупорядоченных пар не было
   for i := 1 to N-1 do //просматриваем все пары рядом стоящих элементов
      begin
       if a[i] > a[i+1] then //если пара стоит в неправильном порядке
         begin
            f := true; //есть неупорядоченная пара
            c:=a[i] ;a[i]:=a[i+1]; a[i+1]:=c; //меняем местами элементы массива
         end;
      end;
until not f; //ждём, пока не исчезнут все неупорядоченные пары
```

Шейкерная сортировка (сортировка перемешиванием, двунаправленная пузырьковая сортировка)

Сортировка перемешиванием, или Шейкерная сортировка, или двунаправленная (англ. *Cocktail sort*) — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства.

- Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива **уже отсортирована** и, следовательно, её можно исключить из рассмотрения.
- Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте **последнего** обмена на каждой итерации. Массив просматривается **поочередно** справа налево и слева направо.

Сортировка методом выбора

- Находится **минимальный** элемент массива (рисунок 5.3) и записывается в **первую** ячейку массива, содержимое которой записывается **на место** найденного минимального элемента.
- После чего находится минимальный элемент массива, начиная со второго элемента, он записывается во **вторую** ячейку массива, содержимое которой записывается на место найденного минимального элемента.
- Таким образом, постепенно выстраивается упорядоченный массив.

Специалисты советуют использовать метод выбора для сортировки сложных структур данных, в случаях, когда на одно сравнение выполняется большое число присваиваний.

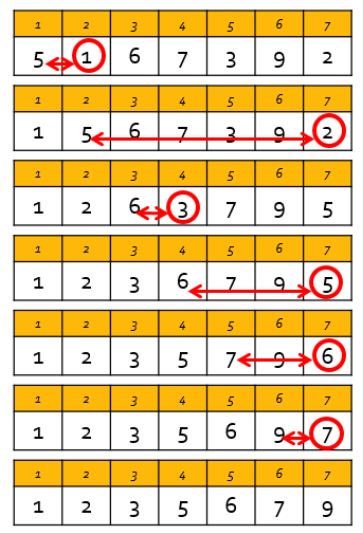


Рис. 5.3: Сортировка методом выбора

Алгоритм, реализующий сортировку выбором, может быть реализован следующим образом:

```
const N=10; //количество элементов массива
var a: array[1..N] of integer; //массив
  i, j: integer; //счётчики для цикла
  c: integer; //переменная для промежуточного хранения
  c2: integer; //переменная для промежуточного хранения
...
for i := 1 to N-1 do
   begin //цикл по первому обрабатываемому элементу массива
     c2 := i; //индекс предполагаемого минимального элемента
```

```
    for j := i+1 to N do //поиск минимального элемента среди оставшихся
      if a[j] < a[c2] //если в c2 индекс не минимального элемента,
        then c2 := j; //то в c2 записывается индекс меньшего элемента
    c:=a[i]; a[i]:=a[c2]; a[c2]:=c; //меняем местами элементы массива
  end;
```

Сортировка вставками

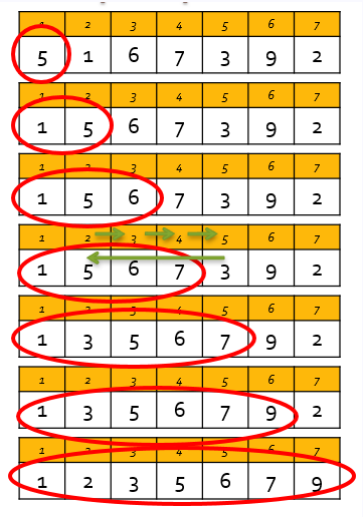


Рис. 5.4: Сортировка вставками

- Сортируемый массив можно разделить на две части – отсортированная часть и неотсортированная (рисунок 5.4).
- В начале сортировки первый элемент массива считается отсортированным, все остальные – не отсортированные.

- Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в нужную позицию в отсортированной части массива.
- Таким образом, за один шаг сортировки отсортированная часть массива увеличивается на один элемент, а неотсортированная часть массива уменьшается на один элемент.

Алгоритм сортировки вставками может быть реализован следующим образом:

```
const N=10; //количество элементов массива
var a: array[1..N] of integer; //массив
  i, j: integer; //счётчики для цикла
  c: integer; //переменная для промежуточного хранения
for i := 2 to N do
  begin
    c := A[i];
    j := i;
    while (j > 0) and (A[j-1] > c) do
       begin
          A[j] := A[j-1];
          j := j – 1;
      end;
    A[j] := c;
  end;
```

Пройти тест для проверки своих знаний по пройденному материалу.

РАЗДЕЛ 6
Множества

Множество – это неупорядоченная совокупность неповторяющихся элементов одного порядкового типа.

Множество напоминает перечислимый тип, но отличается от него тем, что элементы в нём **не упорядочены** (во множестве нет ни самого младшего, ни самого старшего элементов).

Базовым типом для множества могут быть: **логический тип, символьный тип Char, перечислимый тип и тип-диапазон** (например, 0..255), содержащие не более 256 элементов. Большинство целых типов и вещественные типы не могут быть базовыми типами для множества, поскольку они заведомо содержат более 256 элементов.

Для задания элементов множества может использоваться любой порядковый тип, однако порядковые номера элементов множества, т. е. значения функции ord(), должны находиться **в пределах от 0 до 255**.

Во множество входят также все допустимые подмножества (все мыслимые комбинации его элементов), что приводит к огромному числу вариантов, поэтому количество элементов во множестве **не может быть больше, чем 256**. Множество, не содержащее элементов, называют **пустым**. Пустое множество **включено** в любое другое.

Так как элементы во множестве неупорядочены, то к ним нельзя обратиться по номеру или индексу – его у элементов множества просто нет.

Все элементы множества **различны** (если вы и попытаетесь добавить элемент, который уже имеется в множестве, то просто ничего не произойдёт).

Множество описывается так:

var <*имя множества*> : **set of** <*диапазон значений множества*>;

В качестве *диапазона значений множества* указывается любой порядковый тип, число элементов в котором не более 256: Char, Byte, 0..255 и так далее. При этом

верхняя граница числового диапазона не должна быть больше 255, а нижняя - меньше 0.

Например, так описываются типы-множества с различными диапазонами значений:

```
type
MySet = set of 0..255;
MySet = set of 1..25;
MySet = set of 'a'..'z';
MySet = set of Byte;
MySet = set of Boolean;
TColor = (Red, Orange, Yellow, Green, Cyan, Blue, Violet);
Color = set of TColor;
```

Два множества считаются **эквивалентными** тогда и только тогда, когда все их элементы одинаковы, причем порядок следования элементов во множестве не имеет значения.

Если все элементы одного множества входят также в другое, говорят, что первое множество **включено** во второе.

Конкретные значения множества задаются перечислением списка значений через запятую, который берётся в квадратные скобки.

