

**Министерство экономического
развития и торговли
Российской Федерации**

**Министерство образования
Российской Федерации**

**ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ – ВЫСШАЯ ШКОЛА ЭКОНОМИКИ
Нижегородский Филиал**

**Основы алгоритмизации
и императивного программирования**

*Рекомендовано Ученым советом Нижегородского филиала Государственного
университета-Высшей школы экономики в качестве учебного пособия
для студентов всех форм обучения*

Нижний Новгород 2003

УДК 681.3.06
ББК 32.973.26
Д 30

В.М.Дёмкин. Основы алгоритмизации и императивного программирования: Учебное пособие / НФ ГУ-ВШЭ – Нижний Новгород, 2003. 144 с.

ISBN 5-901956-05-2

Учебное пособие предназначено для самостоятельного изучения одного из основных разделов курса “Информатика” – основ алгоритмизации и императивного программирования. Обсуждаются классические типы алгоритмов – линейные, условные, циклические, рекурсивные и эвристические, – а также способы их описания. Управляющие структуры императивной программы – следование, выбор и итерация, – ее структуры данных – массивы и файлы, – а также ее составные части – процедуры и функции – представлены в примерах на языке программирования Turbo Pascal.

Библиогр.: 5 назв.

Рецензенты: д-р физ.-мат. наук, проф. Е.М.Громов
д-р физ.-мат. наук, проф. С.Н.Митяков

ББК 32.973.26

ISBN 5-901956-05-2

© В.М. Дёмкин
© Государственный Университет-Высшая Школа Экономики
Нижегородский филиал, 2003

Государственный Университет-Высшая Школа Экономики Нижегородский филиал
Н.Новгород, ул. Б.Печерская, 25

Подп. к печ. 20.03.2003. Формат 60х84/16. Бумага газетная.
Печать офсетная. Усл. п. л.: 8,37. Тираж 300 экз. Заказ № 63.

Отпечатано ООО «Растр НН» г. Н.Новгород, ул. Белинского, 61
ИД № 05407 от 20.07.2001

Предисловие

Курс “Информатика” является одним из базовых курсов блока дисциплин естественнонаучного направления, изучаемый студентами высшей школы всех форм обучения. Среди основных разделов курса “Информатика”, предназначенных для профессиональной подготовки специалистов, особое место по праву принадлежит ставшему уже классическим как по форме, так и по содержанию разделу – основы алгоритмизации и императивного программирования.

С одной стороны, этому разделу отводится роль своеобразного связующего звена между средней и высшей школой, так как выпускник средней школы в той или иной мере уже владеет навыками алгоритмизации и императивного программирования. Как правило, в средней школе изучается именно какой-либо один из императивных языков программирования, например, Basic, Turbo Pascal или C.

С другой стороны, современные языки программирования, которые в том числе поддерживают и императивную парадигму программирования, до сих пор остаются той притягательной силой, заставляющей значительную часть выпускников высшей школы испытывать неослабеваемый интерес к программированию как к одной из сторон их будущей профессиональной деятельности независимо от полученной ими во время учебы специальности.

Имея многолетний опыт разработки программного обеспечения широкого ряда уникальных систем автоматизации физических экспериментов, опыт преподавания курса “Информатика” студентам различных форм обучения и специальностей НГТУ и Нижегородского филиала ГУ ВШЭ, а также опыт проведения студенческих олимпиад по программированию и подготовки студенческих команд НГТУ к командным чемпионатам мира по программированию ACM (Association for Computing Machinery), автор может смело утверждать, что для формирования у студентов младших курсов целостного взгляда на информатику требуется немало усилий. Реальным помощником в этом могут служить отдельные учебные пособия и практикумы по основным разделам курса “Информатика”, связанные единой формой изложения. Прежде всего, сама идея издания подобной литературы связана со стремлением автора рассматривать информатику под углом зрения проектирования алгоритмов и абстрактных типов данных, придерживаясь рамок требуемой для решения задачи парадигмы программирования, например, императивной, объектно-ориентированной или обобщенной при помощи таких популярных языков программирования для персональных компьютеров как Fortran, Basic, Turbo Pascal, C, C++ и Java.

Вниманию читателя предлагается одно из первых таких учебных пособий, предназначенных для самостоятельного изучения основ алгоритмизации и императивного программирования в примерах на Turbo Pascal. Выбор Turbo Pascal обусловлен тем, что он до сих пор остается одним из основных стандартных языков обучения программированию, а также является одним из двух (наряду с языком C) официальных языков командных чемпионатов мира по программированию ACM среди сборных команд высших учебных заведений.

Введение

Парадигмы программирования и Turbo Pascal

Язык программирования Turbo Pascal является одним из представителей семейства так называемых гибридных языков, он поддерживает три парадигмы программирования: императивную (от слов *imperative programming*), называемой также процедурно-ориентированной (от слов *procedure-oriented programming*) или просто процедурной (от слов *procedural programming*), модульную (от слов *modular programming*) и объектно-ориентированную (от слов *object-oriented programming*).

Язык Turbo Pascal является одной из самых популярных модернизированных версий стандартного языка Pascal. Язык Pascal был создан Никлаусом Виртом (Niklaus Wirth) в 1967-71 годах на основе языка Algol в качестве инструмента для систематизированного обучения программированию. Используемый сначала как инструмент обучения, Pascal затем стал стандартным языком для обучения основам вычислительной техники и программирования (ISO-Pascal). Впоследствии принятие и использование расширенной версии языка Pascal фирмой Borland International в 1984 году с названием Turbo Pascal – высокоэффективной и недорогой версии компилятора языка Pascal для операционной системы MS-DOS – позволило этому языку занять достойное место среди современных языков программирования.

В дальнейшем, развиваясь от версии к версии, Turbo Pascal 7.0 стал одной из эффективных интегрированных сред разработки программ для персональных компьютеров, которой он по праву остается и до сих пор.

В основе *императивной парадигмы программирования* лежит классическая фон-неймановская модель вычислений, разделяющая абстракцию состояния и поведения. Программа, написанная на императивном языке, рассматривается как процесс изменения состояния путем последовательного выполнения отдельных команд (императив), т.е. императивная программа явно указывает способ получения желаемого результата при помощи определенной процедуры, не определяя при этом ожидаемых свойств результата.

Императивный подход наиболее естественен для человека, общающегося с компьютером, поэтому класс императивных языков до сих пор остается самым распространенным классом языков программирования. Первые императивные языки программирования высокого уровня Fortran I (1956 г.), Algol-58 (1958 г.), Fortran II (1958 г.), Cobol (1960 г.) и Algol-60 (1960 г.) оказали весьма существенное влияние на дальнейшее развитие науки программирования. Так, начиная с середины 1960-х годов, во многом благодаря именно теории и практике императивного программирования стали осознавать роль подпрограмм – своеобразных частей императивной программы – как важного промежуточного звена между решаемой задачей и компьютером, что в конечном итоге привело к становлению принципа процедурного структурирования программ, где подпрограммы рассматривались как абстрактные программные функции. С той поры наряду с понятием “императивный” вошло в обиход и понятие “процедурный” или “процедурно-ориентированный”.

Под процедурой понимается часть программы, которая выполняет некоторые определенные действия над данными, в общем случае определяемыми параметрами. Процедура может быть вызвана из любого места программы (или другой процедуры), и при каждом вызове процедуры ей могут быть переданы различные параметры. Одной из разновидностей процедур являются функции, которые также выполняют некоторые определенные действия над данными, но результат всегда передают в программу (или процедуру, или функцию) в виде значения функции посредством своего имени. Язык Turbo Pascal предоставляет пользователю широкий выбор средств, позволяющих осуществлять требуемый вид передачи параметров вызова процедурам и функциям, а также возврата значений из функций.

Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, самостоятельность процедур и функций позволяет локализовать в них все детали программной реализации какого-либо алгоритма. Проектирование программы и ее составных частей при этом основывается на применении методов нисходящего проектирования программ – алгоритм программы представляется в виде последовательности относительно крупных подпрограмм, которые в свою очередь представляются в виде последовательности менее крупных подпрограмм и т.д., тем самым реализуется принцип “от сложного – к простому”.

Типичными представителями императивных языков программирования, которые проектировались по мере развития идей процедурного структурирования программ, начиная с 1950-х и до середины 1970-х годов, являются, например, такие языки, как Fortran, Algol, Cobol, Basic, Pascal, C, Modula.

В основе **модульной парадигмы программирования** лежит принцип разбиения всей программы на отдельно компилируемые компоненты, называемых модулями, каждый из которых представляет собой набор связанных процедур вместе с данными, которые обрабатываются только этими процедурами. Физическим представлением каждого модуля является файл. Модули после этапа отдельной компиляции могут участвовать в сборке программы в виде почти независимых частей, что позволяет использовать их как мощный инструмент для разработки библиотек прикладных программ. Модульная парадигма программирования также известна как “принцип сокрытия данных”, который легко расширяется до понятия “сокрытие информации”, что позволяет скрыть от пользователя не только детали реализации каждого модуля, но и исполняемый код процедур.

Дальнейшим развитием идей модульного программирования стал отказ от использования только единственной глобальной области видимости при взаимодействии отдельных частей программы, что привело к созданию механизма логической группировки связанных процедур и данных в пространства имен. Этот механизм группировки позволяет при сборке программы из отдельных частей, например, избегать конфликта имен, не засорять глобальное пространство имен, существенно сократить издержки как на этапе компоновки программы, так и во время ее выполнения. Например, такой подход уже реализован в стандартном C++.

Типичными представителями языков программирования, поддерживающих модульную парадигму программирования, являются, например, такие языки как Ada, Modula-2, Modula-3, Turbo Pascal, C и C++.

Объектно-ориентированное программирование – это основная методология программирования 1990-х годов и начала XXI века. Она представляет собой продукт более 35 лет практики и опыта, которые восходят к использованию языка Simula 67, затем Smalltalk и не так давно Objective C, Object Pascal, Turbo Pascal, CLOS, а теперь, например, C++ и Java.

В основе объектно-ориентированной парадигмы программирования лежат три принципа:

- инкапсуляция (от слова *encapsulation* или *incapsulation*) – объединение данных и процедур для работы с этими данными в единое целое – объект, который является экземпляром класса;
- наследование (от слова *inheritance*) – средство получения новых классов из существующих путем добавления в них новых данных и процедур или модифицирования наследуемых данных и процедур;
- полиморфизм (от слова *polymorphism*) – создание общего интерфейса для группы близких по смыслу действий, выполнение каждого из которых определяется типом данных.

Класс выступает в роли абстрактного типа данных, определяемого пользователем, и отражает базовые концепции предметной области. Объекты типов, определяемых пользователем, называют либо переменными классов, либо экземплярами классов. Каждый объект содержит необходимую информацию, свою для каждого типа. Классы обеспечивают сокрытие информации, гарантированную инициализацию данных, неявное преобразование определяемых пользователем типов, динамическое определение типа, контроль пользователя над управлением памятью, механизм перегрузки стандартных операторов языка – и это далеко не полный перечень того, что классы могут обеспечить.

Преимущество объектно-ориентированной парадигмы программирования по сравнению с императивной в полной мере проявляется при разработке только сложных программных систем, которые проектируются по принципу восходящего проектирования, т.е. “от простого – к сложному”. Императивное программирование наиболее пригодно для реализации небольших по сложности задач, где очень важна скорость исполнения на современных компьютерах.

Следует заметить, что Turbo Pascal существенно отличается от других языков в реализации объектно-ориентированной парадигмы программирования. Например, здесь нет классов как абстрактных типов данных и соответственно нет иерархии классов, вместо классов есть типы-объекты и иерархия типов-объектов; здесь нет параметрического полиморфизма, на основе которого строятся шаблоны функций и классов, есть только динамический полиморфизм для выполнения близких по смыслу действий на основе механизма виртуальных методов; здесь нет перегрузки стандартных операторов языка для переопределения смысла выполняемых действий над объектами. Однако это нисколько не мешает разработчикам использовать язык Turbo Pascal для разработки сложных программных систем на основе объектного подхода. Примером такой сложной программной системы является коммерческая объектно-ориентированная библиотека программирования Turbo Vision, которая входит в комплект поставки языка Turbo Pascal.

Императивное программирование и Turbo Pascal

Это пособие знакомит читателя с основами алгоритмизации и подмножеством Turbo Pascal, которое поддерживает традиционную для языков Basic, Pascal и C императивную парадигму программирования. Автор полагает, что читателю знакомы основные принципы императивного программирования, которые в первую очередь непосредственно связаны с проектированием алгоритмов. Для описания алгоритмов, как правило, используются метаязыки, называемые иногда псевдокодами. При описании алгоритмов автор пользуется структурированным метаязыком, придерживаясь классических управляющих структур – следование, выбор и итерация. В метаязыках, как правило, ограничено применяются средства языков программирования, однако при этом автор всегда стремится к тому, чтобы используемый метаязык оставался бы к ним по возможности максимально нейтрален. Но бывает и наоборот – эти средства иногда трудно заменить соответствующими эквивалентными конструкциями метаязыка, например, в случае использования явных и неявных инструкций возврата из подпрограмм или, например, в случае использования подпрограмм стандартных библиотек для выполнения тех операций, которые связаны рамками только конкретной среды программирования.

В пособии рассматриваются фундаментальные типы, выражения и управляющие структуры языка Turbo Pascal, здесь же наряду с двумя основными структурами данных программы – массивами и файлами – обсуждаются и ее составные части – процедуры и функции. Основные средства языка Turbo Pascal иллюстрируются на примерах программирования классических алгоритмов – линейных, условных, циклических, рекурсивных и эвристических.

Язык Turbo Pascal поддерживает императивную парадигму программирования, предоставляя пользователю широкий выбор средств, позволяющих осуществлять не только требуемый вид передачи параметров вызова процедурам и функциям, а также возврата значений из функций, но и непосредственное встраивание в текст программы ассемблерного кода.

Алгоритмы и алгоритмизация

Понятие алгоритма и алгоритмического процесса

В своей повседневной деятельности каждому из нас постоянно приходится сталкиваться с разнообразными правилами, предписывающим последовательность действий, цель которых состоит в достижении некоторого необходимого результата. Подобные правила очень многочисленны. Например, мы должны следовать вполне определенной системе правил, чтобы вычислить произведение двух многозначных чисел или найти корни квадратного уравнения. Примеры такого рода можно продолжать неограниченно. Такие системы правил часто называют **алгоритмами**.

Алгоритм или алгоритм – одно из основных понятий (категорий) математики, не обладающих формальным определением в терминах более простых понятий, а абстрагируемых непосредственно из опыта. Понятие алгоритма занимает одно из центральных мест в современной математике, прежде всего вычислительной.

Само слово “алгоритм” происходит от *algorithmi*, являющегося, в свою очередь, латинской транслитерацией арабского имени выдающегося персидского математика Абу-Джафара Мохамеда ибн-Мусы аль-Хорезми (IX век н.э.). Он изложил общие правила выполнения действий над числами, представленными в десятичной форме, которыми мы пользуемся до сих пор. Существенным было то, что эти правила могли быть применены к любым числам.

Действия согласно той или иной инструкции, являющейся алгоритмом, для получения определенного результата предполагают наличие некоторых исходных данных. Например, такими исходными данными при выполнении арифметической операции над двумя числами является эта пара чисел, а результатом – одно число. Таким образом, под алгоритмом понимается всякое точное предписание, которое задает процесс, начинающийся с произвольных исходных данных и направленный на получение полностью определенного этими исходными данными результата.

Первым свойством алгоритма является **дискретный**, т.е. пошаговый характер определяемого им процесса. Кроме исходных данных для алгоритма также должно быть определено правило, по которому процесс признается закончившимся ввиду достижения результата. При этом вовсе не предполагается, что результат будет обязательно получен, т.е. процесс применения алгоритма к конкретным исходным данным может также оборваться безрезультатно или не закончиться вовсе. Если процесс заканчивается (соответственно не заканчивается) получением результата, то в этом случае говорят, что алгоритм применим (соответственно не применим) к рассматриваемым исходным данным.

Другим важнейшим свойством алгоритма является **массовость**. Смысл данного понятия заключается в том, что существует некоторое множество объектов, которые могут служить возможными исходными данными для рассматриваемого алгоритма. Например, для алгоритмов выполнения арифметических операций – сложения, вычитания, умножения и деления – такими данными являются все действительные или целые числа. Смысл массовости таких алгоритмов как раз и состоит в том, что

они одинаково пригодны для всех случаев, т.е. они требуют лишь механического выполнения цепочки некоторых простых действий.

Отметим еще одну важную особенность, присущую каждому алгоритму. Предполагается, что алгоритм **понятен** для исполнителя, т.е. исполнитель алгоритма знает, как его выполнять. При этом исполнитель алгоритма, выполняя его, действует “механически”. Очевидно, что формулировка алгоритма должна быть настолько точна и однозначна, чтобы могла полностью определять все действия исполнителя.

Анализ алгоритмов показывает, что если применять алгоритмы повторно к одним и тем же исходным данным, то мы всегда будем получать один и тот же результат. Например, если применять алгоритм деления отрезка пополам с помощью циркуля и неразмеченной линейки к одному и тому же отрезку, то каждый раз будем получать одну и ту же точку – середину отрезка. Таким образом, можно говорить об **определенности** и **однозначности** алгоритмов.

Итак, алгоритм можно определить как систему правил, сформулированную на языке, понятном исполнителю, и определяющую последовательность действий, в результате выполнения которых мы приходим от исходных данных к искомому результату. Такая последовательность действий называется **алгоритмическим процессом**, а каждое действие – его **шагом**. Число шагов для достижения результата обязательно должно быть конечным. Кроме того, алгоритм должен обладать свойствами массовости, определенности и однозначности.

Отметим здесь, что наряду с совокупностями возможных исходных данных и возможных искомых результатов для каждого алгоритма имеется еще совокупность промежуточных результатов, представляющих собой ту рабочую среду, в которой развивается алгоритмический процесс, каждый дискретный шаг которого состоит в целенаправленной смене одного конструктивного объекта другим. Например, при применении алгоритма вычитания столбиком к паре целых чисел (307, 49) последовательно возникнут такие конструктивные объекты:

$$\begin{array}{r}
 307 \\
 - 49 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 3\dot{0}7 \\
 - 49 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 \dot{3}07 \\
 - 49 \\
 \hline
 8
 \end{array}
 \quad
 \begin{array}{r}
 \dot{3}\dot{0}7 \\
 - 49 \\
 \hline
 58
 \end{array}
 \quad
 \begin{array}{r}
 \ddot{3}\dot{0}7 \\
 - 49 \\
 \hline
 258
 \end{array}$$

Неправильно было бы думать, что для любой задачи существует лишь один алгоритм ее решения. Например, для подсчета количества зрителей на трибунах стадиона можно предложить несколько алгоритмов, применяя которые, будем получать один и тот же результат.

Однако далеко не всегда для задачи удастся найти хоть какой-нибудь алгоритм ее решения. Так, на протяжении тысячелетий математики пытались решить задачу о квадратуре круга: с помощью циркуля и неразмеченной линейки построить квадрат, равновеликий кругу с заданным радиусом r , т.е. так, чтобы $\pi r^2 = x^2$, где x – сторона искомого квадрата. Лишь в XIX веке усилиями выдающихся математиков А.М.Лежандра и К.Ф.Линдемана была строго доказана неразрешимость этой задачи. В самом деле, из условия задачи следует, что $x = \sqrt{\pi} \cdot r$, поэтому надо осуществить

построение так, чтобы получить отрезок, длина которого в $\sqrt{\pi}$ раз больше длины данного отрезка. Оказывается “умножить” графически с помощью циркуля и неразмеченной линейки отрезок на число можно лишь тогда, когда это число является корнем алгебраического уравнения с целыми коэффициентами. Лежандр установил, что π – число иррациональное (т.е. не представимо в виде $\frac{p}{q}$, где p и q –

взаимно простые числа), а Линдеман – что оно, к тому же, и трансцендентное, т.е. не удовлетворяет никакому алгебраическому уравнению с целыми коэффициентами.

Столь же известна и другая задача древнегреческой математики – задача о трисекции угла, т.е. о делении угла на три равные части с помощью циркуля и неразмеченной линейки. Как и в случае задачи о квадратуре круга, неразрешимость этой задачи была доказана лишь в XIX веке и тоже алгебраическими методами. Заметим, что для этих знаменитых задач (и это строго доказано) не существует алгоритмов решения (т.е. задачи неразрешимы) в рамках фиксированного набора допустимых действий, в данном случае – построения только с помощью циркуля и неразмеченной линейки.

Есть также множество проблем, о которых мы и сейчас не можем сказать, разрешимы они или нет, и если разрешимы, то каким образом. Например, известна гипотеза о том, что любое четное число представимо в виде суммы двух простых чисел. Но утверждение это пока не доказано, и неясно, удастся ли его когда-либо доказать или опровергнуть.

Итак, для некоторых задач существует несколько алгоритмов их решения, для некоторых таких алгоритмов вообще не существует, и, наконец, есть задачи, для которых мы не знаем, существуют или нет алгоритмы их решения.

Алгоритмические системы

Когда речь идет о построении алгоритма решения какой-либо задачи, то мы явно или неявно предполагаем известными те объекты, которые будут исходными данными для нашего алгоритма. Например, в задаче деления отрезка пополам в качестве таких объектов выступают отрезки произвольной длины. При построении алгоритма нахождения корней алгебраического уравнения $ax^2 + bx + c = 0$ такими объектами являются тройки любых чисел (a, b, c) , определяющие коэффициенты этого уравнения.

Второе, что мы предполагаем известным, – это те действия, с помощью которых строятся шаги алгоритма. Например, в той же задаче о делении отрезка мы разрешаем использовать только действия с циркулем и неразмеченной линейкой, а в задачах решения алгебраического уравнения разрешается выполнять действия сложения, вычитания, умножения, деления и извлечения квадратного корня. Иными словами, мы предполагаем известными все возможности исполнителя алгоритма.

Мы также должны знать, как формулировать шаги заданий, чтобы их понял исполнитель, т.е. знать, какой язык понятен исполнителю алгоритма. Наконец, при построении алгоритма решения какой-либо конкретной задачи мы должны знать,

что собой будет представлять результат: число, точку на прямой и т.д., т.е. мы должны знать множество объектов, к которым принадлежит результат. Однако с помощью циркуля и линейки решается не только задача деления отрезка пополам, но и целый ряд геометрических построений. С помощью арифметических операций также строятся алгоритмы решения самых различных задач.

Набор средств и понятий, позволяющих строить не один алгоритм, а множество алгоритмов, решающих различные задачи, будем называть **алгоритмической системой**. Алгоритмических систем может быть много и каждая из них определяется:

- множеством входных объектов, подлежащих обработке алгоритмами данной системы;
- свойствами исполнителя алгоритмов, т.е. набором тех действий, которые может выполнять исполнитель;
- множеством выходных объектов, представляющих собой искомые результаты выполнения алгоритмов данной системы;
- языком, на котором формулируются алгоритмы, адресованные исполнителю.

Заметим, что множество входных объектов (исходных данных) и множество выходных объектов (результатов) часто совпадают. Например, в алгоритмах вычислений арифметических выражений и исходные данные, и результаты – числа. Но в ряде случаев исходные данные и результаты могут оказаться совершенно разной природы. Например, исходными данными алгоритма поиска номера телефона в справочнике служат фамилия абонента и его адрес места жительства.

Что касается действий, которые может выполнять исполнитель, то они также могут резко отличаться в различных алгоритмических системах. Язык алгоритмической системы тесно связан с исполнителем и не должен включать в свой состав указаний на недопустимые или невозможные для исполнителя действия, а также обращения к входным объектам, если они не принадлежат алгоритмической системе (это же касается и результатов). Язык должен быть точно понят исполнителем.

Если исполнитель алгоритма – человек, то в качестве языка для формулировки алгоритмов вполне можно использовать естественный язык. Зачастую естественный язык служит основой для другого формального языка – метаязыка или псевдокода.

Если исполнитель алгоритма – компьютер, набор действий которого весьма ограничен, то соответственно язык, на котором задаются шаги заданий, значительно беднее, чем язык, адресованный человеку. В то же время такой язык более точен, его предложения не допускают различных толкований и алгоритм, написанный на таком языке, представляет собой последовательность команд (инструкций), адресованных различным устройствам компьютера, совсем не похожую на привычные для нас фразы естественного языка. По своей сути язык компьютера, называемый также машинным языком, формален, а то, что он все-таки назван языком, имеет под собой довольно глубокое основание. Несмотря на несхожесть его с естественным языком, он имеет свой алфавит, свою грамматику.

Алгоритм, адресованный компьютеру как исполнителю, представляет собой некий текст, т.е. некоторую последовательность символов. Этот текст называют

программой, а язык, с помощью которого записывают программу, соответственно называют языком программирования. В связи с этим отметим здесь одно очень важное обстоятельство – если мы имеем дело с алгоритмической системой, предназначенной для составления алгоритмов обработки данных, т.е. любых символьных последовательностей, то открывается принципиально важная возможность составлять алгоритмы, которые в свою очередь будут преобразовывать алгоритмы, обрабатывая тексты, реализующие эти алгоритмы. В частности, именно таким образом системные программы – трансляторы – преобразуют программу, написанную на одном языке программирования, в программу, написанную на другом языке. При решении задач с помощью компьютера автоматическое преобразование алгоритмов из одной формы в другую (как делают, например, трансляторы и компиляторы) или автоматическое изменение алгоритмов в ходе вычислений (как делают, например, интерпретаторы) представляется чрезвычайно важным. Это как раз и создает ту удивительную логическую гибкость, которая превратила компьютер в принципиально новый инструмент обработки данных.

Понятие о математической модели

Мощным инструментом познания мира является математическое моделирование. **Математическая модель** – приближенное описание какого-либо класса явлений внешнего мира, выраженное с помощью математической символики. Чтобы описать явление, необходимо выявить самые существенные его свойства, закономерности, внутренние связи, роль отдельных характеристик явления. Выделив наиболее важные факторы, влияющие на происходящие процессы, можно пренебречь менее существенными. Тем самым несколько упрощается и огрубляется модель явления.

Учет важнейших факторов, влияющих на изучаемое явление, и взаимосвязей этих факторов производится с помощью математической символики, как правило, в виде уравнений и неравенств. Получившиеся математические соотношения в совокупности с некоторыми известными исходными данными и образуют математическую модель явления.

Итак, математическая модель представляет собой аналог явления, сохраняющий его существенные черты и служащий для его изучения.

Процесс построения математической модели обычно состоит из нескольких взаимосвязанных этапов, каждый из которых, как в целом и сам процесс, по своей сути являются итеративными.

Первый этап состоит в том, что выделяются основные закономерности, характеризующие моделируемое явление. Эти закономерности зачастую носят гипотетический характер. Затем закономерности облачаются в математическую форму. Однако на этом этапе возникает целый ряд сложностей. С одной стороны, хотелось бы выявить и формализовать все факторы, влияющие на изучаемое явление. С другой стороны, запись их в виде уравнений и неравенств обычно сопряжена с большими сложностями. Дело в том, что информацию для составления математических соотношений, как правило, приходится черпать из наблюдений.

Результаты же наблюдений всегда сопровождаются некоторыми погрешностями, которые тем самым приносятся и в математические соотношения. Как следствие, может оказаться, что решение одних уравнений или неравенств, характеризующих данное явление, не удовлетворяет другим соотношениям той же самой модели. Конечно же, такая модель должна быть изменена, т.е. на этом этапе главное – выявить все основные закономерности явления, максимально точно замерить все его параметры, записать все это в математической форме. При этом необходимо следить за тем, чтобы получившиеся соотношения были совместными, т.е. чтобы существовало решение задачи.

Следующий этап состоит в решении поставленной математической задачи. Решив задачу, необходимо проанализировать результаты ее решения. Если окажется, что результаты вычислений не соответствуют наблюдениям, то такая модель требует изменений. Придется воспользоваться какими-то другими гипотезами, написать соответствующие математические соотношения, несколько иначе описывающие исследуемое явление, т.е. на самом деле построить новую математическую модель. Может случиться, что и на этот раз результаты окажутся неудовлетворительными. Тогда опять надо поправить модель – и так далее. Разумеется, все это делается в предположении, что сама математическая задача, возникающая для каждого варианта модели, решается достаточно точно и если возникают расхождения с ожидаемыми результатами, то только из-за недостаточно удачно выбранных гипотез, положенных в основу модели.

В действительности, и само решение математической задачи зачастую вызывает значительные трудности. Поэтому при построении модели необходимо учитывать, к какой математической задаче она сведется, – имеет ли эта задача решение, легко ли получить это решение. Заметим при этом, что решение таких задач, как правило, возможно только с помощью компьютера.

Алгоритмизация

Под *алгоритмизацией* понимают процесс разработки алгоритма решения какой-либо задачи. Переход от содержательной постановки задачи к разработке алгоритма ее решения иногда вызывает большие трудности. Разработчик алгоритма всегда должен четко представлять себе ту алгоритмическую систему, в рамках которой составляется алгоритм. Если речь идет о составлении алгоритма для компьютера, необходимо хорошо представлять возможности исполнителя алгоритма и тщательно их учитывать.

Процесс алгоритмизации по своей методике очень близок к процессу построения математических моделей. Мало того, при построении собственно формального математического описания явления и задач, которые нужно решать в связи с анализом такого явления, необходимо прямо учитывать возможность построения такого алгоритма решения, который мог бы выполнить компьютер. В этом смысле математическое моделирование и процесс алгоритмизации теснейшим и неразрывным образом связаны между собой.

Несмотря на то, что разработка каждого нового алгоритма требует своего собственного подхода, тем не менее, есть некоторые общие приемы и этапы этого рода деятельности.

Первый необходимый этап разработки алгоритма может быть охарактеризован как содержательный анализ задачи. На этом этапе следует вникнуть в суть задачи, выяснить, что дано и что требуется получить. Пожалуй, самое важное – это оценить множество исходных данных, ведь речь, как правило, идет о разработке алгоритма, обладающего свойством массовости. Первый этап алгоритмизации, состоящий в анализе задачи, отвечает нам на вопрос: может ли быть задача решена вообще и при каких исходных данных мы можем получить имеющий смысл результат.

Второй этап алгоритмизации состоит в точной постановке задачи или в построении математической модели исходной задачи.

Третий этап алгоритмизации можно назвать анализом возможностей исполнителя или более точно – анализом алгоритмической системы, в которой мы будем строить алгоритм.

Следующим этапом алгоритмизации является разработка идеи алгоритмического процесса и анализа этой идеи. И, наконец, на завершающем этапе осуществляется кодирование каждого шага алгоритмического процесса с помощью языка, понятном исполнителю алгоритма.

Типы алгоритмов


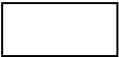

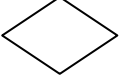

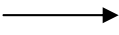
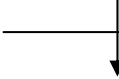

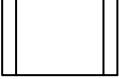
Рассмотрим классические типы алгоритмов – *линейный*, *условный* (или *разветвляющийся*), *циклический*, *рекурсивный* и *эвристический*. Для каждого типа алгоритма представим его описание как на естественном языке, так и на языке блок-схем. При этом описание алгоритма на естественном языке представим как в традиционном, так и в структурированном видах.

Следует заметить, что только на первоначальном этапе знакомства с различными типами алгоритмов автор вынужден прибегать ко всем выше перечисленным способам их описания, в дальнейшем при разработке алгоритмов предпочтение будет отдано лишь их описанию на естественном языке, как правило, в структурированном виде и, как исключение, в традиционном виде. Следует также заметить, что при описании алгоритмов можно использовать структурограммы (диаграммы Насси-Шнайдермана) либо словесно-формульную запись в стиле языка математических формул или запись в стиле какого-либо языка программирования.

При описании алгоритмов на естественном языке автор придерживается следующих принципов:

- порядковая нумерация действий алгоритма;
- структуризация номера для структурированного описания алгоритма;
- объявление константы с помощью глаголов “*определить*” и “*положить*”;
- инициализация переменной с помощью глаголов “*ввести*” и “*положить*”;
- визуализация значений выражений с помощью глагола “*вывести*”.

При описании алгоритмов на языке блок-схем автор придерживается следующих графических символов:

	Пограничные точки – начало и конец исполнения
	Операция общего вида
	Ввод и Вывод
	Разветвление
	Линия передачи управления – сверху вниз и слева направо
	Линия передачи управления – снизу вверх и справа налево
	Схождение
	Точка перехода
	Вызов подпрограммы

Линейный алгоритм

Линейный алгоритм – это линейная (однонаправленная) последовательность действий, т.е. в линейных алгоритмах допускается только последовательная передача управления.

В качестве примера разработки линейного алгоритма рассмотрим задачу вычисления значения арифметического выражения $ax^2 + bx + c$. Прежде, чем приступать к разработке какого-либо алгоритма, обсудим некоторые характерные особенности этапа его проектирования.

В общем случае для решения этой задачи можно разработать, например, три различных алгоритма, которые в зависимости от способа объявления исходных данных a , b , c , x и способа представления результата вычисления арифметического выражения $ax^2 + bx + c$ по-разному определяют интерфейс, связанный с объявлением и инициализацией этих объектов.

Например, для первого алгоритма объекты a , b , c и x пусть все будут константами, для второго – только объекты a , b и c пусть будут константами, а объект x пусть будет переменной, для третьего – объекты a , b , c и x пусть все будут переменными.

В первом случае результат вычисления арифметического выражения $ax^2 + bx + c$ может быть значением как константы, так и переменной. Использование такого алгоритма приведет к вычислению только одного значения арифметического выражения, чтобы получить другое значение этого выражения, необходимо осуществить переопределение значений констант. Если результат вычисления арифметического выражения должен быть значением константы, то при разработке алгоритма кроме арифметических операций могут использоваться либо операции объявления и вывода, либо – только вывода. Если этот результат должен быть значением переменной, то теперь к уже обязательным операциям объявления и вывода необходимо добавить еще и операцию присваивания.

Во втором и в третьем случаях результат вычисления арифметического выражения $ax^2 + bx + c$ может быть только значением переменной, при этом использование этих алгоритмов может привести к вычислению совокупности значений – либо функции $f(x)$, либо функции $f(a, b, c, x)$.

Остановим свой выбор, например, на втором алгоритме, когда объекты a , b и c являются константами, а объект x является переменной. При разработке этого алгоритма, как и в дальнейшем всех последующих, будем использовать только необходимый минимум операций. Например, здесь можно воспользоваться только операциями объявления и ввода-вывода.

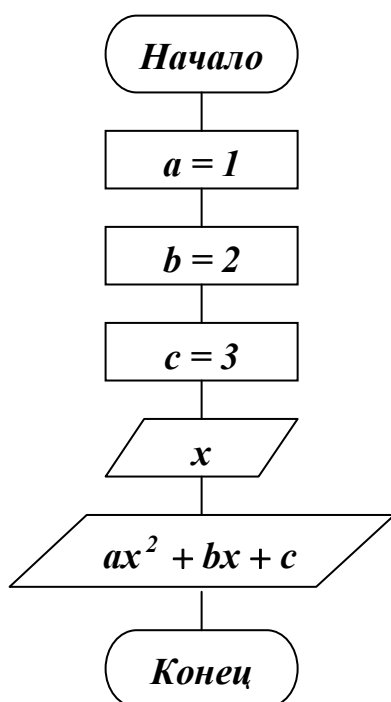
Чтобы ответить на вопрос, почему даже при таком выборе операций результат вычисления арифметического выражения $ax^2 + bx + c$ может быть только значением переменной, следует напомнить, что в программной реализации этого алгоритма при выполнении операции вывода для вычисления значения арифметического выражения будет использоваться временный объект, обладающий

свойствами переменной и хранящийся во вспомогательной памяти, называемой стеком времени выполнения программы.

Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $a = 1$*
3. *Положить $b = 2$*
4. *Положить $c = 3$*
5. *Ввести значение x*
6. *Вывести значение $ax^2 + bx + c$*
7. *Конец*

А теперь представим описание алгоритма на языке блок-схем:



Условные алгоритмы

Условные алгоритмы (алгоритмы выбора или алгоритмы ветвления) позволяют организовать выполнение альтернативных действий. В общем случае существуют две разновидности условных алгоритмов, которые отличаются друг от друга как количеством, так и механизмом исполнения альтернативных действий.

Если выполнение действий алгоритма зависит от одного из двух возможных значений некоторого логического выражения, иногда называемого условием, то возможны две ветви алгоритма – одна для значения “истина”, а другая для значения “ложь”. Каждая из двух ветвей такого алгоритма в общем случае обладает свойством независимости и в свою очередь может состоять из последовательности

действий. В зависимости от значения логического выражения управление в таком условном алгоритме передается одной из двух ветвей.

Если выполнение действий алгоритма зависит от одного из множества заданных значений некоторого выражения, обладающего свойством упорядочивания конечного числа всех своих возможных значений и называемого иногда переключателем, то возможно множество ветвей алгоритма – одни для одного значения переключателя, а другие для нескольких его значений. Каждая из множества ветвей такого алгоритма является независимой и в свою очередь может состоять из последовательности действий. В зависимости от значения переключателя управление в таком условном алгоритме в общем случае передается какой-либо одной из множества ветвей. При этом может выполняться либо только одно альтернативное действие, связанное с соответствующей ему ветвью, либо несколько альтернативных действий подряд друг за другом, начиная с того, на чью соответствующую ему ветвь передается управление. Как исключение, возможен и такой случай, когда значение переключателя не позволяет передать управление ни одной из множества ветвей.

В качестве примера разработки условного алгоритма с двумя ветвями рассмотрим задачу вычисления значения модуля числа $|x| = \begin{cases} x, & \text{если } x \geq 0 \\ -x, & \text{если } x < 0 \end{cases}$

При разработке этого алгоритма рассмотрим случай, когда объект x является переменной. При этом будем придерживаться принятого соглашения о необходимом минимуме операций.

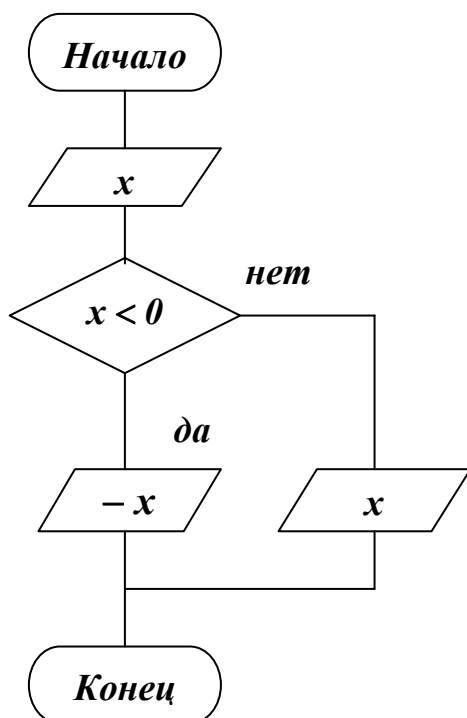
Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Если $x < 0$, то перейти к пункту 4, иначе перейти к пункту 6*
4. *Вывести значение $-x$*
5. *Конец*
6. *Вывести значение x*
7. *Перейти к пункту 5*

Представим теперь структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Если $x < 0$*
то
 - 3.1. *Вывести значение $-x$*
 - 3.2. *Вывести значение x**иначе*
4. *Конец*

А теперь представим описание алгоритма на языке блок-схем:



В качестве примера разработки условного алгоритма с множеством ветвей рассмотрим задачу вычисления значения функции $y = \begin{cases} a + x & \text{при } a = 1 \\ (a + x)^2 & \text{при } a = 2 \\ (a + x)^3 & \text{при } a = 3 \end{cases}$

При разработке этого алгоритма рассмотрим случай, когда объект a , называемый переключателем, и объект x оба являются переменными. При этом по-прежнему будем придерживаться принятого соглашения о необходимом минимуме операций.

Заметим, что как при традиционном описании алгоритмов на естественном языке, так и при описании их на языке блок-схем нет средств, чтобы представить этот алгоритм именно как условный алгоритм с множеством ветвей. Однако это становится возможным в случае структурированного описания алгоритма на естественном языке.

1. **Начало**
2. **Ввести значение x**
3. **Ввести значение a**
4. **По значению переключателя a выбрать**
 - Ветвь 1 :**
 - 4.1. **Вывести значение $a + x$**
 - Ветвь 2 :**
 - 4.2. **Вывести значение $(a + x)^2$**
 - Ветвь 3 :**
 - 4.3. **Вывести значение $(a + x)^3$**

иначе

4.4. Вывести значение ‘Ошибка!’

5. Конец

Циклические алгоритмы

Циклические алгоритмы позволяют организовать повторное, т.е. неоднократное выполнение какой-либо последовательности действий алгоритма. Заметим при этом, что здесь понятие повторности выполнения последовательности действий алгоритма трактуется в широком смысле – от одного раза или фиксированного числа раз и многократно до тех пор, пока не будет выполнено предписанное условие. Мало того, в ряде случаев эта последовательность действий алгоритма может либо вообще не выполняться, либо выполняться бесконечно много раз.

Циклом принято называть неоднократное выполнение последовательности действий алгоритма. Последовательность действий, которая однократно выполняется в цикле, принято называть **телом цикла**. Переменную, значение которой определяет, выполнять или не выполнять тело цикла, принято называть **переменной цикла**. Однократное выполнение тела цикла принято называть **итерацией**.

В общем случае рассматривают две разновидности циклов – цикл с условием продолжения или так называемый цикл с предусловием и цикл с условием завершения или так называемый цикл с постусловием.

Цикл с условием продолжения

Первым действием тела цикла с условием продолжения является проверка условия продолжения цикла – вычисление значения логического выражения. Выполнение тела цикла осуществляется всякий раз, если значение логического выражения есть “истина”. Чтобы выполнить конечное число итераций, в теле цикла необходимо обеспечить возможность смены значения логического выражения на “ложь”, а до тела цикла – инициализацию переменной цикла. Тело цикла с условием продолжения может либо вообще не выполняться, либо выполняться требуемое число раз, либо выполняться бесконечно много раз.

В качестве примера разработки циклического алгоритма, реализованного в виде цикла с условием продолжения, рассмотрим задачу табулирования функции одного переменного, например, $y = x^2$ для пяти узлов. Здесь, как и в дальнейшем, будем придерживаться принятого соглашения о необходимом минимуме операций.

Представим вначале традиционное описание алгоритма на естественном языке:

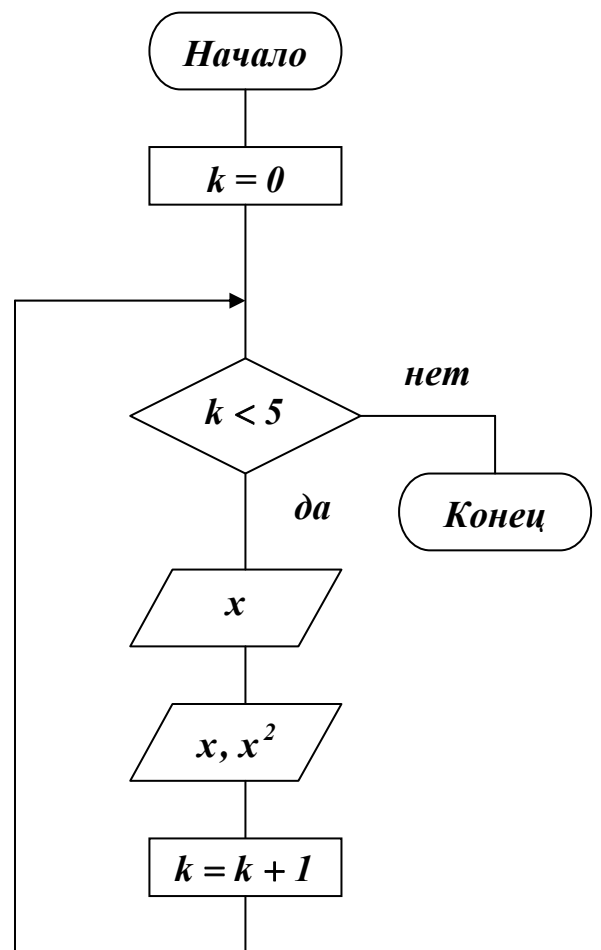
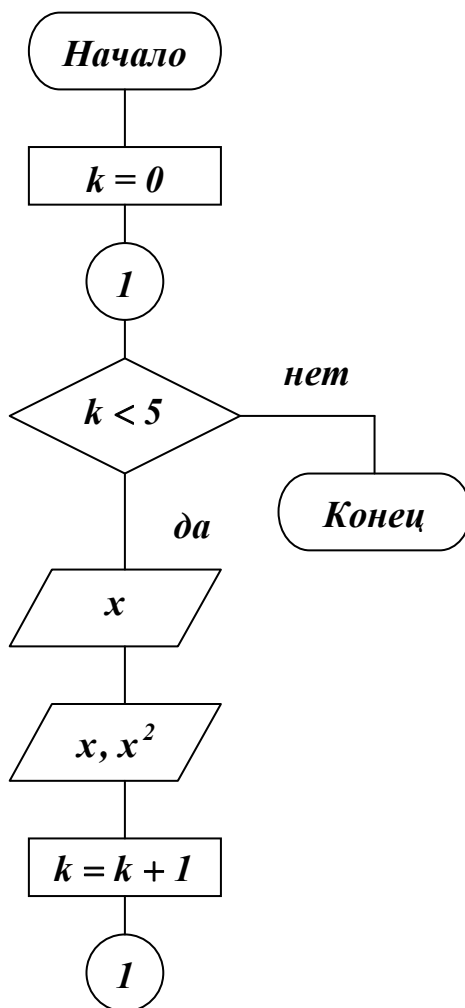
- 1. Начало**
- 2. Положить $k = 0$**
- 3. Если $k < 5$, то перейти к пункту 4, иначе перейти к пункту 8**
- 4. Ввести значение x**

5. Вывести значения x и x^2
6. Положить $k = k + 1$
7. Перейти к пункту 3
8. Конец

Представим теперь структурированное описание алгоритма на естественном языке:

1. Начало
2. Положить $k = 0$
3. Пока $k < 5$ повторить
 - 3.1. Ввести значение x
 - 3.2. Вывести значения x и x^2
 - 3.3. Положить $k = k + 1$
4. Конец

А теперь представим два варианта описания алгоритма на языке блок-схем:



В первом варианте описания алгоритма для организации цикла используется точка перехода, помеченная, например, номером *1*, а во втором – только линии передачи управления.

Частным случаем цикла с условием продолжения является так называемый счетный цикл, тело которого выполняется фиксированное число раз благодаря явному или неявному изменению переменной цикла с постоянным шагом в заданном диапазоне значений либо от начального к конечному, либо от конечного к начальному. Значение шага при этом может быть задано как в явной, так и в неявной форме (так называемое стандартное значение или значение по умолчанию).

В качестве примера разработки циклического алгоритма, реализованного в виде счетного цикла, обратимся к уже известной задаче табулирования функции одного переменного $y = x^2$ для пяти узлов. Заметим, что как при традиционном описании алгоритмов на естественном языке, так и при описании их на языке блок-схем нет средств, чтобы представить этот алгоритм именно в виде счетного цикла. Однако это становится возможным в случае структурированного описания алгоритма на естественном языке.

Представим общую форму структурированного описания алгоритма на естественном языке:

1. *Начало*
2. *От $k = 1$ до $k = 5$ с шагом 1 повторить*
 - 2.1. *Ввести значение x*
 - 2.2. *Вывести значения x и x^2*
3. *Конец*

Как видим, переменная цикла здесь изменяется неявно с шагом, который явно задан единицей. Можно предложить и краткую форму описания этого алгоритма:

1. *Начало*
2. *От $k = 1$ до $k = 5$ повторить*
 - 2.1. *Ввести значение x*
 - 2.2. *Вывести значения x и x^2*
3. *Конец*

А здесь, как видим, переменная цикла изменяется неявно с шагом, который неявно задан единицей.

Теперь следует обратить внимание на одно важное обстоятельство, связанное с указанием свойств объекта, являющегося переменной цикла. Для реализации счетного цикла переменная цикла обязательно должна обладать свойством упорядочивания конечного числа всех своих возможных значений. В дальнейшем станет известно, что такие объекты будут называться объектами порядкового типа. Например, переменная цикла может относиться к объектам целого типа, что означает, что ее значения принадлежат диапазону целых чисел.

Для реализации же цикла с условием продолжения, как и цикла с условием завершения, переменная цикла не обязательно должна обладать свойством упорядочивания конечного числа всех своих возможных значений. Например, переменная цикла может относиться как к объектам целого типа, так и к объектам вещественного типа, что означает, что ее значения могут принадлежать диапазону как целых, так и действительных чисел. При этом следует помнить, что выполнение вычислений в арифметике действительных чисел всегда сопряжено с ошибками округления, что в принципе может привести к потере одной итерации.

Цикл с условием завершения

Последним действием тела цикла с условием завершения является проверка условия завершения цикла – вычисление значения логического выражения. Выполнение тела цикла осуществляется всякий раз, если значение логического выражения есть “ложь”. Чтобы реализовать конечное число итераций, в теле цикла необходимо обеспечить возможность смены значения логического выражения на “истину”, а до тела цикла – инициализацию переменной цикла. Тело цикла с условием завершения может либо выполниться только один раз, либо выполниться требуемое число раз, либо выполняться бесконечно много раз.

В качестве примера разработки циклического алгоритма, реализованного в виде цикла с условием завершения, рассмотрим уже известную задачу табулирования функции одного переменного $y = x^2$ для пяти узлов.

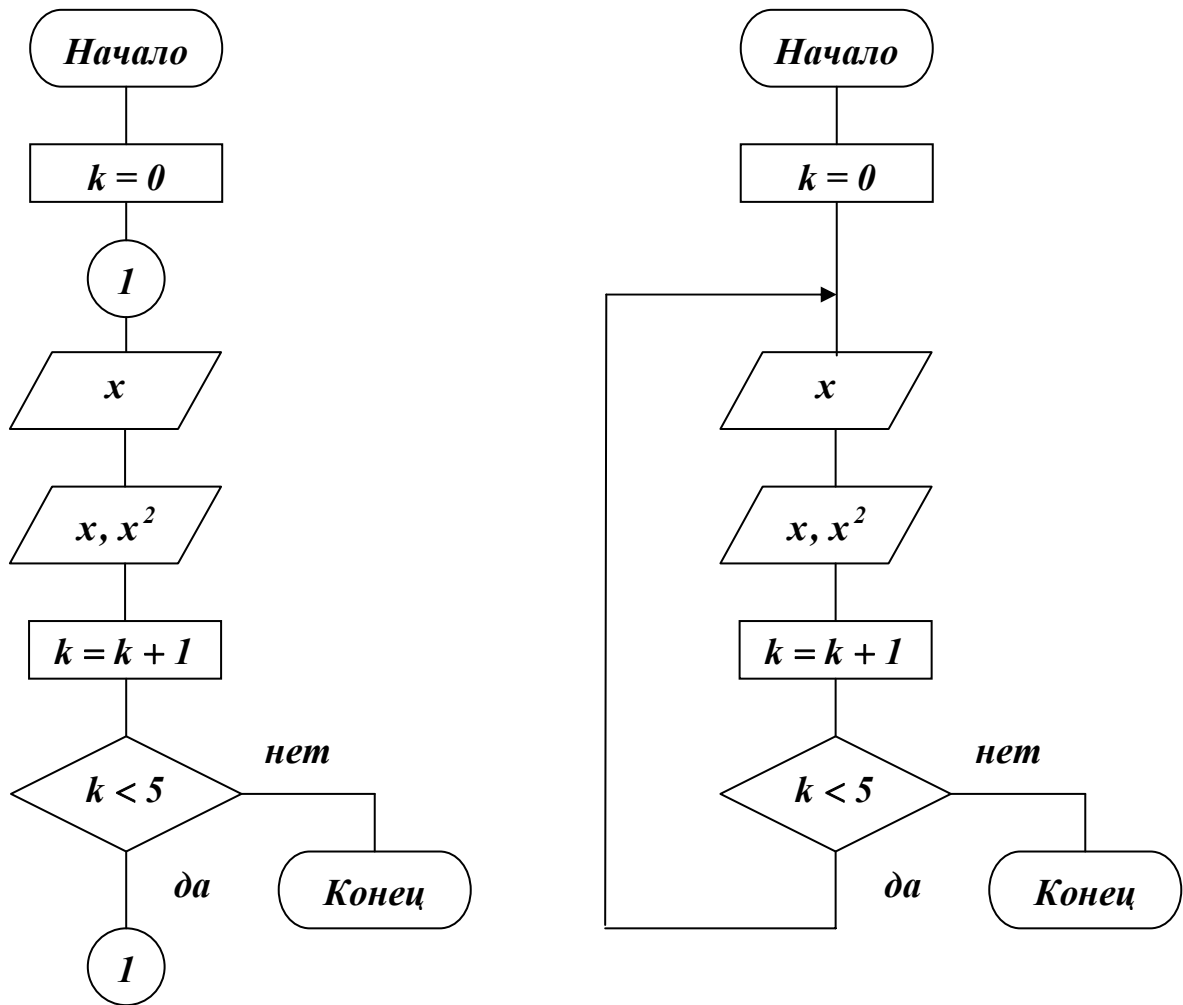
Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $k = 0$*
3. *Ввести значение x*
4. *Вывести значения x и x^2*
5. *Положить $k = k + 1$*
6. *Если $k < 5$, то перейти к пункту 3, иначе перейти к пункту 7*
7. *Конец*

Представим теперь структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $k = 0$*
3. *Повторить*
 - 3.1. *Ввести значение x*
 - 3.2. *Вывести значения x и x^2*
 - 3.3. *Положить $k = k + 1$**пока не будет $k = 5$*
4. *Конец*

А теперь представим два варианта описания алгоритма на языке блок-схем:



И в заключение обсудим весьма важную проблему, связанную с так называемым “зацикливанием” алгоритмов, реализованных в виде циклов с условием продолжения или завершения. В предложенных реализациях этих алгоритмов условием выхода из цикла является значение переменной цикла k , равное 5 . Переменная цикла изменяется от 0 до 5 с шагом, равным 1 , что позволяет выполнить требуемое число итераций, именно 5 , начиная с первой при $k = 0$ и заканчивая последней при $k = 4$. В общем случае это означает, что если требуется выполнить n итераций, то переменная цикла должна изменяться от 0 до n с шагом, равным 1 , чтобы условием выхода из цикла всегда было значение $k = n$.

Рассмотрим теперь случай, когда переменная цикла изменяется от 1 до n с шагом, равным 1 . Чтобы выполнить n итераций, условием выхода из цикла должно быть значение переменной цикла, равное $n + 1$. Однако, теперь операция изменения значения переменной цикла $k = k + 1$ становится в некотором роде спусковым механизмом “зацикливания” алгоритма, если значение $n + 1$ не будет следующим по порядку за значением n . Такое бывает возможным, например, при программировании такого алгоритма, если значение n является верхней границей некоторого выбранного диапазона целых чисел.

Если значение шага по изменению переменной цикла будет отличным от 1 , то необходимо заранее до реализации алгоритма вычислить такое допустимое верхнее значение этой переменной, чтобы оно обязательно было следующим по порядку за своим предыдущим значением. Именно это верхнее значение переменной цикла и должно проверяться в условии выхода из цикла.

Рекурсивные алгоритмы

Рекурсия – это такой способ организации алгоритма, когда он явно или неявно вызывает сам себя. Напомним, что какой-либо алгоритм может быть вызван лишь в том случае, если он организован либо как процедура, либо как функция. Способ организации алгоритма в виде процедуры или функции предполагает наличие у него заголовка, состоящего из имени, за которым в круглых скобках может следовать список параметров. Вызвать алгоритм – означает обратиться к нему по его имени с указанием, если это необходимо, списка аргументов вызова.

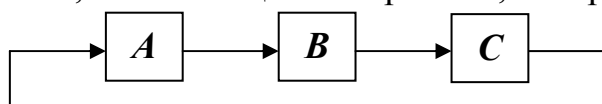
Явный вызов алгоритмом самого себя принято называть **прямой рекурсией**. Одним из классических примеров прямой рекурсии является функция для вычисления факториала неотрицательного целого числа:

$$n! = \begin{cases} 1 & \text{при } n = 0, \\ n \cdot (n-1)! & \text{при } n > 0. \end{cases}$$

Другим ярким примером является последовательность чисел Фибоначчи, в которой каждое число, начиная с третьего, является суммой двух предыдущих:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \quad \text{при } n > 1. \end{aligned}$$

Неявный или опосредованный вызов алгоритмом самого себя принято называть **косвенной рекурсией**. Для реализации косвенной рекурсии необходимо наличие некоторой последовательности вложенных друг в друга вызовов одними алгоритмами других, в которой какой-нибудь алгоритм последнего уровня вложения вызывает какой-либо алгоритм предыдущего уровня вложения. Глубина вложения вызовов при этом может быть произвольной. Например, рассмотрим такую последовательность вложенных вызовов, в которой пусть алгоритм A вызывает алгоритм B , вызывающий алгоритм C , который вновь вызывает алгоритм A :



Для рекурсии в любой ее форме типична необходимость отсрочки некоторых действий. Реализация механизма этих отложенных действий осуществляется при помощи вспомогательной памяти, организованной в виде динамической структуры данных и называемой стеком времени выполнения. Стек организован по принципу **LIFO** (от слов **Last In First Out**), что означает “последний пришел” – “первый ушел”. В стек записываются или как говорят “вталкиваются” (от слова **push**)

аргументы вызова рекурсивных алгоритмов, которые затем извлекаются оттуда в обратном порядке или как говорят “выталкиваются” (от слова *pop*) при выполнении отложенных действий. Операции вталкивания и выталкивания выполняются путем модифицирования значения указателя вершины стека.

В дальнейшем станет известно, что при программировании рекурсивных алгоритмов глубина рекурсии во время выполнения программы будет зависеть от размера стека, а для реализации косвенной рекурсии будет необходим механизм опережающего объявления процедур и функций.

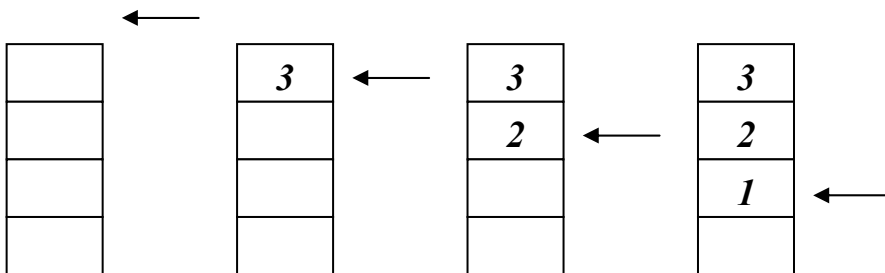
В качестве первого примера разработки рекурсивного алгоритма для случая прямой рекурсии рассмотрим функцию с одним параметром ***Factorial(n)*** для вычисления факториала неотрицательного целого числа. Напомним, что функции возвращают результат в виде значения функции посредством своего имени.

Представим вначале традиционное описание алгоритма на естественном языке:

Функция ***Factorial(n)***

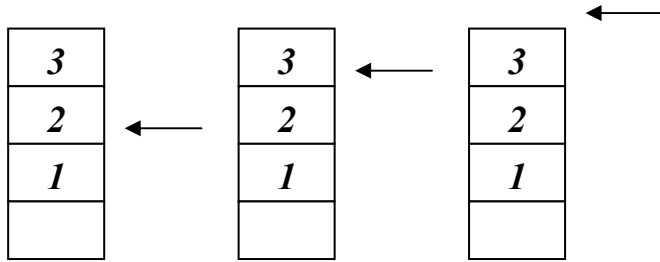
1. **Начало**
2. **Если $n = 0$, то перейти к пункту 3, иначе перейти к пункту 5**
3. **Положить $Factorial = 1$**
4. **Конец**
5. **Положить $Factorial = n \cdot Factorial(n - 1)$**
6. **Перейти к пункту 4**

При вычислении, например, значения $3!$ три раза будет откладываться операция умножения, в результате чего в стек будут вталкиваться друг за другом три аргумента – в первый раз значение 3 , во второй раз значение 2 , а в третий раз значение 1 :



Операция вталкивания представляет собой изменение значения указателя вершины стека и последующую запись элемента данных в стек. Здесь стрелкой показаны позиции указателя вершины стека при вталкивании в стек аргументов вызова функции ***Factorial(n)***.

Затем при $n = 0$ выполнится операция присваивания ***Factorial = 1***, после которой друг за другом будут выполняться отложенные операции умножения, первым операндом которых будут вытолкнутые из стека значения аргументов – в первый раз ***Factorial = 1 \cdot 1 = 1***, во второй раз ***Factorial = 2 \cdot 1 = 2***, а в третий раз ***Factorial = 3 \cdot 2 = 6***:



Заметим, что выталкивание данных из стека не означает, что они из него исчезают. Операция выталкивания представляет собой чтение элемента данных из стека и последующее изменение значения указателя вершины стека. Здесь стрелкой показаны позиции указателя вершины стека после выталкивания очередного элемента данных.

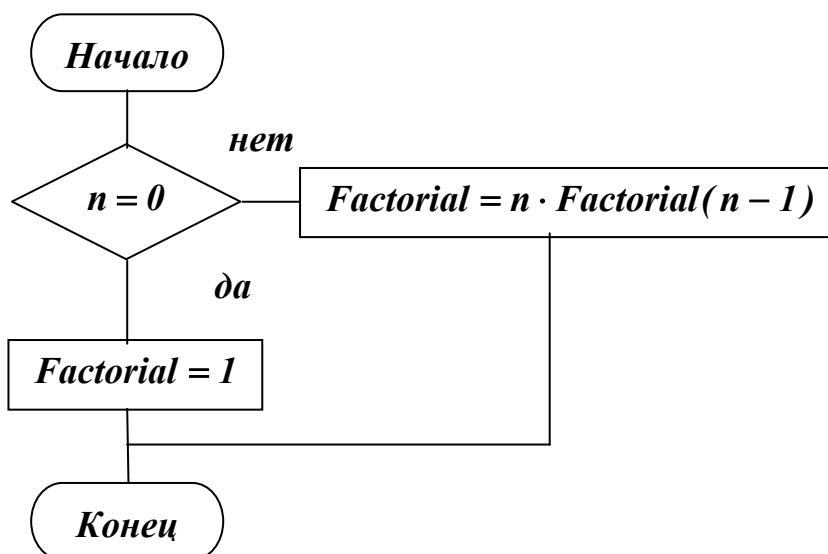
Представим теперь структурированное описание алгоритма на естественном языке:

Функция *Factorial(n)*

1. *Начало*
2. *Если* $n = 0$
то
 - 2.1. Положить $Factorial = 1$
 - иначе
 - 2.2. Положить $Factorial = n \cdot Factorial(n - 1)$
3. *Конец*

А теперь представим описание алгоритма на языке блок-схем:

Функция *Factorial(n)*



В качестве второго примера разработки рекурсивного алгоритма для случая прямой рекурсии рассмотрим функцию с одним параметром *Fibonacci(n)* для вычисления члена последовательности чисел Фибоначчи.

Представим только структурированное описание этого алгоритма на естественном языке:

Функция *Fibonacci(n)*

1. **Начало**
2. **Если $(n = 0)$ или $(n = 1)$**
то
 - 2.1. **Положить $Fibonacci = n$****иначе**
 - 2.2. **Положить $Fibonacci = Fibonacci(n - 1) + Fibonacci(n - 2)$**
3. **Конец**

В завершение разговора о последовательности чисел Фибоначчи уместным будет здесь напомнить, что всякое положительное число *m* можно единственным образом представить в виде суммы чисел Фибоначчи, причем в этом разложении наибольшее число не будет превосходить *m* и никакие два числа не будут соседними членами последовательности чисел Фибоначчи.

И в заключение обсудим одну важную проблему, связанную с эффективностью программных реализаций рассматриваемых рекурсивных алгоритмов. Как видим, рекурсивные алгоритмы ясны и прозрачны для понимания, изящны и совершенны как строгие математические формулы. Однако у рекурсивных алгоритмов есть один существенный недостаток – это большие затраты времени при их исполнении, связанные с издержками по управлению стеком при выполнении отложенных действий. Традиционное решение этой проблемы – это отказ от рекурсии и переход к итеративным реализациям алгоритмов.

В качестве примера такого перехода от рекурсивных алгоритмов к циклическим представим структурированные описания этих алгоритмов на естественном языке, а также обсудим их достоинства и недостатки:

Функция *Factorial(n)*

1. **Начало**
2. **Если $n = 0$**
то
 - 2.1. **Положить $Factorial = 1$****иначе**
 - 2.2. **Положить $p = 1$**
 - 2.3. **От $k = 1$ до $k = n$ повторить**
 - 2.3.1. **Положить $p = p \cdot k$**
 - 2.4. **Положить $Factorial = p$**
3. **Конец**

Функция *Fibonacci(n)*

1. *Начало*
2. *Если ($n = 0$) или ($n = 1$)*
то
 - 2.1. *Положить $Fibonacci = n$**иначе*
 - 2.2. *Положить $a = 0$*
 - 2.3. *Положить $b = 1$*
 - 2.4. *От $k = 2$ до $k = n$ повторить*
 - 2.4.1. *Положить $c = a + b$*
 - 2.4.2. *Положить $a = b$*
 - 2.4.3. *Положить $b = c$*
 - 2.5. *Положить $Fibonacci = c$*
3. *Конец*

Как видим, циклические алгоритмы в отличие от рекурсивных не так ясны и прозрачны для понимания, кроме того, они требуют дополнительных переменных. Зато программные реализации циклических алгоритмов весьма эффективны по времени выполнения. Если у задачи есть две схемы решения – итерационная и рекурсивная, – то предпочтение, как правило, отдается итеративным алгоритмам.

С другой стороны, есть задачи, для решения которых применяются только рекурсивные схемы, так как построение итерационной схемы может оказаться либо слишком сложным, либо не естественным.

Эвристические алгоритмы

Существует определенный класс задач, решение которых возможно лишь на основе применения специальных методов, построенных на проверке всяких догадок, удачных идей или любых разумных способов поиска требуемых величин.

Такие процессы поиска решения задачи называются **эвристическими**, а сами методы – **эвристиками**. Исторически понятие эвристики возникло в Древней Греции как метод обучения, применявшийся Сократом: в ходе беседы ее участники с помощью системы наводящих вопросов и приемов наталкивались учителем на правильный ответ. В современном понимании одно из значений понятия эвристики представляет собой метод или алгоритм выработки правильного решения задачи с помощью неформализованных или самообучающихся методик.

В качестве примера разработки эвристических алгоритмов рассмотрим несколько задач, решение которых основано на применении различных эвристик.

Многим известна задача о перевозчике, который должен перевезти с одного берега реки на другой козу, капусту и волка при условии, что за один раз в лодке можно перевезти либо козу, либо капусту, либо волка. В результате анализа исходных данных эвристика здесь может быть построена на предположении, что на берегу реки нельзя оставлять вместе как козу с капустой, так и волка с козой.