```
MySet := [1, 2 , 5, 10];
MySet := [2, 4, 8, 12..21];
MySet := ['1', '2' , '3'];
MySet := ['3', '1' , '2'];
```

6.1 Операции над множествами

Операция «+» (объединение) – результирующее множество (рисунок 6.1, пункт А) состоит из элементов обоих множеств, указанных в качестве операндов (оди-

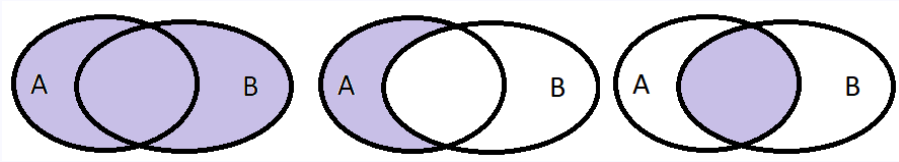


Рис. 6.1: Операции с множествами: А) сложение, Б) разность, В) пересечение

наковые элементы не дублируются). Например: [1,2,3] + [3,4,5] = [1,2,3,4,5].

Операция «–» (разность) – результирующее множество (рисунок 6.1, пункт Б) состоит из тех элементов множества, указанного в качестве левого операнда, которые отсутствуют во множестве, указанном в качестве правого операнда. Например: [1,2,3] - [3,4,5] = [1,2].

Операция «*» (пересечение) – результирующее множество (рисунок 6.1, пункт В) состоит из элементов, имеющихся в каждом из множеств, указанных в качестве операндов. Например: [1,2,3] * [3,4,5] = [3].

Операция «=» (эквивалентность) – логическая операция для сравнения двух множеств. Результат равен *true*, если сравниваемые множества эквивалентны. Например: [1,2,3] = [3,4,5] –*false*; [1,2,3] = [3,2,1] –*true*.

Операции «<=» «>=» (вхождение) – логические операции для проверки включённости одного множества в другое. Результат операций *true* или *false*. Например: [1,2,3] <= [3,2,1] –*true*; [1,2,3] <= [3,2,1,4] –*true*; [1,2,3] <= [1,3,4,5] –*false*; [1,2,3] >= [3,2,1] –*true*; [1,2,3] >= [3,2,1,4] –*false*; [1,2,3] >= [1,3] –*true*.

Операция «in» (проверка принадлежности) – логическая операция, результат равен true, если левый операнд (элемент) включен во множество, являющееся правым операндом. Например, если задано множество S:=[3,2,1,4], то: операция (3 **in** S) даст результат *true*; операция (2*2 **in** S) –*true*; операция (5 **in** S) –*false*.

Также с множествами реализованы процедуры:

- **Include (S, i)** – процедура включает элемент i во множество S.
- **Exclude (S, i)** – процедура исключает элемент i из множества S.

Каждая из этих процедур реализуется одной машинной командой, в то время как для реализации операций «+» и «–» требуется по *13+6*N* команд (N – количество битов во множестве, т.е. количество элементов).

Вывод элементов множества

Значение переменной, определяющей множество, невозможно напрямую вывести в диалоговое окно или на консоль. Для этого используется следующий прием:

- с помощью цикла **For** просматривается весь диапазон возможных элементов множества (он задаётся после служебных слов **set of** при описании переменной типа множество в блоке **var**);
- проверяется наличие возможного элемента во множестве с помощью операции **in** (операция принадлежности элемента множеству);
- если элемент во множестве есть (значение *I **in** S* равно *true*), то этот элемент выводится на экран (консоль, форму).

Например, выведем на экран все элементы множества s, описанного как множество целых чисел в интервале 1..255:

```
var s : set of 1..255;
    i : integer;
В порядке возрастания:
 for i := 1 to 255 do
    if i in s then write(i);
В порядке убывания:
```

```
for i := 255 downto 1 do
    if i in s then write(i);
```

Если по условию задачи необходимо обрабатывать числа, выходящие за допустимые для множеств границы (меньше 0 или больше 255), но при этом числа образуют отрезок длиной не более 256, то используется следующий прием:

- определяется, на какую величину D необходимо изменить имеющийся диапазон, чтобы получить диапазон включенный в интервал 0..255;
- при занесении элементов во множество изменять каждый на найденную величину D;
- при выводе элементов множества каждый элемент изменять в обратную сторону на найденную величину D.

Например, пусть необходимо включить во множество числа -125..125. Вычислим D=125. Если нужно внести во множество число -31, то в него вносится -31+125. Тогда нижняя граница станет равной не -125, а 0; а верхняя граница изменится соответственно со 125 до 250.

Задача: В строковой величине найти все гласные буквы, которые входят в каждое слово.

```
var s, gl, sl, rez1 : set of char;
    i : char;
    Str : string;
    j : integer;

procedure propuskProbelov;
 begin //пропуск пробелов между словами
  while (j<=length(Str)) and (Str[j] = ' ') do j:=j+1;
```

```
 end;

procedure propuskSlova;
 begin //пропуск слова
  while (j<=length(Str)) and (Str[j] <> ' ') do
   begin
    sl:=sl+[Str[j]]; //добавление каждого символа слова во множество sl
    j:=j+1;
   end;
 end;

   begin
    s := ['a'..'z']; //заполнение множества s символами алфавита
    gl := ['e','y','u','i','o','a','j']; //заполнение множества gl гласными
    writeln('Введите предложение, состоящее из слов:');
    readln(Str); //введенное пользователем предложение
    j:=1; //индекс символа в строке Str
    rez1:=gl; //сначала в результате - все гласные
     while j<length(Str) do
      begin
        propuskProbelov;
        sl:=[]; //очищается перед каждым словом
        propuskSlova;
        j:=j+1;
        rez1:=rez1*sl; //операция «*» оставляет в rez1 только гласные, входящие
в это слово
      end;
    writeln('Гласные буквы в каждом слове: ');
    for i:='a' to 'z' do //вывод rez1
```