Представим традиционное описание двух алгоритмов на естественном языке:

1. *Начало*
2. *Перевезти козу*
3. *Вернуться обратно*
4. *Перевезти волка*
5. *Вернуться обратно с козой*
6. *Перевезти капусту*
7. *Вернуться обратно*
8. *Перевезти козу*
9. *Конец*

1. *Начало*
2. *Перевезти козу*
3. *Вернуться обратно*
4. *Перевезти капусту*
5. *Вернуться обратно с козой*
6. *Перевезти волка*
7. *Вернуться обратно*
8. *Перевезти козу*
9. *Конец*

Рассмотрим еще одну известную задачу, в которой требуется поделить поровну воду из полного ведра емкостью **8 л** с помощью двух пустых ведер соответственно емкостью **5 л** и **3 л** за минимальное количество переливаний. Существует и другая формулировка задачи, где используются, например, ведра емкостью **10 л**, **7 л** и **4 л**.

В результате анализа исходных данных эвристика здесь может быть построена на предположении, что ведро максимальной емкости должно служить своеобразным накопителем остатков воды из других ведер. Также следует предположить, что перед последней операцией переливания в одном из первых двух ведер должен остаться **1 л** воды, а ведро минимальной емкости при этом должно быть полным.

Представим описание двух алгоритмов переливания воды для ведер, которые назовем, например, **A**, **B** и **C**, где с помощью стрелки будем указывать, из какого ведра в какое переливается вода, а после переливания будем отмечать состояние всех ведер, т.е. количество литров воды в каждом ведре:

	<i>A</i>	<i>B</i>	<i>C</i>		<i>A</i>	<i>B</i>	<i>C</i>
	8	0	0		10	0	0
1. $A \rightarrow B$	3	5	0	1. $A \rightarrow C$	6	0	4
2. $B \rightarrow C$	3	2	3	2. $C \rightarrow B$	6	4	0
3. $C \rightarrow A$	6	2	0	3. $A \rightarrow C$	2	4	4
4. $B \rightarrow C$	6	0	2	4. $C \rightarrow B$	2	7	1
5. $A \rightarrow B$	1	5	2	5. $B \rightarrow A$	9	0	1
6. $B \rightarrow C$	1	4	3	6. $C \rightarrow B$	9	1	0
7. $C \rightarrow A$	4	4	0	7. $A \rightarrow C$	5	1	4
				8. $C \rightarrow B$	5	5	0

Заметим, что здесь каждый алгоритм представлен только одной своей оптимальной версией, конечно же, существуют и другие версии как оптимальные, так и не оптимальные по количеству переливаний.

Рассмотрим теперь задачу о программировании автомата с конечным набором операций и известным начальным состоянием, который позволяет за минимальное количество операций перейти от исходного числа, например, равного нулю, к заданному натуральному числу при помощи, например, двух операций – прибавить единицу и умножить на два.

При построении эвристики здесь необходимо воспользоваться результатом процесса перехода от заданного натурального числа к исходному при помощи набора обратных операций – вычесть единицу и разделить на два. Операция вычитания единицы позволит переходить от нечетных к четным числам, которые затем можно будет делить на два. Зная последовательность выполнения обратных операций и соответственно их тип, легко получить решение этой задачи.

Представим словесно-формульное описание алгоритма для обратного и прямого хода решения, например, для числа **21**:

- | | | |
|----|-----------------------|---------------|
| 1. | <i>Начало</i> | |
| 2. | <i>Вычесть 1</i> | $21 - 1 = 20$ |
| 3. | <i>Разделить на 2</i> | $20 : 2 = 10$ |
| 4. | <i>Разделить на 2</i> | $10 : 2 = 5$ |
| 5. | <i>Вычесть 1</i> | $5 - 1 = 4$ |
| 6. | <i>Разделить на 2</i> | $4 : 2 = 2$ |
| 7. | <i>Разделить на 2</i> | $2 : 2 = 1$ |
| 8. | <i>Вычесть 1</i> | $1 - 1 = 0$ |
| 9. | <i>Конец</i> | |

Обратный ход решения

- | | | |
|----|----------------------|-------------------|
| 1. | <i>Начало</i> | |
| 2. | <i>Прибавить 1</i> | $0 + 1 = 1$ |
| 3. | <i>Умножить на 2</i> | $1 \cdot 2 = 2$ |
| 4. | <i>Умножить на 2</i> | $2 \cdot 2 = 4$ |
| 5. | <i>Прибавить 1</i> | $4 + 1 = 5$ |
| 6. | <i>Умножить на 2</i> | $5 \cdot 2 = 10$ |
| 7. | <i>Умножить на 2</i> | $10 \cdot 2 = 20$ |
| 8. | <i>Прибавить 1</i> | $20 + 1 = 21$ |
| 9. | <i>Конец</i> | |

Прямой ход решения

Как видим, здесь неявно подразумевается, что автомат “знает”, какое число является четным, а какое – нечетным. Чтобы от частного решения задачи перейти к общему, можно воспользоваться, например, стеком времени выполнения, в который во время обратного хода будут вталкиваться мнемонические обозначения прямых операций, а во время прямого хода, выталкивая их из стека, можно формировать строки решения для каждой операции.

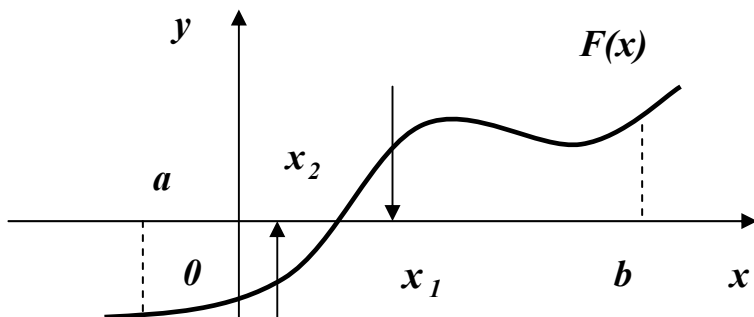
Представим структурированное описание алгоритма на естественном языке для обратного и прямого хода, например, для заданного натурального числа **n**:

1. *Начало*
2. *Пока $n > 0$ повторить*
 - 2.1. *Если n нечетное*
то
 - 2.1.1. *Положить $n = n - 1$*
 - 2.1.2. *Втолкнуть в стек символ +*
 - иначе*

- 2.1.3. Положить $n = \frac{n}{2}$
- 2.1.4. Втолкнуть в стек символ *
3. Положить $p = 0$
4. Положить $r = 0$
5. Пока стек не пустой повторить
 - 5.1. Вытолкнуть из стека оператор
 - 5.2. Если оператор +
 - то
 - 5.2.1. Положить $r = r + 1$
 - 5.2.2. Вывести строку для слагаемых p и 1 и суммы r
 - 5.2.3. Положить $p = r$
 - иначе
 - 5.2.4. Положить $r = r \cdot 2$
 - 5.2.5. Вывести строку для множителей p и 2 и произведения r
 - 5.2.6. Положить $p = r$
6. Конец

Здесь символы $+$ и $*$ являются общепринятыми обозначениями соответственно операторов сложения и умножения.

В заключение рассмотрим задачу отыскания действительных корней нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$:



$$x_1 = \frac{a + b}{2}$$

$$x_2 = \frac{a + x_1}{2}$$

Рассмотрим эвристику, которая лежит в основе метода дихотомии или так называемого метода деления отрезка пополам. Если на границах некоторого отрезка (на рисунке отрезок $[a, b]$) функция $F(x)$ меняет знак, то существует, по крайней мере, одна точка на этом отрезке, в которой функция $F(x)$ обращается в нуль. Если разделить этот отрезок пополам и определить знак функции $F(x)$ в точке деления, то тем самым можно определить другой отрезок, на границах которого функция $F(x)$ меняет знак (на рисунке отрезок $[a, x_1]$). В принципе, повторное применение этого метода, т.е. деление отрезка локализации корня пополам, может позволить сколь угодно близко подойти к корню. Например, на рисунке показана вторая итерация отыскания корня, которая позволяет перейти от отрезка $[a, x_1]$ к отрезку $[x_2, x_1]$.

“В принципе”, потому что существуют погрешности вычислений, т.е. определение знака функции $F(x)$ для какой-либо очередной итерации может быть неверным из-за ошибки округления.

Рассмотрим функцию с тремя параметрами $Dichotomy(a, b, Tolerance)$ для отыскания действительного корня нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$ с заданной точностью $Tolerance$. Для реализации этой функции можно выполнить разработку как циклического, так и рекурсивного алгоритмов.

Представим только структурированное описание циклического алгоритма на естественном языке:

Функция $Dichotomy(a, b, Tolerance)$

1. **Начало**
2. **Положить $x_left = a$**
3. **Положить $f_left = F(x_left)$**
4. **Положить $x_right = b$**
5. **Положить $f_right = F(x_right)$**
6. **Повторить**
 - 6.1. **Положить $x_root = \frac{x_left + x_right}{2}$**
 - 6.2. **Положить $f_root = F(x_root)$**
 - 6.3. **Если $f_left \cdot f_root > 0$**
то
 - 6.3.1. **Положить $x_left = x_root$**
 - 6.3.2. **Положить $f_left = f_root$****иначе**
 - 6.3.3. **Положить $x_right = x_root$****пока не будет $|f_root| < Tolerance$**
7. **Положить $Dichotomy = x_root$**
8. **Конец**

Итерационная процедура деления отрезка локализации корня пополам завершается благодаря выполнению неравенства $|F(x_root)| < Tolerance$, где x_root является найденным значением действительного корня для заданного значения “нуля” $Tolerance$.

Введение в Turbo Pascal

Алфавит языка Turbo Pascal

Набор символов или групп символов, рассматриваемых как единое целое, с помощью которых составляется текст программы, принято называть *алфавитом* языка программирования. В языке Turbo Pascal элементы алфавита строятся из упорядоченного набора символов, каждому из которых соответствует индивидуальный числовой код от 0 до 127 основной таблицы ASCII-кодов персонального компьютера.

Текст программы – это последовательность строк, состоящих из элементов алфавита языка. Максимальная длина строки 127 символов. Текст программы, представляемый в виде текстового файла, является входным текстом или исходным кодом для компилятора языка Turbo Pascal.

К элементам алфавита языка Turbo Pascal относятся:

- буквы:
 - прописные латинские буквы
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 - строчные латинские буквы
a b c d e f g h i j k l m n o p q r s t u v w x y z
 - символ подчеркивания
_
- арабские цифры:
0 1 2 3 4 5 6 7 8 9
- специальные символы:
+ - * / = > < . , ; : @ ' () [] { } # \$ % ^
- составные символы, рассматриваемые как единое целое:
<> <= >= := (* *) (.) ..

Лексическая структура языка Turbo Pascal

Элементы алфавита языка используются для построения базовых смысловых единиц, называемых *лексемами*. Лексический анализатор компилятора языка разделяет входной текст, представляемый в виде потока символов, на лексемы.

К лексемам относятся:

- идентификаторы (имена);
- разделители;
- литералы;
- операторы (знаки операций);
- ключевые (служебные) слова;
- стандартные директивы;
- директивы компилятора.

Идентификаторы

Идентификаторы служат для именования таких объектов программы, как типы, константы, переменные, метки, функции и процедуры. В качестве идентификаторов можно использовать любые последовательности символов, которые удовлетворяют следующим ограничениям:

- идентификатор может состоять из букв и цифр;
- идентификатор не может начинаться с цифры;
- идентификатор не может совпадать ни с одним из ключевых слов;
- идентификаторы различаются по первым 63 символам.

Разделители

К разделителям относятся:

- символ пробел (ASCII-код 32);
- управляющие символы (ASCII-коды от 0 до 31);
- комментарий.

Основное назначение разделителей – разделять в тексте программы ключевые слова и идентификаторы, разделять слова в строках текстовых файлов, а также закрывать эти строки или сами эти файлы.

Комментарии заключаются либо в скобки { }, либо в скобки вида (* *) и могут быть как однострочными, так и многострочными.

Литералы

Литералы используются для обозначения в тексте программы чисел, символов и символьных строк, а также ASCII-кодов. Например:

1	–1	десятичный литерал;
\$1F		шестнадцатеричный литерал;
0.001	–0.1	литерал с фиксированной точкой;
1E+2	–1E–3	литерал с плавающей точкой;
'A'		символьный литерал (символ A);
^C		символьный литерал (символ ETX или ASCII-код 3);
'Hello'		строковый литерал;
#65		символьный литерал (ASCII-код символа A);

Операторы

В языках программирования операторы или так называемые знаки операций используются для указания операций над объектами программы. К сожалению, в русскоязычных учебных пособиях по программированию термин “оператор” связывали и теперь иногда продолжают связывать не с операциями, а с синтаксическими единицами языка – предложениями (от слова *statement*), – которые

теперь в соответствии с их первоначальным смыслом при переводе стандарта современных языков программирования с английского на русский язык уже стали называть инструкциями. Причину такого положения можно объяснить тем, что для языков низкого уровня термин “инструкция”, как и его синоним “команда”, ранее связывали с командой ассемблера, поэтому термин “statement” для языков высокого уровня стали переводить как “оператор”, а не как “инструкцию”.

В современных языках программирования большинство операторов допускает перегрузку, синтаксически и семантически напоминающую (пере)определение функций, поэтому автор оставляет за собой право все, что имеет отношение к термину “statement” в языке Turbo Pascal, называть инструкциями, а не операторами.

В языке Turbo Pascal представлены только два класса операторов – унарные и бинарные. Например, к унарным операторам относятся:

- +** унарный плюс;
- унарный минус (операция смены знака);
- @** адрес (операция получения адреса объекта программы);
- not** логическая операция *not* (отрицание).

Например, к бинарным операторам целочисленной арифметики относятся:

- +** операция сложения;
- операция вычитания;
- *** операция умножения;
- div** операция деления;
- mod** операция по модулю (получение остатка операции деления);
- and** логическая операция *and* (операция **И** или конъюнкция).

Например, к бинарным операторам вещественной арифметики относятся:

- +** операция сложения;
- операция вычитания;
- *** операция умножения;
- /** операция деления.

Более подробное описание операторов будет дано вместе с типами данных.

Ключевые слова

Приведем список ключевых слов языка Turbo Pascal 7.0:

and	downto	inherited	procedure	type
array	else	interface	program	unit
asm	end	label	record	until
begin	file	mod	repeat	uses
case	for	nil	set	var
const	function	not	shl	while
constructor	goto	object	shr	with
destructor	if	of	string	xor
div	implementation	or	then	
do	in	packed	to	

Ключевые слова можно использовать только по своему прямому назначению и их нельзя переопределять, т.е. использовать в качестве идентификаторов.

Стандартные директивы

Стандартные директивы первоначально связаны с некоторыми стандартными объявлениями в программе. Пользователь может переопределить любую стандартную директиву. Ключевые слова и стандартные директивы принадлежат к так называемым зарезервированным словам.

Приведем список стандартных директив языка Turbo Pascal:

absolute	far	interrupt	public
assembler	forward	near	virtual
external	inline	private	

Директивы компилятора

Директивы компилятора представляют собой особым образом оформленные комментарии в тексте программы и используются для управления компилятором в процессе компиляции программы. Директивы могут быть переключающими, параметрическими и условными, а в зависимости от области видимости – глобальными или локальными.

Переключающие директивы воздействуют на те опции компилятора, которые включены в диалоговое окно *Options/Compiler*, например, директивы *{ $\$N+$ }* и *{ $\$N-$ }* позволяют использовать или не использовать числовой сопроцессор для операций вещественной арифметики.

Параметрические директивы задают параметры, которые должен учитывать компилятор, например, директива *{ $\$L$ имя-файла}*, позволяет при компоновке программы включить код объектного файла *имя-файла*, связанного с внешней ассемблерной процедурой или функцией, которая объявлена в тексте программы с помощью стандартной директивы *external*.

Условные директивы определяют условия, при которых компилируются те или иные фрагменты программы, например, директива *{ $\$Define$ условный_символ}* устанавливает переменную *условный_символ*, которая затем будет управлять компиляцией какого-либо фрагмента программы.

Структура программы Turbo Pascal

Программа Turbo Pascal – это набор единиц трансляции, комбинируемых посредством компоновки. Единица трансляции, которую часто называют исходным файлом, – это последовательность объявлений. В общем случае программа Turbo Pascal состоит из двух частей – раздела объявлений и объявления блока программы. Объявление – это один из видов инструкций языка Turbo Pascal.

Раздел объявлений может состоять из заголовка программы, подраздела объявления глобальных директив компилятора, подраздела объявления модулей программы, подраздела объявления глобальных типов, определяемых пользователем, подраздела объявления глобальных констант, подраздела объявления глобальных переменных, подраздела объявления глобальных меток, подраздела объявления функций пользователя и подраздела объявления процедур пользователя. В разделе объявлений после заголовка программы первые два подраздела должны следовать друг за другом в указанном порядке, остальные подразделы могут следовать друг за другом в произвольном порядке, повторяясь при необходимости сколь угодно раз.

Для объявления блока программы используется составная инструкция языка Turbo Pascal, с помощью которой здесь обозначается начало и конец тела программы. В дальнейшем будут рассматриваться еще и другие инструкции, из которых можно составлять тело программы Turbo Pascal. Отправной точкой при этом будет программирование всех рассматриваемых в пособии типов алгоритмов.

Представим структуру программы Turbo Pascal:

```
{ Раздел объявлений }  
program имя_программы;           { заголовок программы }  
  { Подраздел объявления глобальных директив компилятора }  
  { Подраздел объявления модулей программы }  
  { Подраздел объявления глобальных типов пользователя }  
  { Подраздел объявления глобальных констант }  
  { Подраздел объявления глобальных переменных }  
  { Подраздел объявления глобальных меток }  
  { Подраздел объявления функций пользователя }  
  { Подраздел объявления процедур пользователя }  
{ Объявление блока программы }  
begin  
  { Тело программы }  
end.
```

Такая структура обязательна для любой программы Turbo Pascal, что является следствием строгого контроля типов данных языка: любой нестандартный идентификатор, используемый в программе, должен быть предварительно объявлен в своем соответствующем подразделе объявлений. Объявление нестандартного идентификатора программы зачастую означает явное или неявное указание типа связанного с ним объекта программы, например, в случае объявления типа, определяемого пользователем, константы, переменной или функции пользователя. Объявление же стандартного идентификатора означает его переопределение.

Понятие типа является одним из фундаментальных понятий языка Turbo Pascal. Тип определяет диапазон возможных значений данных, допустимые операции, применимых к этим данным, и способ хранения этих данных в памяти. Подробное описание типов данных будет дано позднее в настоящей главе.

Демонстрационная программа Turbo Pascal

Пример 1. Представим первую демонстрационную программу:

```
{ Пример 1 }  
{ Демонстрационная программа Turbo Pascal }  
{ My_First_Program – это имя программы }  
program My_First_Program;  
{ text – это константа типа string (текстовая константа) }  
const text = 'Я программирую на языке Turbo Pascal';  
begin  
  writeln(text);  
  writeln('I love Turbo Pascal')  
end.
```

Результат работы программы:

```
Я программирую на языке Turbo Pascal  
I love Turbo Pascal
```

А теперь проанализируем форму представления текста программы:

- Расположение текста по строкам – дело вкуса, а не требование синтаксиса языка Turbo Pascal.
- Символ точка с запятой является разделителем, который отмечает конец инструкции. Использование этого разделителя позволяет располагать несколько инструкций в одной строке текста программы. В дальнейшем при обсуждении пустых инструкций языка Turbo Pascal будет сказано, в каких случаях можно опустить этот разделитель.
- Символ пробел используется как разделитель лексем языка Turbo Pascal.
- Игнорируется различие между прописными и строчными буквами (кроме символьных и строковых литералов). Например, слово **program** можно записать как **Program** или как **PROGRAM**.

Рассмотрим теперь смысловое содержание строк программы:

- Фигурные скобки { } заключают в себе комментарий, который может быть как однострочным, так и многострочным.
- Слова **program**, **const**, **begin** и **end** – это ключевые слова языка Turbo Pascal.
- Слово **program** используется для объявления имени программы. Объявление имени программы является необязательным.
- Слово **const** открывает подраздел объявления глобальных констант. Объявление константы связано с указанием ее имени, типа (если это необходимо) и значения. Здесь тип константы определяется неявно.
- Строковые литералы – это тоже текстовые константы.

- Слово ***begin*** открывает объявление блока программы, а слово ***end*** закрывает это объявление. Символ точка – признак конца текста программы.
- В теле программы представлены всего две инструкции вызова стандартной процедуры вывода ***writeln***. Параметром этой процедуры при каждом ее вызове является текстовая константа. По умолчанию устройством вывода является экран дисплея.

Первое знакомство с организацией ввода-вывода

В языке Turbo Pascal, как и в языке Pascal, вообще нет инструкций ввода-вывода. Для обмена информацией между основной памятью персонального компьютера и различными его стандартными устройствами используются стандартные процедуры ввода-вывода – ***read*** (от слова ***read*** – прочитать), ***readln*** (от слов ***read*** и ***line*** – прочитать и перейти к новой строке), ***write*** (от слова ***write*** – записать), ***writeln*** (от слов ***write*** и ***line*** – записать и перейти к новой строке), ***blockread*** (от слов ***block*** и ***read*** – прочитать блок данных) и ***blockwrite*** (от слов ***block*** и ***write*** – записать блок данных).

Понятие процедуры – одно из центральных понятий языка Turbo Pascal. Процедура – это некоторая последовательность инструкций языка Turbo Pascal, к которой можно обратиться по имени. Стандартные процедуры не нуждаются в предварительном объявлении, они доступны в любом месте программы. Имя стандартной процедуры не является ключевым словом языка Turbo Pascal, следовательно, пользователь может использовать свою процедуру ввода-вывода.

Обсудим здесь лишь некоторые характерные особенности рассматриваемых стандартных процедур ввода-вывода. Подробное обсуждение процедур и функций, предназначенных для управления обменом информацией, будет проведено в главе, посвященной работе с файлами.

Во-первых, стандартные процедуры ввода-вывода – одни из немногих, при вызове которых допускается использование произвольного количества фактических параметров. Параметры процедур передаются в виде списка, элементы которого разделяются знаком препинания запятой.

Во-вторых, любой обмен информацией реализован как работа с файлами, которые доступны либо только для чтения, либо только для записи, либо для чтения и записи одновременно. Под файлом понимается либо именованная область внешней памяти персонального компьютера, либо логическое устройство ввода-вывода. Все файлы подразделяются на два класса – файлы последовательного доступа и файлы прямого доступа. В стандартных процедурах ввода-вывода для файлов последовательного доступа ***read (readln)*** и ***write (writeln)*** первым элементом списка фактических параметров может быть переменная файлового типа или так называемая файловая переменная. Если первый элемент списка фактических параметров не является переменной файлового типа, по умолчанию для файлов последовательного доступа устройством ввода является клавиатура, а устройством вывода – экран дисплея. Логическое имя этих физически различных устройств – ***con*** (от слова ***console*** –

консоль). В стандартных процедурах ввода-вывода для файлов прямого доступа **blockread** и **blockwrite** первым элементом списка фактических параметров обязательно должна быть переменная файлового типа, так как эти процедуры предназначены только для работы с диском по каналу прямого доступа.

В-третьих, до первого вызова процедуры ввода-вывода должно произойти явное или неявное назначение переменной файлового типа файлу, т.е. либо файлу с его путевым именем, либо логическому устройству ввода-вывода, а также явное указание направления обмена информацией – чтение, запись или чтение и запись одновременно. Назначение переменной файлового типа файлу осуществляется при помощи стандартной процедуры **assign** (от слова **assign** – назначать). Указание направления обмена информацией осуществляется либо при помощи стандартных процедур инициации **reset** (от слова **reset** – сбросить), **rewrite** (от слова **rewrite** – перезаписать) и **append** (от слова **append** – присоединять), либо благодаря выбору соответствующей стандартной процедуры ввода-вывода.

В-четвертых, любой программе на языке Turbo Pascal доступны два предварительно объявленных файла со стандартными переменными файлового типа: **input** – для ввода информации с клавиатуры и **output** – для вывода информации на экран дисплея. При этом пользователь в программе по своему усмотрению может либо переопределить назначение этих переменных, либо объявить и назначить их вновь.

В заключение приведем несколько примеров, иллюстрирующих особенности работы с файлами последовательного доступа при программировании вывода информации на различные устройства вывода – консоль и принтер.

Пример 2. Неявное указание консоли устройством вывода было уже представлено в первой демонстрационной программе. Теперь это, например, можно продемонстрировать с помощью следующей программы:

```
{ Пример 2 }  
{ Неявное указание консоли устройством вывода }  
begin  
  { вывод текстовой константы на экран дисплея }  
  writeln('I love Turbo Pascal')  
end.
```

Пример 3. Явное указание консоли устройством вывода при помощи стандартного имени переменной файлового типа **output**, например, можно продемонстрировать с помощью следующей программы:

```
{ Пример 3 }  
{ Явное указание консоли устройством вывода }  
begin  
  { вывод текстовой константы на экран дисплея }  
  writeln(output, 'I love Turbo Pascal')  
end.
```

Пример 4. Явное указание принтера устройством вывода при помощи имени переменной файлового типа *lst* (от слова *listing* – распечатка), объявленной в стандартном системном модуле *Printer*, например, можно продемонстрировать с помощью следующей программы:

```
{ Пример 4 }  
{ Явное указание принтера устройством вывода }  
{ объявление системного модуля Printer }  
uses Printer;  
begin  
    { вывод текстовой константы на принтер }  
    writeln(lst, 'I love Turbo Pascal')  
end.
```

Пример 5. Явное указание принтера устройством вывода при помощи стандартного имени переменной файлового типа *output* после переопределения ее назначения с логического имени консоли *con* на логическое имя печатающего устройства *prn* (от слова *printer* – принтер), например, можно продемонстрировать с помощью следующей программы:

```
{ Пример 5 }  
{ Явное указание принтера устройством вывода }  
begin  
    { вывод текстовой константы на экран дисплея }  
    writeln(output, 'I love Turbo Pascal');  
    { назначение принтера устройством вывода }  
    assign(output, 'prn');  
    { инициация файла для записи }  
    rewrite(output);  
    { вывод текстовой константы на принтер }  
    writeln(output, 'I love Turbo Pascal')  
end.
```

Пример 6. Явное указание консоли устройством вывода при помощи объявленного пользователем имени переменной файлового типа *output*, например, можно продемонстрировать с помощью следующей программы:

```
{ Пример 6 }  
{ Явное указание консоли устройством вывода }  
{ объявление переменной файлового типа }  
var output : text;  
begin  
    { назначение консоли устройством вывода }  
    assign(output, 'con');
```

```

{ инициация файла для записи }
rewrite(output);
{ вывод текстовой константы на экран дисплея }
writeln(output, 'I love Turbo Pascal')
end.

```

Типы данных

В языке Turbo Pascal все данные, которые используются в программе, должны принадлежать к какому-либо заранее известному типу – или стандартному (предопределенный в языке Turbo Pascal тип данных), или пользовательскому (определяемый пользователем тип данных).

Тип данных определяет:

- формат представления данных в памяти персонального компьютера;
- множество допустимых значений, которые могут принимать данные;
- множество допустимых операций, применимых к этим данным.

Язык Turbo Pascal характеризуется разветвленной структурой типов данных:



Формат представления данных в памяти персонального компьютера, т.е. их размер в байтах, можно получить с помощью стандартной функции **sizeof** (от слов **size of** – размер), аргументом которой может быть либо имя данного, либо имя типа.

Во введении в Turbo Pascal ограничимся изложением только простых типов, в дальнейшем будут рассматриваться еще и структурированные типы – массивы и файлы, – а затем еще и процедурные типы. Такая форма изложения позволяет естественным образом перейти к изучению основ императивной парадигмы программирования, неизбежно ограниченных временными рамками курса, которые при желании всегда можно расширить, привлекая дополнительную литературу.

Простые типы

К простым типам принадлежат порядковые и вещественные типы. Порядковые типы подразделяются на целые или целочисленные, логические или булевы, символьные, перечисляемые и интервальные типы.

Порядковые типы

Порядковые типы характеризуются тем, что данные этих типов имеют конечное число возможных значений. Эти значения можно определенным образом упорядочить и, следовательно, с каждым из них можно сопоставить некоторое целое число – порядковый номер значения.

В языке Turbo Pascal имеется семь стандартных порядковых типов, а именно пять целых типов, логический и символьный типы, а также два определяемых пользователем типа – перечисляемый и интервальный типы.

К данным порядковых типов x применимы стандартные функции:

- ***ord(x)*** (от слова ***order*** – порядок) – возвращает значение порядкового типа x ;
- ***pred(x)*** (от слова ***predecessor*** – предшественник) – возвращает предыдущее значение порядкового типа x , т.е. значение, соответствующее порядковому номеру $ord(x) - 1$;
- ***succ(x)*** (от слова ***successor*** – преемник) – которая возвращает следующее значение порядкового типа x , т.е. значение, соответствующее порядковому номеру $ord(x) + 1$.

Функция ***pred(x)*** не определена для нижнего значения, а функция ***succ(x)*** – для верхнего значения диапазона возможных значений порядкового типа x .

Целые типы. В языке Turbo Pascal имеется пять стандартных целых типов: ***byte*** (от слова ***byte*** – байт), ***word*** (от слова ***word*** – слово), ***shortint*** (от слов ***short integer*** – короткое целое), ***integer*** (от слова ***integer*** – целое) и ***longint*** (от слов ***long integer*** – длинное целое).

Приведем в таблице сведения о формате представления данных целого типа в памяти персонального компьютера и диапазоне их возможных значений:

Тип	Диапазон значений	Формат
byte	$[0, 255]$ или $[0, 2^8 - 1]$	1 байт без знака
word	$[0, 65535]$ или $[0, 2^{16} - 1]$	2 байта без знака
shortint	$[-128, 127]$ или $[-2^7, 2^7 - 1]$	1 байт со знаком
integer	$[-32768, 32767]$ или $[-2^{15}, 2^{15} - 1]$	2 байта со знаком
longint	$[-2147483648, 2147483647]$ или $[-2^{31}, 2^{31} - 1]$	4 байта со знаком

При использовании процедур и функций с целочисленными параметрами следует руководствоваться так называемой вложенностью типов, т.е. тип **word** можно заменить типом **byte**, а тип **longint** – типом **integer**, который в свою очередь можно заменить типом **shortint**.

Перечень стандартных процедур и функций, применимых к данным целого типа, будет представлен в главе, посвященной программированию линейных алгоритмов. Здесь же отметим, что для выражения *x* целого типа стандартная функция **ord(x)** возвращает само значение *x*.

В тексте программы целые числа записываются как в десятичном, так и в шестнадцатеричном форматах данных. Например, числа **1000** и **-1000** в десятичном формате данных можно записать так:

1000 или **+1000**

-1000

В шестнадцатеричном формате эти же числа записываются так:

\$3e8 или **\$3E8**

-\$3e8 или **-\$3E8**

В языке Turbo Pascal определены пять операций целочисленной арифметики:

Оператор	Операция
+	сложение
-	вычитание
*	умножение
div	деление
mod	остаток операции деления

Тип результата в целочисленной арифметике будет соответствовать типу операндов, а если операнды относятся к различным целым типам – типу того операнда, который имеет максимальный диапазон значений. Возможное переполнение результата никак не контролируется, что может привести к недоразумениям, например, для типа **integer** вычисление суммы двух чисел **32767** и **2** приведет к неправильному результату:

32767 + 2 = -32767

В то время как для типа **longint** результат вычисления суммы этих же чисел будет правильным:

32767 + 2 = 32769

Итак, результат применения в целочисленной арифметике операций сложения, вычитания и умножения предсказуем при условии отсутствия его переполнения. Результат же применения двух последних операций – деления **div** (от слова **divide** – делить) и получения остатка операции деления **mod** (от слов **modulo operation** – операция по модулю) – всегда очевиден и предсказуем, например:

5 div 2 = 2 **1 div 2 = 0**

5 mod 2 = 1 **1 mod 2 = 1**

Логический (булев) тип. В языке Turbo Pascal имеется четыре стандартных логических или булевых типа: *boolean* (от слова *boolean* – булев) длиной в байт, *bytebool* (от слов *byte* и *boolean*) длиной в байт, *wordbool* (от слов *word* и *boolean*) длиной в слово (2 байта) и *longbool* (от слов *long* и *boolean*) длиной в двойное слово (4 байта). Логические типы: *bytebool*, *wordbool* и *longbool* были введены в язык Turbo Pascal 7.0 для совместимости с другими языками программирования и операционной системой *Windows*. Заметим, что во всех примерах, где необходимо применение логических типов, будет использоваться только тип *boolean*.

Значениями логического типа, например, *boolean* может быть одна из предварительно объявленных констант – *false* (от слова *false* – ложь) или *true* (от слова *true* – истина). Это обусловлено тем, что логический тип *boolean* определен как перечисляемый тип:

type

boolean = (false, true);

Так как соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления, поэтому для выражения *x* логического типа стандартная функция *ord(x)* возвращает целое число в диапазоне от 0 до 1.

Для констант *false* и *true* справедливы следующие правила:

ord(false) = 0

ord(true) = 1

false < true

pred(true) = false

succ(false) = true

Логические переменные необходимы для хранения значений логических выражений. Для логических переменных, принимающих одно из двух допустимых значений, можно использовать операции отношения, например: равно (оператор =), не равно (оператор <>), меньше (оператор <), больше (оператор >).

В языке Turbo Pascal определены четыре логические операции:

Оператор	Операция
not	логическое НЕ (логическое отрицание)
and	логическое И (логическое умножение, конъюнкция)
or	логическое ИЛИ (логическое сложение, дизъюнкция)
xor	исключительное ИЛИ (сложение по модулю 2)

Операторы *not* (от слова *not* – не), *and* (от слова *and* – и), *or* (от слова *or* – или) и *xor* (от слов *exclusive or* – исключительное или) являются одними из наиболее распространенных операторов булевой алгебры, названной в честь английского математика XIX века Джорджа Буля. Булевы операторы используются в булевых выражениях, иногда называемых логическими или условными, для вычисления одного из двух булевых значений – “истина” или “ложь”.

Булевы или логические операторы применимы к операндам целого и логического типов. Если операнды – целые числа, то результат логической операции есть тоже целое число. Логические операции над операндами логического типа дают результат логического типа.

В заключение приведем таблицы истинности каждой логической операции:

Операнд	Оператор	Операнд	Результат
	not	false	true
	not	true	false
false	and	false	false
false	and	true	false
true	and	false	false
true	and	true	true
false	or	false	false
false	or	true	true
true	or	false	true
true	or	true	true
false	xor	false	false
false	xor	true	true
true	xor	false	true
true	xor	true	false

Символьный тип. Значением стандартного символьного типа *char* (от слова *character* – символ) является множество всех символов таблицы ASCII-кодов (от слов *American Standard Code for Information Interchange* – американский стандартный код для обмена информацией). Каждому символу этой таблицы приписывается целое число в диапазоне $[0, 255]$, которое служит кодом внутреннего представления символа. Символьный тип, как и логические типы, также является перечисляемым типом.

Первая половина таблицы ASCII-кодов персонального компьютера с кодами символов из диапазона $[0, 127]$ соответствует стандарту ASCII, ее иногда называют основной таблицей ASCII-кодов, вторая половина таблицы с кодами символов из диапазона $[128, 255]$ не ограничена жесткими рамками стандарта и может меняться на персональных компьютерах разных типов. Символы с кодами из диапазона $[0, 31]$ относятся к служебным или управляющим кодам, которым в операциях ввода-вывода отведена особая роль.

Для выражения x символьного типа стандартная функция *ord(x)* возвращает целое число в диапазоне от 0 до 255.

В тексте программы символьные литералы могут записываться либо с помощью двух символов апостроф (например, 'A' – это символ *A*, а 'a' – это символ *a*), либо ASCII-кодами (например, #65 – это код символа *A*, а #97 – это код символа *a*), либо с помощью символа ^ и последующей литеры (например, ^A или ^a – это #1).

Перечисляемый тип. Перечисляемый тип или тип перечисление является типом, определяемым пользователем, и его значение определяется перечислением идентификаторов, которые могут использоваться в программе в качестве операндов в операциях сравнения и присваивания. Идентификаторы располагаются в списке, заключенном в круглых скобках, и разделяются знаком препинания запятой:

type

имя_перечисляемого_типа = (список_идентификаторов);

Значением перечисляемого типа *имя_перечисляемого_типа* является множество элементов перечисления *список_идентификаторов*. Один и тот же идентификатор можно использовать в объявлении только одного уникального перечисляемого типа. Идентификаторы из списка являются именованными константами, значение которых устанавливается порядком перечисления: первый идентификатор в списке получает порядковый номер 0, второй – 1 и т.д. Перечисляемый тип задает подмножество целочисленных констант типа *word*, т.е. для выражения *x* перечисляемого типа стандартная функция *ord(x)* возвращает целое число в диапазоне от 0 до 65535.

Например, определим перечисляемый тип *week* (от слова *week* – неделя), значение которого будет связано с одним из семи дней недели:

type

week = (*Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday*);

Здесь, например, *ord(Monday)* = 0, а *ord(Sunday)* = 6.

Интервальный тип. Интервальный тип является типом, определяемым пользователем, и представляет собой подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме самого интервального типа. Интервальный тип определяется границами своих значений внутри базового типа:

type

имя_интервального_типа = *нижняя_граница* .. *верхняя_граница*;

Здесь *нижняя_граница* и *верхняя_граница* являются константами.

Например, определим интервальный тип *digit* (от слова *digit* – цифра), значение которого связано с символьным значением одной из десяти цифр:

type

digit = '0'..'9';

После такого объявления интервального типа *digit* любой объект программы этого типа не должен выходить за границы указанного интервала. Попытка присвоить такому объекту программы значение вне интервала допустимых значений приводит к ошибке либо времени компиляции, либо времени выполнения.

Если выражение *x* интервального типа, результат *ord(x)* зависит от свойств базового порядкового типа, так как интервальный тип сохраняет все свойства своего базового порядкового типа.

Вещественные типы

Вещественные типы характеризуются тем, что данные этих типов (действительные или так называемые вещественные числа) тоже имеют конечное число значений, которое определяется форматом внутреннего представления вещественного числа. Однако количество возможных значений какого-либо вещественного типа настолько велико, что сопоставить с каждым из них целое число (порядковый номер) не представляется возможным.

В отличие от порядковых типов, значения которых представляются абсолютно точно, значения вещественных типов определяют какое-либо число лишь с некоторой конечной точностью, зависящей от формата внутреннего представления вещественного числа.

В языке Turbo Pascal имеется пять стандартных вещественных типов: *single* (от слова *single* – одинарный), *real* (от слова *real* – вещественный), *double* (от слова *double* – двойной), *extended* (от слова *extended* – расширенный) и *comp* (от слова *complex* – сложный).

Вещественные числа типа *single*, *real*, *double* и *extended* имеют следующую структуру:

s	e	m
----------	----------	----------

Здесь *s* (от слова *sign* – знак) – знаковый разряд, *e* (от слова *exponent* – экспонент или показатель степени) – экспоненциальная часть (содержит двоичный порядок), а *m* (от слова *mantissa* – мантисса) – мантисса или дробная часть числа.

Приведем в таблице сведения о формате представления данных вещественного типа *single*, *real*, *double* и *extended* в памяти персонального компьютера, диапазоне возможных значений десятичного порядка и количестве значащих цифровых разрядов мантиссы:

Тип	Точность мантиссы	Диапазон десятичного порядка	Формат
single	7 или 8	[– 45, 38]	4 байта s = 1 e = 8 m = 23
real	11 или 12	[– 39, 38]	6 байт s = 1 e = 8 m = 39
double	15 или 16	[– 324, 308]	8 байт s = 1 e = 11 m = 52
extended	19 или 20	[– 4932, 4932]	10 байт s = 1 e = 16 m = 63

Мантисса *m* имеет длину от 23 (для *single*) до 63 (для *extended*) двоичных разрядов, что и обеспечивает ее точность для указанного в таблице количества десятичных цифр. Десятичная точка или запятая подразумевается перед старшим разрядом мантиссы, но при арифметических операциях с вещественным числом ее

положение сдвигается влево или вправо в соответствии с двоичным порядком числа, поэтому действия над вещественными числами называют арифметикой с плавающей точкой или запятой – *floating-point arithmetic*.

Особая роль принадлежит вещественному типу **comp**, поскольку данные этого типа являются целыми числами со знаком и точностью 19 или 20 десятичных цифр:

Тип	Диапазон значений	Формат
comp	$[-2^{63}, 2^{63} - 1]$	8 байт со знаком

Доступ к типам *single*, *double*, *extended* и *comp* возможен только при особых режимах компиляции, так как эти типы рассчитаны на аппаратную поддержку арифметики с плавающей точкой и для их эффективного использования в состав персонального компьютера должен входить числовой сопроцессор (от слов *numeric coprocessor*). Числовой сопроцессор всегда обрабатывает числа в формате *extended*, а три остальных вещественных типа в этом случае получаются простым усечением результатов до нужных размеров и применяются в основном для экономии памяти. Работу числового сопроцессора можно эмулировать и программно, о чем будет сказано в главе, посвященной программированию линейных алгоритмов.

Заметим, что тип *real* оптимизирован для работы без числового сопроцессора, поэтому использование этого типа при работе с числовым сопроцессором приведет к дополнительным затратам времени на преобразование типа от *real* к *extended* и может свести на нет все преимущества числового сопроцессора.

В тексте программы вещественные числа записываются как в формате с фиксированной точкой, так и в экспоненциальном формате.

Например, числа **5.0** и **-0.5** можно записать так:

5. или **5.0** или **5e+0** или **5E+0** или **5.e+0** или **5.E+0** или **5.0e+0** или **5.0E+0**
-0.5 или **-0.5e+0** или **-0.5E+0**

Здесь символ *e* или *E* означает десятичный порядок и имеет смысл – умножить на 10 в степени, показатель которой записывается следом за этим символом в виде целого числа со знаком.

При выводе вещественные числа по умолчанию отображаются в экспоненциальном формате, при этом шаблон вывода зависит от типа числа, к тому же положительные числа выводятся с ведущим пробелом.

Например, для числа **5.0** формат вывода будет таким:

5.0000000000E+00 для типа *real*,
5.000000000000000E+0000 для остальных типов.

Например, для числа **-5.0** формат вывода будет таким:

-5.0000000000E+00 для типа *real*,
-5.000000000000000E+0000 для остальных типов.

При выводе вещественные числа также могут отображаться и в формате с фиксированной точкой.

Подробное обсуждение механизма управления форматным выводом данных вещественного типа будет проведено в главе, посвященной работе с файлами.

В языке Turbo Pascal определены четыре операции вещественной арифметики:

Оператор	Операция
+	сложение
–	вычитание
*	умножение
/	деление

Перечень стандартных процедур и функций, применимых к данным вещественного типа, будет представлен в главе, посвященной программированию линейных алгоритмов.

Раздел объявлений программы

При обсуждении структуры программы ранее уже было сказано, из каких подразделов может состоять раздел объявлений программы. Перейдем теперь к подробному обсуждению подразделов, связанных с объявлением как глобальных, так и локальных объектов программы – типов, определяемых пользователем, констант, переменных и меток. Подразделы объявления функций и процедур пользователя, которые также могут принадлежать как к глобальным, так и к локальным объектам программы, будут рассматриваться в главе, посвященной процедурам и функциям.

Понятие “глобальный” связано с понятием “глобальная область видимости”. Объекты программы, объявленные в глобальной области видимости, доступны как самой программе, так и всем ее частям – модулям, внешним функциям и процедурам. Наряду с глобальными объектами в программе могут быть объявлены и локальные объекты. Понятие “локальный” связано с понятием “локальная область видимости”. Объекты программы, объявленные в локальной области видимости, доступны только функциям и процедурам пользователя. К локальным объектам относятся объекты, объявленные в разделе объявлений функций и процедур пользователя, а также формальные параметры этих функций и процедур, объявленные как параметры-значения.

Объявление типов

При использовании в программе стандартных типов данных нет необходимости в их предварительном объявлении в отличие от типов данных, определяемых пользователем. Язык Turbo Pascal допускает определение пользовательских типов данных как на основе стандартных типов, так и на основе ранее определенных пользовательских типов, при этом следует придерживаться следующего правила:

каждый новый тип должен определяться через уже известный компилятору тип. Имена пользовательских типов – это уникальные идентификаторы программы.

Подраздел объявления типов начинается с ключевого слова ***type*** (от слова ***types*** – типы) и продолжается до любого другого ключевого слова. В этом подразделе могут быть использованы ранее объявленные константы.

Определение пользовательского типа на основе стандартного типа:

type

имя_пользовательского_типа = имя_стандартного_типа;

Здесь для присваивания объекту программы ***имя_пользовательского_типа*** значения ***имя_стандартного_типа*** используется оператор присваивания ***=***.

Например:

type

integer_type = word;

real_type = real;

Определение пользовательского типа на основе ранее определенного типа:

type

имя_пользовательского_типа_1 = имя_стандартного_типа;

имя_пользовательского_типа_2 = имя_пользовательского_типа_1;

Например, определим пользовательский тип ***matrix*** (матрица) для описания такого объекта программы, как двумерный массив элементов типа ***integer***, состоящий из 3-х строк и 4-х столбцов, используя ранее определенный пользовательский тип ***column_size*** (длина столбца) как одномерный массив элементов типа ***integer***:

type

column_size = array[1..3] of integer;

matrix = array[1..4] of column_size;

Здесь в определении пользовательского типа используется структурированный тип массив, при определении которого используется интервальный тип.

Например, другим решением этой проблемы является использование ранее определенного пользовательского типа ***row_size*** (длина строки) как одномерного массива элементов типа ***integer***:

type

row_size = array[1..4] of integer;

matrix = array[1..3] of row_size;

Объявление констант

Подраздел объявления констант начинается с ключевого слова ***const*** (от слова ***constants*** – константы) и продолжается до любого другого ключевого слова. Константа именует объект программы, который не может изменять свое значение. Имена констант – это уникальные идентификаторы программы.

Определение константы:

const

имя_константы = значение_константы;

Здесь для присваивания объекту программы *имя_константы* значения *значение_константы* используется оператор присваивания =.

По умолчанию тип константы определяется способом записи ее значения. Например:

```
const
  c1 = 1;           { целый mun integer }
  c2 = 1.0;         { вещественный mun real }
  c3 = 'a';         { символьный mun char }
  c4 = 'hello';     { mun строка string }
  c5 = true;        { логический mun boolean }
```

Язык Turbo Pascal допускает использование типизированных констант. Определение типизированной константы:

```
const
  имя_константы : имя_типа_константы = значение_константы;
```

Здесь в объявлении используется знак препинания двоеточие.

Типизированным константам можно присваивать значения в ходе выполнения программы, поэтому фактически типизированные константы представляют собой переменные с начальными значениями.

Объявление переменных

Подраздел объявления переменных начинается с ключевого слова **var** (от слова *variables* – переменные) и продолжается до любого другого ключевого слова. Переменная именует объект программы, который может изменять свое значение. Имена переменных – это уникальные идентификаторы программы.

Определение переменной:

```
var
  имя_переменной : имя_типа_переменной;
```

Здесь в объявлении используется знак препинания двоеточие.

Например:

```
var
  a : byte;
```

Несколько однотипных переменных можно объединять в список, разделяя их знаком препинания запятая. Например:

```
var
  a, b, c : shortint;
```

Объявление меток

Подраздел объявления меток начинается с ключевого слова **label** (от слова *labels* – метки) и продолжается до любого другого ключевого слова. Метка позволяет пометить какую-либо инструкцию тела программы и таким образом сослаться на нее. Ссылка на помеченную инструкцию осуществляется посредством использования инструкции перехода **goto**, которая позволяет передавать управление

этой помеченной инструкции. Метка располагается перед помечаемой инструкцией и отделяется от нее знаком препинания двоеточие. Инструкцию можно помечать несколькими метками, которые в этом случае разделяются знаком препинания двоеточие.

Объявление метки:

label

имя_метки;

Имена меток – это уникальные идентификаторы программы. Имя метки в отличие от остальных идентификаторов программы может начинаться с цифры, а также в языке Turbo Pascal допускается использование в качестве меток целых чисел без знака. Несколько меток можно объединять в список, разделяя их знаком препинания запятая.

Например:

label

1, 1_label, restart;

Первое знакомство с инструкциями языка Turbo Pascal

При обсуждении структуры программы Turbo Pascal уже состоялся первый этап знакомства с понятием “инструкция” и была обоснована позиция автора, связанная как с традиционной, так и современной трактовкой этого понятия. На втором этапе был представлен ряд примеров использования некоторых видов инструкций языка, а именно инструкции объявления, составной инструкции и инструкции вызова процедуры. Следующий этап знакомства был связан с изложением главы, посвященной разделу объявлений программы. Здесь к уже перечисленным видам инструкций добавились помеченная инструкция и инструкция перехода. Как видно из примеров и структуры раздела объявлений программы, объявления могут делать нечто большее, чем просто связывание типа с именем. Большинство объявлений являются еще и определениями, т.е. они определяют некую сущность, которая соответствует имени. Такими сущностями при определении могут быть, например, участки выделенной компилятором памяти, значения констант, новые типы.

Приведем описание набора инструкций языка Turbo Pascal, стараясь передать удобную для понимания синтаксическую форму для каждой инструкции:

инструкция объявление

объявление;

составная инструкция

begin последовательность_инструкций end

последовательность_инструкций:

инструкция

последовательность_инструкций инструкция

пустая инструкция

инструкция вызова процедуры

имя_процедуры (список_параметров);

список_параметров:

параметр

список_параметров , параметр

инструкция присваивания

имя_переменной := выражение;

помеченная инструкция

метка : инструкция;

метка:

идентификатор

целое_число_без_знака

инструкция перехода

goto метка;

условная инструкция

if логическое_выражение **then** инструкция **else** инструкция;

if логическое_выражение **then** инструкция;

инструкция выбора

case ключ_селектор **of**

последовательность_case-инструкций **else** инструкция **end;**

case ключ_селектор **of**

последовательность_case-инструкций **end;**

ключ_селектор:

выражение_порядкового_типа

последовательность_case-инструкций:

case-инструкция

последовательность_case-инструкций case-инструкция

case-инструкция:

список_константных_выражений : инструкция;

константное_выражение:

выражение_порядкового_типа

инструкция повторения

while логическое_выражение **do** инструкция;

repeat инструкция **until** логическое_выражение;

for инструкция_инициализации **for to** выражение **do** инструкция;

for инструкция_инициализации **for downto** выражение **do** инструкция;

инструкция_инициализации **for**:

имя_переменной_порядкового_типа := выражение

выражение:

выражение_порядкового_типа

инструкция присоединения

with имя_записи **do** инструкция;

inline-инструкция

(последовательность_машинных_кодов);

последовательность_машинных_кодов:

машинный_код

последовательность_машинных_кодов / машинный_код

машинный_код:

шестнадцатеричный_литерал_без_знака

asm-инструкция

asm последовательность_инструкций_ассемблера **end**;

последовательность_инструкций_ассемблера:

инструкция_ассемблера

последовательность_инструкций_ассемблера инструкция_ассемблера

Такие инструкции, как условная инструкция и инструкция выбора, могут иметь различные синтаксические формы – полную и краткую, связанную с отсутствием части **else** от полной синтаксической формы.

Инструкция повторения представлена группой из четырех инструкций: инструкцией повторения с предусловием, инструкцией повторения с постусловием и двумя счетными инструкциями повторения.

В программе Turbo Pascal на языке встроенного ассемблера можно написать как отдельные фрагменты кода, так и подпрограммы – процедуры или функции пользователя. В этом случае программный код таких фрагментов или подпрограмм записывается с помощью **asm**-инструкций. Также в программах Turbo Pascal можно использовать объектный код уже скомпилированных внешних подпрограмм, исходный код которых написан на языке ассемблера.

В отдельных случаях для повышения быстродействия программы Turbo Pascal в ее исходный текст с помощью *inline*-инструкций можно вставлять команды ассемблера, записанные непосредственно в машинных кодах.

Подробное обсуждение инструкций языка Turbo Pascal будет распределено по главам, связанных с программированием всех рассматриваемых в пособии типов алгоритмов, здесь же представим только пустую инструкцию.

Пустая инструкция не содержит никаких значимых действий, в основном она используется для передачи управления в потоке инструкций. Пустые инструкции могут быть представлены как в явной, так и в неявной форме, а также могут присутствовать и в других инструкциях, например, в составных инструкциях:

```
begin    { пустая инструкция в составной инструкции }  
end
```

```
begin    { составная и пустая инструкции в составной инструкции }  
    begin { пустая инструкция в составной инструкции }  
    end;  { пустая инструкция в составной инструкции }  
end
```

Другие случаи использования пустой инструкции будут представлены позже.

Программирование линейных алгоритмов

Для линейных алгоритмов процесс решения задачи представлен в виде ряда последовательно выполняемых этапов, на каждом из которых по некоторым значениям, известным к началу выполнения этого этапа, вычисляется новое значение. Одни из этих вычисленных значений являются окончательными результатами решения задачи, а другие значения являются промежуточными результатами, которые могут быть использованы в качестве исходных данных для некоторых из последующих этапов.

Выражения

Для задания правил вычисления новых значений в императивных языках программирования служит такое понятие, как **выражение**, причем каждое выражение задает правила вычисления только одного значения. Выражения строятся из операндов, операторов и круглых скобок, с помощью которых можно задать любой порядок выполнения операций. Различают три вида операндов – **постоянные**, **переменные** и **вычисляемые**.

Постоянный операнд задает значение, которое определяется к началу вычисления выражения и не меняется в процессе выполнения программы, т.е. постоянный операнд – это константа языка Turbo Pascal.

Переменный операнд задает значение, которое определяется к началу вычисления выражения и может изменяться в процессе выполнения программы, т.е. переменный операнд – это переменная языка Turbo Pascal.

Вычисляемый операнд задает значение, которое не определяется к началу вычисления выражения, а вычисляется непосредственно в процессе вычисления самого выражения, т.е. вычисляемый операнд – это вызов стандартной функции языка Turbo Pascal или функции пользователя.

Операторы указывают на операцию, которую следует выполнить над операндами выражения. В языке Turbo Pascal используются только унарные и бинарные операторы.

Заметим, что выражение ничего не говорит о том, что следует делать с самим вычисленным значением, и потому выражение не задает какого-то логически завершенного этапа вычислений. Наиболее типичной является ситуация, когда вычисленное значение выражения необходимо запомнить для его использования на последующих этапах вычислительного процесса – такое запоминание достигается путем присваивания вычисленного значения некоторой переменной. Для задания такого действия в языке Turbo Pascal используется инструкция присваивания.

Итак, в программе результат вычисления выражения можно либо присвоить переменной, либо передать его в качестве параметра какой-либо функции или процедуре.

В языке Turbo Pascal используются только два вида выражений – **арифметические** и **логические**.

Выражения, в которых используются только операнды арифметического типа (целочисленный или вещественный тип) и арифметические операторы, называются арифметическими. Тип результата вычисления выражения – арифметический.

Выражения, в которых используются только операнды логического типа и логические операторы, называются логическими. Специфическим видом операнда логического типа является отношение, которое является результатом сравнения двух выражений. Тип результата вычисления выражения – логический.

Приведем таблицу операций отношения:

Оператор	Операция
=	равно
<>	не равно
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно

Заметим, что операции отношения используются также для указателей, строк и множеств.

При вычислении значения выражения приоритет вычислений определяется расставленными круглыми скобками, а при их отсутствии – по таблице:

Приоритет	Операция	Операторы
1	Вычисления в круглых скобках	
2	Вычисления значений функций	
3	Унарные операции	@ , not , + , –
4	Операции типа умножения	* , / , div , mod , and , shl , shr
5	Операции типа сложения	+ , – , or , xor
6	Операции отношения	= , <> , < , <= , > , >= , in

Заметим, что булевы операторы можно использовать еще и для побитовых логических операций с целыми числами. Тип результата при этом определяется типом операнда, имеющего наименьший размер. К этой группе операторов относятся еще два оператора для побитовых логических операций с целыми числами – *shl* (от слов *shift left* – сдвиг влево) и *shr* (от слов *shift right* – сдвиг вправо).

Оператор *in* используется в операциях с множествами.

Арифметические функции

Язык Turbo Pascal предоставляет пользователю ряд стандартных арифметических функций для вычисления значения выражения с заданной точностью. При компиляции программы в режиме использования числового сопроцессора или его эмуляции арифметические функции возвращают значение типа *extended*, в противном случае – типа *real*. Аргументом арифметических функций могут быть данные только арифметического типа.

Приведем в таблице сведения о некоторых арифметических функциях:

Функция	Назначение	Тип результата
$abs(x)$	модуль аргумента	тип аргумента
$arctan(x)$	арктангенс аргумента	вещественный
$cos(x)$	косинус аргумента	вещественный
$exp(x)$	e^x	вещественный
$ln(x)$	натуральный логарифм аргумента	вещественный
pi	$\pi = 3.1415926535897932385$	вещественный
$sin(x)$	синус аргумента	вещественный
$sqr(x)$	x^2	тип аргумента
$sqrt(x)$	\sqrt{x}	вещественный

Полезным будет напомнить некоторые формулы приведения:

$$\begin{aligned}
 a^x &= e^{x \ln a} & x^a &= e^{a \ln x} & \arcsin x &= \arctg \frac{x}{\sqrt{1-x^2}} \\
 x^n &= e^{n \ln x} & \sqrt[n]{x} &= e^{\frac{\ln x}{n}} & \arccos x &= \frac{\pi}{2} - \arcsin x \\
 \lg x &= \frac{\ln x}{\ln 10} & & & \operatorname{arcctg} x &= \frac{\pi}{2} - \arctg x
 \end{aligned}$$

Как видим, в языке Turbo Pascal недостаточно средств для выполнения численных расчетов, так как при проектировании языка такая цель не преследовалась. Однако пользователь всегда может разработать некоторый набор функций, позволяющих реализовать необходимые ему численные расчеты.

Преобразования типов

Как уже отмечалось, типы данных позволяют не только устанавливать формат их внутреннего представления, но и контролировать те действия, которые выполняются над этими данными. В программе данные одного типа могут преобразовываться в данные другого типа, преобразования типов при этом могут быть неявными или явными. Контроль за использованием типов данных как на этапе времени компиляции программы, так и на этапе времени выполнения программы – одна из характерных особенностей языка Turbo Pascal. Строгое соблюдение концепции типов означает, что все выполняемые операции над данными определены только для совместимых типов. Проблему совместимости типов рассмотрим на примере целочисленных и вещественных данных.

Неявные преобразования типов

Неявное преобразование типов возможно только в двух случаях:

- в выражениях, составленных из вещественных и целочисленных констант или переменных, которые автоматически преобразуются к вещественному типу, и все выражение в целом преобразуется к вещественному типу;
- одна и та же область памяти попеременно трактуется как содержащая данные то одного, то другого типа (так называемое совмещение в памяти данных разного типа).

Рассмотрим два примера, иллюстрирующих особенности механизма неявного преобразования типов, например, для первого случая.

Пример 7. Неявные преобразования типов на этапе времени компиляции программы можно продемонстрировать с помощью следующей программы:

```
{ Пример 7 }
{ Неявные преобразования типов }
begin
  writeln(1 / 2)
end.
```

Результат работы программы:

5.0000000000E-01

При вычислении значения выражения $1/2$ целочисленные константы 1 и 2 неявно преобразуются в вещественные константы 1.0 и 2.0 на этапе времени компиляции программы, так как тип выражения $1/2$ благодаря оператору деления $/$ трактуется как вещественный (по умолчанию *real*).

Пример 8. Неявные преобразования типов на этапе времени выполнения программы можно продемонстрировать с помощью следующей программы:

```
{ Пример 8 }
{ Неявные преобразования типов }
const  a = 1;
        b = 2;
var    c : real;
begin
    c := a / b;
    writeln(c)
end.
```

Результат работы программы:

5.000000000000E-01

При вычислении значения выражения a/b целочисленные константы a и b неявно преобразуются в вещественные константы 1.0 и 2.0 на этапе времени выполнения программы, так как тип выражения a/b благодаря оператору деления $/$ трактуется как вещественный (по умолчанию *real*).

Явные преобразования типов

Для явного преобразования типов в языке Turbo Pascal определены стандартные функции, которые получают в качестве параметра значение одного типа, а возвращают результат в виде значения другого типа. Таковой является уже представленная ранее функция *ord*. Представим здесь еще две функции, используемые для преобразования *real* в *integer*: функцию *trunc* (от слова *truncate* – усекать), которая усекает *real* путем отбрасывания дробной части, и функцию *round* (от слова *round* – округлять), которая округляет *real* до ближайшего целого.

Рассмотрим два примера, иллюстрирующих особенности механизма явного преобразования типов.

Пример 9. Явные преобразования типов при помощи стандартных функций *trunc* и *round* можно продемонстрировать с помощью следующей программы:

```
{ Пример 9 }
{ Явные преобразования типов }
begin
    writeln(trunc(4.1), ' ', trunc(4.5));
    writeln(round(4.1), ' ', round(4.5))
end.
```

Результат работы программы:

4 4
4 5

В языке Turbo Pascal может использоваться и более общий механизм преобразования типов, согласно которому преобразование достигается применением идентификатора стандартного типа или определяемого пользователем типа как идентификатора функции преобразования к выражению преобразуемого типа (так называемое автоопределенное преобразование типов).

Пример 10. Механизм автоопределенного преобразования типов, например, для типа *longint* можно продемонстрировать с помощью следующей программы:

```
{ Пример 10 }
{ Явные преобразования типов }
var  a : integer;
      b : longint;
begin
  a := 32767;
  b := longint(a) + 2;
  writeln(' ', b);
  writeln(a + 2, ' ', longint(a) + 2)
end.
```

Результат работы программы:

```
32769
-32767 32769
```

Арифметические выражения

Рассмотрим ряд примеров, в которых вычисляются значения одного и того же арифметического выражения $\cos \sqrt{ax^2 + bx} + \lg(ax + b)$.

Первые два примера демонстрируют особенности интерфейса, связанного с определением исходных данных и результата.

Пример 11. Пусть данные *a*, *b* и *x* будут константами, тип которых задается по умолчанию как *real*. Тип арифметического выражения в этом случае будет приводиться к типу *real* благодаря неявному преобразованию типов. Результат вычисления выражения пусть будет присвоен переменной типа *real*.

Представим программную реализацию:

```
{ Пример 11 }
{ Арифметические выражения }
const line = 'cos(sqrt(a*sqr(x)+b*x))+lg(a*x+b) = ';
      a = 0.1;
      b = 0.2;
      x = 0.3;
var  y : real;
```

```
begin
  y := cos(sqrt(a * sqr(x) + b * x)) + ln(a * x + b) / ln(10);
  writeln(line, y)
end.
```

Результат работы программы:

```
cos (sqrt (a*sqr (x) +b*x) )+lg (a*x+b) = 3.2742575532E-01
```

Пример 12. Пусть данные a , b и x будут теперь переменными типа *real*. Тип арифметического выражения в этом случае также будет приводиться к типу *real* благодаря неявному преобразованию типов. Пусть выражение будет элементом списка вывода.

Представим программную реализацию:

```
{ Пример 12 }
{ Арифметические выражения }
const line = 'cos(sqrt(a*sqr(x)+b*x))+lg(a*x+b) = ';
var a, b, x : real;
begin
  write('a ? '); readln(a);
  write('b ? '); readln(b);
  write('x ? '); readln(x);
  writeln(line, cos(sqrt(a * sqr(x) + b * x)) + ln(a * x + b) / ln(10))
end.
```

Результат работы программы:

```
cos (sqrt (a*sqr (x) +b*x) )+lg (a*x+b) = 3.2742575532E-01
```

Два следующих примера демонстрируют точность вычислений от минимальной до максимально возможной, которая будет определяться типом исходных данных.

Пример 13. Пусть данные a , b и x будут типизированными константами типа *single*, тип арифметического выражения в этом случае будет приводиться к типу *single* благодаря неявному преобразованию типов. Результат вычисления выражения пусть будет присвоен переменной типа *single*.

Представим программную реализацию:

```
{ Пример 13 }
{ Арифметические выражения }
const line = 'cos(sqrt(a*sqr(x)+b*x))+lg(a*x+b) = ';
  a : single = 0.1;
  b : single = 0.2;
  x : single = 0.3;
```



```
var    y : single;
begin
  y := cos(sqrt(a * sqr(x) + b * x)) + ln(a * x + b) / ln(10);
  writeln(line, y)
end.
```

Результат работы программы:

```
cos (sqrt (a*sqr (x)+b*x)) +lg (a*x+b)  =  3.27425748109818E-0001
```

Пример 14. Пусть данные *a*, *b* и *x* будут теперь типизированными константами типа *extended*, тип арифметического выражения в этом случае будет приводиться к типу *extended* благодаря неявному преобразованию типов. Результат вычисления выражения пусть будет присвоен переменной типа *extended*.

Представим программную реализацию:

```
{ Пример 14 }
{ Арифметические выражения }
const  line = 'cos(sqrt(a*sqr(x)+b*x))+lg(a*x+b) = ';
       a : extended = 0.1;
       b : extended = 0.2;
       x : extended = 0.3;
var    y : extended;
begin
  y := cos(sqrt(a * sqr(x) + b * x)) + ln(a * x + b) / ln(10);
  writeln(line, y)
end.
```

Результат работы программы:

```
cos (sqrt (a*sqr (x)+b*x)) +lg (a*x+b)  =  3.27425755316843E-0001
```

Как ранее уже было сказано, если в программе типы данных определяются как *single*, *double*, *extended* или *comp*, то для компиляции таких программ требуется настройка компилятора, связанная с указанием либо непосредственного использования числового сопроцессора, либо его программной эмуляции для операций арифметики с плавающей точкой.

Такую настройку компилятора можно осуществить одним из двух способов: либо через диалоговое окно *Numeric processing* диалогового окна *Compiler Options*, вызываемого с помощью пункта меню *Compiler* в меню *Options* главного меню интегрированной среды Turbo Pascal, либо посредством директив компилятора *{ \$N+ }* (от слов *numeric coprocessor* – числовой сопроцессор) и *{ \$E+ }* (от слова *emulation* – эмуляция), записываемых в подразделе объявления глобальных директив компилятора раздела объявлений программы.

Заметим, что настройки в диалоговых окнах реализованы на использовании так называемых радио-кнопок. Приведем примеры настроек компилятора:

- использование числового сопроцессора

Numeric processing

☒ 8087/80287

☐ Emulation

- эмуляция числового сопроцессора

Numeric processing

☒ 8087/80287

☒ Emulation

Таким образом, тексты программ для последних двух примеров можно компилировать лишь при условии, что настройка компилятора была осуществлена через диалоговое окно *Numeric processing*.