```
    if i in rez1 then write(i);
end.
```

Пройти тест для проверки своих знаний по пройденному материалу.

РАЗДЕЛ 7
Основы работы с файлами

Файл – это именованная структура данных, представляющая собой последовательность данных одного типа, причём количество элементов последовательности практически не ограничено. В первом приближении файл можно рассматривать как массив переменной длины неограниченного размера.

Данные, которые заполняют файл, называют **компонентами** файла. Компоненты **пронумерованы** начиная с нуля. Компоненты файла могут быть различных типов: строки, символы, числа, массивы, записи и проч.

По типам содержащихся в них элементов различают файлы трёх видов:

текстовые файлы – последовательности символов, разбитых на строки. В **Delphi** предопределен тип **TextFile**, соответствующий текстовому файлу. Таким образом, объявление файловой переменной имеет вид:

var <*имя файловой переменной*> : **TextFile**;

Например:

var F : **TextFile**;

типизированные файлы – двоичные файлы, содержащие последовательность однотипных данных. Тип данных, содержащихся в файле, может быть не только простым типом, но и структурированным (массив, запись и проч.). Объявление файловых переменных для таких файлов имеет вид:

var <*имя файловой переменной*>: **file of** <*тип данных*>;

Например:

var F : **file of real**;

нетипизированные файлы – двоичные файлы, которые могут содержать данные самых различных типов в виде последовательности байтов. Программист при чтении этих данных сам должен разбираться, какие байты к чему относятся. Файловая переменная для нетипизированного файла объявляется следующим образом:

var <*имя файловой переменной*> : **file**;

Например:

var F : **File**;

Общая технология работы с файлами в Delphi требует определённого порядка действий:

1. Связывание файла с файловой переменной (**AssignFile**(F, S)).
2. Открытие файла для работы:
   a. для чтения (**Reset**(F));
   b. для записи (**ReWrite**(F));
   c. для чтения и записи (**Append**(F)).
3. Работа с файлом.
4. Закрытие файла (**CloseFile**(F)).

Рассмотрим подробнее каждый из этапов:

Связывание файла с файловой переменной (AssignFile(F, S))

Для доступа к файлам чаще всего используется специальная **файловая переменная**, которая должна быть описана одним из трёх приведённых выше способов. Она связывается с существующим или вновь создаваемым файлом процедурой **AssignFile**. Эта процедура имеет синтаксис:

AssignFile(F: **File**, S: **string**);

где F — файловая переменная любого типа, S — строка, содержащая имя файла. Если полный путь поиска не указан, то файл будет разыскиваться в текущем каталоге. Например:

AssignFile(F1, 'Test.txt'); *//связывает файловую переменную F1 с файлом «Test.txt»*

AssignFile(F2, 'c:/projects/test.out'); *//связывает файловую переменную F2 с файлом «c:/ projects/test.out»*

AssignFile(F3, OpenDialog1.**FileName**); *//связывает файловую переменную F3 с файлом, имя которого записано в свойстве* ***FileName*** *компонента — диалога* ***OpenDialog1***

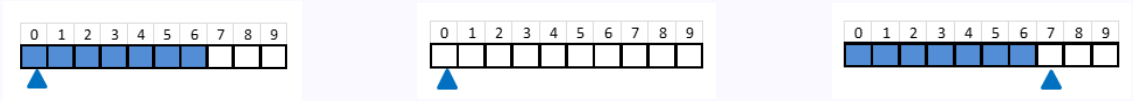
В дальнейшем после связывания файла с файловой переменной процедурой **AssignFile** при обращении к файлу нужно указывать не его имя (например, Test.txt), а имя файловой переменной.

Открытие файла для работы

Для работы с файлом его необходимо открыть (*инициализировать*). Это означает, что операционная система «даёт добро» на внесение изменений в данный файл (например, на запись) и следит, чтобы обращения других пользователей и программ к этому файлу (если компьютер подключен к сети) выполнялись корректно. Так, считывание данных из файла, в который другой пользователь в этот момент вносит изменения, невозможно.

В файле есть невидимый курсор, который принято называть **указателем**, посредством которого можно работать с содержимым. Его можно представить, как курсор в текстовом файле. Если требуется записать данные в файл, указатель ста-

вится в нужную позицию и далее выполняется запись. Если нужно прочитать – указатель перемещается на нужную компоненту файла и считывает её. Указатель при этом каждый раз движется вперёд к следующей компоненте файла. Рано или поздно он достигнет конца файла. Инициализировать (открыть) файл – означает указать для этого файла направление передачи данных. В **Delphi** можно открыть файл для **чтения**, для **записи** или для чтения и записи одновременно, при этом при открытии указатель файла будет располагаться в разных позициях (рисунок 7.1):



а) Reset(F) б) Rewrite(F) в) Append(F)

Рис. 7.1: Расположение указателя при открытии файла разными процедурами

Открытие файла для чтения – осуществляется процедурой

Reset(<файловая переменная>);

С помощью процедуры **Reset** можно открыть любой из трёх видов файлов (текстовый, типизированный, нетипизированный). В результате выполнения этой процедуры указатель, связанный с этим файлом, будет указывать на начало файла (рисунок 7.1, а), то есть на компонент с порядковым номером 0.

Открытие файла для записи – осуществляется процедурой

Rewrite (<*файловая переменная*>);

Процедура **Rewrite** может быть использована для любого из трёх видов файлов. При этом при открытии уже существующего файла вся информация из него стирается, а указатель устанавливается в начало файла. Если попытаться открыть с помощью **Rewrite** не существующий файл, то он будет создан в той

же папке, где находится вызвавшая его программа. Новый файл получит имя, написанное в процедуре **AssignFile**, подготавливается к приему информации и его *указатель* устанавливается (рисунок 7.1, б) на компоненту с номером 0.

Открытие файла для чтения и записи (т.е. для дозаписи информации в ранее существовавший файл)

Append (<*файловая переменная*>);

Процедура **Append** применима только к текстовым файлам, т.е. файловая переменная должна иметь тип TextFile. Процедурой **Append** нельзя инициировать запись в типизированный или нетипизированный файл. Указатель файла устанавливается в конец файла (рисунок 7.1, в) для дополнения его новыми компонентами.

Работа с файлом. Это может быть считывание из него данных, запись новой информации (компонентов), поиск и другие операции. Для разных видов файлов работа с файлом отличается. Более подробно рассмотрим её при ниже.