В случае настройки компилятора посредством директив компилятора $\{ \$N+ \}$ и $\{ \$E+ \}$ в тексте программы в подразделе объявления глобальных директив компилятора могут быть использованы следующие директивы:

- использование числового сопроцессора

$\{ \$N+ \}$

- эмуляция числового сопроцессора

$\{ \$N+, E+ \}$

Пример 15. Пусть настройка компилятора выполняется посредством директив компилятора для случая использования числового сопроцессора, тогда программная реализация для последнего примера могла бы выглядеть так:

{ Пример 15 }

{ Арифметические выражения }

$\{ \$N+ \}$

*const line = 'cos(sqrt(a*sqr(x)+b*x))+lg(a*x+b) = ';*

a : extended = 0.1;

b : extended = 0.2;

x : extended = 0.3;

var y : extended;

begin

*y := cos(sqrt(a * sqr(x) + b * x)) + ln(a * x + b) / ln(10);*

writeln(line, y)

end.

Программирование условных алгоритмов

Условные алгоритмы (алгоритмы выбора или алгоритмы ветвления) позволяют организовать выполнение альтернативных действий.

Если выполнение действий алгоритма зависит от одного из двух возможных значений выражения логического типа (*true* – “истина” или *false* – “ложь”), то возможны две ветви алгоритма. При программировании таких алгоритмов на языке Turbo Pascal используется условная инструкция *if*.

Если выполнение действий алгоритма зависит от одного из множества заданных значений выражения порядкового типа, то возможно множество ветвей алгоритма. При программировании таких алгоритмов на языке Turbo Pascal используется инструкция выбора *case*.

Условная инструкция

Для определения условной инструкции можно использовать две синтаксические формы – полную и краткую:

if условие then инструкция else инструкция;

if условие then инструкция;

Здесь *условие* – логическое выражение.

Полная форма условной инструкции *if* позволяет выполнить какую-либо одну из двух альтернативных ветвей алгоритма. Если значение логического выражения есть *true*, то выполняется инструкция в части *then*, в противном случае – инструкция в части *else*. Краткая форма условной инструкции *if* позволяет выполнить или не выполнить только одну из двух альтернативных ветвей алгоритма. Если значение логического выражения есть *true*, то выполняется инструкция в части *then*, в противном случае условная инструкции *if* становится пустой.

Заметим, что синтаксис языка Turbo Pascal разрешает использование пустых инструкций как в части *then*, так и в части *else*:

if условие then else инструкция;

if условие then инструкция else;

Если первая форма условной инструкции *if* с пустой инструкцией в части *then* еще находит применение, то вторая ее форма с пустой инструкцией в части *else* обычно не используется. К пустым инструкциям в части *then* прибегают, например, в случае нежелания или невозможности сравнения с нулем в логических выражениях вещественного типа.

Теперь обратимся к одной весьма важной особенности, связанной с вложенностью условных инструкций *if* друг в друга. Поскольку любая из инструкций как в части *then*, так и в части *else* может быть в свою очередь условной, и в то же время не каждая из них может иметь часть *else*, то в этом случае возникает неоднозначность трактовки условий для каждой условной инструкции *if*. Язык Turbo Pascal эту проблему решает следующим образом: любая часть *else* соответствует своей ближайшей к ней части *then* условной инструкции *if*.

Рассмотрим несколько условных алгоритмов и их программные реализации.

Пример 16. Пусть требуется вычислить одно из трех возможных значений некоторой функции двух переменных $U(x, y)$ в зависимости от условий, накладываемых на ее аргументы x и y :

$$U(x, y) = \begin{cases} \sin(x + y) & \text{для } x + y < 0 \\ \sin x \cdot \cos y & \text{для } x + y = 0 \\ \cos(x + y) & \text{для } x + y > 0 \end{cases}$$

Представим традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Ввести значение y*
4. *Если $x + y < 0$, то перейти к пункту 5, иначе перейти к пункту 8*
5. *Положить $U(x, y) = \sin(x + y)$*
6. *Вывести значения x , y и $U(x, y)$*
7. *Конец*
8. *Если $x + y > 0$, то перейти к пункту 9, иначе перейти к пункту 11*
9. *Положить $U(x, y) = \cos(x + y)$*
10. *Перейти к пункту 6*
11. *Положить $U(x, y) = \sin(x) * \cos(y)$*
12. *Перейти к пункту 6*

Такое описание алгоритма дает представление о структуре программы, в которой упор сделан на логические взаимосвязи элементарных операций.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Ввести значение y*
4. *Если $x + y < 0$*
то
 - 4.1. *Положить $U(x, y) = \sin(x + y)$**иначе*
 - 4.2. *Если $x + y > 0$*
то
 - 4.2.1. *Положить $U(x, y) = \cos(x + y)$*
 - иначе
 - 4.2.2. *Положить $U(x, y) = \sin(x) * \cos(y)$*
5. *Вывести значения x , y , $U(x, y)$*
6. *Конец*

Такое описание алгоритма дает представление о структуре программы, в которой элементарные операции объединены в управляющие структуры следования и выбора – блоки с одной точкой входа и одной точкой выхода, свойственные многим языкам, которые поддерживают императивную парадигму программирования.

Как видим, такой способ передачи управления, основанный на использовании управляющих структур следования и выбора, позволяет теперь трактовать условный алгоритм как линейный.

Теперь обратимся к программной реализации:

```
{ Пример 16 }
{ Условные алгоритмы }
type  real_type = real;
var    x, y, u : real_type;
begin
  write('x? '); readln(x);
  write('y? '); readln(y);
  if x + y < 0
    then u := sin(x + y)
    else if x+y > 0
      then u := cos(x + y)
      else u := sin(x) * cos(y);
  writeln('x = ', x);
  writeln('y = ', y);
  writeln('U(x,y) = ', u)
end.
```

Результат работы программы:

```
x? 1
y? 2
x =  1.00000000000E+00
y =  2.00000000000E+00
U(x,y) = -9.8999249660E-01
```

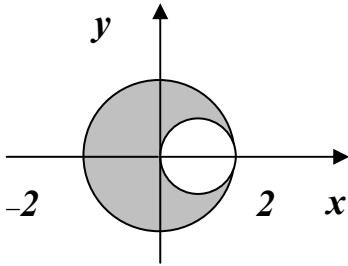
Такая программная реализация является весьма традиционной для структурного программирования, в ней осуществляется вычисление значений только двух логических выражений из трех возможных.

Приведем фрагмент из трех инструкций другой не менее традиционной программной реализации для вычисления значения функции $U(x, y)$:

```
if x + y < 0 then u := sin(x + y);
if x + y = 0 then u := sin(x) * cos(y);
if x + y > 0 then u := cos(x + y);
```

Очевидным недостатком такой реализации является вычисление значения всех трех логических выражений вместо двух, другой недостаток связан с тем обстоятельством, что в арифметике с плавающей точкой следует по возможности избегать сравнения с нулем. Проблема так называемого "машинного нуля" заслуживает отдельного разговора, который продолжим в главе, посвященной программированию циклических алгоритмов.

Пример 17. Рассмотрим плоскую мишень, границами которой являются две окружности $x^2 + y^2 = 4$ и $(x - 1)^2 + y^2 = 1$. Задавая координаты выстрела (x, y) , требуется определить состояние попадания или непопадания в мишень:



Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Ввести значение y*
4. *Если $((x - 1)^2 + y^2 \geq 1)$ и $(x^2 + y^2 \leq 4)$*
то
 - 4.1. *Вывести значение "Попадание в мишень!"*
 - иначе
 - 4.2. *Вывести значение "Непопадание в мишень!"*
5. *Конец*

Теперь представим программную реализацию:

```
{ Пример 17 }
{ Условные алгоритмы }
type  real_type = real;
var   x, y : real_type;
begin
  writeln('Ваш выстрел в мишень');
  write('x? '); readln(x);
  write('y? '); readln(y);
  if (sqr(x - 1) + sqr(y) >= 1) and (sqr(x) + sqr(y) <= 4)
    then writeln('Попадание в мишень!')
    else writeln('Непопадание в мишень!')
end.
```

Результат работы программы:

Ваш выстрел в мишень

х? 0

у? 1

Попадание в мишень!

Пример 18. Проверить, является ли заданное четырехзначное натуральное число палиндромом. Палиндромом называют такое число, которое одинаково читается как слева направо, так и справа налево, например, число **1221** будет палиндромом, а число **1223** – нет. Заметим, что наряду с числами палиндромами могут быть как слова, так и фразы, например: слова **казак**, **кабак** и **шалаш** – это палиндромы, а фразы **А роза упала на лапу Азора** и **Леша на полке клопа нашел** – это также палиндромы.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение натурального числа n*
3. *Если $(n \geq 1000)$ и $(n \leq 9999)$*
то
 - 3.1. *Положить $n_{1000} = n \div 1000$*
 - 3.2. *Положить $n = n \bmod 1000$*
 - 3.3. *Положить $n_{100} = n \div 100$*
 - 3.4. *Положить $n = n \bmod 100$*
 - 3.5. *Положить $n_{10} = n \div 10$*
 - 3.6. *Положить $n_1 = n \bmod 10$*
 - 3.7. *Если $(n_{1000} = n_1)$ и $(n_{100} = n_{10})$*
то
 - 3.7.1. *Вывести значение “Это палиндром!”*
 - 3.7.2. *Вывести значение “Это не палиндром!”*
4. *Конец*

Теперь представим программную реализацию:

```
{ Пример 18 }
{ Условные алгоритмы }
type   integer_type = integer;
var     n, n_1000, n_100, n_10, n_1 : integer_type;
begin
  write('Число из диапазона 1000..9999? '); readln(n);
  if (n >= 1000) and (n <= 9999) then begin
    n_1000 := n div 1000;
    n := n mod 1000;
```

```

n_100 := n div 100;
n := n mod 100;
n_10 := n div 10;
n_1 := n mod 10;
if (n_1000 = n_1) and (n_100 = n_10)
    then writeln('Это палиндром!')
    else writeln('Это не палиндром!')
end
end.

```

Результат работы программы:

Число из диапазона 1000..9999? 5665
Это палиндром!

Здесь с помощью условной инструкции *if* реализован так называемый механизм защиты переменной *n* от некорректного ввода ее значений – если какое-либо число при вводе не принадлежит диапазону четырехзначных чисел, то это число не следует проверять, является оно палиндромом или нет. Создание механизма защиты с использованием условной инструкции *if* является одним из традиционных способов решения этой проблемы. Существуют и другие традиционные способы организации такой защиты, которые будут представлены в главе, посвященной программированию циклических алгоритмов.

Пример 19. Представим здесь еще одну реализацию механизма защиты переменной *n* от некорректного ввода ее значений с использованием условной инструкции *if* и инструкции перехода *goto*. Заметим, что использование инструкции перехода *goto* противоречит парадигме структурного программирования, здесь же применение этой инструкции можно оправдать лишь в рамках демонстрационной программы как своеобразную дань традиции, исторически предшествовавшей принципам структурного программирования.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение натурального числа n*
3. *Если (n >= 1000) и (n <= 9999)*
то
 - 3.1. *Положить $n_{1000} = n \div 1000$*
 - 3.2. *Положить $n = n \bmod 1000$*
 - 3.3. *Положить $n_{100} = n \div 100$*
 - 3.4. *Положить $n = n \bmod 100$*
 - 3.5. *Положить $n_{10} = n \div 10$*
 - 3.6. *Положить $n_1 = n \bmod 10$*
 - 3.7. *Если ($n_{1000} = n_1$) и ($n_{100} = n_{10}$)*
то

- 3.7.1. Вывести значение “Это палиндром!”
 иначе
 3.7.2. Вывести значение “Это не палиндром!”
 иначе
 3.8. Перейти к пункту 2
 4. Конец

Теперь представим программную реализацию:

```
{ Пример 19 }
{ Условные алгоритмы }
type integer_type = integer;
var n, n_1000, n_100, n_10, n_1 : integer_type;
label 1;
begin
    1 : write(' Число из диапазона 1000..9999? '); readln(n);
    if (n >= 1000) and (n <= 9999) then begin
        n_1000 := n div 1000;
        n := n mod 1000;
        n_100 := n div 100;
        n := n mod 100;
        n_10 := n div 10;
        n_1 := n mod 10;
        if (n_1000 = n_1) and (n_100 = n_10)
            then writeln('Это палиндром!')
            else writeln('Это не палиндром!')
        end
        else goto 1
    end.
```

Как видим, использование условной инструкции *if* и инструкции перехода *goto* позволяет реализовать итеративный механизм защиты переменной от некорректного ввода ее значений. В главе, посвященной программированию циклических алгоритмов, будут представлены управляющие структуры, допускающие итеративную передачу управления, которые не противоречат парадигме структурного программирования.

Инструкция выбора

Для определения инструкции выбора можно использовать две синтаксические формы – полную и краткую:

```
case ключ_селектор of ветви_выбора else инструкция end;
case ключ_селектор of ветви_выбора end;
```

Здесь *ключ_селектор* – выражение порядкового типа, *ветви_выбора* – последовательность *case*-инструкций.

Case-инструкция представляет собой помеченную инструкцию вида:

список_констант_выбора : инструкция;

Здесь *список_констант_выбора* – список константных выражений того же типа, что и *ключ_селектор*. Элементы списка разделяются знаком препинания запятой.

Полная форма инструкции выбора *case* позволяет выполнить какую-либо одну из множества альтернативных ветвей алгоритма. Если значение выражения *ключ_селектор* совпадает со значением какого-либо константного выражения *константа_выбора*, то выполняется инструкция, связанная с этой *ветвью_выбора*, в противном случае – инструкция в части *else*.

Краткая форма инструкции выбора *case* позволяет либо выполнить только одну из множества альтернативных ветвей алгоритма, либо не выполнить ни одну из них. Если значение выражения *ключ_селектор* совпадает со значением какого-либо константного выражения *константа_выбора*, то выполняется инструкция, связанная с этой *ветвью_выбора*, в противном случае инструкция выбора *case* становится пустой.

Рассмотрим два условных алгоритма с множеством альтернативных ветвей и их программные реализации.

Пример 20. Требуется дать корректный ответ на вопрос о чьем-либо возрасте для заданного числа лет из диапазона, например, от 1 до 99.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Вывести значение ‘Ваш возраст 1..99?’*
3. *Ввести значение n*
4. *По значению ключа_селектора n выбрать*
Ветвь 1..99 :
 - 4.1. *Если $n \div 10 = 1$*
то
 - 4.1.1. *Вывести значение ‘Вам ‘, n, ‘ лет’*
 - иначе*
 - 4.1.2. *По значению ключа_селектора n mod 10 выбрать*
Ветвь 1 :
 - 4.1.2.1. *Вывести значение ‘Вам ‘, n, ‘ год’*
 - Ветвь 2..4 :*
 - 4.1.2.2. *Вывести значение ‘Вам ‘, n, ‘ года’*
 - иначе*
 - 4.1.2.3. *Вывести значение ‘Вам ‘, n, ‘ лет’*
 - иначе*
 - 4.2. *Вывести значение ‘Ошибка!’*
5. *Конец*

Теперь представим программную реализацию:

```

{ Пример 20 }
{ Условные алгоритмы }
type   integer_type = integer;
var     n : integer_type;
begin
  write('Ваш возраст 1..99? '); readln(n);
  case n of
    1..99 : if n div 10 = 1
              then writeln('Вам ', n, ' лет')
              else case n mod 10 of
                    1   : writeln('Вам ', n, ' год');
                    2..4 : writeln('Вам ', n, ' года');
                    else  writeln('Вам ', n, ' лет')
                  end
              else  writeln('Ошибка!')
            end
  end.

```

Результат работы программы:

```

Ваш возраст 1..99? 13
Вам 13 лет

```

Здесь с помощью инструкции выбора *case* создан один из традиционных способов реализации механизма защиты переменной *n* от некорректного ввода ее значений.

Теперь для иллюстрации *case*-инструкции со списком константных выражений представим ветвь для значения константы *2..4* в другом виде:

```

2, 3, 4 : writeln('Вам ', n, ' года');

```

Пример 21. Построить имитатор калькулятора для выполнения арифметических операций сложения, вычитания, умножения и деления для операндов вещественного типа *real*.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить флаг = false*
3. *Вывести значение '1-ый операнд x? '*
4. *Ввести значение x*
5. *Вывести значение '2-ой операнд y? '*
6. *Ввести значение y*
7. *Вывести значение 'Оператор? '*
8. *Ввести значение operator*
9. *По значению ключа_селектора operator выбрать Ветвь '+' :*
 - 9.1. *Положить $z = x + y$*

Ветвь ‘-’ :

9.2. Положить $z = x - y$

Ветвь ‘’ :*

9.3. Положить $z = x * y$

Ветвь ‘/’ :

9.4. Положить $z = x / y$

иначе

9.5. Положить $flag = true$

10. Если *not* флаг

то

10.1. Вывести значения ‘ x ’, *operator*, ‘ $y =$ ’, z

11. Конец

Теперь представим программную реализацию:

{ Пример 21 }

{ Условные алгоритмы }

const flag : boolean = false;

var operator : char;

x, y, z : real;

begin

write('1-ый операнд x? '); readln(x);

write('2-ой операнд y? '); readln(y);

write('Оператор? '); readln(operator);

case operator of

‘+’ : z := x + y;

‘-’ : z := x - y;

‘’ : z := x * y;*

‘/’ : z := x / y;

else flag := true

end;

if not flag then writeln('x ', operator, ' y = ', z)

end.

Результат работы программы:

1-ый операнд x ? 1

2-ой операнд y ? 2

Оператор? /

$x / y = 5.000000000000E-01$

Программирование циклических алгоритмов

Циклические алгоритмы позволяют организовать многократное выполнение какой-либо последовательности действий алгоритма до тех пор, пока не будет выполнено предписанное условие. При программировании циклических алгоритмов в языке Turbo Pascal используется одна из четырех инструкций повторения – инструкция повторения с предусловием, инструкция повторения с постусловием и две счетные инструкции повторения.

Инструкции повторения являются основой построения управляющих структур, допускающих итеративную передачу управления, которые не противоречат парадигме структурного программирования. Управляющие структуры следования и итерации позволяют трактовать циклические алгоритмы как линейные.

Инструкция повторения с предусловием

Определение инструкции повторения с предусловием:

while условие do инструкция;

Здесь ***условие*** – логическое выражение.

Инструкция повторения с предусловием позволяет организовать в программе цикл с условием продолжения. Первым действием тела цикла с условием продолжения является проверка условия продолжения цикла – вычисление значения логического выражения. Выполнение тела цикла осуществляется всякий раз, если значение логического выражения есть ***true***. Чтобы реализовать конечное число итераций, в теле цикла необходимо обеспечить возможность смены значения логического выражения на ***false***, а до тела цикла – инициализацию переменной цикла. Тело цикла с условием продолжения может либо вообще не выполниться, либо выполниться требуемое число раз, либо выполняться бесконечно много раз.

Рассмотрим первый циклический алгоритм и его программную реализацию.

Пример 22. Получить таблицу умножения для заданного натурального числа ***m*** из диапазона от ***1*** до ***9***, используя цикл с условием продолжения, и реализовать механизм защиты переменной ***m*** от некорректного ввода ее значений с помощью инструкции повторения с предусловием.

Представим традиционное описание алгоритма на естественном языке:

1. ***Начало***
2. ***Положить $m = 0$***
3. ***Если ($m < 1$) или ($m > 9$), то перейти к пункту 4, иначе перейти к пункту 6***
4. ***Ввести значение m***
5. ***Перейти к пункту 3***
6. ***Положить $k = 0$***
7. ***Если $k < 9$, то перейти к пункту 8, иначе перейти к пункту 11***
8. ***Положить $k = k + 1$***

9. Вывести значения m , 'х', k , '=' , $m * k$
10. Перейти к пункту 7
11. Конец

Представим структурированное описание алгоритма на естественном языке:

1. Начало
2. Положить $m = 0$
3. Пока $(m < 1)$ или $(m > 9)$ повторить
 - 3.1. Ввести значение m
4. Положить $k = 0$
5. Пока $k < 9$ повторить
 - 5.1. Положить $k = k + 1$
 - 5.2. Вывести значения m , 'х', k , '=' , $m * k$
6. Конец

Прежде, чем перейти к программной реализации алгоритма получения таблицы умножения и результатам работы программы, следует обратить внимание на механизм управления форматным выводом строк таблицы. Подробное изложение организации форматного вывода данных будет представлено в главе, посвященной работе с файлами. Здесь же отметим, что управление форматом вывода для элемента списка целочисленного типа означает указание ширины поля вывода при помощи константного выражения, которое отделяется от элемента списка вывода знаком препинания двоеточие. Для символьного представления значения $m * k$ следует отвести поле вывода шириной в две позиции.

Теперь представим программную реализацию:

```
{ Пример 22 }
{ Циклические алгоритмы }
type   integer_type = integer;
const  low_limit : integer_type = 0;
        high_limit : integer_type = 9;
        m : integer_type = 0;
var     k : integer_type;
begin
    while (m < 1) or (m > high_limit) do begin
        write('Множитель? '); readln(m)
    end;
    k := low_limit;
    while k < high_limit do begin
        k := k + 1;
        writeln(m, 'х', k, '=', m * k : 2)
    end
end.
```

Результат работы программы:

Множитель? 5

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

При обсуждении циклических алгоритмов в главе, посвященной типам алгоритмов, уже рассматривались условия запуска механизма “зацикливания” алгоритмов, сейчас же обратимся к механизму инициализации переменной цикла.

Особенностью рассматриваемой реализации является инициализация переменной цикла – типизированной константы *m* значением, которое равно значению нижней границы диапазона изменения переменной цикла *k*. Такая инициализация является необходимым условием для осуществления ввода корректного значения этой типизированной константы, принадлежащего диапазону изменения переменной цикла *k*, при помощи инструкции повторения с предусловием. Однако очевидно, что такая реализация механизма защиты переменных от некорректного ввода их значений не может являться универсальной, так как возможна ситуация, когда цикл может не выполниться вообще. Например, если значение *low_limit* будет являться минимальным значением диапазона допустимых значений целочисленного типа или если значение *high_limit* будет являться максимальным значением этого диапазона, то никакая инициализация не позволит запустить этот механизм защиты.

Инструкция повторения с постусловием

Определение инструкции повторения с постусловием:

repeat инструкция until условие;

Здесь *условие* – логическое выражение.

Инструкция повторения с постусловием позволяет организовать в программе цикл с условием завершения. Последним действием тела цикла с условием завершения является проверка условия завершения цикла – вычисление значения логического выражения. Выполнение тела цикла осуществляется всякий раз, если значение логического выражения есть *false*. Чтобы реализовать конечное число итераций, в теле цикла необходимо обеспечить возможность смены значения логического выражения на *true*, а до тела цикла – инициализацию переменной цикла. Тело цикла с условием завершения может либо выполниться только один раз, либо выполниться требуемое число раз, либо выполняться бесконечно много раз.

Продолжим обсуждение алгоритма получения таблицы умножения и его программной реализации.

Пример 23. Получить таблицу умножения для заданного натурального числа m из диапазона от 1 до 9, используя цикл с условием завершения, и реализовать механизм защиты переменной m от некорректного ввода ее значений с помощью инструкции повторения с постусловием.

Представим традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение m*
3. *Если ($m \geq 1$) и ($m \leq 9$), то перейти к пункту 4, иначе перейти к пункту 2*
4. *Положить $k = 0$*
5. *Положить $k = k + 1$*
6. *Вывести значения m , 'x', k , '=', $m * k$*
7. *Если $k = 9$, то перейти к пункту 8, иначе перейти к пункту 5*
8. *Конец*

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Повторить*
 - 2.1. *Ввести значение m пока не будет ($m \geq 1$) и ($m \leq 9$)*
3. *Положить $k = 0$*
4. *Повторить*
 - 4.1. *Положить $k = k + 1$*
 - 4.2. *Вывести значения m , 'x', k , '=', $m * k$ пока не будет $k = 9$*
5. *Конец*

Теперь представим программную реализацию:

```
{ Пример 23 }  
{ Циклические алгоритмы }  
type   integer_type = integer;  
const  low_limit : integer_type = 0;  
        high_limit : integer_type = 9;  
var     k , m : integer_type;  
begin  
    repeat  
        write('Множитель? '); readln(m)  
    until (m >= 1) and (m <= high_limit);  
    k := low_limit;
```



```

repeat
   $k := k + 1$ ;
  writeln( $m$ , ' x ',  $k$ , '=',  $m * k : 2$ )
until  $k = high\_limit$ 
end.

```

Как видим, предложенная реализация механизма защиты переменной цикла от некорректного ввода ее значений уже может являться универсальной, так как при вводе значение переменной цикла – типизированной константы *m* – никогда не выйдет за границы диапазона допустимых значений своего целочисленного типа.

Итак, программирование циклических алгоритмов в случае использования целочисленных переменных цикла требует от пользователя умения предвидеть возможные так называемые “аварийные” ситуации и избегать их. При этом следует придерживаться следующего правила: значения целочисленных переменных, от которых зависит, каким будет цикл, не должны выходить за границы диапазона допустимых значений своего целочисленного типа.

А теперь обратимся к проблеме вычисления так называемого “машинного нуля”. В дискретной арифметике с плавающей точкой всегда существуют такие числа $0 < x < \varepsilon$, для которых будет справедливым, например, равенство $1.0 + x = 1.0$, т.е. аппроксимация бесконечного непрерывного множества вещественных чисел конечным (пусть даже и очень большим) множеством их внутреннего представления как раз и приводит к появлению “машинного нуля”.

Будем рассматривать только программные реализации для вещественных типов *real*, *single*, *double* и *extended* для вычисления “машинного нуля”. Вычисления будем проводить методом дихотомии.

Пример 24. Вычислить “машинный нуль” для типа *real*:

```

{ Пример 24 }
{ Циклические алгоритмы }
type  real_type = real;
const epsilon : real_type = 1.0;    { “машинный нуль” }
begin
  while  $1.0 + epsilon > 1.0$  do  $epsilon := epsilon / 2$ ;
  writeln(epsilon)
end.

```

Результат работы программы:

9.0949470177E-13

Пример 25. Чтобы вычислить “машинный нуль” для остальных вещественных типов (*single*, *double* и *extended*), оказывается, что в тексте этой программы недостаточно будет лишь определить пользовательский тип *real_type* для соответствующего стандартного типа, например, для типа *double* так:

```
{ Пример 25 }
{ Циклические алгоритмы }
{$N+}
type   real_type = double;
const  epsilon : real_type = 1.0;      { “машинный нуль” }
begin
    while 1.0 + epsilon > 1.0 do epsilon := epsilon / 2;
    writeln(epsilon)
end.
```

Результат работы программы:

5.42101086242752E-0020

Попытка вычислить “машинный нуль” для вещественных типов *single* и *extended* (и даже для типа *real*) приведет к некорректным результатам, а именно для всех: этих типов получится одно и то же значение — **5.42101086242752E-0020**.

Причина здесь в том, что в случае использования числового сопроцессора выражение **1.0 + epsilon** будет неявно преобразовываться к типу *extended*. Чтобы получить правильные результаты, значение вычисленного выражения **1.0 + epsilon** в теле цикла необходимо присваивать переменной соответствующего стандартного типа. В этом случае для каждой переменной какого-либо вещественного типа (кроме типа *extended*) будет происходить усечение результатов до нужных размеров.

Пример 26. Вычислить “машинный нуль” для типа *real* в случае использования числового сопроцессора:

```
{ Пример 26 }
{ Циклические алгоритмы }
{$N+}
type   real_type = real;
const  epsilon : real_type = 1.0;      { “машинный нуль” }
var    nearly_one : real_type;
begin
    repeat
        epsilon := epsilon / 2;
        nearly_one := 1.0 + epsilon;
    until nearly_one = 1.0;
    writeln(epsilon)
end.
```

Результат работы программы:

4.54747350886464E-0013

Как видим, значение “машинного нуля” для вещественного типа *real* стало вдвое меньше (т.е. точнее по сравнению с предыдущим вычислением) благодаря использованию для вычислений числового сопроцессора (или его программной эмуляции). Теперь чтобы вычислить “машинный нуль” для остальных вещественных типов (*single*, *double* и *extended*), в тексте этой программы достаточно будет лишь определить пользовательский тип *real_type* для соответствующего стандартного типа:

<i>5.96046447753906E-0008</i>	для <i>muna single</i>
<i>1.11022302462516E-0016</i>	для <i>muna double</i>
<i>5.42101086242752E-0020</i>	для <i>muna extended</i>

Счетная инструкция повторения

Для определения счетной инструкции повторения используются две синтаксические формы:

for имя_переменной := выражение to выражение do инструкция;

for имя_переменной := выражение downto выражение do инструкция;

Здесь *имя_переменной* – переменная цикла порядкового типа, *выражение* – выражение порядкового типа. Счетная инструкция повторения позволяет организовать в программе счетный цикл. Тело счетного цикла выполняется фиксированное число раз благодаря неявному изменению переменной цикла с постоянным шагом в заданном диапазоне значений либо от начального к конечному (так называемый автоинкремент – первая синтаксическая форма), либо от конечного к начальному (так называемый автодекремент – вторая синтаксическая форма).

Продолжим обсуждение алгоритма получения таблицы умножения и его программной реализации.

Пример 27. Получить таблицу умножения для заданного натурального числа *m* из диапазона от *1* до *9*, используя счетный цикл, и реализовать механизм защиты переменной *m* от некорректного ввода ее значений с помощью инструкции повторения с постусловием.

Как известно, счетный цикл является частным случаем цикла с условием продолжения, поэтому представим здесь только структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Повторить*
 - 2.1. *Ввести значение m*
пока не будет (m >= 1) и (m <= 9)
3. *От k = 1 до k = 9 повторить*
 - 3.1. *Вывести значения m, ' x ', k, '=', m * k*
4. *Конец*

Теперь представим программную реализацию:

```
{ Пример 27 }
{ Циклические алгоритмы }
type   integer_type = integer;
const  low_limit : integer_type = 1;
        high_limit : integer_type = 9;
var     k , m : integer_type;
begin
  repeat
    write('Множитель? '); readln(m)
  until (m >= low_limit) and (m <= high_limit);
  for k := low_limit to high_limit do writeln(m, ' x ', k, '= ', m * k : 2)
end.
```

В заключение обратимся к одной из классических задач вычислительной математики – проблеме табулирования функции одного переменного $f(x)$ для заданного числа узлов табулирования n на некотором отрезке $x \in [a, b]$. В результате табулирования получаются таблицы, содержащие значения аргумента и соответствующие значения табулируемой функции.

Чтобы вычислить значения аргумента для каждого узла табулирования и соответствующие значения табулируемой функции, можно поступить двояким образом: в качестве переменной цикла можно выбрать либо порядковый номер узла, либо само значение аргумента для узла. В случае эквидистантного шага таблицы значений аргумента (т.е. постоянной величины приращения аргумента) для каждой из рассматриваемых альтернатив значения аргумента табулируемой функции каких-либо двух соседних узлов будут связаны соотношением $x_{k+1} = x_k + h$ для любого $k \in [1, n]$, где $k + 1$ и k – порядковые номера этих узлов, при этом $x_1 = a$ и $x_n = b$, а шаг табулирования $h = \frac{b - a}{n - 1}$.

Если переменная цикла будет целочисленного типа, то для организации цикла можно использовать любую инструкцию повторения, обычно предпочтение отдают счетным инструкциям повторения, которые сами автоинкрементируют (или автодекрементируют) переменную цикла после очередной итерации.

Пример 28. Протабулировать функцию $f(x) = e^x - e^{-x} - 2$ на отрезке $[0, 1]$ для заданного числа узлов табулирования n , используя счетный цикл, и реализовать механизм защиты переменной n , а также переменных a и b , которые являются соответственно нижней и верхней границами отрезка, от некорректного ввода их значений с помощью инструкции повторения с постусловием.

Заметим, что счетный цикл предпочтительней будет организовать так, чтобы переменная цикла $k \in [0, n - 1]$, что позволит в теле цикла значение аргумента x для каждого узла вычислять по формуле $x = a + k \cdot h$.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $a = 0$*
3. *Положить $b = 1$*
4. *Повторить*
 - 4.1. *Ввести значение числа узлов табулирования n пока не будет $n > 1$*
5. *Положить $h = (b - a) / (n - 1)$*
6. *От $k = 0$ до $k = n - 1$ повторить*
 - 6.1. *Положить $x = a + k * h$*
 - 6.2. *Вывести значения $x, f(x)$*
7. *Конец*

Теперь представим программную реализацию:

```
{ Пример 28 }
{ Циклические алгоритмы }
type   integer_type = integer;
       real_type = real;
const  a : real_type = 0.0;
       b : real_type = 1.0;
var     k, n : integer_type;
       x, h : real_type;
begin
  repeat
    write('Число узлов табулирования? '); readln(n)
  until n > 1;
  h := (b - a) / (n - 1);
  for k := 0 to n - 1 do begin
    x := a + k * h;
    writeln((k + 1) : 2, x : 19, (exp(x) - exp(-x) - 2) : 22)
  end
end.
```

Результат работы программы:

```
Число узлов табулирования? 6
1    0.0000000000E+00    -2.0000000000E+00
2    2.0000000000E-01    -1.5973279949E+00
3    4.0000000000E-01    -1.1784953484E+00
4    6.0000000000E-01    -7.2669283570E-01
5    8.0000000000E-01    -2.2378803563E-01
6    1.0000000000E+00    3.5040238729E-01
```

Если переменная цикла будет вещественного типа, то для организации цикла можно использовать только одну из двух инструкций повторения – с предусловием или с постусловием. Однако программные реализации таких циклов отличаются одним неприятным свойством – возможной потерей последней итерации. Причина здесь в том, что значение переменной цикла для последней итерации в логическом выражении инструкции повторения с предусловием или постусловием проверяется на равенство со значением верхней границы отрезка табулирования функции, в ряде случаев значение переменной цикла оказывается несколько больше значения верхней границы отрезка из-за так называемого накопления ошибок округления при суммировании переменной цикла и приращения аргумента.