Закрытие файла – обязательный процесс по окончании работы с ним. Осуществляется с помощью процедуры **CloseFile**:

CloseFile(<*файловая переменная*>);

Закрытие файла означает, что файл снова доступен другим приложениям без ограничений. Кроме того, закрытие файла гарантирует, что все внесённые в него изменения не пропадут, потому что для повышения скорости работы результаты промежуточных действий обычно сохраняются в специальных буферах операционной системы.

Дополнительные процедуры для работы с файлом:

Erase(F) – **уничтожает** физический файл на диске, который был связан с файловой переменной F. Файл к моменту вызова процедуры **Erase** должен быть закрыт.

Rename(F, NewName: String) – **переименовывает** физический файл на диске, связанный с логическим файлом F. Переименование возможно после закрытия файла.

7.1 Текстовые файлы

Текстовые файлы связываются с файловыми переменными, принадлежащими стандартному типу **TextFile**. Текстовые файлы предназначены для хранения текстовой информации и представляют собой последовательность строк, а строки - последовательность символов. Строки имеют переменную длину, каждая строка завершается признаком конца строки.

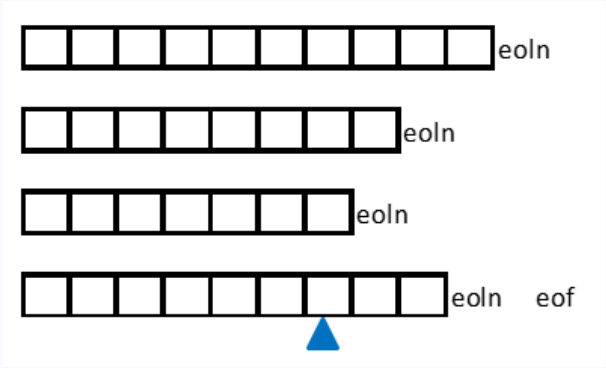


Рис. 7.2: Структура текстового файла

Доступ к каждой компоненте (строке или символу) текстового файла возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой строки (рисунок 7.2) ставится специальный признак конца строки **EOLN** (End Of Line), а в конце файла – признак конца файла **EOF** (End Of File). Эти признаки можно протестировать одноименными функциями, которые имеют логи-

ческий тип и могут принимать значения **True** или **False**:

while not EOF(F) do ... // *цикл продолжается, пока указатель не достигнет конца файла*;

while not EOLN(F) do ... // *цикл продолжается, пока указатель не достигнет конца строки.*

Для чтения и записи строк и символов в файл применяются процедуры:

Read (F, V1, ...) – читает из текстового файла F в переменные V1, ... значения соответствующего типа, игнорируя признаки **EOLN**. Не может читать признак конца строки **EOLN**(F).

Readln (F, V1, ...) – читает из текстового файла F в переменные V1, ... значения соответствующего типа, с учётом признаков **EOLN** (т.е. во всех строках файла).

Write (F, V1, ...) – записывает значения переменных V1, ... в файл F.

Writeln (F, V1, ...) – записывает значения переменных V1, ... и признак конца строки **EOLN** в файл F.

В качестве параметров V1, ... могут использоваться символы, строки и числа. Если V1, ... – символьного типа, то в эти переменные будут считываться символы, если V1, ... строкового типа, то считываться будут строки, если V1, ... числа, то числа. Процедура **Read** не способна «перепрыгнуть» через признак конца строки **EOLN**, поэтому она будет читать только первую строку до тех пор, пока не будет использована процедура **Readln**.

Чтобы прочесть какую-то компоненту текстового файла, нужно сначала прочесть все предыдущие. Такой доступ к компонентам файла называется **последовательным доступом**. Последовательный доступ к компонентам есть только у текстовых файлов.

После применения процедуры **Read** указатель файла **автоматически** переместится **на следующую компоненту** - в зависимости от типа параметра V1: если

параметр символьного типа, то указатель переместиться на следующий символ в той же строке. Если параметр строкового типа, то указатель переместиться на следующую строку.

После применения процедуры **Readln** указатель файла **автоматически** переместится **на следующую строку**, даже если параметр V1 символьного или числового типа.

При вызове **Readln** без параметров V1, ... указатель файла станет на начало следующей строки. При вызове **Writeln** можно опускать параметры V1, ... В этом случае в файл передаётся пустая строка.

Программа «зависнет» т.к. вторая строка файла не будет прочитана	Корректная программа посимвольного вывода на консоль содержимого текстового файла

var F : **TextFile**;
 c : **char**;
AssignFile(F,'Test.txt');
Reset(F);
while not EOF(F) **do**
 begin
 read(F,c);
 write(c);
 end;
	var F : **TextFile**;
	 c : **char**;
	AssignFile(F,'Test.txt');
	Reset(F);
	while not EOF(F) **do**
	 begin
	 while not EOLN(F) **do**
	 begin
	 read(F,c);
	 write(c);
	 end;
	 readln(F);
	 writeln;
	 end;

7.2 Типизированные файлы

Длина любого компонента типизированного файла строго постоянна, что даёт возможность организовать **прямой доступ** к каждой из них (т.е. доступ к компоненте по её порядковому номеру).

Процедур, аналогичных **Readln** и **Writeln**, для типизированных файлов нет. Зато есть процедура **Seek**, позволяющая перемещаться по файлу не только последовательно, как в текстовых файлах, но сразу переходить к требуемому элементу.

Типизированные файлы можно открывать только с помощью процедур **Reset** и **Rewrite**. Дозаписывать типизированный файл можно, обратившись к нему через Reset, с помощью процедуры **Write**.

После открытия файла указатель стоит в его начале и указывает на первый компонент с номером 0. После каждого чтения или записи с помощью процедур ввода-вывода (**Read** или **Write**) указатель сдвигается к следующей компоненте файла. Переменные в списках ввода-вывода должны иметь тот же тип, что и компоненты файла.

Подпрограммы для работы с типизированными файлами:

Read (F, V1, ...) – процедура читает из типизированного файла F в переменные V1,... значения соответствующего типа.

Write (F, V1, ...) – процедура записывает значения переменных V1,... соответствующего типа в файл F.

Seek (F, N) – процедура перемещает указатель файла F к требуемому компоненту с номером N (первый компонент файла имеет номер 0).

FileSize (F) – функция возвращает количество компонент файла F. Чтобы переместить указатель в конец типизированного файла, можно написать:

Seek(F, FileSize(F)).

FilePos (F) – функция возвращает текущую позицию в файле, т.е. номер компонента, который будет обрабатываться следующей операцией ввода-вывода.

Задача 1: записать в типизированный файл 10 целых чисел, найденных случайным образом.