Пример 29. Представить программную реализацию для цикла с условием продолжения для иллюстрации возможности потери последней итерации:

{ Пример 29 }

{ Циклические алгоритмы }

```

type    integer_type = integer;
        real_type = real;
const  a : real_type = 0.0;
        b : real_type = 1.0;
var    k, n : integer_type;
        x, h : real_type;
begin
  repeat
    write('Число узлов табулирования? '); readln(n)
  until n > 1;
  h := (b - a) / (n - 1);
  k := 0;
  x := a;
  while x <= b do begin
    writeln((k + 1) : 2, x : 19, (exp(x) - exp(-x) - 2) : 22)
    k := k + 1;
    x := x + h
  end
end.

```

Результат работы программы:

```

Число узлов табулирования? 6
1    0.0000000000E+00    -2.0000000000E+00
2    2.0000000000E-01    -1.5973279949E+00
3    4.0000000000E-01    -1.1784953484E+00
4    6.0000000000E-01    -7.2669283570E-01
5    8.0000000000E-01    -2.2378803563E-01
6    1.0000000000E+00     3.5040238729E-01

```

Как видим, при таком шаге табулирования потери последней итерации не произошло. Однако, в результате уменьшения шага табулирования всего вдвое произойдет потеря последней итерации:

Число узлов табулирования? 11

1	0.0000000000E+00	-2.0000000000E+00
2	1.0000000000E-01	-1.7996665000E+00
3	2.0000000000E-01	-1.5973279949E+00
4	3.0000000000E-01	-1.3909594131E+00
5	4.0000000000E-01	-1.1784953484E+00
6	5.0000000000E-01	-9.5780938901E-01
7	6.0000000000E-01	-7.2669283570E-01
8	7.0000000000E-01	-4.8283259632E-01
9	8.0000000000E-01	-2.2378803563E-01
10	9.0000000000E-01	5.3033451426E-02

Чтобы исключить эффект накопления ошибок округления при выполнении операции $x = x + h$, запишем, например, цикл с условием продолжения в другом виде, где значение переменной цикла будет вычисляться как $x = a + k \cdot h$:

```
while x <= b do begin
  writeln((k + 1) : 2, x : 19, (exp(x) - exp(-x) - 2) : 22)
  k := k + 1;
  x := a + k * h
end
```

В этом случае потери последней итерации не происходит:

Число узлов табулирования? 11

1	0.0000000000E+00	-2.0000000000E+00
2	1.0000000000E-01	-1.7996665000E+00
3	2.0000000000E-01	-1.5973279949E+00
4	3.0000000000E-01	-1.3909594131E+00
5	4.0000000000E-01	-1.1784953484E+00
6	5.0000000000E-01	-9.5780938901E-01
7	6.0000000000E-01	-7.2669283570E-01
8	7.0000000000E-01	-4.8283259632E-01
9	8.0000000000E-01	-2.2378803563E-01
10	9.0000000000E-01	5.3033451426E-02
11	1.0000000000E+00	3.5040238729E-01

В заключение отметим, что в языке Turbo Pascal версии 7.0 в программных реализациях циклов для управления переменной цикла можно использовать две стандартные процедуры – *break* и *continue*. Процедура *break* позволяет досрочно

выйти из цикла, не дожидаясь выполнения условия выхода, а процедура *continue* позволяет перейти к следующей итерации цикла.

Рассмотрим пример использования процедур *break* и *continue*.

Пример 30. Вычислить сумму четных чисел среди заданных от одного до девяти целых чисел, используя при этом число 0 в качестве признака завершения операции ввода чисел.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $s = 0$*
3. *От $k = 1$ до $k = 9$ повторить*
 - 3.1. *Вывести значения k , ‘–е число? ‘*
 - 3.2. *Ввести значение n*
 - 3.3. *Если $n = 0$*
то
 - 3.3.1. *Вызвать процедуру *break**
 - иначе*
 - 3.3.2. *Если $n \bmod 2 \neq 0$*
то
 - 3.3.2.1. *Вызвать процедуру *continue**
 - 3.3.2.2. *Положить $s = s + n$*
4. *Если $s \neq 0$*
то
 - 4.1. *Вывести значения ‘Сумма = ‘, s*
5. *Конец*

Представим программную реализацию:

```
{ Пример 30 }
{ Циклические алгоритмы }
type   integer_type = integer;
const  s : longint = 0;
var    k, n : integer_type;
begin
  for k := 1 to 9 do begin
    write(k, '–е число? '); readln(n);
    if n = 0
      then break
      else if n mod 2 <> 0 then continue else s := s + n
  end;
  if s <> 0 then writeln('Сумма = ', s)
end.
```


Результат работы программы:

```
1-е число? 1
2-е число? 2
3-е число? 4
4-е число? -2
5-е число? 0
Сумма = 4
```

Вместо стандартных процедур *break* и *continue* можно использовать логическую переменную – флаг, значение которого определяет условие досрочного выхода из цикла или перехода к следующей итерации. В частности, этим приемом приходится часто пользоваться в программах Turbo Pascal для версии 6.0.

Пример 31. Представить альтернативный вариант программной реализации предыдущего примера для версии 6.0 языка Turbo Pascal:

```
{ Пример 31 }
{ Циклические алгоритмы }
type   integer_type = integer;
const  s : longint = 0;
       k : integer_type = 0;
       flag : boolean = false;
var     n : integer_type;
begin
  repeat
    k := k + 1;
    write(k, '-е число? '); readln(n);
    if n = 0
      then flag := true
      else if n mod 2 <> 0 then s := s + n
  until (k = 9) or flag;
  if s <> 0 then writeln('Сумма = ', s)
end.
```

Структурированные типы. Массивы

В языке Turbo Pascal имеется четыре структурированных типа: массивы, записи, множества и файлы. Любой из структурированных типов характеризуется множественностью элементов, т.е. константа или переменная структурированного типа всегда имеет несколько компонентов. Каждый компонент, в свою очередь, может принадлежать структурированному типу, что позволяет организовать произвольную глубину вложенности типов, однако принято, что суммарная длина любого из них во внутреннем представлении не должна превышать 65520 байт. Из четырех структурированных типов рассмотрим только два типа – массивы и файлы.

Необходимость в массивах возникает всякий раз, когда при решении задачи приходится иметь дело с большим, но конечным количеством однотипных упорядоченных данных. Массивы представляют собой такие структуры данных, которые состоят из упорядоченного набора фиксированного количества однотипных компонентов, причем индивидуальное имя получает только весь набор, а для компонентов этого набора определяется лишь порядок следования и общее их количество. Каждый элемент массива, представляющий собой минимальную структурную единицу, имеет однозначно определенное множество индексов, которые определяют его положение в упорядоченном наборе, т.е. задают правило вычисления его номера. Каждый элемент массива может быть явно обозначен посредством указания имени массива и множества индексов.

Размерность массива определяется числом его индексов. Одномерный массив, или вектор, представляет собой список элементов, обозначаемых одиночными индексами. Если a – одномерный массив, а i – значение индекса, то i -ый элемент a обозначается через a_i . Двумерный массив, или матрица, представляет собой список векторов, или таблицу элементов, состоящую из фиксированного числа строк и столбцов. Элементы двумерного массива обозначаются двумя индексами. Если a – двумерный массив, а i и j – значения индексов, то i, j -ый элемент a , который располагается в i -ой строке и j -ом столбце, обозначается через a_{ij} . Трехмерный массив представляет собой список двумерных массивов. Элементы трехмерного массива обозначаются тремя индексами. Если a – трехмерный массив, а i, j и k – значения индексов, то i, j, k -ый элемент a , который располагается в i -ом двумерном массиве в его j -ой строке и k -ом столбце, обозначается через a_{ijk} . Аналогичным образом определяются и массивы более высоких размерностей – n -мерный массив представляет собой список компонент, размерность каждого из которых $n - 1$.

Объявление типа массив:

array [список_индексных_типов] of тип_элемента;

Здесь ***список_индексных_типов*** – список порядковых типов, который определяет размерность массива, ***тип_элемента*** – тип языка Turbo Pascal. Элементы списка индексных типов разделяются знаком препинания запятой.

Например, определим одномерный массив элементов типа ***char***:

var a : array[byte] of char;

Число элементов массива **a** равно 256, так как оно определяется количеством возможных значений индексного типа **byte**. Диапазон изменения значений индекса элемента массива соответствует диапазону возможных значений типа **byte**, который составляет $[0, 255]$. Компилятор выделит память под массив **a** размером $256 \times 1 = 256$ байт, где 1 соответствует размеру в байтах элемента массива.

Например, определим одномерный массив элементов типа **integer**:

var a : array[shortint] of integer;

Число элементов массива **a** равно 256, так как оно определяется количеством возможных значений индексного типа **shortint**. Диапазон изменения значений индекса элемента массива соответствует диапазону возможных значений типа **shortint**, который составляет $[-128, 127]$. Компилятор выделит память под массив **a** размером 256×2 байт, где 2 соответствует размеру в байтах элемента массива.

После объявления константы или переменной типа массив для доступа к какому-либо элементу массива необходимо обратиться к имени массива, следом за которым в квадратных скобках указывается список индексов.

Если объявления типизированных констант простых типов не вызывают затруднений, так как в качестве их значений используются нетипизированные константы или их идентификаторы, то при определении типизированных констант типа массив (или констант-массивов) для записи их значений требуется соблюдение особых правил синтаксиса.

При определении одномерных констант-массивов для инициализации используется заключенный в круглые скобки список констант, элементы списка при этом разделяются знаком препинания запятой.

При определении многомерных констант-массивов для инициализации используется заключенный в круглые скобки список, каждый элемент которого заключается в дополнительные круглые скобки, элементы списка разделяются знаком препинания запятой, при этом каждый элемент представляет собой вложенный список констант, соответствующих каждому измерению, глубина вложения должна при этом соответствовать количеству измерений (размерности массива). В этих вложенных структурах списков констант самые внутренние списки констант связаны с изменением самого правого индекса массива.

В оперативной памяти персонального компьютера элементы массива следуют друг за другом в порядке возрастания самого правого индекса массива.

С помощью инструкции присваивания можно передать значения всех элементов одного массива другому массиву того же типа. В результате выполнения такой операции присваивания можно получать копии массивов. Однако над массивами не определены операции отношения, т.е. сравнение двух массивов должно осуществляться только поэлементно.

В качестве индексных типов можно использовать любые порядковые типы, кроме **longint** и интервальных типов с базовым типом **longint**. Обычно в качестве индексных типов используется интервальный тип, который задает границы изменения индекса. При этом в качестве значений границ интервального типа обычно используются либо нетипизированные константы, либо их идентификаторы.

Заметим, что в правильно спроектированной программе любой индекс массива не должен выходить за пределы, определенные интервальным типом, при этом использование индексов можно контролировать как на этапе времени компиляции, так и на этапе времени выполнения программы.

Одномерные массивы

Рассмотрим несколько традиционных способов определения констант-массивов и переменных типа массив, используя интервальный тип с базовым типом *integer*:

- определение константы-массива и переменной типа массив с использованием определяемого пользователем типа массив

```
type    array_type = array [1..5] of integer;
const   c : array_type = (1, 2, 3, 4, 5);
var     a : array_type;
```

- определение константы-массива и переменной типа массив непосредственно при их объявлении

```
const   c : array [-2..2] of shortint = (1, 2, 3, 4, 5);
var     a : array [-2..2] of shortint;
```

- определение константы-массива и переменной типа массив с использованием определяемого пользователем интервального типа

```
type    subrange_type = 0..4;
const   c : array [subrange_type] of char = ('1', '2', '3', '4', '5');
var     a : array [subrange_type] of char;
```

- определение константы-массива и переменной типа массив с использованием интервального типа, границы которого являются идентификаторами

```
const   low_limit = 0;
        high_limit = 1;
        c : array [low_limit..high_limit] of boolean = (false, true);
var     a : array [low_limit..high_limit] of boolean;
```

Теперь рассмотрим несколько примеров, связанных с характерными приемами работы с одномерными массивами.

Пример 32. Представить механизмы инициализации, копирования, визуализации и сравнения элементов одномерных массивов с помощью программы, в которой определены константы-массивы и переменные типа массив:

{ Пример 32 }

{ Структурированные типы. Массивы }

```
type    subrange_type = 0..4;
        array_type = array [1..5] of integer;
const   low_limit = 0;
        high_limit = 1;
        c1 : array_type = (1, 2, 3, 4, 5);
        c2 : array [-2..2] of shortint = (1, 2, 3, 4, 5);
```

```

    c3 : array [subrange_type] of char = ('1', '2', '3', '4', '5');
    c4 : array [low_limit..high_limit] of boolean = (false, true);
var    k : integer;
    a1 : array_type;
    a2 : array [-2..2] of shortint;
    a3 : array [subrange_type] of char;
    a4 : array [low_limit..high_limit] of boolean;
begin
    { Визуализация массива c1 }
    for k := 1 to 5 do write(c1[k] : 3);
    writeln;
    { Визуализация массива c2 }
    for k := -2 to 2 do write(c2[k] : 3);
    writeln;
    { Визуализация массива c3 }
    for k := 0 to 4 do write(c3[k] : 3);
    writeln;
    { Визуализация массива c4 }
    for k := low_limit to high_limit do write(c4[k] : 6);
    writeln;
    { Копирование массивов }
    a1 := c1;
    { Визуализация массива a1 как копии массива c1 }
    for k := 1 to 5 do write(a1[k]:3);
    writeln;
    { Инициализация массива a2 }
    a2[-2] := -2;
    a2[-1] := -1;
    a2[0] := 0;
    a2[1] := 1;
    a2[2] := 2;
    { Визуализация массива a2 }
    for k := -2 to 2 do write(a2[k] : 3);
    writeln;
    { Инициализация массива a3 }
    for k := 0 to 4 do begin
        write('array[', k, ']? '); readln(a3[k])
    end;
    { Визуализация массива a3 }
    for k := 0 to 4 do write(a3[k] : 3);
    writeln;
    { Поэлементное сравнение массивов }
    { Инициализация массива a4 значениями булевых констант }
    for k := 1 to 2 do if a1[k] = c1[k] then a4[k - 1] := true else a4[k - 1] := false;

```

```

{ Визуализация массива a4 }
for k := low_limit to high_limit do write(a4[k] : 6);
writeln;
{ Инициализация массива a4 значениями логических выражений }
a4[0] := a1[1] = c1[1];
a4[1] := a1[1] <> c1[1];
{ Визуализация массива a4 }
for k := 0 to 1 do write(a4[k] : 6);
writeln
end.

```

Результат работы программы:

```

 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
FALSE TRUE
 1  2  3  4  5
-2 -1  0  1  2
array[1]? 5
array[2]? 4
array[3]? 3
array[4]? 2
array[5]? 1
 5  4  3  2  1
 TRUE  TRUE
 TRUE FALSE

```

В оперативной памяти элементы одномерного массива, например, массива **a1** при переходе от младших адресов к старшим располагаются в следующем порядке:

a1[1], a1[2], a1[3], a1[4], a1[5].

Пример 33. Вычислить сумму $\sum_{k=1}^{10} a_k$ и произведение $\prod_{k=1}^{10} a_k$ для заданных значений элементов одномерного массива **a(10)**.

Представим структурированное описание алгоритма на естественном языке:

1. **Начало**
2. **Положить число элементов массива $n = 10$**
3. **От $k = 1$ до $k = n$ повторить**
 - 3.1. **Определить значение элемента массива $a[k]$**
4. **Положить сумму $S = 0$**
5. **Положить произведение $P = 1$**
6. **От $k = 1$ до $k = n$ повторить**

- 6.1. Положить $S = S + a[k]$
- 6.2. Положить $P = P * a[k]$
7. Вывести значение S
8. Вывести значение P
9. Конец

Теперь представим программную реализацию:

```
{ Пример 33 }
{ Структурированные типы. Массивы }
type integer_type = integer;
const n = 10;
type array_type = array [1..n] of integer_type;
const a : array_type = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
var k : integer_type;
    s, p : longint;
begin
    s := 0;
    p := 1;
    for k := 1 to n do begin
        s := s + a[k];
        p := p * a[k]
    end;
    writeln('Сумма: ', s);
    writeln('Произведение: ', p)
end.
```

Результат работы программы:

```
Сумма : 55
Произведение : 3628800
```

Пример 34. Упорядочить элементы одномерного массива по возрастанию методом “пузырьковой” сортировки.

Сортировка или упорядочение – это процесс, позволяющий упорядочить множество данных в возрастающем или убывающем порядке. Одним из самых известных (и самых скверных) методов сортировки является метод “пузырьковой” сортировки, который иногда называют злым духом перестановок. Например, чтобы упорядочить n элементов одномерного массива в возрастающем или убывающем порядке, необходимо выполнить $\frac{n^2 - n}{2}$ сравнений. “Пузырьковой” сортировку называют так потому, что в процессе упорядочения самый “легкий” элемент массива “всплывает” первым, затем – следующий и т.д.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Определить значение числа элементов массива n*
3. *От $i = 1$ до $i = n$ повторить*
 - 3.1. *Определить значение элемента массива $a[i]$*
4. *От $i = 1$ до $i = n$ повторить*
 - 4.1. *Вывести значение элемента массива $a[i]$*
5. *От $i = 1$ до $i = n - 1$ повторить*
 - 5.1. *От $j = i + 1$ до $j = n$ повторить*
 - 5.1.1. *Если $a[j] < a[i]$*
то
 - 5.1.1.1. *Положить $copy = a[i]$*
 - 5.1.1.2. *Положить $a[i] = a[j]$*
 - 5.1.1.3. *Положить $a[j] = copy$*
6. *От $i = 1$ до $i = n$ повторить*
 - 6.1. *Вывести значение элемента массива $a[i]$*
7. *Конец*

Теперь представим программную реализацию:

```
{ Пример 34 }
{ Структурированные типы. Массивы }
const  n = 10;
type   array_type = array [1..n] of integer;
const  a : array_type = (1, 3, 5, 0, 2, 4, 9, 7, 8, 6);
var     i, j, copy : integer;
begin
  { Визуализация массива }
  for i := 1 to n do write(a[i] : 3);
  writeln;
  { Сортировка массива }
  for i := 1 to n - 1 do
    for j := i + 1 to n do
      if a[j] < a[i] then begin
        copy := a[i];
        a[i] := a[j];
        a[j] := copy;
      end;
  { Визуализация массива }
  for i := 1 to n do write(a[i] : 3);
  writeln
end.
```


Результат работы программы:

```
1  3  5  0  2  4  9  7  8  6
0  1  2  3  4  5  6  7  8  9
```

Двумерные массивы

Рассмотрим несколько традиционных способов определения констант-массивов и переменных типа массив как двумерных массивов, например, матриц из двух строк и четырех столбцов, используя интервальный тип с базовым типом *integer*:

- определение константы-массива и переменной типа массив с использованием определяемого пользователем типа массив

```
type  matrix = array [1..2, 1..4] of integer;
const c : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
var   a : matrix;
```

- определение константы-массива и переменной типа массив с использованием определяемых пользователем типов массив

```
type  row_size = array [1..4] of integer;
      matrix = array [1..2] of row_size;
const c : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
var   a : matrix;
```

- определение константы-массива и переменной типа массив с использованием определяемых пользователем типов массив, для которых границы интервальных типов являются идентификаторами

```
const m = 2;
      n = 4;

type  row_size = array [1..n] of integer;
      matrix = array [1..m] of row_size;
const c : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
var   a : matrix;
```

Теперь рассмотрим несколько примеров, связанных с характерными приемами работы с двумерными массивами.

Пример 35. Представить механизмы инициализации и визуализации элементов двумерных массивов с помощью программы, в которой определены константа-массив и переменная типа массив.

```
{ Пример 35 }
{ Структурированные типы. Массивы }
const m = 2;
      n = 4;

type  row_size = array [1..n] of integer;
      matrix = array [1..m] of row_size;
const c : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
```

```

var    a : matrix;
       i, j : integer;
begin
  { Визуализация массива c }
  for i := 1 to m do begin
    for j := 1 to n do write(c[i,j] : 3);
    writeln
  end;
  { Инициализация массива a }
  for i := 1 to m do
    for j := 1 to n do begin
      write('a[', i, ', ', j, ']? '); readln(a[i,j])
    end;
  { Визуализация массива a }
  for i := 1 to m do begin
    for j := 1 to n do write(a[i,j] : 3);
    writeln
  end
end.

```

Результат работы программы:

```

  1  2  3  4
  5  6  7  8
a[1,1]? 8
a[1,2]? 7
a[1,3]? 6
a[1,4]? 5
a[2,1]? 4
a[2,2]? 3
a[2,3]? 2
a[2,4]? 1
  8  7  6  5
  4  3  2  1

```

В оперативной памяти элементы двумерного массива, например, массива *a* при переходе от младших адресов к старшим располагаются в следующем порядке:

a[1,1], a[1,2], a[1,3], a[1,4], a[2,1], a[2,2], a[2,3], a[2,4].

Пример 36. Построить так называемый “магический квадрат” – квадратную матрицу заданного размера, для которой суммы элементов каждой строки и каждого столбца равны одному и тому же числу.

Рассмотрим один из вариантов решения этой проблемы, в котором используется вектор, представляющий собой образец для заполнения “магического квадрата” произвольными числами.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Определить значение числа строк и столбцов m*
3. *От $i = 1$ до $i = m$ повторить*
 - 3.1. *Определить значение элемента массива $pattern[i]$*
4. *От $i = 1$ до $i = m$ повторить*
 - 4.1. *Положить индекс $k = i$*
 - 4.2. *От $j = 1$ до $j = m$ повторить*
 - 4.2.1. *Положить $a[i,j] = pattern[k]$*
 - 4.2.2. *Если $k = m$*
то
 - 4.2.2.1. *Положить $k = 1$*
 - иначе*
 - 4.2.2.2. *Положить $k = k + 1$*
5. *От $i = 1$ до $i = m$ повторить*
 - 5.1. *От $j = 1$ до $j = m$ повторить*
 - 5.1.1. *Вывести значение элемента массива $a[i,j]$*
6. *Конец*

Теперь представим программную реализацию:

```
{ Пример 36 }
{ Структурированные типы. Массивы }
const  m = 10;
type    row_size = array [1..m] of integer;
        matrix = array [1..m] of row_size;
const  pattern : row_size = (1, 3, 5, 0, 2, 4, 9, 7, 8, 6);
var     i, j, k : integer;
        a : matrix;
begin
  for i := 1 to m do begin
    k := i;
    for j := 1 to m do begin
      a[i,j] := pattern[k];
      if k = m then k := 1 else k := k + 1
    end
  end;
  { Визуализация массива }
  for i := 1 to m do begin
    for j := 1 to m do write(a[i,j] : 3);
    writeln
  end
end.
```

Результат работы программы:

1	3	5	0	2	4	9	7	8	6
3	5	0	2	4	9	7	8	6	1
5	0	2	4	9	7	8	6	1	3
0	2	4	9	7	8	6	1	3	5
2	4	9	7	8	6	1	3	5	0
4	9	7	8	6	1	3	5	0	2
9	7	8	6	1	3	5	0	2	4
7	8	6	1	3	5	0	2	4	9
8	6	1	3	5	0	2	4	9	7
6	1	3	5	0	2	4	9	7	8

Пример 37. В матрице из трех строк и четырех столбцов переставить местами любые две строки или любые два столбца.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить число строк $m = 3$*
3. *Положить число столбцов $n = 4$*
4. *От $i = 1$ до $i = m$ повторить*
 - 4.1. *От $j = 1$ до $j = n$ повторить*
 - 4.1.1. *Определить значение элемента массива $a[i,j]$*
5. *От $i = 1$ до $i = m$ повторить*
 - 5.1. *От $j = 1$ до $j = n$ повторить*
 - 5.1.1. *Вывести значение элемента массива $a[i,j]$*
6. *Повторить*
 - 6.1. *Вывести значение 'Поменять в матрице строку'*
 - 6.2. *Ввести значение номера строки k пока не будет ($k \geq 1$) и ($k \leq m$)*
7. *Повторить*
 - 7.1. *Вывести значение 'на строку'*
 - 7.2. *Ввести значение номера строки l пока не будет ($l \geq 1$) и ($l \leq m$) и ($l \neq k$)*
8. *От $j = 1$ до $j = n$ повторить*
 - 8.1. *Положить $сору = a[k,j]$*
 - 8.2. *Положить $a[k,j] = a[l,j]$*
 - 8.3. *Положить $a[l,j] = сору$*
9. *От $i = 1$ до $i = m$ повторить*
 - 9.1. *От $j = 1$ до $j = n$ повторить*
 - 9.1.1. *Вывести значение элемента массива $a[i,j]$*
10. *Конец*

Теперь представим программную реализацию:

```

{ Пример 37 }
{ Структурированные типы. Массивы }
const  m = 3;
        n = 4;
type    row_size = array [1..n] of integer;
        matrix = array [1..m] of row_size;
const  a : matrix = ((1, 2, 3, 4), (5, 6, 7, 8), (9,10,11,12));
var     i, j, k, l, copy: integer;
begin
    { Визуализация массива }
    for i := 1 to m do begin
        for j := 1 to n do write(a[i,j] : 3);
        writeln
    end;
    repeat
        write('Поменять в матрице строку '); readln(k)
    until (k >= 1) and (k <= m);
    repeat
        write('на строку '); readln(l)
    until (l >= 1) and (l <= m) and (l <> k);
    { Поэлементная перестановка местами строк }
    for j := 1 to n do begin
        copy := a[k,j];
        a[k,j] := a[l,j];
        a[l,j] := copy
    end;
    { Визуализация массива }
    for i := 1 to m do begin
        for j := 1 to n do write(a[i,j] : 3);
        writeln
    end
end.

```

Результат работы программы:

```

1  2  3  4
5  6  7  8
9 10 11 12

```

```

Поменять в матрице строку 1
на строку 3
9 10 11 12
5  6  7  8
1  2  3  4

```

Представим фрагмент текста программы, с помощью которого в этой матрице можно переставить местами любые два столбца:

```
repeat
  write('Поменять в матрице столбец '); readln(k)
until (k >= 1) and (k <= n);
repeat
  write('на столбец '); readln(l)
until (l >= 1) and (l <= n) and (l <> k);
{ Поэлементная перестановка местами столбцов }
for i := 1 to m do begin
  copy := a[i,k];
  a[i,k] := a[i,l];
  a[i,l] := copy;
end;
```

Результат работы программы:

```
1  2  3  4
5  6  7  8
9 10 11 12
Поменять в матрице столбец 1
на столбец 4
4  2  3  1
8  6  7  5
12 10 11 9
```

Пример 38. В матрице из трех строк и четырех столбцов найти минимальный и максимальный элементы и поменять их местами.

Представим структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить число строк $m = 3$*
3. *Положить число столбцов $n = 4$*
4. *От $i = 1$ до $i = m$ повторить*
 - 4.1. *От $j = 1$ до $j = n$ повторить*
 - 4.1.1. *Определить значение элемента массива $a[i,j]$*
5. *От $i = 1$ до $i = m$ повторить*
 - 5.1. *От $j = 1$ до $j = n$ повторить*
 - 5.1.1. *Вывести значение элемента массива $a[i,j]$*
6. *Положить минимум $minitit = a[1,1]$*
7. *Положить максимум $maxitit = a[1,1]$*
8. *Положить индекс $i_minitit = 1$*
9. *Положить индекс $j_minitit = 1$*

10. Положить индекс $i_maximum = 1$
11. Положить индекс $j_maximum = 1$
12. Положить флаг $flag = false$
13. От $i = 1$ до $i = m$ повторить
 - 13.1. От $j = 1$ до $j = n$ повторить
 - 13.1.1. Если $a[i,j] < minimum$
то
 - 13.1.1.1. Положить $minimum = a[i,j]$
 - 13.1.1.2. Положить $i_minimum = i$
 - 13.1.1.3. Положить $j_minimum = j$
 - 13.1.1.4. Положить $flag = true$
 - иначе
 - 13.1.1.5. Если $a[i,j] > maximum$
то
 - 13.1.1.5.1. Положить $maximum = a[i,j]$
 - 13.1.1.5.2. Положить $i_maximum = i$
 - 13.1.1.5.3. Положить $j_maximum = j$
 - 13.1.1.5.4. Положить $flag = true$
14. Если $flag = true$
то
 - 14.1. Положить $a[i_minimum, j_minimum] = maximum$
 - 14.2. Положить $a[i_maximum, j_maximum] = minimum$
 - 14.3. От $i = 1$ до $i = m$ повторить
 - 14.3.1. От $j = 1$ до $j = n$ повторить
 - 14.3.1.1. Вывести значение элемента массива $a[i,j]$
15. Конец

Теперь представим программную реализацию:

```
{ Пример 38 }
{ Структурированные типы. Массивы }
const  m = 3;
       n = 4;
type   row_size = array [1..n] of integer;
       matrix = array [1..m] of row_size;
const  a : matrix = ((1, 2, 3, 4), (5, 6, 7, 8), (9,10,11,12));
       i_minimum : integer = 1;    j_minimum : integer = 1;
       i_maximum : integer = 1;    j_maximum : integer = 1;
       flag : boolean = false;
var     i, j : integer;
       minimum, maximum : integer;
begin
  { Визуализация массива }
  for i := 1 to m do begin
```

```

    for j := 1 to n do write(a[i,j] : 3);
    writeln
end;
minimum := a[1,1];
maximum := a[1,1];
{ Поиск минимального и максимального элементов массива }
for i := 1 to m do
    for j := 1 to n do
        if a[i,j] < minimum
            then begin
                minimum := a[i,j];
                i_minimum := i;
                j_minimum := j;
                flag := true
            end
        else if a[i,j] > maximum then begin
                maximum := a[i,j];
                i_maximum := i;
                j_maximum := j;
                flag := true
            end;
    { Перестановка местами минимального и максимального элементов }
    if flag then begin
        a[i_minimum,j_minimum] := maximum;
        a[i_maximum,j_maximum] := minimum
        { Визуализация массива }
        for i := 1 to m do begin
            for j := 1 to n do write(a[i,j] : 3);
            writeln
        end
    end
end.

```

Результат работы программы:

1	2	3	4
5	6	7	8
9	10	11	12
12	2	3	4
5	6	7	8
9	10	11	1

Структурированные типы. Файлы

Под файлом понимается либо именованная область внешней памяти персонального компьютера (например, жесткого, гибкого или оптического диска), либо логическое устройство ввода-вывода – источник или приемник информации.

Отметим характерные особенности файлов:

- У файла есть имя, что позволяет программе одновременно работать с несколькими файлами.
- Файл содержит компоненты одного типа. Типом компонента может быть любой тип языка Turbo Pascal, кроме файлового типа.
- Длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип или переменную файлового типа можно объявить одним из трех способов:

- файловый тип для текстовых файлов

type

имя_файлового_типа = text;

- файловый тип для типизированных файлов

type

имя_файлового_типа = file of имя_типа;

- файловый тип для нетипизированных файлов

type

имя_файлового_типа = file;

Итак, в зависимости от способа объявления в программе файлового типа пользователь может работать с тремя видами файлов: текстовыми, типизированными и нетипизированными. Вид файла, вообще говоря, определяет способ хранения информации в файле. Однако в языке Turbo Pascal нет средств контроля вида ранее созданных файлов, поэтому при объявлении уже существующих файлов пользователь должен сам следить за соответствием вида объявления характеру файла.

Доступ к файлам

На первом этапе знакомства со стандартными процедурами ввода-вывода уже обсуждалось, что любой программе на языке Turbo Pascal доступны два предварительно объявленных файла со стандартными переменными файлового типа: *input* – для ввода информации с клавиатуры и *output* – для вывода информации на экран дисплея. Логическое имя этих устройств ввода-вывода – *con*. Пользователь в программе по своему усмотрению может либо переопределить назначение этих файловых переменных, либо объявить и назначить их вновь. Любые другие файлы или логические устройства ввода-вывода становятся доступны только после их открытия, которое заключается в назначении ранее объявленной

переменной файлового типа либо имени существующего или вновь создаваемого файла, либо логическому имени устройства ввода-вывода, а также в указании направления обмена информацией (чтение или запись). Переменная файлового типа назначается имени файла или логическому имени устройства ввода-вывода при помощи стандартной процедуры *assign*:

assign (имя_файловой_переменной, имя_файла);

assign (имя_файловой_переменной, логическое_устройство);

Параметры процедуры *имя_файла* и *логическое_устройство* должны быть типа *string*. Указание направления обмена информацией осуществляется либо при помощи стандартных процедур инициации *reset*, *rewrite* и *append*, либо благодаря выбору соответствующей стандартной процедуры ввода-вывода.

Если имя файла задается в виде пустой строки (''), то в зависимости от направления обмена информацией файловая переменная назначается стандартному файлу *input* или *output*.