```
program Project2;
var F : File of Byte;
    c : Byte;
    i : Integer;
begin
  AssignFile(F, 'Test.txt');
  Rewrite(F);
  For i := 1 to 10 do
     begin
      c:=random(100);
      Write(F,c);
     end;
  CloseFile(F);
end.
```

Задача 2: Найти квадрат последнего целого числа, записанного в типизированный файл.

```
program Project2;
var F : File of Byte;
   c : Byte;
begin
    AssignFile(F, 'Test.txt');
    Reset(F);
    Seek(F,FileSize(F)-1);
    Read(F,c);
```

```
    Write(c*c:5);
    CloseFile(F);
end.
```

Задача 3: В типизированном файле целых чисел поменять местами элементы, стоящие на 1-м и 5-м местах.

```
  var f : file of integer;
      N1, N5, i : integer;
begin
  Assign(f, 'int.dat');
  Reset(f);
  Seek(f, 0); read(f, N1);
  Seek(f, 4); read(f, N5);
  Seek(f, 0); Write(f, N5);
  Seek(f, 4); Write(f, N1);
  Seek(f, 0);
  Close(f);
end.
```

7.3 Нетипизированные файлы

Нетипизированные файлы объявляются, как файловые переменные типа **File** и отличаются тем, что для них не указан тип компонентов. Отсутствие единого типа делает эти файла, с одной стороны, совместимыми с любыми другими файлами, а с другой – позволяет организовать высокоскоростной обмен данными между диском и памятью.

В нетипизированный файл можно записать компоненты **разных типов**: они расположатся друг за другом, так же, как и в других видах файлов, будут пронумерованы (начиная от 0).

При инициализации нетипизированного файла процедурами **Reset** и **Rewrite** можно указать длину записи нетипизированного файла в байтах.

```
var F : File;
begin
AssignFile(F,'Test.txt');
Reset(F, 512);
...
 CloseFile(F);
end.
```

Длина записи нетипизированного файла указывается вторым параметром при обращении к процедурам **Reset** и **Rewrite**. Если длина записи не указана, она принимается равной 128 байт.

Delphi не накладывает каких-либо ограничений на длину записи нетипизированного файла, за исключением требования положительности и ограничения максимальной длины значением 2 Гбайт.

При работе с нетипизированными файлами могут применяться все процедуры и функции, доступные типизированным файлам, за исключением **Read** и**Write**, которые заменяются соответственно высокоскоростными процедурами **BlockRead** и **BlockWrite**.

BlockRead (F, B, C) – считывание значений переменной B в файл F. C – количество записей, которые должны быть прочитаны за одно обращение к диску.

BlockWrite (F, B, C) – запись значений переменной B в файл F. C – количество записей, которые должны быть записаны за одно обращение к диску.

Прямой и последовательный доступ

Смысл **последовательного доступа** к файлу заключается в том, что в каждый момент времени доступна лишь одна компонента из всей последовательности. Для того, чтобы обратиться (получить доступ) к компоненте с номером N, необходимо

просмотреть от начала файла последовательно все предшествующие N-1 компоненты. После обращения к компоненте с номером N указатель автоматически переместится с компоненте с номером N+1. Отсюда следует, что процессы формирования (записи) компонент файла и просмотра (чтения) не могут произвольно чередоваться. Таким образом, файл вначале строится при помощи последовательного добавления компонент в конец, а затем может последовательно просматриваться от начала до конца.

Delphi позволяет применять к типизированным и нетипизированным файлам, записанным на диск, способ **прямого доступа**. **Прямой доступ** означает возможность заранее определить в файле блок, к которому будет применена операция ввода-вывода. В случае нетипизированных файлов блок равен размеру буфера, для типизированных файлов блок - это одна компонента файла.

Прямой доступ предполагает, что файл представляет собой линейную последовательность пронумерованных блоков. Если файл содержит n блоков, то они нумеруются от 0 до n-1.

Реализация прямого доступа осуществляется с помощью функций и процедур **FileSize**, **FilePos**, **Seek**.

РАЗДЕЛ 8
Тип данных Запись

Запись – это структура данных, объединяющая элементы одного или различных типов. Каждый элемент записи называется **поле** и имеет свое имя и тип. Запись можно рассматривать как описание одного объекта. Поля записи – различные характеристики одного объекта. Записи удобны для создания структурированных баз данных с разнотипными элементами.

Чтобы работать в программе с записями, нужно сначала описать соответствующий тип данных. Запись описывается следующим образом:

```
Type
  имя типа записи = record
      поле записи : тип поля записи;
  end;
```

Например:

```
Type
TStudent = record                  //объявление типа Запись
Fio : string[20];                  //поле ФИО
Group : integer;                   //поле номера студ. группы
Ocn : array[1..3] of integer;      //поле массива оценок
end;

Var
Student: TStudent;                 //объявление переменной типа
                                   Запись
```

Доступ к каждому полю осуществляется указанием имени записи и поля, разделённых точкой, например:

```
Student.Fio := 'Иванов А.И.';      //внесение данных в поля записи
Student.Group := 72 0603;
```

Доступ к полям можно осуществлять также при помощи оператора **with**:

```
with Student do
  begin
    Fio := 'Иванов А.И.';
    Group := 720603;
  end;
```

Для работы с большим количеством записей (т.е. создания и обработки базы данных) лучше всего хранить записи в типизированных файлах. Типом файловой переменной в этом случае является запись:

```
Var
   Student : TStudent;        //объявление переменной типа запись
   f : file of TStudent;      //объявление файла типа запись
```

В этом случае в файле можно сохранить множество записей.

Задача: написать программу, вводящую в файл и читающую из файла список покупателей, совершивших покупку в магазине. Каждая запись должна содержать фамилию, имя, отчество, адрес покупателя и дату покупки. Вывести список всех покупателей, удалив записи об одном и том покупателе, совершавшем покупки в разное время. Записать эту информацию в текстовый файл.

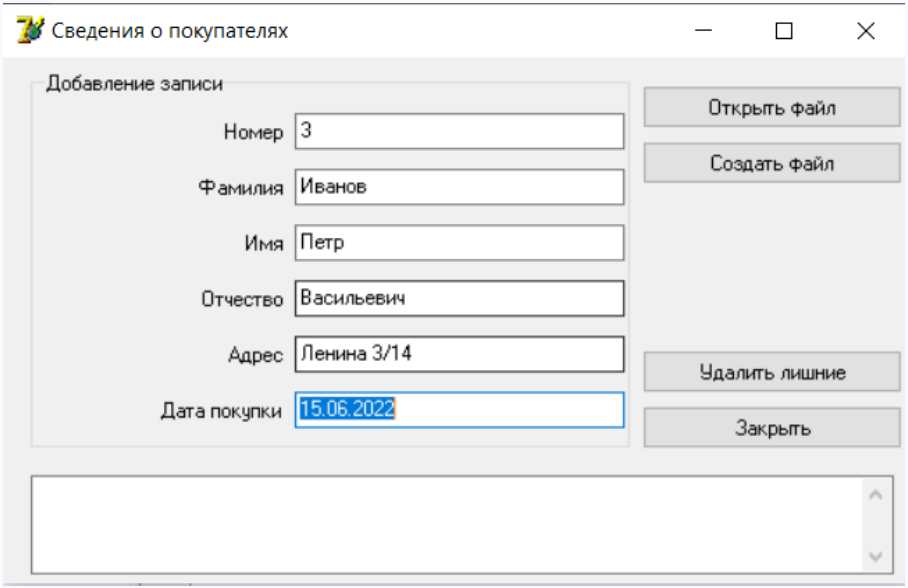


Рис. 8.1: Пример формы для обработки записей из типизированного файла

Программа реализована в визуальной среде Delphi. После запуска программы на выполнение появится диалоговое окно программы (рисунок 8.1). Кнопка *«Добавить запись»* видна не будет. Необходимо создать новый файл записей, нажав на кнопку *«Создать файл»* или открыть ранее созданный, нажав кнопку *«Открыть файл»*. После этого станет видна кнопка *«Добавить запись»* и можно будет вводить записи. При нажатии на кнопку *«Удалить лишние»* будет проведено удаление записей о повторяющихся покупателях. Файл записей закрывается одновременно с программой при нажатии на кнопку *«Закрыть»*.

```
type Pokypatel = record //описание типа-запись, содержащей информацию об 1 покупателе
  Nom : integer;
  Fam : string[15];
  Im : string[10];
  Otch : string[15];
  Adr : string[30];
  Dat : string[15];
end;

var chel : Pokypatel;
    f : file of Pokypatel;
    m, k : array[1..100]of Pokypatel ;
    s : string;
    i, j, n : integer;

procedure TForm1.Button2Click(Sender: TObject); //кнопка «Закрыть»
begin
      Closefile(f);
      Form1.Close;
```

```
end;

procedure TForm1.Button3Click(Sender: TObject); //кнопка «Создать файл»
begin
  if OpenDialog1.Execute then
   begin
    s:=OpenDialog1.FileName;
    Assignfile(f,s);
    Rewrite(f);
    Button1.Show;
   end;
end;

procedure TForm1.Button1Click(Sender: TObject); // кнопка «Добавить запись»
begin
  seek(f,filesize(f));
  with chel do
   begin
    Nom:=strtoint(Edit1.Text);
    Fam:=Edit2.Text;
    Im:=Edit3.Text;
    Otch:=Edit4.Text;
    Adr:=Edit5.Text;
    Dat:=Edit6.Text;
   Memo1.Lines.Add(inttostr(nom)+' '+Fam+' '+Im+' '+Otch+' '+Adr+' '+Dat);
   end;
write(f,chel);
Edit1.Text:='';
Edit2.Text:='';
```

```
Edit3.Text:=’’;
Edit4.Text:=’’;
Edit5.Text:=’’;
Edit6.Text:=’’;
end;

procedure TForm1.Button4Click(Sender: TObject); //кнопка «Открыть»
begin
    if openDialog1.Execute then
      begin
        s:=OpenDialog1.FileName;
        AssignFile(f,s);
        Reset(f);
      end;
Memo1.Lines.Clear;
While not eof(f) do
   begin
    read(f,chel);
    with chel do Memo1.Lines.Add(inttostr(nom)+’ ’+Fam+’ ’+Im+’ ’+
    +Otch+ ’ ’+Adr+’ ’+Dat);
  end;
Button1.Show;
end;

procedure TForm1.Button5Click(Sender: TObject); //кнопка «Удалить лишние»
 var i, j : integer;
   f1, f2 ,f3 : file of Pokypatel;
   a, b : Pokypatel;
```

```
   h : boolean;

function Sovpad(a,b : Pokypatel) : boolean;
  begin
     result := not( (a.Otch = b.Otch) and (a.Fam = b.Fam) and (a.Im = b.Im)
       and (a.Adr = b.Adr) )
   end;

begin
  AssignFile(f1, s+'.1');
  Rewrite(f1);
  n := 1;
  Seek(f, 0);
  While not Eof(f) do
     begin
      Read(f, m[n]);
      n := n+1;
    end;
  Seek(f, 0);
  for i := 1 to n-1 do
     begin
       h := true;
       for j := i+1 to n-1 do
          if not Sovpad(m[i],m[j]) then h := false;
      if h then Write(f1, m[i]);
    end;
Memo1.Lines.Clear;
seek(f1, 0);
While not Eof(f1) do
```