Пример 39. Продемонстрировать механизм указания направления обмена информацией для текстовых файлов:

{ Пример 39 }

{ Структурированные типы. Файлы }

{ Определение файловой переменной для ввода-вывода }

var input_output : text;

{ Определение строковой переменной для ввода строки символов }

text_line : string;

begin

{ Назначение файловой переменной текстовому файлу }

assign(input_output, '');

{ Инициация текстового файла для вывода }

rewrite(input_output);

{ Вывод текстовой константы на экран дисплея }

writeln(input_output, 'I love Turbo Pascal');

{ Инициация текстового файла для ввода }

reset(input_output);

readln(input_output, text_line);

{ Инициация текстового файла для вывода }

rewrite(input_output);

{ Вывод текстовой строки на экран дисплея }

writeln(input_output, text_line)

end.

Результат работы программы:

I love Turbo Pascal

Я люблю Турбо Паскаль

Я люблю Турбо Паскаль

Имена файлов

Имя файла – это выражение типа *string*, которое записывается по правилам определения имен в операционной системе, например, в MS DOS:

[логическое_имя_диска:][\][маршрут\]имя[.расширение]

Здесь в квадратных скобках, как это принято, указаны необязательные компоненты имени файла.

Логическое_имя_диска состоит из двух символов, первый символ – это одна из прописных или строчных латинских букв, после которой ставится второй символ – двоеточие. Имена *A:(a:)* и *B:(b:)* относятся к дисковым накопителям или приводам (от слова *drive* – привод) на гибких дисках, *C:(c:)*, *D:(d:)* и т.д. – к остальным накопителям (как физическим, так и виртуальным устройствам внешней памяти). Если имя диска не указано, то подразумевается устройство по умолчанию.

Маршрут (от слова *path* – маршрут) – это путь в иерархической файловой системе к файлу либо от корневого каталога (если это абсолютный маршрут), либо от текущего каталога (если это относительный маршрут). Синтаксически маршрут записывается как последовательность имен каталогов, разделенных символом обратная косая черта (от слова *backslash* – обратная косая черта). Если запись маршрута начинается с символа обратная косая черта, то такой маршрут называют абсолютным. Максимальная длина маршрута ограничена операционной системой, например, в MS DOS она составляет 64 символа.

Имя содержит от одного до восьми символов, *расширение* – от одного до трех символов из следующего набора:

- прописные и строчные латинские буквы
- цифры
- специальные символы `~ ! @ # $ % ^ & () ' - _`

Логические устройства

Стандартные аппаратные средства персонального компьютера, например, такие как клавиатура, экран дисплея, принтер и коммуникационные порты ввода-вывода, определяются в языке Turbo Pascal специальными именами, которые являются логическими именами устройств или логическими устройствами (*logical unit*). Стиль написания этих имен прописными буквами является лишь данью традиции:

- *CON* – логическое имя консоли. Различие между физическими устройствами (клавиатурой и экраном дисплея) устанавливается по направлению передачи данных.
- *PRN* – логическое имя принтера (от слова *printer* – устройство печати или принтер). *PRN* обычно обозначает первый параллельный порт для принтера, известный также под именем *LPT1* (от слов *line printer* – устройство построчной печати). Если к персональному компьютеру подключено несколько принтеров, доступ к ним осуществляется по логическим именам *LPT1*, *LPT2* и *LPT3*.

- **AUX** – логическое имя внешнего устройства (от слов *auxiliary unit* – внешнее устройство). **AUX** обычно обозначает первый последовательный коммуникационный порт, известный также под именем **COM1** (от слов *communications serial port* – последовательный коммуникационный порт). Как правило, в составе персонального компьютера имеются четыре коммуникационных порта, доступ к которым осуществляется по логическим именам **COM1**, **COM2**, **COM3** и **COM4**.
- **NUL** – логическое имя фиктивного устройства (от слова *null* – ноль). **NUL** устройство сбрасывает любую посланную ему информацию. **NUL** – это в некотором роде “мусорная корзина”, в которую можно отправлять любую информацию.

Инициация файла

Инициировать (от слова *initiate* – устанавливать) файл – означает указать для этого файла направление передачи информации. В языке Turbo Pascal файл можно открыть для чтения, для записи, а также для чтения и записи одновременно.

Явная инициация файлов осуществляется при помощи стандартных процедур: **reset** (от слова *reset* – сбросить), **rewrite** (от слова *rewrite* – перезаписать) и **append** (от слова *append* – присоединять).

Процедура **reset** иницирует чтение информации из файла:

reset(имя_файловой_переменной);

При выполнении этой процедуры уже существующий файл подготавливается к чтению из него информации, при этом специальная переменная-указатель, связанная с этим файлом (файловый указатель), будет указывать на начало файла. Заметим, что в языке Turbo Pascal разрешается обращаться к типизированным файлам, открытых процедурой **reset**, с помощью процедуры **write** или **writeln**. Такая возможность позволяет обновлять ранее созданные типизированные файлы и при необходимости расширять их.

Процедура **rewrite** иницирует запись информации в файл:

rewrite(имя_файловой_переменной);

При выполнении этой процедуры новый файл подготавливается к записи в него информации, при этом файловый указатель будет указывать на начало файла.

Процедура **append** иницирует запись информации в текстовый файл для его расширения:

append(имя_файловой_переменной);

При выполнении этой процедуры уже существующий текстовый файл подготавливается к записи в него информации, при этом файловый указатель будет указывать на конец файла. Если текстовый файл ранее уже был открыт с помощью процедуры **reset** или процедуры **rewrite**, использование процедуры **append** приведет к закрытию этого файла и открытию его вновь, но уже для его расширения.

Напомним, что неявная инициация файлов может осуществляться и благодаря выбору соответствующей стандартной процедуры ввода-вывода.

Процедуры и функции для работы с файлами

Среди множества процедур и функций, используемых для работы с файлами любого вида, рассмотрим лишь некоторые, такие, как: *close* (от слова *close* – закрыть), *eof* (от слов *end of file* – конец файла), *ioresult* (от слов *input-output result* – результат операции ввода-вывода).

Процедура *close* закрывает файл:

close(имя_файловой_переменной);

При выполнении этой процедуры закрывается файл, при этом связь файловой переменной с именем файла, установленная ранее процедурой *assign*, сохраняется, что позволяет повторно открывать файл. Заметим, что в случае нормального завершения программы все открытые файлы всегда закрываются автоматически.

Логическая функция *eof* тестирует конец файла:

eof(имя_файловой_переменной) : boolean;

Функция возвращает значение *true*, если файловый указатель указывает на конец файла. В случае чтения файла это означает, что файл исчерпан, в случае записи – что очередная порция информации будет добавлена в конец файла.

Логическая функция *ioresult* тестирует результат операции ввода-вывода:

ioresult : word;

Функция возвращает значение *false*, если операция ввода-вывода завершилась успешно, в остальных случаях – отличные от нуля коды ошибок ввода-вывода. Заметим, что вызов функции *ioresult* необходимо осуществлять только при отключенном автоконтроле ошибок ввода-вывода, в противном случае любая ошибка ввода-вывода приведет к аварийному завершению программы. Директива компилятора *{SI-}* отключает, а директива *{SI+}* включает автоконтроль.

Пример 40. Пусть в текущем каталоге есть текстовый файл *text.dat*, в котором были записаны две такие строки:

I love Turbo Pascal!

Я люблю Турбо Паскаль!

Продемонстрировать использование функций *ioresult* и *eof* для проверки существования в текущем каталоге текстового файла *text.dat* и подсчета в нем количества строк:

{ Пример 40 }

{ Структурированные типы. Файлы }

{ Определение текстовой константы для имени файла }

const my_file = 'text.dat';

{ Определение файловой переменной для ввода }

var input : text;

{ Определение строковой переменной для ввода строки файла }

line : string;

{ Определение целой переменной для подсчета числа строк }

counter : word;

```

begin
  { Назначение файловой переменной текстовому файлу }
  assign(input, my_file);
  { Отключить контроль ошибок ввода-вывода }
  {$I-}
  { Инициация текстового файла для ввода }
  reset(input);
  { Включить контроль ошибок ввода-вывода }
  {$I+}
  if ioresult <> 0
    then writeln('Файл ', my_file, ' не существует!')
    else begin
      counter := 0;
      while not eof(input) do begin
        readln(input, line);
        counter := counter + 1
      end;
      writeln('Количество строк в файле ', my_file, ' : ', counter)
    end
end.

```

Результат работы программы:

Количество строк в файле text.dat : 2

Заметим, что по завершению этой программы открытый для чтения файл закроется автоматически. Явный вызов процедуры *close(input)* здесь был бы неуместен, так как в случае отсутствия файла *text.dat* эта операция привела бы к аварийному завершению программы: нельзя закрыть файл, если он не был открыт. Обычно директивами компилятора отключения *{\$I-}* и включения *{\$I+}* автоконтроля ошибок ввода-вывода в тексте программы выделяют фрагмент для контроля каких-либо операций ввода-вывода, результаты которых необходимо исследовать, не вызывая аварийного завершения программы. При этом пользователь сам определяет реакцию программы на те или иные ошибки времени выполнения программы (от слов *runtime error* – ошибка времени выполнения).

Пример 41. Продемонстрировать механизм контроля операций ввода-вывода и с помощью функции *ioresult* получить коды ошибок времени выполнения:

```

{ Пример 41 }
{ Структурированные типы. Файлы }
  { Определение текстовой константы для имени файла }
const  my_file = 'text.dat';
  { Определение файловой переменной для ввода }
var    input : text;

```

begin

{ Назначение файловой переменной текстовому файлу }

assign(input, my_file);

{ Отключить контроль ошибок ввода-вывода }

{SI-}

{ Инициация текстового файла для ввода }

reset(input);

writeln('Код ошибки процедуры reset: ', ioresult);

close(input);

writeln('Код ошибки процедуры close: ', ioresult)

end.

Если текстовый файл *text.dat* существует в текущем каталоге, результат выполнения программы будет следующим:

Код ошибки процедуры reset: 0

Код ошибки процедуры close: 0

Если текстовый файл *text.dat* не существует в текущем каталоге, результат выполнения программы будет следующим:

Код ошибки процедуры reset: 2

Код ошибки процедуры close: 103

Справочная служба интегрированной среды разработки Turbo Pascal версий 6.0 и 7.0 интерпретирует эти коды следующим образом:

2 File not found **(Файл не найден)**

103 File not open **(Файл не открыт)**

Текстовые файлы

Текстовые файлы предназначены для хранения текстовой информации. В языке Turbo Pascal текстовый файл трактуется как совокупность строк переменной длины. Доступ к каждой строке текстового файла возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой строки ставится специальный признак *EOLN* (от слов *end of line* – конец строки), а в конце файла – признак *EOF* (от слов *end of file* – конец файла). Эти признаки можно протестировать одноименными логическими функциями.

Признак *EOLN* – это последовательность из ASCII-кода 13 или *CR* (от слов *carriage return* – возврат каретки) и ASCII-кода 10 или *LF* (от слова *linefeed* – перевод строки).

Признак *EOF* – это ASCII-код 26 или *SUB* (от слова *substitution* – замена).

Логическая функция *eoln* тестирует конец строки:

eoln(имя_файловой_переменной) : boolean;

Функция *eoln* возвращает значение *true*, если в текстовом файле достигнут признак *EOLN*. В случае отсутствия аргумента *имя_файловой_переменной* функция проверяет стандартный файл ввода *input*.

Существует некоторое отличие в работе функций *eoln* и *eof* с логическими устройствами и дисковыми файлами. Дело в том, что для логических устройств невозможно предвидеть, каким будет результат чтения очередного символа. Поэтому эти функции тестируют соответствующие признаки для логических устройств после очередного чтения, а для дисковых файлов – перед очередным чтением.

При работе с логическими устройствами функция *eoln* возвращает значение *true*, если последним считанным символом был *EOLN* или *EOF*, а при чтении дискового файла значение *true* возвращается в случае, если следующим считываемым символом будет *EOLN* или *EOF*.

При работе с логическими устройствами функция *eof* возвращает значение *true*, если последним считанным символом был *EOF*, а при чтении дискового файла значение *true* возвращается в случае, если следующим считываемым символом будет *EOF*.

Для доступа к строкам текстового файла применяются процедуры *read*, *readln*, *write* и *writeln*. Как уже отмечалось, при вызове этих процедур в списках ввода-вывода можно указывать произвольное количество фактических параметров. Первым параметром в списке может быть файловая переменная, в этом случае осуществляется обращение либо к дисковому файлу, либо к логическому устройству, при этом при помощи процедуры *assign* файловая переменная должна быть назначена какому-либо файлу. Если файловая переменная в списке не указана, происходит обращение к стандартным файлам ввода-вывода *input* и *output*.

Процедура *read*

Процедура обеспечивает ввод из файла символов, строк и чисел целого или вещественного типов:

read(имя_файловой_переменной, список_ввода);

read(список_ввода);

Список_ввода – это либо одна переменная, либо последовательность перечисленных через запятую переменных типа *char*, *string*, *pchar*, а также любого целого или вещественного типа.

При вводе переменных типа *char* выполняется чтение одного символа из строки файла и присваивание прочитанного значения переменной из списка ввода. Если файловый указатель достиг конца строки файла, то результатом чтения вначале будет символ *CR*, затем – символ *LF*, а если достигнут конец файла, то – символ *EOF*. При вводе с клавиатуры символы *CR* и *LF* вводятся при нажатии клавиши *Enter*, а символ *EOF* – при одновременном нажатии клавиш *Ctrl* и *Z*.

При вводе переменных типа *string* или *pchar* количество прочитанных символов из строки файла равно максимальной длине строки из списка ввода, если только раньше в строке файла не встретились символы **CR** или **EOF**. При этом сами символы **CR** и **EOF** в строку из списка ввода не передаются. Если при чтении строки файла еще не встретились символы **CR** или **EOF**, а строка из списка ввода уже заполнена прочитанными символами, то оставшиеся символы строки файла могут быть прочитаны при помощи последующих вызовов процедуры *read*, пока не встретятся символы **CR** или **EOF**.

Если в текстовом файле записано несколько строк, то при помощи процедуры *read* можно прочитать только первую строку, все последующие строки для процедуры *read* недоступны – при каждом очередном вызове процедуры *read* будет возвращаться пустая строка. Для чтения последовательности строк следует использовать процедуру *readln*.

При вводе переменных целого или вещественного типа процедура *read* из строки файла для каждой переменной из списка ввода вначале выделяет подстроку, которая рассматривается теперь как символьное представление числовой константы соответствующего типа и затем преобразуется во внутренний формат, полученное значение числовой константы присваивается соответствующей переменной.

Выделение подстроки осуществляется по следующему правилу: все ведущие пробелы и символы табуляции до первого значащего символа числовой константы и признаки **EOLN** или **EOF** пропускаются, затем после выделения первого значащего символа числовой константы любой из перечисленных символов служит признаком конца подстроки. Если в подстроке был нарушен требуемый формат представления числовой константы, возникает ошибка ввода-вывода. Если при пропуске ведущих пробелов и символов табуляции встретился признак **EOF**, переменная при вводе получает значение 0.

При вводе с клавиатуры символьные строки запоминаются в буфере ввода, который передается процедуре *read* только после нажатия клавиши **Enter**, что позволяет редактировать данные при их вводе.

Если текстовый файл представляет собой последовательность строк чисел целого или вещественного типа, то процедура *read* позволяет осуществлять ввод этих чисел так, как будто все они принадлежат одной строке файла благодаря свойству пропускать признаки **EOLN** или **EOF**. Например, инициализацию элементов одномерного и двумерного массивов можно осуществить при помощи одного текстового файла, каждая строка которого может содержать произвольное количество чисел целого или вещественного типа, при этом количество чисел в файле должно быть не меньше числа элементов массива.

Пример 42. Продемонстрировать механизм инициализации одномерного массива с помощью текстового файла *numbers.dat*, в котором были записаны четыре строки следующего вида:

```
1 2 3 4
5 6 7
8 9
10
```

Представим программу для инициализации пяти элементов одномерного массива:

```
{ Пример 42 }
{ Структурированные типы. Файлы }
const  n = 5;
      { Определение текстовой константы для имени файла }
      my_file = 'numbers.dat';
      { Определение файловой переменной для ввода }
var    input : text;
      k : integer;
      a : array [1..n] of integer;
begin
  { Назначение файловой переменной текстовому файлу }
  assign(input, my_file);
  { Инициация текстового файла для ввода }
  reset(input);
  k := 1;
  { Инициализация элементов массива }
  while not eof(input) and (k <= n) do begin
    read(input, a[k]);
    k := k + 1
  end;
  { Визуализация массива }
  for k := 1 to n do write(a[k] : 3)
end.
```

Результат работы программы:

1 2 3 4 5

Пример 43. Продемонстрировать механизм инициализации двумерного массива, например, матрицы из двух строк и пяти столбцов с помощью того же самого текстового файла *numbers.dat*.

```
{ Пример 43 }
{ Структурированные типы. Файлы }
const  m = 2;
      n = 5;
      { Определение текстовой константы для имени файла }
      my_file = 'numbers.dat';
      { Определение файловой переменной для ввода }
var    input : text;
      i, j : integer;
      a : array [1..m, 1..n] of integer;
```

```

begin
  { Назначение файловой переменной текстовому файлу }
  assign(input, my_file);
  { Инициация текстового файла для ввода }
  reset(input);
  i := 1;
  { Инициализация элементов массива }
  while not eof(input) and (i <= m) do begin
    j := 1;
    while not eof(input) and (j <= n) do begin
      read(input, a[i,j]);
      j := j + 1
    end;
    i := i + 1
  end;
  { Визуализация массива }
  for i := 1 to m do begin
    for j := 1 to n do write(a[i,j] : 3);
    writeln
  end
end.

```

Результат работы программы:

```

1  2  3  4  5
6  7  8  9 10

```

Такой реализации механизма инициализации массивов можно противопоставить, например, более простую и наглядную альтернативу. Обычно текстовые файлы для инициализации массивов хранят такие структуры, которые идентичны структурам самих массивов, например, для одномерного массива в текстовом файле хранится вектор (строка или столбец) соответствующей длины, для двумерного массива – матрица соответствующего размера. В этом случае фрагменты программ для инициализации массивов в двух предыдущих примерах могли бы стать и такими:

- для одномерного массива


```

      { Инициализация элементов массива }
      for k := 1 to n do read(input, a[k]);
      
```
- для двумерного массива


```

      { Инициализация элементов массива }
      for i := 1 to m do
        for j := 1 to n do read(input, a[i,j]);
      
```

Здесь по-прежнему количество чисел в файле должно быть не меньше числа элементов массива.

Процедура **readln**

Процедура обеспечивает ввод из файла символов, строк и чисел целого или вещественного типов:

```
readln(имя_файловой_переменной, список_ввода);  
readln(список_ввода);  
readln;
```

Процедура **readln** полностью идентична процедуре **read** за исключением того, что после ввода последней переменной из списка ввода оставшаяся часть строки файла до признака **EOLN** или **EOF** пропускается, поэтому следующий ввод переменных из списка ввода с помощью процедуры **readln** или **read** начинается с первого символа уже следующей строки файла, если только это не последняя строка.

Кроме того, процедуру **readln** можно вызвать без параметра **список_ввода**, что приведет к пропуску всех символов текущей строки вместе с признаком **EOLN** или **EOF**. При вводе с клавиатуры нажатие на клавишу **Enter** приводит к перемещению курсора в начало следующей строки экрана.

Процедура **write**

Процедура обеспечивает вывод в файл данных символьного, логического, строкового, целого и вещественного типа:

```
write(имя_файловой_переменной, список_вывода);  
write(список_вывода);
```

Элемент списка вывода в общем случае является выражением (арифметическим или логическим) и может иметь следующую форму:

```
выражение[ : выражение_1 [ : выражение_2 ] ]
```

Здесь **выражение** – выражение арифметического или логического типа, **выражение_1** и **выражение_2** – выражения типа **word**, определяющие формат вывода, в квадратных скобках, как это принято, указаны необязательные параметры.

Параметр **выражение_1** определяет минимальную ширину поля вывода для символьного представления значения **выражение**. Если символьное представление имеет меньшую длину, чем параметр **выражение_1**, оно будет дополнено слева пробелами, если – большую длину, то параметр **выражение_1** игнорируется и выводится необходимое число символов.

Параметр **выражение_2** определяет количество выводимых десятичных цифр дробной части вещественного числа. Он может использоваться только совместно с параметром **выражение_1**.

Если параметр **выражение_1** не указан, элементы списка вывода выводятся друг за другом без какого-либо их разделения.

Например, рассмотрим такую программу:

```
begin  
  write('T', 'u', 'r', 'b', 'o', 'P', 'a', 's', 'c', 'a', 'l')  
end.
```

Здесь результатом вывода на экран списка текстовых констант будет строка:

TurboPascal

Символы и строки выводятся без изменений, но перед ними могут появиться ведущие пробелы, если ширина поля вывода окажется больше требуемой длины.

Например, рассмотрим такую программу:

begin

write('Turbo', 'Pascal' : 7)

end.

Здесь результатом вывода на экран списка текстовых констант будет строка:

Turbo Pascal

Логические выражения в зависимости от их значения выводятся как строки **FALSE** или **TRUE**.

Например, рассмотрим такую программу:

begin

write(1 < 2, ' ', 2 > 1)

end.

Здесь результатом вывода на экран значений логических выражений будет строка:

FALSE TRUE

Целые числа выводятся в виде символьной последовательности, которая в случае отрицательных чисел начинается с символа '—'.

Например, рассмотрим такую программу:

begin

write(1, ' ', -2)

end.

Здесь результатом вывода на экран значений целых чисел будет строка:

1 -2

Вещественные числа выводятся в виде символьной последовательности либо в экспоненциальном формате, либо в формате с фиксированной точкой.

По умолчанию вывод вещественного числа в экспоненциальном формате осуществляется в одном из двух видов:

- если числовой сопроцессор не используется или не эмулируется, ширина поля вывода принимается равной 17, мантисса представляется 11 цифрами
s#.#####Es##
- если числовой сопроцессор используется или эмулируется, ширина поля вывода принимается равной 23, мантисса представляется 15 цифрами
s#.#####Es####

Здесь *s* — знак числа (символ пробел для положительных или символ '—' для отрицательных чисел) или знак порядка (символ '+' для положительных или символ '—' для отрицательных порядков), *#* — десятичная цифра, *E* — символ десятичного основания. Последняя цифра символьного представления мантиссы округляется по правилам округления.

Если не указан параметр *выражение_1*, ширина поля вывода определяется по умолчанию и принимается равной 17 или 23.

Например, рассмотрим такую программу:

```
begin  
  write(1.2, ' ', -4.5)  
end.
```

Здесь результатом вывода на экран значений вещественных чисел будет строка:

1.2000000000E+00 -4.5000000000E+00

Например, рассмотрим такую программу:

```
{ $N+}  
begin  
  write(1.2, ' ', -4.5)  
end.
```

Здесь результатом вывода на экран значений вещественных чисел будет строка:

1.2000000000000000E+0000 -4.5000000000000000E+0000

Если значение параметра **выражение_1** указано в диапазоне от 0 до 8, а числовой сопроцессор при этом не используется или не эмулируется, ширина поля вывода принимается равной 8. В этом случае мантисса представляется 2 цифрами, и вторая цифра мантиссы округляется по правилам округления.

Например, рассмотрим такую программу:

```
begin  
  write(1.23 : 0, ' ', -4.56 : 8)  
end.
```

Здесь результатом вывода на экран значений вещественных чисел будет строка:

1.2E+00 -4.6E+00

В противном случае, если числовой сопроцессор используется или эмулируется, а значение параметра **выражение_1** при этом указано в диапазоне от 0 до 10, ширина поля вывода принимается равной 10. В этом случае мантисса также представляется 2 цифрами, и вторая цифра мантиссы также округляется.

Например, рассмотрим такую программу:

```
{ $N+}  
begin  
  write(1.23 : 0, ' ', -4.56 : 10)  
end.
```

Здесь результатом вывода на экран значений вещественных чисел будет строка:

1.2E+0000 -4.6E+0000

Если значение параметра **выражение_2** больше 0, вещественное число всегда выводится в формате с фиксированной точкой с требуемым количеством десятичных цифр в дробной части независимо от значения параметра **выражение_1**. В общем случае последняя цифра дробной части округляется.

Например, рассмотрим такую программу:

```
begin  
  write(0.123 : 1 : 3, ' ', -4.56 : 1 : 1)  
end.
```

Здесь результатом вывода на экран значений вещественных чисел будет строка:

0.123 -4.6

Если значение параметра *выражение_2* равно 0, вещественное число выводится с округлением до целого в формате целого числа.

Например, рассмотрим такую программу:

begin

write(0.123 : 1 : 0, -4.56 : 3 : 0)

end.

Здесь результатом вывода на экран значений вещественных чисел будет строка:

0 -5

Если значение параметра *выражение_2* отрицательное, этот параметр игнорируется, и вещественное число выводится в экспоненциальном формате с учетом значения параметра *выражение_1*.

Например, рассмотрим такую программу:

begin

write(0.123 : 1 : -3, ' ', -4.56 : 1 : -3)

end.

Здесь результатом вывода на экран значений вещественных чисел будет строка:

1.2E-01 -4.6E+00

Если значение параметра *выражение_2* больше 11, а числовой сопроцессор при этом не используется или не эмулируется, значение этого параметра принимается равным 11. В противном случае, если числовой сопроцессор используется или эмулируется, а значение параметра *выражение_2* больше 18, значение этого параметра принимается равным 18.

Процедура *writeln*

Процедура обеспечивает вывод в файл данных символьного, логического, строкового, целого и вещественного типов:

writeln(имя_файловой_переменной, список_вывода);

writeln(список_вывода);

writeln;

Процедура *writeln* полностью идентична процедуре *write* за исключением того, что выводимая строка символов завершается признаком *EOLN*. Кроме того, процедуру *writeln* можно вызвать без параметра *список_вывода*, что приведет к выводу в файл признака *EOLN*. Вывод на экран признака *EOLN* приводит к перемещению курсора в начало следующей строки экрана.

Процедуры и функции

Во введении уже было отмечено, что в основе императивной парадигмы программирования лежит принцип процедурного структурирования программ. Так, под процедурой понимают часть программы, которая выполняет некоторые определенные действия над данными, в общем случае определяемыми параметрами. Процедура при этом может быть вызвана из любого места программы (или другой процедуры), и при каждом вызове процедуры ей могут быть переданы различные параметры. Одной из разновидностей процедур являются функции, которые также выполняют некоторые определенные действия над данными, но результат всегда передают в программу (или процедуру, или функцию) в виде значения функции посредством своего имени.

Обратим здесь внимание на одну терминологическую особенность именования частей программы, связанную с проектированием в 1958 году языка Fortran II для научных расчетов, где впервые была реализована идея независимой компиляции подпрограмм (от слова *subroutine* – подпрограмма), а именно собственно подпрограмм и подпрограмм-функций. Использование подпрограмм позволило существенно экономить исходный программный код, поскольку записывалась подпрограмма в программе только раз, а вызываться могла из различных мест программы сколь угодно раз. Кроме того, использование подпрограмм в дальнейшем облегчило проектирование больших программ, поскольку из подпрограмм можно было образовывать различные библиотеки для общего пользования. Поэтому и даже теперь термином “подпрограмма” обозначают те части программы, которые имеют отношение к процедурам и функциям, независимо от того, о каком языке программирования идет речь.

Отметим характерные особенности программ, разработанных в соответствии с императивной парадигмой программирования:

- описание программы как совокупности подпрограмм – процедур и функций, каждая из которых может вызывать другую процедуру или функцию и может быть вызвана основной программой;
- использование механизмов передачи параметров подпрограммам и механизмов вложения подпрограмм в результате применения методов структурного программирования;
- создание библиотек подпрограмм для конкретной предметной области;
- создание больших программных систем на основе подпрограмм и использование механизма раздельной компиляции модулей в результате применения методов структурного проектирования.

Итак, процедуры и функции представляют собой весьма важный инструмент языка Turbo Pascal, который наряду с управляющими структурами структурного программирования – следование, выбор и итерация – и структурами данных – строки, массивы, записи, множества и файлы – позволяют пользователю проектировать хорошо структурированные программы.

В структурированных программах обычно легко прослеживается основной алгоритм, они проще в отладке и менее чувствительны к ошибкам программирования. Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, самостоятельность процедур и функций позволяет локализовать в них все детали программной реализации какого-либо алгоритма. Проектирование программы и ее составных частей при этом основывается на применении методов нисходящего проектирования программ: алгоритм программы представляется в виде последовательности относительно крупных подпрограмм, которые, в свою очередь, представляются в виде последовательности менее крупных подпрограмм и т.д., тем самым реализуется принцип “от сложного – к простому”.

Процедурой в языке Turbo Pascal называется особым образом оформленный фрагмент программы, имеющий собственное имя. Упоминание этого имени в блоке программы (или процедуры, или функции) приводит к активизации процедуры и называется ее вызовом. По отношению к вызываемой процедуре программу (или процедуру, или функцию), в которой присутствует инструкция вызова процедуры, называют вызывающей. Сразу после активизации процедуры, если речь не идет о рекурсивных процедурах (или функциях), начинают выполняться инструкции ее тела, после выполнения последней инструкции тела процедуры управление передается в вызывающую программу (или процедуру, или функцию) непосредственно за инструкцией вызова процедуры. Для обмена данными между вызывающей программой (или процедурой, или функцией) и вызываемой процедурой может использоваться список параметров вызова, который заключается в круглые скобки за именем процедуры. Наряду с таким механизмом обмена данными между вызываемой процедурой и вызывающей программой (или процедурой, или функцией) существуют и другие механизмы, в которых не используется список параметров вызова. Заметим, что при объявлении функций ее параметры вызова в силу традиции иногда называют аргументами вызова.

Функция отличается от процедуры тем, что результат ее работы возвращается в вызывающую программу (или процедуру, или функцию) в виде значения этой функции, и, следовательно, вызовы функций могут использоваться в качестве операндов в выражениях. Имена процедур и функций пользователя могут использоваться в списках параметров вызова других процедур и функций пользователя. Для реализации такого механизма вызова в языке Turbo Pascal существуют процедурные типы – тип-процедура и тип-функция. Существует также возможность вызова функции пользователя как процедуры, что означает отказ от использования значения, возвращаемого этой функцией.

Неотъемлемой частью языка Turbo Pascal являются библиотеки его стандартных процедур и функций, которые, как уже отмечалось, не требуют своего предварительного объявления. Каждый пользователь по своему усмотрению может разрабатывать свои процедуры и функции, объявления которых помещаются в разделе объявлений программы (или процедуры, или функции), если речь не идет о механизме раздельной компиляции частей программы – внешних процедурах и функциях, а также модулей.

Объявление процедур и функций пользователя

Структура процедуры или функции пользователя, как и структура программы, состоит из раздела объявлений и объявления блока. Для объявления блока используется составная инструкция языка Turbo Pascal, с помощью которой обозначается начало и конец тела процедуры или функции. Раздел объявлений начинается с заголовка процедуры или функции. Заголовок процедуры пользователя объявляет имя процедуры, за которым в круглых скобках может следовать список параметров вызова, а заголовок функции пользователя кроме имени функции, за которым в круглых скобках также может следовать список параметров вызова, объявляет еще и тип возвращаемого значения. Список параметров вызова иногда называют списком формальных параметров.

Заголовок процедуры:

procedure имя_процедуры(список_параметров);

procedure имя_процедуры;

Заголовок функции:

function имя_функции(список_параметров) : тип_функции;

function имя_функции : тип_функции;

Здесь ***имя_процедуры*** и ***имя_функции*** – уникальные идентификаторы в глобальной области видимости программы (или процедуры, или функции), ***список_параметров*** – список формальных параметров процедуры или функции, ***тип_функции*** – тип возвращаемого функцией значения.

За заголовком процедуры или функции может следовать одна из стандартных директив для уточнения действий компилятора:

- ***assembler*** – использование встроенного ассемблера для программного кода тела процедуры или функции;
- ***external*** – объявление внешней процедуры или функции, программный код которой написан на языке ассемблера; тело процедуры или функции в этом случае отсутствует;
- ***far*** – использование дальней модели вызова для программного кода процедуры или функции;
- ***forward*** – объявление опережающего описания процедуры или функции;
- ***inline*** – использование программного кода встроенной процедуры или функции вместо оператора вызова процедуры или функции; тело встроенной процедуры или функции представляет собой ***inline***-инструкцию;
- ***interrupt*** – объявление процедуры обработки прерываний;
- ***near*** – использование ближней модели вызова для программного кода процедуры или функции.

Заметим, что в соответствии с архитектурой микропроцессора персонального компьютера в программах Turbo Pascal могут использоваться две модели памяти – ближняя и дальняя. Модель памяти определяет возможность вызова процедуры или функции из различных частей программы: если используется ближняя модель памяти, вызов возможен только в пределах 64 Кбайт (размер одного сегмента

оперативной памяти, который выделяется основной программе и каждому используемому в ней модулю); для дальней модели памяти вызов возможен из любого сегмента оперативной памяти.

Заметим также, что заголовки процедур и функций могут выступать в роли объявлений, когда они используются при объявлении глобальных подпрограмм в интерфейсной части модуля программы. Вместе со стандартными директивами заголовки процедур и функций также могут использоваться еще при так называемом опережающем объявлении этих процедур и функций.

Раздел объявлений процедур или функций кроме заголовка и стандартных директив может состоять также из подразделов локальных объектов – подраздела объявления локальных директив компилятора, подраздела объявления локальных типов, определяемых пользователем, подраздела объявления локальных констант, подраздела объявления локальных переменных, подраздела объявления локальных меток, подраздела объявления локальной или вложенной функции пользователя и подраздела объявления локальной или вложенной процедуры пользователя. Все подразделы объявлений локальных объектов могут следовать друг за другом в произвольном порядке, повторяясь при необходимости сколь угодно раз.