```
      begin
        Read(f1, a);
        with a do
          Memo1.Lines.Add(InTtoStr(nom)+' '+Fam+' '+Im+' '+Otch+' '+Adr+'
'+Dat);
      end;
    end;
```

Приложения

Вопросы к зачету

1. Переменные: определение, назначение и типы. Предопределенные (стандартные) типы. Раздел описания переменных. Область действия переменных.
2. Арифметические операции над целыми и вещественными данными. Приоритеты операций. Изменение приоритета. Целочисленные операции. Тип-диапазон.
3. Логические операции И, ИЛИ, НЕ. Таблицы истинности для этих операций. Использование данных логических операций в условных выражениях.
4. Математические функции. Функции преобразования типов.
5. Оператор присваивания. Назначение, синтаксис и порядок использования. Совместимость типов данных в левой и правой частях оператора присваивания.
6. Условный оператор. Назначение, синтаксис и семантика. Принципиальные отличия от оператора выбора (варианта).
7. Оператор выбора (варианта). Назначение, синтаксис и семантика. Принципиальные отличия от условного оператора.
8. Циклический процесс. Определение, назначение. Оператор цикла с заранее известным количеством повторений (синтаксис и семантика). А-циклы.

9. Циклический процесс. Определение, назначение. Оператор цикла с предусловием (синтаксис и семантика). КМВ-циклы.

10. Циклический процесс. Определение, назначение. Оператор цикла с постусловием (синтаксис и семантика). КМВ-циклы.

11. Рекурсия. Пример.

12. Символьный тип данных.

13. Строковый тип данных. Операции над строками. Стандартные процедуры и функции для работы со строками.

14. Структурированные типы данных. Массивы. Операции с массивами. Динамические массивы.

15. Способы поиска в массивах.

16. Способы сортировки элементов массива.

17. Консольное приложение. Операции ввода-вывода при работе с консольным приложением.

18. Множества. Операции над множествами.

19. Понятие файла с точки зрения его использования в программе. Типы файлов в языке Delphi. Дескриптор файла. Общая схема работы с файлом.

20. Файлы с прямым и последовательным доступом. Понятие указателя файла. Действия над указателем файла в зависимости от способа доступа к файлу.

21. Текстовые файлы: процедуры и функции обработки данных; действия над указателем файла. Операции ввода-вывода при работе с текстовыми файлами.

22. Типизированные файлы: процедуры и функции обработки данных; действия над указателем файла. Операции ввода-вывода при работе с типизированными файлами.

23. Нетипизированные файлы: процедуры и функции обработки данных; действия над указателем файла. Операции ввода-вывода при работе с нетипизированными файлами.

24. Структура программы на языке Pascal, Delphi (все разделы). Назначение каждого раздела. Очередность описания разделов программы.

25. Подпрограммы. Виды подпрограмм. Структура подпрограмм. Сходства и отличия процедур и функций. Порядок описания и вызова подпрограмм. Использование директивы Forward при описании подпрограмм.

26. Механизм передачи параметров. Фактические и формальные параметры.

Основные определения

Алгоритм – это конечная последовательность команд исполнителю, приводящая к решению поставленной задачи.

Алгоритм с ветвлением – алгоритм, в котором после проверки условий в разных ситуациях исполняется один из двух разных наборов команд.

Алгоритм с повторением – алгоритм, который в своем теле содержит команду повторения.

Линейный алгоритм – алгоритм, в котором исполнитель все команды алгоритма исполняет одну за другой в порядке их записи.

Величина – это отдельный информационный объект, отдельная единица данных.

Запись – это структура данных, объединяющая элементы одного или различных типов, называемыми полями.

Именованная константа – это имя (идентификатор), которое в программе используется вместо самой константы.

Обычная константа – это целое или дробное число, строка символов или отдельный символ, логическое значение.

Исполнитель – исполнителем будем называть человека, живое существо или автоматическое устройство, которое способно к восприятию и исполнению команд (предписаний).

Итерация цикла– единичное выполнение тела цикла.

Компилятор – специальная программа, которая выполняет задачу преобразования исходной программы в машинный код.

Компиляция – это процесс преобразования исходной программы в исполняемую при помощи компилятора.

Консоль – это монитор и клавиатура, рассматриваемые как единое устройство.

Массив – это совокупность величин одного типа, обозначенных одним именем.

Множество – это набор логически связанных друг с другом однотипных элементов.

Переменная – это именованный участок памяти для хранения данных определенного типа.

Описание переменной – резервирование определенного именованного участка памяти для хранения значения переменной.

Идентификация переменной – присваивание переменной первоначального конкретного значения.

Повторение – это набор команд, который исполняется до тех пор, пока выполняется некоторое условие.

Проект – это набор файлов, используя которые компилятор создает исполняемый файл программы (EXE-файл).

Рекурсия – это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих ее операторов обращается сама

к себе.

Структурное программирование – процесс разработки алгоритмов с помощью структурных блок-схем.

Структурное программирование сверху-вниз – это процесс пошагового разбиения алгоритма на более мелкие части с целью получить такие элементы, для которых можно легко написать конкретные команды.

Тело цикла – это последовательность операторов, повторяемых в процессе выполнения оператора цикла.

Условие – это выражение логического типа (Boolen), которое может принимать одно из двух значений: True (истина) или False (ложь).

Файл – информация, которая хранится на компьютерном носителе под специальным именем.

Типизированные файлы – это двоичные файлы, содержащие последовательность однотипных данных.

Нетипизированные файлы – это двоичные файлы, которые могут содержать самые различные данные в виде последовательности байтов.

Список рекомендуемой литературы

- Абрамов, В.Г. Введение в язык Паскаль: учебное пособие для студентов высших учебных заведений, обучающихся по специальности 010501 «Прикладная математика и информатика», направлению 010400 «Информационные технологии» / В.Г. Абрамов, Н.П. Трифонов, Г.Н. Трифонова. - Москва: КноРус, 2011.- 380 с.
- Бакнелл, Дж. Фундаментальные алгоритмы и структуры данных в Delphi : [пер. с англ] / Дж. Бакнелл. –М. : ДиаСофтЮП : СПб. : Питер, 2006. – 557 с.
- Боровский, А. Н. Программирование в Delphi 2005 / А.Н. Боровский. – СПб.: БХВ-Петербург, 2005. - 448 с. http://bit.ly/1YCpmAy
- Дарахвелидзе, П. Г. Delphi – среда визуального программирования / П. Г. Дарахвелидзе, Е. П. Марков. – СПб. : BHV-Санкт-Петербург, 1996. – 352 с.
- Орлов, С.А. Теория и практика языков программирования: учебник для вузов. Стандарт 3-го поколения / С.А. Орлов. – Санкт-Петербург: Питер, 2013. – 688 с.
- Прищепов, М. А. Программирование на языках Basic, Pascal и Object Pascal в среде Delphi : учеб. Пособие [для вузов] / М. А. Прищепов, Е. В. Севернева, А. И. Шакирин ; ред. М. А. Прищепов. – Мн. : ТетраСистемс, 2006. – 320 с.
- Севернева, Е.В. Основы алгоритмизации и программирования: учебно-методический комплекс для студентов высших учебных заведений / Е.В. Севернева, Н.М. Жалобкевич. – Минск: БГАТУ, 2009. – 112 с.
- Семакин, И.Г. Основы алгоритмизации и программирования: учебник / И.Г. Семакин, А.П. Шестаков. – Москва: Академия, 2011. – 391 с.
- Симанович, С.В. Информатика. Базовый курс: учебник для вузов. Стандарт 3-го поколения. / С.А. Симанов. – Санкт-Петербург: Питер, 2011. – 640 с.

- Фаронов, В.В. Delphi. Программирование на языке высокого уровня : учебник для вузов по направлению «Информатики и вычислительная техника». / В.В. Фаронов. – СПб.; М. ; Нижний Новгород : Питер, 2006. – 640 с.
- Чиртик, А. А. Программирование в DELPHI / А. А. Чиртик. – СПб. ; М. ; Нижний Новгород : Питер, 2010. – 400 с.
- Архангельский, А.Л. Язык Pascal и основы программирования в Delphi/ А. Л. Архангельский. – М.: Бином, 2004. – 496 с.
- Бобровский, С.И. Delphi 7. Учебный курс / С. И. Бобровский. – СПб.: Питер, 2003. – 736 с.
- Буславский, А.А. Начальный уровень обучения программированию на языке Pascal / А.А. Буславский. – Минск: МОИРО, 2010. – 89 с.
- Вабищевич, С.В. Основы программирования в среде Delphi / С.В. Вабищевич, А.Л. Воробев. – Минск: БГПУ, 2004. – 112 с.
- Долинский, М.С. Решение сложных и олимпиадных задач по про-граммированию / М.С. Долинский. - СПб.: Питер, 2006. – 366 с.
- Котов, В.М. Алгоритмы и структуры данных / В.М. Котов, Е.П. Соболевская, А.А. Толстиков. – Минск, БГУ, 2011. – 267 с.
- Кричевцов, О.В. Лабораторные работы по дисциплине «Основы алгоритмизации и программирования». Язык Pascal / О.В. Кричевцов. – Витебск: ВГТК, 2010. – 133 с.
- Культин, Н.Б. Программирование в Turbo Pascal 7.0 и Delphi / Н.Б. Культин. – Санкт-Петербург: БХВ-Петербург, 2012. – 380 с.
- Москаленко, А.А. Решение прикладных задач в интегрированной среде Турбо Паскаль: методическое пособие / А.А. Москаленко, З.И. Кононенко. – Минск: БИТУ, 2011. – 60 с.

- Потапова, Л.Е. Алгоритмизация и программирование на языке Паскаль: учебно-методическое пособие / Л.Е. Потапова, Т.Г. Алейникова. –Витебск: ВГУ, 2008. – 129 с.
- Пшеничнов, Ю.А. Информатика: лабораторный практикум для студентов технических специальностей / Ю.А. Пшеничнов. – Гомель: БелГУТ, 2011, – 47 с.
- Пянкрат, В.У. ІнтАл і Pascal у паралельным выкладанні / В.У. Пянкрат – Мінск: БДПУ, 2008. – 74 с.
- Расолька, Г.А. Заданні вылічальнай практыкі па курсу «Метады праграмірання і інфарматыка»: дапаможнік для студэнтаў механіка-матэматычнага факультэта спецыяльнасці 1-31 03 01-02 «Матэматыка (навукова-педагагічная дзейнасць)» / Г.А. Расолька, Е.В. Крэмень, Ю.А. Крэмень. – Мінск: БДУ, 2013. – 111 с.
- Расолько, Г.А. Система тестов для самоподготовки по курсу «Методы программирования и информатика». Язык Pascal: пособие для студентов механико-математического факультета специальности 1-31 03 01- 02 «Математика (научно-педагогическая деятельность)» / Г.А. Расолько, Е.В. Кремень, Ю.А. Кремень. – Минск: БГУ, 2013. – 70 с.
- Силаев, Н.В. Методы решения задач информатики: пособие для студентов математического факультета специальностей 1-02 05 05-01 Информатика. Иностранный язык (английский), 1-02 05 03-02 Математика и информатика, 1-02 05 03 Математика / Н.В. Силаев, З.Н. Силаева. – Брест: БрГУ, 2011 – 62 с.
- Фаронов, В. В. DELPHI. Программирование на языке высокого уровня : учебник для вузов по направлению "Информатика и вычислительная техника"/ В. В. Фаронов. - СПб. ; М. ; Нижний Новгород : Питер, 2006. – 640 с.
- Фаронов, В. В. Система программирования Delphi : учебное пособие / В. В. Фаронов. - СПб. ; БХВ-Петербург, 2003. – 912 с.

- Фонасов, С.А. Числовые задачи в программировании: сборник / С.А. Фонасов. – Гродно: Гродненский областной институт развития образования, 2012. – 66 с.