В блоке функции обязательно должна присутствовать хотя бы одна инструкция присваивания, с помощью которой имени функции присваивается значение выражения, что позволит реализовать механизм передачи значения этой функции в вызывающую программу (или процедуру, или функцию).

Формальные параметры процедур и функций

В списке формальных параметров процедуры или функции для каждого параметра должны быть указаны его идентификатор и тип, при этом типом любого параметра может быть только стандартный или ранее объявленный тип. Список формальных параметров рассматривается как своеобразное расширение раздела объявлений: любой параметр из этого списка может использоваться в теле процедуры или функции, в списке все параметры разделяются знаком препинания точка с запятой.

Любой из формальных параметров процедуры или функции в зависимости от способа его объявления может быть отнесен к одной из четырех разновидностей:

- параметр-значение;
- параметр-переменная;
- параметр-константа (только для версии 7.0);
- параметр-процедура или параметр-функция (параметр процедурного типа).

В инструкции вызова процедуры или в выражении вызов функции параметры списка вызова называются фактическими параметрами, они передаются вызываемой процедуре или функции как аргументы. Объявление формальных параметров процедуры или функции тем или иным способом существенно только для вызывающей программы (или процедуры, или функции): например, если формальный параметр объявлен как параметр-значение, то при вызове ему может

соответствовать произвольное выражение; если формальный параметр объявлен как параметр-переменная, то при вызове ему должен соответствовать аргумент в виде константы или переменной требуемого типа. Контроль за соблюдением этого правила, а также контроль за соответствием списка формальных параметров списку параметров вызова (количество, порядок расположения и типы параметров) осуществляется компилятором на этапе времени компиляции.

Реализация механизма передачи аргументов в вызываемую процедуру или функцию определяется способом объявления соответствующих им формальных параметров: для параметров-значений аргументы передаются в виде копий своих значений, для параметров-переменных и параметров-констант – в виде своих адресов, а для параметров процедурного типа – в виде значений процедур или функций, т.е. в виде их адресов.

Параметр-значение. Если формальный параметр процедуры или функции является параметром-значением, то в списке формальных параметров он имеет следующий вид:

имя_параметра : тип_параметра;

Здесь *имя_параметра* – имя формального параметра, *тип_параметра* – тип формального параметра. Однотипные параметры в списке, как это принято в подразделах объявлений, могут быть объединены в группу, в которой они разделяются знаком препинания запятой.

При вызове процедур или функций для какого-либо аргумента, которому соответствует формальный параметр-значение, вызываемой процедуре или функции через стек передается копия значения этого аргумента.

Формальные параметры-значения в теле процедуры или функции имеют локальное время жизни, т.е. они существуют лишь на время работы этой вызванной процедуры или функции, и либо локальную область видимости, т.е. они доступны только самой этой процедуре или функции, либо глобальную область видимости, т.е. они доступны всем вложенным в эту процедуру или функцию другим процедурам или функциям.

Параметр-переменная. Если формальный параметр процедуры или функции является параметром-переменной, то в списке формальных параметров он имеет следующий вид:

var имя_параметра : тип_параметра;

Здесь *имя_параметра* – имя формального параметра, *тип_параметра* – тип формального параметра. Однотипные параметры в списке, как это принято в подразделах объявлений, могут быть объединены в группу, в которой они разделяются знаком препинания запятой. Действие ключевого слова *var* при этом распространяется только до ближайшего разделителя точка с запятой, т.е. в пределах одной группы.

При вызове процедур или функций для какого-либо аргумента, которому соответствует формальный параметр-переменная, вызываемой процедуре или функции через стек передается непосредственно адрес этого аргумента.

Формальные параметры-переменные в теле процедуры или функции имеют глобальное время жизни и глобальную область видимости.

Параметр-константа. Если формальный параметр процедуры или функции является параметром-константой, то в списке формальных параметров он имеет следующий вид:

const имя_параметра : тип_параметра;

Здесь *имя_параметра* – имя формального параметра, *тип_параметра* – тип формального параметра. Однотипные параметры в списке, как это принято в подразделах объявлений, могут быть объединены в группу, в которой они разделяются знаком препинания запятой. Действие ключевого слова ***const*** при этом распространяется только до ближайшего разделителя точка с запятой, т.е. в пределах одной группы.

При вызове процедур или функций для какого-либо аргумента, которому соответствует формальный параметр-константа, вызываемой процедуре или функции через стек передается непосредственно адрес этого аргумента.

Формальные параметры-константы в теле процедуры или функции имеют глобальное время жизни и глобальную область видимости, а их значения не могут быть изменены.

Параметр процедурного типа. Если формальный параметр процедуры или функции является параметром процедурного типа, т.е. параметром-процедурой или параметром-функцией, то в списке формальных параметров он имеет следующий вид:

имя_параметра : тип_параметра;

Здесь *имя_параметра* – имя формального параметра, *тип_параметра* – процедурный тип формального параметра. Процедурные типы определяются в подразделе объявления типов следующим образом:

type

имя_параметра-процедуры = procedure(список_параметров);

type

имя_параметра-процедуры = procedure;

type

имя_параметра-функции = function(список_параметров) : тип_функции;

type

имя_параметра-функции = function : тип_функции;

При вызове процедур или функций для какого-либо аргумента, которому соответствует формальный параметр процедурного типа, вызываемой процедуре или функции через стек передается непосредственно адрес этого аргумента, что позволяет в теле вызываемой процедуры или функции осуществить вызов другой процедуры или функции, именем которой является этот аргумент вызова. Заметим, что процедуры или функции, чьи имена являются аргументами вызова, должны компилироваться с директивой ***far***, не должны быть стандартными или локальными процедурами или функциями и не должны иметь директив ***inline*** или ***interrupt***.

Заметим также, что переменные процедурных типов могут использоваться не только как аргументы вызова, но и как инструкции или операнды выражений.

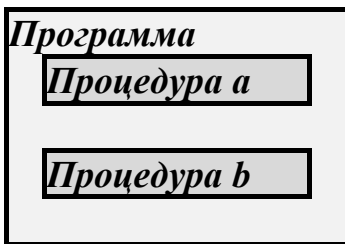
Формальные параметры-процедуры и параметры-функции в теле процедуры или функции имеют глобальное время жизни и глобальную область видимости.

А теперь продолжим обсуждение понятий времени жизни и видимости на конкретных примерах для таких объектов программы (или процедуры, или функции), как константы, переменные и процедуры.

По отношению к вызываемым процедурам или функциям объекты вызывающей программы имеют глобальное время жизни и глобальную область видимости – они существуют на время работы программы и доступны ей самой и всем ее процедурам или функциям, а объекты вызывающей процедуры или функции имеют глобальное время жизни и либо локальную область видимости, либо глобальную область видимости для всех вложенных в эту процедуру или функцию других процедур или функций. Напомним, что объекты, имеющие глобальную область видимости, принято называть глобальными, а объекты, имеющие локальную область видимости, принято называть локальными.

Имена локальных объектов процедур или функций могут совпадать с именами глобальных объектов программы (или процедуры, или функции), в этом случае локальные объекты “закрывают” доступ к этим глобальным объектам, т.е. в теле локальных процедур или функций теперь эти глобальные объекты становятся недоступными.

Пример 44. Представим программу, в которой определены две глобальные процедуры *a* и *b*:



{ Пример 44 }

{ Процедуры и функции }

```

const c = 1;      { константа c – глобальный объект программы }
procedure a;      { процедура a – глобальный объект программы }
  const a = 2;    { константа a – локальный объект процедуры a }
    b = 3;        { константа b – локальный объект процедуры a }
    c = 4;        { константа c – локальный объект процедуры a }
  begin          { тело процедуры a }
    write(a : 2); { визуализация локального объекта процедуры a }
    write(b : 2); { визуализация локального объекта процедуры a }
    write(c : 2); { визуализация глобального объекта программы }
  end; { a }
procedure b;      { процедура b – глобальный объект программы }
  const b = 5;    { константа b – локальный объект процедуры b }
  begin          { тело процедуры b }
    write(b : 2); { визуализация локального объекта процедуры b }
    write(c : 2); { визуализация глобального объекта программы }
  end
  
```

```

    a                { вызов процедуры a }
  end; { b }
begin
  write(c : 2);      { визуализация глобального объекта программы }
  a;                { вызов процедуры a }
  b;                { вызов процедуры b }
  writeln
end.

```

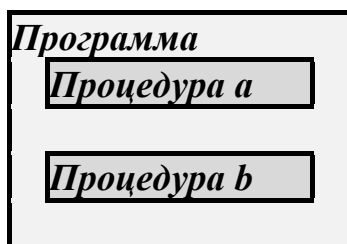
Результат работы программы:

1 2 3 4 5 1 2 3 4

Теперь проанализируем особенности этой программы:

- в программе определены три глобальных объекта – константа *c*, процедура *a* и процедура *b*;
- каждый глобальный объект программы имеет глобальное время жизни;
- каждый глобальный объект программы имеет глобальную область видимости;
- процедуру *a* можно вызвать как из программы, так и из процедуры *b*;
- процедуру *b* можно вызвать только из программы, чтобы реализовать вызов этой процедуры из процедуры *a*, перед ее определением необходимо поместить опережающее объявление процедуры *b*;
- в каждой процедуре определены локальные объекты, имена которых совпадают с именами глобальных объектов;
- локальный объект процедуры *a* – константа *a* – “закрывает” доступ к глобальному объекту программы – имени процедуры *a* (это означает, что теперь невозможен рекурсивный вызов процедуры *a*);
- локальный объект процедуры *a* – константа *b* – “закрывает” доступ к глобальному объекту программы – имени процедуры *b* (это означает, что теперь в теле процедуры *a* невозможен вызов процедуры *b*);
- глобальный объект программы – константа *c* – доступен процедуре *b* и не доступен процедуре *a*, так как локальный объект процедуры *a* – константа *c* – “закрывает” доступ к этому глобальному объекту программы.

Пример 45. Представим программу, в которой определены две глобальные процедуры *a* и *b*, а также реализована возможность вызова процедуры *b* из процедуры *a* благодаря опережающему объявлению процедуры *b*:



{ Пример 45 }

{ Процедуры и функции }

```

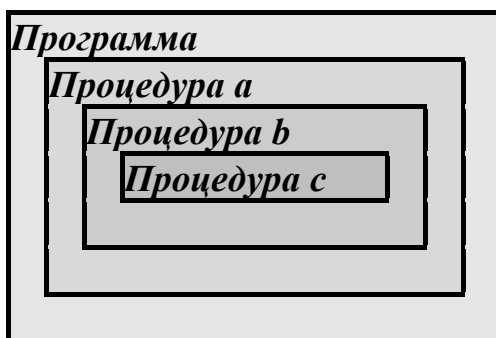
const c = 1;      { константа c – глобальный объект программы }
procedure b;      { опережающее объявление процедуры b }
forward;
procedure a;      { процедура a – глобальный объект программы }
  const a = 2;    { константа a – локальный объект процедуры a }
  c = 3;          { константа c – локальный объект процедуры a }
begin            { тело процедуры a }
  write(a : 2);   { визуализация локального объекта процедуры a }
  write(c : 2);   { визуализация локального объекта процедуры a }
  b              { вызов процедуры b }
end; { a }
procedure b;      { процедура b – глобальный объект программы }
  const b = 4;    { константа b – локальный объект процедуры b }
begin            { тело процедуры b }
  write(b : 2);   { визуализация локального объекта процедуры b }
  write(c : 2);   { визуализация глобального объекта программы }
end; { b }
begin
  write(c : 2);   { визуализация глобального объекта программы }
  a;              { вызов процедуры a }
  b;              { вызов процедуры b }
  writeln
end.

```

Результат работы программы:

1 2 3 4 1 4 1

Пример 46. Представим программу, в которой определены три процедуры *a*, *b* и *c*, при этом локальная процедура *c* (2-ой уровень вложения) определена в локальной процедуре *b* (1-ый уровень вложения), а локальная процедура *b* – в глобальной процедуре *a* (0-ой уровень вложения):



{ Пример 46 }

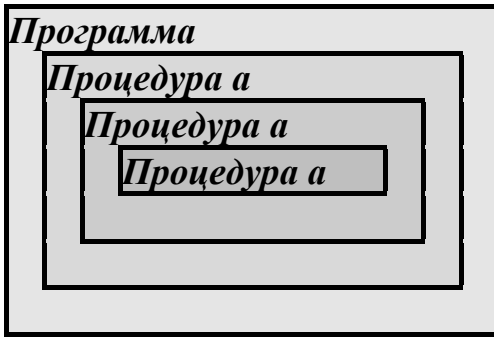
{ Процедуры и функции }

const <i>e</i> = 1;	{ константа <i>e</i> – глобальный объект программы }
<i>d</i> = 2;	{ константа <i>d</i> – глобальный объект программы }
procedure <i>a</i> ;	{ процедура <i>a</i> – глобальный объект программы }
const <i>a</i> = 3;	{ константа <i>a</i> – локальный объект процедуры <i>a</i> }
procedure <i>b</i> ;	{ процедура <i>b</i> – локальный объект процедуры <i>a</i> }
const <i>b</i> = 4;	{ константа <i>b</i> – локальный объект процедуры <i>b</i> }
procedure <i>c</i> ;	{ процедура <i>c</i> – локальный объект процедуры <i>b</i> }
const <i>c</i> = 5;	{ константа <i>c</i> – локальный объект процедуры <i>c</i> }
<i>e</i> = 6;	{ константа <i>e</i> – локальный объект процедуры <i>c</i> }
begin	{ тело процедуры <i>c</i> }
write(<i>c</i> : 2);	{ визуализация локального объекта процедуры <i>c</i> }
write(<i>b</i> : 2);	{ визуализация глобального объекта процедуры <i>c</i> }
write(<i>a</i> : 2);	{ визуализация глобального объекта процедуры <i>c</i> }
write(<i>d</i> : 2);	{ визуализация глобального объекта процедуры <i>c</i> }
write(<i>e</i> : 2)	{ визуализация локального объекта процедуры <i>c</i> }
end ; { <i>c</i> }	
begin	{ тело процедуры <i>b</i> }
write(<i>b</i> : 2);	{ визуализация локального объекта процедуры <i>b</i> }
<i>c</i>	{ вызов процедуры <i>c</i> }
end ; { <i>b</i> }	
begin	
write(<i>a</i> : 2);	{ визуализация локального объекта процедуры <i>a</i> }
<i>b</i>	{ вызов процедуры <i>b</i> }
end ; { <i>a</i> }	
begin	
write(<i>e</i> : 2);	{ визуализация глобального объекта программы }
write(<i>d</i> : 2);	{ визуализация глобального объекта программы }
<i>a</i> ;	{ вызов процедуры <i>a</i> }
writeln	
end.	

Результат работы программы:

1 2 3 4 5 4 3 2 6

Пример 47. Представим программу, в которой определены три процедуры с одним и тем же именем *a*, при этом одна из процедур является глобальной (0-ой уровень вложения), в которой определена другая локальная процедура (1-ый уровень вложения), в которой в свою очередь определена третья локальная процедура (2-ой уровень вложения):



{ Пример 47 }

{ Процедуры и функции }

<i>const e = 1;</i>	<i>{ константа e – глобальный объект программы }</i>
<i>procedure a;</i>	<i>{ процедура a (0–ой уровень) – глобальный объект программы }</i>
<i>const b = 2;</i>	<i>{ константа b – локальный объект процедуры a (0–ой уровень) }</i>
<i>procedure a;</i>	<i>{ процедура a (1–ый уровень) – локальный объект процедуры a (0–ой уровень) }</i>
<i>const c = 3;</i>	<i>{ константа c – локальный объект процедуры a (1–ый уровень) }</i>
<i>procedure a;</i>	<i>{ процедура a (2–ой уровень) – локальный объект процедуры a (1–ый уровень) }</i>
<i>const d = 4;</i>	<i>{ константа d – локальный объект процедуры a (2–ой уровень) }</i>
<i>begin</i>	<i>{ тело процедуры a (2–ой уровень) }</i>
<i>write(d : 2);</i>	<i>{ визуализация локального объекта процедуры a (2–ой уровень) }</i>
<i>write(c : 2);</i>	<i>{ визуализация глобального объекта процедуры a (1–ый уровень) }</i>
<i>write(b : 2);</i>	<i>{ визуализация глобального объекта процедуры a (0–ой уровень) }</i>
<i>write(e : 2)</i>	<i>{ визуализация глобального объекта программы }</i>
<i>end; { a }</i>	
<i>begin</i>	<i>{ тело процедуры a (1–ый уровень) }</i>
<i>write(c : 2);</i>	<i>{ визуализация локального объекта процедуры a (1–ый уровень) }</i>
<i>a</i>	<i>{ вызов процедуры a (2–ой уровень) }</i>
<i>end; { a }</i>	
<i>begin</i>	<i>{ визуализация локального объекта процедуры a (0–ой уровень) }</i>
<i>write(b : 2);</i>	
<i>a</i>	<i>{ вызов процедуры a (1–ый уровень) }</i>
<i>end; { a }</i>	

```
begin
  write(e : 2);           { визуализация глобального объекта программы }
  a;                     { вызов процедуры a (0-ой уровень) }
  writeln
end.
```

Результат работы программы:

1 2 3 4 3 2 1

Теперь рассмотрим несколько примеров, связанных с характерными приемами работы с процедурами. Представим реализации механизма визуализации элементов двумерных массивов с помощью программ, в которых определены различные варианты процедуры *print_matrix*.

Пример 48. Механизм обмена данными между вызываемой процедурой и вызывающей программой не использует список параметров вызова:

```
{ Пример 48 }
{ Процедуры и функции }
const  m = 2;
       n = 4;
type   row_size = array [1..n] of integer;
       matrix = array [1..m] of row_size;
const  a : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
procedure print_matrix;
  var  i, j : integer;
  begin
    for i := 1 to m do begin
      for j := 1 to n do write(a[i,j] : 3);
      writeln
    end
  end; { print_matrix }
begin
  print_matrix    { вызов процедуры }
end.
```

Пример 49. Механизм обмена данными между вызываемой процедурой и вызывающей программой использует список параметров вызова, состоящий из трех параметров-значений:

```
{ Пример 49 }
{ Процедуры и функции }
const  m = 2;
       n = 4;
```

```

type   row_size = array [1..n] of integer;
       matrix = array [1..m] of row_size;
const  a : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
procedure print_matrix(m, n : integer; a : matrix);
  var  i, j : integer;
  begin
    for i := 1 to m do begin
      for j := 1 to n do write(a[i,j] : 3);
      writeln
    end
  end; { print_matrix }
begin
  print_matrix(m, n, a)    { вызов процедуры }
end.

```

Пример 50. Механизм обмена данными между вызываемой процедурой и вызывающей программой использует список параметров вызова, состоящий из двух параметров-значений и одного параметра-переменного:

```

{ Пример 50 }
{ Процедуры и функции }
const  m = 2;
       n = 4;
type   row_size = array [1..n] of integer;
       matrix = array [1..m] of row_size;
const  a : matrix = ((1, 2, 3, 4), (5, 6, 7, 8));
procedure print_matrix(m, n : integer; var a : matrix);
  var  i, j : integer;
  begin
    for i := 1 to m do begin
      for j := 1 to n do write(a[i,j] : 3);
      writeln
    end
  end; { print_matrix }
begin
  print_matrix(m, n, a)    { вызов процедуры }
end.

```

Теперь рассмотрим пример, связанный с характерными приемами работы с функциями.

Пример 51. Протабулировать функцию $f(x) = e^x - e^{-x} - 2$ на отрезке $[0, 1]$ для заданного числа узлов табулирования n :

```

{ Пример 51 }
{ Процедуры и функции }
type   integer_type = integer;
       real_type = real;
const  a : real_type = 0;
       b : real_type = 1;
var     k, n : integer_type;
       x, h : real_type;
function f(x : real_type) : real_type;
begin
    f := exp(x) - exp(-x) - 2
end; { f }
begin
    repeat
        write('Число узлов табулирования? '); readln(n)
    until n > 1;
    h := (b - a) / (n - 1);
    for k:=0 to n - 1 do begin
        x := a + k * h;
        writeln((k + 1) : 2, x : 19, f(x) : 22)
    end
end.

```

Рекурсивные процедуры и функции

Рекурсия – это такой способ организации программы, когда какая-либо ее подпрограмма явно или неявно вызывает сама себя. Явный вызов подпрограммой самой себя принято называть прямой рекурсией, т.е. подпрограмму называют прямо рекурсивной, если в теле этой подпрограммы явно используется вызов этой же подпрограммы. Одним из ярких классических примеров прямо рекурсивной подпрограммы является функция для вычисления факториала неотрицательного целого числа:

$$n! = \begin{cases} 1 & \text{при } n = 0, \\ n \cdot (n - 1)! & \text{при } n > 0 \end{cases}$$

Пример 52. Вычислить факториал неотрицательного целого числа при помощи рекурсивной функции:

```

{ Пример 52 }
{ Процедуры и функции }
var  n : integer;
function factorial(n : integer) : longint;
begin
    if n = 0 then factorial := 1 else factorial := n * factorial(n - 1)

```

```

end; { factorial }
begin
  repeat
    write('n? '); readln(n)
  until (n >= 0) and (n <= 12);
  writeln(n, '!' = ', factorial(n))
end.

```

Рекурсивная функция *factorial(n)* позволяет вычислить корректное значение *n!* только для $n \in [0, 12]$, значение *13!* уже выходит за границы диапазона *longint*, верхняя граница которого равна значению *2147483647*:

$$13! = 6227020800$$

Рекурсивная форма организации алгоритма обычно выглядит изящнее итерационной и дает более компактный текст программы, но выполняется медленнее и может вызвать переполнение стека, так как при каждом вызове рекурсивной подпрограммы ее локальные переменные записываются во временную память – стек. Например, при вычислении *n!* с помощью рекурсивной функции *factorial(n)* ее локальная переменная *n*-раз записывалась в стек и соответственно столько же раз откладывалось завершение вычисления выражения в правой части инструкции присваивания *factorial := n * factorial(n - 1)*.

Рекурсивное определение функции *factorial(n)* можно заменить и нерекурсивным, например, с использованием счетной инструкции повторения.

Пример 53. Вычислить факториал неотрицательного целого числа при помощи функции для итерационного алгоритма:

```

{ Пример 53 }
{ Процедуры и функции }
var n : integer;
function factorial(n : integer) : longint;
  var i : integer;
      k : longint;
  begin
    if n = 0
    then factorial := 1
    else begin
      k := 1;
      for i := 1 to n do k := k * i;
      factorial := k
    end
  end; { factorial }

```

```

begin
  repeat
    write('n? '); readln(n)
  until (n >= 0) and (n <= 12);
  writeln(n, '! = ', factorial(n))
end.

```

Теперь обратимся к программным реализациям эвристических алгоритмов, взяв за основу метод дихотомии – один из известных численных методов отыскания действительных корней нелинейных уравнений.

Пример 54. Найти действительный корень нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$ с заданной точностью *Tolerance* с помощью функции с тремя параметрами *Dichotomy(a, b, Tolerance)*, реализация которой выполнена на основе циклического алгоритма:

```

{ Пример 54 }
{ Процедуры и функции }
type  real_type = real;
const a : real_type = 0.0;
      b : real_type = 1.0;
      tolerance : real_type = 1e-6;
function f(x : real_type) : real_type;
begin
  f := exp(x) - exp(-x) - 2
end; { f }
function dichotomy(a, b : real_type; tolerance : real_type) : real_type;
var  x_left, x_right : real_type;
     f_left, f_right : real_type;
     x_root, f_root : real_type;
begin
  f_left := f(a);
  f_right := f(b);
  if abs(f_left) < tolerance
  then if abs(f_left) < abs(f_right)
       then dichotomy := a
       else dichotomy := b
  else if abs(f_right) < tolerance
       then dichotomy := b
       else begin
            x_left := a;
            x_right := b;
            repeat
              x_root := (x_left + x_right) / 2;

```

```

        f_root := f(x_root);
        if f_left * f_root > 0
            then begin
                x_left := x_root;
                f_left := f_root
            end
            else x_right := x_root
        until abs(f_root) < tolerance
    end;
    dichotomy := x_root
end; { dichotomy }
begin
    writeln('Метод дихотомии');
    writeln('Корень = ', dichotomy(a, b, tolerance))
end.

```

Результат работы программы:

Метод дихотомии

Корень = 8.8137340546E-01

Пример 55. Найти действительный корень нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$ с заданной точностью *Tolerance* с помощью рекурсивной функции с четырьмя параметрами вызова *Dichotomy(F, a, b, Tolerance)*, где параметр *F* является параметром-функцией:

```

{ Пример 55 }
{ Процедуры и функции }
type    real_type = real;
        function_type = function(x : real_type) : real_type;
const   a : real_type = 0.0;
        b : real_type = 1.0;
        tolerance : real_type = 1e-6;
function f(x : real_type) : real_type;
begin
    f := exp(x) - exp(-x) - 2
end; { f }
function dichotomy(f : function_type; a, b : real_type; tolerance : real_type) : real_type;
var     f_left, f_right : real_type;
        x_root, f_root : real_type;
begin
    f_left := f(a);
    f_right := f(b);

```



```

    if abs(f_left) < tolerance
    then if abs(f_left) < abs(f_right)
        then dichotomy := a
        else dichotomy := b
    else if abs(f_right) < tolerance
        then dichotomy := b
        else begin
            x_root := (a + b) / 2;
            f_root := f(x_root);
            if abs(f_root) < tolerance
                then dichotomy := x_root
            else if f_left * f_root > 0
                then dichotomy := dichotomy(f, x_root, b, tolerance)
                else dichotomy := dichotomy(f, a, x_root, tolerance)
        end;
    end; { dichotomy }
begin
    writeln('Метод дихотомии');
    writeln('Корень = ', dichotomy(f, a, b, tolerance))
end.

```

Результат работы программы:

Метод дихотомии

Корень = 8.8137340546E-01

Неявный вызов подпрограммой самой себя принято называть косвенной рекурсией. Неявный или опосредованный вызов некоторой подпрограммой самой себя реализуется посредством механизма вложенных друг в друга вызовов других подпрограмм, пока какая-нибудь подпрограмма последнего уровня вложения не вызовет какую-нибудь подпрограмму предыдущего уровня вложения этой последовательности вызовов, глубина вложения при этом может быть произвольной. Напомним, что для реализации косвенной рекурсии, как правило, используется механизм опережающего объявления подпрограмм.

Для иллюстрации косвенной рекурсии обратимся к примеру с двумя глобальными рекурсивными процедурами, которые вызывают друг друга. В этом случае должен обязательно использоваться механизм опережающего объявления процедуры первого уровня вложения.

Пример 56. Найти действительный корень нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$ с заданной точностью *Tolerance* с помощью двух рекурсивных процедур с четырьмя параметрами вызова *Dichotomy(F, a, b, Tolerance)* и *Next_iteration(F, a, b, Tolerance)*, где параметр *F* является параметром-функцией:

{ Пример 56 }

{ Процедуры и функции }

```

type   real_type = real;
       function_type = function(x : real_type) : real_type;
const  a : real_type = 0.0;
       b : real_type = 1.0;
       tolerance : real_type = 1e-6;
function f(x : real_type) : real_type;
begin
    f := exp(x) - exp(-x) - 2
end; { f }
procedure next_iteration(f : function_type; a, b : real_type; tolerance : real_type);
forward;
procedure dichotomy(f : function_type; a, b : real_type; tolerance : real_type);
var f_left, f_right : real_type;
begin
    f_left := f(a);
    f_right := f(b);
    if abs(f_left) < tolerance
    then if abs(f_left) < abs(f_right)
        then writeln('Корень = ', a)
        else writeln('Корень = ', b)
    else if abs(f_right) < tolerance
        then writeln('Корень = ', b)
        else next_iteration(f, a, b, tolerance)
    end; { dichotomy }
procedure next_iteration(f : function_type; a, b : real_type; tolerance : real_type);
var x_root : real_type;
begin
    x_root := (a + b) / 2;
    if f(a) * f(x_root) > 0
    then dichotomy(f, x_root, b, tolerance)
    else dichotomy(f, a, x_root, tolerance)
end; { next_iteration }
begin
    writeln('Метод дихотомии');
    dichotomy(f, a, b, tolerance)
end.

```

Результат работы программы:

Метод дихотомии

Корень = 8.8137340546E-01

И в заключение для иллюстрации косвенной рекурсии обратимся теперь к примеру с глобальной рекурсивной процедурой и ее локальной рекурсивной процедурой, которые вызывают друг друга. Очевидно, что в этом случае отпадает необходимость использования механизма опережающего объявления процедуры первого уровня вложения, так как она является локальной по отношению к глобальной процедуре нулевого уровня вложения. Отпадает также необходимость и в самом списке параметров вызова локальной процедуры, так как параметры вызова глобальной процедуры для ее локальной процедуры всегда являются глобальными объектами.

Пример 57. Найти действительный корень нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$ с заданной точностью *Tolerance* с помощью рекурсивной процедуры с четырьмя параметрами вызова *Dichotomy(F, a, b, Tolerance)* и ее локальной рекурсивной процедуры *Next_iteration*, где параметр *F* является параметром-функцией:

{ Пример 57 }

{ Процедуры и функции }

type real_type = real;

function_type = function(x : real_type) : real_type;

const a : real_type = 0.0;

b : real_type = 1.0;

tolerance : real_type = 1e-6;

function f(x : real_type) : real_type;

begin

f := exp(x) - exp(-x) - 2

end; { f }

procedure dichotomy(f : function_type; a, b : real_type; tolerance : real_type);

procedure next_iteration;

var x_root : real_type;

begin

x_root := (a + b) / 2;

if f(a) * f(x_root) > 0

then dichotomy(f, x_root, b, tolerance)

else dichotomy(f, a, x_root, tolerance)

end; { next_iteration }

var f_left, f_right : real_type;

begin

f_left := f(a);

f_right := f(b);

if abs(f_left) < tolerance

then if abs(f_left) < abs(f_right)

then writeln('Корень = ', a)

else writeln('Корень = ', b)

```
    else if  $\text{abs}(f\_right) < tolerance$ 
        then writeln('Корень = ', b)
        else next_iteration
    end; { dichotomy }
begin
    writeln('Метод дихотомии');
    dichotomy(f, a, b, tolerance)
end.
```

Результат работы программы:

```
Метод дихотомии
Корень = 8.8137340546E-01
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Власов В.К., Королев Л.Н., Сотников А.Н. Элементы информатики / Под ред. Л.Н. Королева. – М.: “Наука”, 1988. – 320 с.
2. Бауер Ф.Л., Гооз Г. Информатика. Вводный курс: В 2-х ч. Ч. 1. Пер. с нем. – М.: Мир, 1990. – 336 с.
3. Бауер Ф.Л., Гооз Г. Информатика. Вводный курс: В 2-х ч. Ч. 2. Пер. с нем. – М.: Мир, 1990. – 423 с.
4. Фаронов В.В. Турбо Паскаль. В 3 кн. Кн. I. Основы Турбо Паскаля. – М.: “МВТУ–ФЕСТО ДИДАКТИК”, 1992. – 304 с.
5. Епанешников А.М., Епанешников В.А. Программирование в среде Turbo Pascal 7.0. – М.: “ДИАЛОГ–МИФИ”, 1993. – 288 с.

СОДЕРЖАНИЕ

Предисловие	3
Введение	4
Парадигмы программирования и Turbo Pascal	4
Императивное программирование и Turbo Pascal	7
Алгоритмы и алгоритмизация	8
Понятие алгоритма и алгоритмического процесса	8
Алгоритмические системы	10
Понятие о математической модели	12
Алгоритмизация	13
Типы алгоритмов	15
Линейный алгоритм	16
Условные алгоритмы	17
Циклические алгоритмы	20
Цикл с условием продолжения	20
Цикл с условием завершения	23
Рекурсивные алгоритмы	25
Эвристические алгоритмы	29
Введение в Turbo Pascal	34
Алфавит языка Turbo Pascal	34
Лексическая структура языка Turbo Pascal	34
Идентификаторы	35
Разделители	35
Литералы	35
Операторы	35
Ключевые слова	36
Стандартные директивы	37
Директивы компилятора	37
Структура программы Turbo Pascal	37
Демонстрационная программа Turbo Pascal	39
Первое знакомство с организацией ввода-вывода	40
Типы данных	43
Простые типы	44
Порядковые типы	44
Вещественные типы	49

Раздел объявлений программы	51
Объявление типов	51
Объявление констант	52
Объявление переменных	53
Объявление меток	53
Первое знакомство с инструкциями языка Turbo Pascal	54
Программирование линейных алгоритмов	58
Выражения	58
Арифметические функции	60
Преобразования типов	61
Неявные преобразования типов	61
Явные преобразования типов	62
Арифметические выражения	63
Программирование условных алгоритмов	67
Условная инструкция	67
Инструкция выбора	73
Программирование циклических алгоритмов	77
Инструкция повторения с предусловием	77
Инструкция повторения с постусловием	79
Счетная инструкция повторения	83
Структурированные типы. Массивы	90
Одномерные массивы	92
Двумерные массивы	97
Структурированные типы. Файлы	105
Доступ к файлам	105
Имена файлов	107
Логические устройства	107
Инициация файла	108
Процедуры и функции для работы с файлами	109
Текстовые файлы	111
Процедура read	112
Процедура readln	116
Процедура write	116
Процедура writeln	119
Процедуры и функции	120
Объявления процедур и функций пользователя	121
Формальные параметры процедур и функций	123

Рекурсивные процедуры и функции	133
Библиографический список	141